



Лекции по Алгоритми

работна версия, 2025 година

Минко Марков

Съдържание

Част I	Фундамент	2
1	Изчислителни задачи и алгоритми.	3
1.1	Алгоритъм – що е то?	3
1.1.1	Опит за дефиниция	3
1.1.2	Изчислителни задачи	6
1.1.2.1	Дефиниция	6
1.1.2.2	Видове изчислителни задачи	8
1.1.2.3	Как описваме изчислителните задачи	11
1.1.3	Изчислителни задачи и алгоритми	14
1.1.4	Инструкции	16
1.1.5	Knuth за алгоритмите	19
1.1.6	Изчислителни методи. Итератори.	23
1.1.7	Итеративни и рекурсивни алгоритми	24
1.2	Евклидовият алгоритъм	26
1.2.1	Оригиналният Евклидов алгоритъм	27
1.2.2	Съвременни варианти на Евклидовия алгоритъм.	29
1.3	Как описваме алгоритми	29
1.3.1	Псевдокод	29
1.3.2	Други видове описания на алгоритми	31
2	Анализ на алгоритми.	33
2.1	Анализ на коректността	34
2.1.1	Доказателство за финитност	35
2.1.2	Верификация на рекурсивни алгоритми: по индукция	38
2.1.2.1	НАРАСТВАНЕ С ЕДИНИЦА	38
2.1.2.2	МАКСИМУМ НА НЕСОРТИРАН МАСИВ, РЕКУРСИВЕН	39
2.1.2.3	МАКСИМУМ НА БИТОНИЧЕН МАСИВ	40
2.1.2.4	Евклидовият алгоритъм в рекурсивен вариант	43
2.1.3	Верификация на итеративни алгоритми: с инвариант	45
2.1.3.1	МАКСИМУМ НА НЕСОРТИРАН МАСИВ, ИТЕРАТИВЕН	49
2.1.3.2	Странен алгоритъм за сумиране на елементите на масив	51
2.1.3.3	Евклидовият алгоритъм в итеративен вариант	55
2.1.3.4	ALG 2SUM	55
2.1.3.5	ALG 3SUMCOUNT	57
2.1.3.6	ALG MAX SUM PLUS DIST	63
2.1.3.7	Намиране на най-малкото положително число, което липсва	68
2.1.3.8	BOYER–MOORE за намиране на мажорант	80
2.1.3.9	Една транспортна задача	84
2.2	Анализ на сложността	89

2.2.1	Големина на входа	89
2.2.2	Въведение в сложността по време	90
2.2.3	Определение на сложността по време	95
2.2.4	Сложност по памет	101
2.2.5	Проблеми със сложността по време, до които води нашият модел	103
2.3	Асимптотични нотации	103
2.3.1	Опит за намиране на точната сложност по време на прости алгоритми	103
2.3.2	Отказ от точна оценка и търсене на приблизителна оценка на сложността	106
2.3.3	Асимптотична нотация Θ	108
2.3.4	Други асимптотични нотации: O , Ω , o и ω	113
2.3.5	Релации $\asymp$, $\leq$, $<$, $\geq$ и $>$ и някои техни свойства	117
2.3.6	Асимптотични еквивалентности на често срещани функции	130
2.3.7	Задачи с решения върху асимптотични сравнения	140
2.3.7.1	Свойства на релациите на асимптотични сравнения	140
2.3.7.2	Асимптотични сравнения на двойки функции	147
2.3.7.3	Подреждане на функции по асимптотично нарастване	150
2.3.8	Релациите на асимптотични сравнения при целочислена променлива	165
2.3.9	Асимптотичните нотации и релациите на асимптотични сравнения	167
2.3.10	Релациите $\leq$ и $\geq$ като преднаредби	167
2.3.11	Асимптотични сравнения на функции на повече от една променлива	174
2.3.12	Релация $\sim$: друга релация на асимптотична еквивалентност	174
2.4	Примери за изчисляване на сложност по време на итеративни алгоритми	174
2.4.1	Алгоритми с единичен цикъл	174
2.4.2	Алгоритми с вложени цикли	175
2.4.3	Двоично търсене, итеративен вариант	180
2.5	Винаги ли е лоша високата сложност	183
3	Рекурентни уравнения.	187
3.1	Фундамент	187
3.1.1	Определение	187
3.1.2	Обобщения	188
3.1.3	Рекурентните уравнения са рекурсивни алгоритми	189
3.1.4	Класификации на рекурентните уравнения	190
3.1.5	Винаги ли променливата в рекурентно уравнение е целочислена	194
3.1.6	Решаване на рекурентни уравнения	196
3.1.7	Рекурентните уравнения в изследването на алгоритми	197
3.2	Методи за решаване на рекурентни уравнения	201
3.2.1	Чрез развиване	201
3.2.2	Чрез дървото на рекурсията	213
3.2.3	По индукция	218
3.2.4	Чрез Мастър теоремата (MT)	230
3.2.5	Чрез теоремата на Акга-Bazzi	243
3.2.6	Чрез метода с характеристичното уравнение	248
Част II Сортиране		275
4	Въведение. Елементарни сортираци алгоритми.	276
4.1	За задачата СОРТИРАНЕ	276

4.2	Защо сортирането е важно	277
4.2.1	Двоично търсене	278
4.2.2	Най-близки елементи	281
4.2.3	Уникалност на елементите	282
4.2.4	Откриване на еднакви елементи в два масива	283
4.2.5	Мода	284
4.2.6	Медиана	289
4.2.7	2SUM, 3SUM, kSUM	289
4.3	Анализ на сортиращи алгоритми. Стабилност.	292
4.4	Елементарни Сортиращи Алгоритми	294
4.4.1	INSERTION SORT	294
4.4.2	SELECTION SORT	303
5	Двоична пирамида. HEAPSORT. Приоритетна опашка.	306
5.1	Двоични дървета и пирамиди	306
5.1.1	Попълнено двоично дърво	306
5.1.2	Адреси в попълнено двоично дърво	314
5.1.3	Двоична пирамида	319
5.1.4	Реализиране на пирамида чрез масив	320
5.1.5	Подпирамида	324
5.2	Построяване на пирамида	326
5.2.1	Наивно построяване на пирамида	326
5.2.2	Бързо построяване на пирамида: алгоритъм BUILD HEAP	330
5.2.2.1	Предварителни обяснения	330
5.2.2.2	Неформално въвеждане на HEAPIFY	334
5.2.2.3	HEAPIFY в итеративен вариант	334
5.2.2.4	Алгоритъм BUILD HEAP	343
5.3	Сортиращ алгоритъм HEAPSORT	344
5.4	Приоритетни опашки	349
5.4.1	Абстрактен Тип Данни (АТД)	349
5.4.2	Приоритетна опашка: вид АТД	349
5.4.3	Реализация на приоритетни опашки с двоични пирамиди	351
5.4.4	Функцията INCREASE-KEY	352
5.4.5	Задачи, които се решават ефикасно с приоритетни опашки	353
6	Алгоритмична схема Разделяй-и-Владей. MERGESORT и QUICKSORT.	364
6.1	Разделяй-и-Владей	364
6.2	Примери за ефикасни Разделяй-и-Владей алгоритми	365
6.2.1	НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D	365
6.2.2	ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА	379
6.3	MERGESORT	391
6.4	QUICKSORT	397
6.4.1	Имплементация на фазата Разделяй чрез PARTITION-HOARE	399
6.4.2	Имплементация на фазата Разделяй чрез PARTITION-LOMUTO	406
6.4.3	Самият QUICKSORT	411
6.4.4	Сложност по време на QUICKSORT	412

7	Сортиране в линейно време. COUNTING SORT. RADIX SORT.	416
7.1	Въведение	416
7.2	COUNTING SORT	416
7.3	RADIX SORT	423
 Част III Алгоритми върху графи		427
8	Въведение в алгоритмите върху графи. Обхождания на графи.	428
8.1	Рекапитулация на графите	428
8.2	Обхождания на графи	433
8.3	Обща схема за обхождане	435
8.4	BFS	436
8.4.1	BFS от един стартов връх	437
8.4.2	BFS, обхождащ целия граф	444
8.4.3	Приложения на BFS	445
8.5	DFS	445
8.5.1	Неформално въведение и сравнение с BFS	445
8.5.2	Псевдокод на DFS	448
8.5.3	Свойства на DFS	448
8.5.3.1	Скобова структура на интервалите, генерирани от DFS	448
8.5.3.2	Категоризация на ребрата на ориентиран граф от DFS	452
8.5.3.3	Категоризация на ребрата на неориентиран граф от DFS	454
8.5.3.4	Изследване на цикличност на графи чрез DFS	455
9	Топосортиране, силно св. комп., срязващи върхове, мостове.	458
9.1	Фундамент	458
9.2	Алгоритъм на Tarjan за топологическо сортиране	462
9.3	Алгоритъм на Kahn за топологическо сортиране	464
9.4	Алгоритъм на Kosaraju за намиране на силно свързаните компоненти	467
9.4.1	Неуспешен опит за решение чрез едно пускане на DFS	469
9.4.2	Решение чрез две пускания на DFS	471
9.5	Алгоритми за намиране на срязващите върхове и на мостовете	481
9.5.1	Фундамент	481
9.5.2	Алгоритъм за ефикасно намиране на срязващите върхове	484
9.5.3	Алгоритъм за за ефикасно намиране на мостовете	490
10	Минимални покриващи дървета. Алгоритми на Prim и Kruskal.	492
10.1	Фундамент	492
10.2	Алчни алгоритми	495
10.3	Алгоритъм на Prim	496
10.3.1	Базов вариант на алгоритъма на Prim	497
10.3.2	Изтънчен вариант на алгоритъма на Prim	499
10.4	Алгоритъм на Kruskal	503
10.5	Union-Find структури данни	506
10.5.1	Два крайни подхода, които не вършат работа	506
10.5.2	Реализация на разбиване чрез ориентирана гора	507

11 Най-къси пътища. Dijkstra, дагове, Bellman-Ford.	518
11.1 Фундамент	518
11.1.1 Базови дефиниции	518
11.1.2 Варианти на задачата	520
11.1.3 Най-къс път се състои от най-къси подпътища	522
11.1.4 Дърво на най-късите пътища	522
11.1.5 Пак за отрицателните тегла	525
11.1.6 Релаксация	528
11.2 Алгоритъмът на Dijkstra в два варианта	531
11.3 Най-къси пътища в тегловни дагове	536
11.4 Алгоритъмът на Bellman-Ford	539
 Част IV Динамично програмиране	 541
12 Схемата Динамично Програмиране. Примери.	542
12.1 Въведение с примери	542
12.1.1 Задачата за опашката пред касата	542
12.1.2 Числата на Fibonacci	548
12.1.3 Боядисване на къщи в редица	555
12.1.4 Най-къс път за всяка двойка върхове	557
12.1.4.1 Алгоритъм, реализиращ матрично умножение	558
12.1.4.2 Алгоритъмът на Floyd-Warshall	564
12.2 Фундамент	567
12.2.1 Задачи, подзадачи и подподзадачи	567
12.2.2 Конституенти и декомпозиция	568
12.2.3 Състояния	569
12.2.4 От дърво на рекурсията към даг на рекурсията	571
12.2.5 Разделяй-и-Владей vs Динамично Програмиране	574
12.2.6 Какво казват експерти за Динамично Програмиране	574
12.2.7 За конструирането на оптимално решение.	578
12.3 Задачи за броене на комбинаторни структури	579
12.3.1 Биномни коефициенти	579
12.3.2 Числа на Stirling от втори род	583
12.3.3 Числа на Stirling от първи род	584
12.3.4 Брой на целочислени разбивания	586
12.4 Задачи за оптимални скобувания на редици (триъгълна таблица)	590
12.4.1 Matrix-Chain Multiplication	590
12.4.2 Minimum Weight Triangulation	605
12.4.3 Неасоциативна операция върху стринг	612
12.4.4 Деривация в контекстно-свободна граматика	615
12.5 Задачи за оптимални подмножества	617
12.5.1 2-Partition	617
12.5.2 2 Equal Sum Subsets (2-ESS)	626
12.5.3 Задачата за раницата (Knapsack)	631
12.5.3.1 Unbounded Knapsack	632
12.5.3.2 0-1 Knapsack	637
12.5.3.3 Bounded Knapsack	641
12.6 Задачи за планиране (Scheduling)	644

12.6.1	Задачата за безработния актьор (Interval Scheduling)	644
12.6.1.1	Алчно решение	645
12.6.1.2	Решение по схемата Дин. Пр. за тегловния вариант	651
12.7	Задачи върху редици и стрингове	654
12.7.1	Оптимално разбиване на редица	654
12.7.2	Longest Increasing Subsequence	660
12.7.2.1	Дин. Прогр. със сложност $\Theta(n^2)$	661
12.7.2.2	Дин. Прогр. със сложност $\Theta(n \lg n)$	662
12.7.2.3	PATIENCE SORT	668
12.7.2.4	Building Bridges	668
12.7.3	Longest Common Subsequence	668
12.7.4	Sequence Alignment	674
12.7.5	Edit Distance	685
12.7.6	Факторизация на стринг на минимален брой палиндроми	693
12.7.7	Разполагане на текст в параграф	694
12.8	NP -трудни задачи върху дървета	703
12.8.1	Минимално върхово покриване на дърво	703
12.8.2	Минимално доминиращо множество на дърво	709
12.9	Игри	719
12.9.1	The Daisy Petals Game	721
12.9.2	The Coins in a Line Game	725
12.10	Мемоизация	729
12.11	За границите на възможностите на динамичното програмиране	732

Част V Долни граници

734

13 Долни граници върху сложност на задачи.

735

13.1	Въведение	735
13.2	Дървета за вземане на решение	737
13.2.1	Задачите THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE	737
13.2.2	Долна граница $\Omega(n \lg n)$ за СОРТИРАНЕ	741
13.2.2.1	Сортиране, базирано на сравнения	741
13.2.2.2	Дървото на сравненията на INSERTION SORT за $n = 3$	742
13.2.2.3	Дървото на сравненията на SELECTION SORT за $n = 3$	747
13.2.2.4	Дървото на вземане на решение на произволен сортиращ алг.	748
13.2.2.5	Дървото за вземане на решение е изчислителен модел	749
13.2.2.6	За структурата на сортиращите дървета за вземане на решение	750
13.2.2.7	Долна граница $\Omega(n \lg n)$ чрез дървета за вземане на решение	752
13.2.2.8	Минималният брой на сравненията в най-добрия случай е $n - 1$	752
13.2.3	Долна граница $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ	756
13.2.4	Долна граница $\Omega(\lg n)$ за ТЪРСЕНЕ	760
13.2.5	Лампи и ключове	761
13.3	Редукции	766
13.3.1	Долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ	766
13.3.2	Долна граница $\Omega(n \lg n)$ за МОДА	768
13.3.3	Долна граница $\Omega(n \lg n)$ за 2SUM във версия за броене	769
13.3.4	Долна граница $\Omega(n \lg n)$ за INTERVAL SCHEDULING.	770
13.3.5	Относителна долна граница: DEGENERACY TESTING е 3SUM-hard	770

13.4	Аргументиране чрез противник (Adversary argument)	773
13.4.1	Неформално обяснение	773
13.4.2	Формално определение	779
13.4.3	Долна граница $\lceil \log_2 n \rceil$ за намиране на число от $\{1, \dots, n\}$	780
13.4.4	Долна граница $n - 1$ за намиране на максималния елемент	781
13.4.5	Долна гр. $\lceil \frac{3n}{2} \rceil - 2$ за намиране на максималния и минималния елемент	784
13.4.6	Долна гр. $n - 2 + \lceil \log_2 n \rceil$ за намиране на втория по големина елемент	791
13.4.7	Долна граница $\lceil \frac{3n-2}{2} \rceil$ за намиране на медиана	806
13.4.8	Долна граница $2n - 1$ за сливане на сортирани масиви	815
13.4.9	Долна граница $2n - 1$ за търсене в $n \times n$ таблица на Young	816
13.4.10	Долна граница n^2 за установяване на свързаност на граф с $2n$ върха	823
13.4.11	Задачата за яйцето	823
13.4.12	Трудноуловими подстрингове и свойства	823
13.4.12.1	Трудноуловими подстрингове	824
13.4.12.2	Трудноуловими графови свойства	826

Част VI Алгоритмична неподатливост 834

14	NP-пълнота и NP-трудност.	835
14.1	Въведение с пример и илюстрация	835
14.2	Алгоритмична неподатливост	838
14.3	Задачите като формални езици	842
14.3.1	Всяка задача за разпознаване в някакъв смисъл е формален език	842
14.3.2	Подходящи кодираня на задачи за разпознаване	844
14.4	Детерминирани машини на Turing	845
14.4.1	Фундамент	845
14.4.2	Съответствие между разпознаване на езици и решаване на задачи за разпознаване от машини на Turing	848
14.4.3	Пример за машина на Turing	848
14.4.4	Машини на Turing-изчислители на стрингови функции	852
14.4.5	Сложност по време на машина на Turing. Клас на сложност P .	854
14.4.6	Сложност по памет на машина на Turing.	855
14.5	Недетерминизъм и недетерминирани машини на Turing	856
14.5.1	НМТ с размножаване	856
14.5.2	НМТ-верификатор	863
14.5.3	Еквивалентност на двете дефиниции на НМТ	864
14.5.4	Какви нови възможности ни дава недетерминизмът	868
14.5.5	Сложност по време на НМТ. Клас на сложност NP .	868
14.6	Полиномиални редукции	872
14.6.1	Фундамент	873
14.6.2	Основни свойства	874
14.6.3	Примери за полиномиални редукции	878
14.6.4	Karp редукции срещу Turing редукции	881
14.7	NP -пълнота	887
14.8	NP -трудност	888

15 SAT и теоремата на Cook.	890
15.1 Синтаксис на булевите формули	890
15.2 Булеви функции	891
15.3 Семантика на булевите формули. Удовлетворимост.	891
15.4 Конюнктивни и Дизюнктивни нормални форми	894
15.4.1 Дизюнктивна нормална форма	894
15.4.2 Конюнктивна нормална форма	896
15.4.3 Превръщане на формули в КНФ или ДНФ	897
15.5 SATISFIABILITY (SAT) и моделиращата сила на КНФ	899
15.5.1 Моделиране на sudoku позиция чрез КНФ.	899
15.5.2 Моделиране на ограничено поставяне на етикети чрез КНФ.	905
15.5.3 Моделиране на наличие на Хамилтонов път чрез КНФ.	908
15.6 SAT е NP -пълна	909
15.7 SAT е началото на редици от доказателства на NP -пълнота	919
15.8 Географията на NP	920
16 Основни NP-пълни задачи.	924
16.1 Дървото на доказателствата на NP -пълнота.	924
16.2 Редукциите.	926
16.2.1 3SAT, MAX2SAT и NAESAT	926
16.2.2 MAXCUT	946
16.2.3 VERTEX COVER	950
16.2.4 INDEPENDENT SET, CLIQUE и DOMINATING SET	955
16.2.5 0-1 INTEGER PROGRAMMING	962
16.2.6 3-COLORABILITY, k-COLORABILITY и PLANAR 3-COLORABILITY	967
16.2.7 3-DIMENSIONAL MATCHING (3DM)	984
16.2.7.1 Съчетание – дефиниция, частни случаи и приложения	984
16.2.7.2 Хиперграфите като обобщение на обикновените графи	991
16.2.7.3 3DM – обобщение на ПЕРФЕКТНО СЪЧЕТАНИЕ	994
16.2.7.4 3DM е неподатлива	995
16.2.8 HAMILTONIAN CYCLE, HAM. PATH, LONGEST CYCLE, LONGEST PATH	1006
16.2.9 2-PARTITION	1018
16.2.10 SUBSET SUM	1024
16.2.11 UNBOUNDED KNAPSACK	1024
16.2.12 0-1 KNAPSACK	1029
16.2.13 BIN PACKING	1030
16.2.14 MULTIPROCESSOR SCHEDULING	1032
16.3 Когато принадлежността на задача към NP не е очевидна.	1033
16.3.1 $\star$ -FREE REGEXP INEQ	1033
16.3.2 ART GALLERY	1045
17 Заобикаляне на неподатливостта.	1062
17.1 Силна и слаба NP -пълнота	1062
17.2 Апроксимиращи алгоритми	1062
17.3 Параметризирана сложност	1062
Благодарности	1063
Библиография	1065

Списък на фигурите

1.1	Dixit algorismi.	4
1.2	Вход, алгоритъм, изход.	14
1.3	Блок-схема на Евклидовия алгоритъм.	32
2.1	Видове сложност по време.	100
2.2	Графиките на n^2 , $3n^2$ и $(2 + \sin n)n^2$	110
2.3	Графиките на n , $n^{1+\sin n}$, 1 и n^2	121
2.4	Графиките на $2^{2^{[n]}}$ и $2^{2^{[n]}}$	122
2.5	$\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx$	131
2.6	$\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx$	132
2.7	$\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx = \int_1^4 \frac{4- x }{x} dx$	133
2.8	$\int_1^h \frac{1}{x} dx$ като площ на район.	137
2.9	$\sum_{i=1}^{n-1} \frac{1}{i}$ като площ на район.	138
2.10	$\sum_{i=2}^n \frac{1}{i}$ като площ на район.	138
2.11	Графиките на n и $n^{1+\sin 2\pi n}$	166
2.12	Диаграма на преднаредба.	168
2.13	Фактор-релацията на пълна преднаредба.	170
2.14	Преднаредба, която не е пълна.	171
2.15	Фактор-релация на преднаредба, която не е пълна.	172
3.1	Дървото на рекурсията, отговарящо на $T(n) = 2T\left(\frac{n}{2}\right) + n$	214
3.2	Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	214
3.3	Схема на дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	215
3.4	Друга схема на дървото при $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	216
3.5	Дърво на рекурсията на $T(n) = 2T(n-1) + 1$	217
3.6	Дърво на рекурсията на $T(n) = T(n-1) + 2T(n-2) + 2$	218
3.7	MT: дърво на рекурсията при $T(n) = aT\left(\frac{n}{b}\right) + f(n)$	233
4.1	Блок-схема на COMPUTE MODE.	285
5.1	Попълнено двоично дърво съгласно Определение 54.	309
5.2	Височини на върхове в попълнено двоично дърво.	313
5.3	Изтриване на върховете с височини $< k$	314
5.4	Попълнено дърво с 14 върха и техните адреси.	322
5.5	Пирамида с 26 върха.	325
5.6	Линеаризацията на пирамида от Фигура 5.5.	325
5.7	Листата на пирамидата от Фигура 5.5.	325
5.8	Подпирамидата $A[[3]]$	325
5.9	Листата на $A[[3]]$ в червено.	325

5.10	Почти-пирамидата, получена от пирамидата от Фиг. 5.5 с един добавен връх.	327
5.11	Илюстрация на работата на NAIVE BUILD HEAP.	327
5.12	Понякога по-изгодно коренът да бъде разменен с детето с по-малкия ключ.	331
5.13	Ако разменяме връх с детето с по-голям ключ	332
5.14	След 4 размени на връх с детето с по-голям ключ получаваме пирамида.	332
5.15	След 2 размени на връх с детето с по-малък ключ получаваме пирамида.	333
5.16	Може разменянето на връх с по-малкото дете да доведе до $\Omega(n)$ размени.	333
5.17	Пирамидата, получена от дървото на Фигура 5.16 с $\Omega(n)$ размени.	333
5.18	АТД приоритетна опашка като черна кутия.	350
6.1	Точките, които ще ползваме за пример в НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D.	366
6.2	Бисекторът ℓ .	370
6.3	Дегенеративни случаи. ℓ си остава бисектор.	371
6.4	2d-лентата.	373
6.5	2d-лентата в примера от Фигура 6.1.	374
6.6	Възможните други краища на къса прекосяваща отсечка са малко.	375
6.7	Прозорецът върху примера от Фигура 6.1.	376
6.8	Прозорецът може да обхваща най-много осем точки.	377
6.9	Диаграма на Hasse на данните след ред 8 на SELECT.	383
6.10	Диаграма на Hasse на данните след ред 10 на SELECT.	384
6.11	Диаграмата на Hasse след като m стане медианата на медианите.	386
8.1	BFS в момента t_k .	443
8.2	Обхождане на дърво от BFS и от DFS.	447
8.3	Интервалите, генерирани от различни изпълнения на DFS.	449
8.4	Интервалите и гората на обхождането на DFS.	449
8.5	Ориентиран граф със сдвоени ребра и неориентиран граф.	454
8.6	Различен брой ребра назад.	456
9.1	Даг и неговото топо-сортиране.	459
9.2	Ориентиран граф и неговият фактор-граф.	468
9.3	SC-DUMMY при различни наредби π' и π'' на върховете.	469
9.4	Различни наредби на V индуцират различни наредби на $\text{SC}(G)$.	473
9.5	Транспонираният граф на графа от Фигура 9.2.	478
9.6	Срязващ връх и срязващо ребро.	482
9.7	Разцепване на връх.	483
9.8	Блокове в графи.	484
9.9	Кога коренът на дървото на DFS е срязващ връх.	485
9.10	Кога не-листо, не-корен на дървото на DFS е срязващ връх.	486
9.11	В алгоритъма FCV-REC: когато променливата x съдържа не-листо, не-корен.	488
9.12	В алгоритъма FCV-REC: черните върхове се прескачат.	489
10.1	Прехвърляне на връх от границата в дървото при Prim.	498
10.2	След Union(1, 2) и Union(3, 4).	508
10.3	Два варианта за Union(1, 3) върху гората от Фигура 10.2.	508
10.4	Два варианта за Union(1, 5) върху Вар. 1 от Фиг. 10.2.	510
10.5	Union by rank: правилен и неправилен.	511
11.1	Дървесни, гранични и неизвестни върхове в алгоритъма на Dijkstra.	535
12.1	Идентифициране на върхове в дърво на рекурсия.	572

12.2	Минимално върхово покриване в графа на Petersen.	704
12.3	Редуване на нива не дава минимално върхово покриване на дърво.	706
12.4	Намиране на максимална антиклика в дърво с 22 върха.	708
12.5	Минимално доминиращо множество в графа на Petersen.	710
12.6	$D(T)$ е м-во с мин. мощност измежду тези $k + 1$ м-ва.	716
12.7	$\tilde{D}(T)$ е м-во с мин. мощност измежду тези две м-ва.	716
13.1	Схема за измервания за THE BALANCE PUZZLE.	738
13.2	Схема от измервания за THE TWELVE-COIN PUZZLE.	740
13.3	Дървото на сравненията на INSERTION SORT върху $\langle a_1, a_2, a_3 \rangle$	744
13.4	Дървото на сравненията на INSERTION SORT не различава псевдо-пермутациите.	745
13.5	Дървото на сравненията на SELECTION SORT върху $\langle a_1, a_2, a_3 \rangle$	747
13.6	Не е достатъчно всяка двойка да е сравнена някъде в дървото, за да бъдат различавани пермутациите.	750
13.7	Пример за даг на сравненията при гарантирано уникални елементи.	755
13.8	Даг на сравненията с Хамилтонов път.	755
13.9	В най-добрия случай, с две сравнения се намира повтаряне на елементи.	757
13.10	Пример за даг на сравненията при възможни повтаряния на елементи.	758
13.11	ДА-екземпляр на 3SUM поражда ДА-екземпляр на DEGENERACY TESTING.	772
13.12	Аргументация с противник: два алгоритъма.	779
13.13	Диаграма на състоянията и преходите при сравняване на числа.	786
13.14	Турнир с 8 участници.	791
13.15	Дървото на турнир с 33 участници.	792
13.16	Дървото от Фигура 13.15 след редуцирането.	793
13.17	$ S = 1$: противникът няма да допусне това.	795
13.18	$ S = \lfloor \log_2 n \rfloor$: противникът може да постигне това.	796
13.19	$ S = n - 1$, ако турнирът е организиран неоптимално.	796
13.20	Няма Хамилтонов път, но a_2 е в средата на всяко топо-сортиране.	807
13.21	Единствена медиана е сравнима с всички елементи.	809
13.22	Полезни и безполезни ребра за намиране на медианата.	810
14.1	Машина на Turing приема палиндром.	849
14.2	Машина на Turing събира числа.	853
14.3	Композиция на машини-изчислители и на съответните функции.	853
14.4	Дървото на конфигурациите на недетерминирана машина на Turing.	858
14.5	Граф, в който търсим с връщане Хамилтонов цикъл.	860
14.6	Сравнение на търсене с връщане с недетерминирано търсене.	861
14.7	Машина-верификатор симулира машина с размножаване.	866
14.8	Композицията на MT, илюстрираща композиция на редукции.	875
14.9	Схема на полиномиална редукция $L_1 \leq_p L_2$	877
14.10	Схема на Turing редукция $L_1 \leq_T L_2$	883
14.11	Turing редукция HAMILTONIAN CYCLE към HAMILTONIAN PATH.	884
14.12	Карг редукция HAMILTONIAN CYCLE към HAMILTONIAN PATH.	886
15.1	ДА-екземпляр на RESTRICTED LABEL PLACEMENT.	905
15.2	НЕ-екземпляр на RESTRICTED LABEL PLACEMENT.	906
15.3	Началното съдържание на лентата на $M(x)$ в теоремата на Cook.	913
15.4	Пространство-времето в теоремата на Cook.	914
15.5	Географията на NP в светлината на теоремата на Ladner.	921

16.1	Дървовидната структура на доказателствата за NP -трудност.	924
16.2	2SAT формулата ϕ_1 и нейният импликационен граф G_1	933
16.3	Импликационният граф G_1 от Фигура 16.2, нарисуван по-прегледно.	933
16.4	Импликационният граф от Фигура 16.2 след махане на $(x_2 \vee \overline{x_4})$ от ϕ_1	934
16.5	Неудачен импликационен граф.	934
16.6	Фактор-графът на графа от Фигура 16.4.	939
16.7	Топо-сортиране на фактор-графа от Фигура 16.6.	939
16.8	Пример за $\text{NAESAT} \leq_p \text{MAXCUT}$	949
16.9	И двата графа от $\text{VERTEX COVER} \leq_p \text{DOMINATING SET}$	958
16.10	Оптимумът на ILP не опт. на LP след приближение.	964
16.11	Първият вид джаджки в редукцията $3\text{SAT} \leq_p 3\text{-COLORABILITY}$	967
16.12	Частично 3-оцветяване на екземпляра от Фигура 16.11.	968
16.13	ДА-екземпляр на 3SAT и съответният ДА-екземпляр на 3-COLORABILITY.	971
16.14	Превръщане на непланарен в планарен граф с добавяне на връх.	973
16.15	3-оцветяванията на джаджката от $3\text{-COLOR} \leq_p \text{PLANAR } 3\text{-COLOR}$	977
16.16	Граф G и максимално по включване съчетание M в него.	984
16.17	Максимално по мощност съчетание M' в G от Фигура 16.16.	984
16.18	Теоремата на Hall следва от теоремата на König.	990
16.19	Пример за хиперграф.	991
16.20	Двуделният граф, описващ хиперграфа H от Фигура 16.19.	993
16.21	Максимално съчетание в триделен хиперграф.	994
16.22	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): звездите.	997
16.23	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): короните.	998
16.24	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): звездите и короните заедно.	999
16.25	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): слагаме една тройка в съчетанието.	999
16.26	$3\text{SAT} \leq_p 3\text{DM}$ върху екз. от (16.10): u -тройките на $A_{1,1}$ са задължителни.	1000
16.27	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): забранени тройки.	1002
16.28	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): възможност за $A_{2,2}$	1002
16.29	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): възможност за $A_{2,3}$	1003
16.30	$3\text{SAT} \leq_p 3\text{DM}$ върху екземпляра от (16.10): от всяка корона влиза тройка.	1004
16.31	Основното приспособление във $\text{VERTEX COVER} \leq_p \text{HAMILTONIAN PATH}$	1007
16.32	Трите начина за минаване на Хамилтонов цикъл през джаджката.	1008
16.33	Веригата p_z през всички z -страни на приспособления.	1009
16.34	Петте приспособления от примера за $\text{VERTEX COVER} \leq_p \text{HAMILTONIAN CYCLE}$	1010
16.35	Добавянето на ребрата от E'' във $\text{VERTEX COVER} \leq_p \text{HAMILTONIAN CYCLE}$	1011
16.36	Веригите във $\text{VERTEX COVER} \leq_p \text{HAMILTONIAN CYCLE}$	1011
16.37	$\text{VERTEX COVER} \leq_p \text{HAMILTONIAN CYCLE}$: окончателният J	1012
16.38	Заменяне на z -страната с Хамилтонов път между z -краищата.	1014
16.39	Двете вериги от P_u в примера.	1015
16.40	Сега вече веригите минават през всички върхове на джаджки.	1015
16.41	Има Хамилтонов цикъл!	1016
16.42	Multiprocessor scheduling като подреждане в стекове.	1032
16.43	Географията на NP и coNP	1044
16.44	Това е Turing, а не Karp, редукция.	1044
16.45	Вариантите на ART GALLERY имат значение за оптимума.	1045
16.46	n -ъгълник, чието охраняване иска $\lfloor \frac{n}{3} \rfloor$ охранители.	1046
16.47	Пример за $\text{VERTEX COVER} \leq_p \text{ART GALLERY}$: започваме с графа на Petersen.	1049
16.48	$\text{VERTEX COVER} \leq_p \text{ART GALLERY}$: десетоъгълникът.	1049
16.49	Изрязваме ϵ около върховете и получаваме отвори.	1050

16.50	Построяваме алея 1-2.	1051
16.51	Всички алеи през отвора 1. Те са видими от точка 1'.	1052
16.52	Всички алеи.	1053
16.53	Запречваме алеите и получаваме трапéци.	1054
16.54	Конструираният Р е десетоъгълник с коридори.	1055
16.55	Група коридори 5.	1056
16.56	Десет охранители—в жълтите точки—са достатъчни.	1057
16.57	Шест охранители също са достатъчни.	1058

Списък на таблиците

8.1	Сложностите на действията върху графи при двете представяния.	431
12.1	Таблица за изчисляване на биномни коефициенти.	580
12.2	Таблица за изчисляване на числа на Stirling от втори род.	584
12.3	Таблица за изчисляване на числа на Stirling от първи род.	586
12.4	Таблица за изчисляване на броеве на целочислени разбивания.	589
12.5	Пример за работата на ALG UNBOUNDED KNAPSACK, SOLUTION.	637
12.6	Печеливши и губещи целочислени разбивания.	723
13.1	Стратегията на противника при намиране максимум и минимум.	787
14.1	Машина на Turing M_{pal} , която решава задачата ПАЛИНДРОМИ.	849
15.1	Променливите в редукцията в Теоремата на Cook.	915
16.1	Сложност на базови задачи върху КНФ/ДНФ.	1043

Списък на избрани алгоритми

1	Max Bitonic	41
2	Alg 2SUM	56
3	Alg Max Sum Plus Dist	64
4	Alg BSearch Iter	278
5	Insertion Sort	294
6	Naive Build Heap	327
7	Iterative Heapify	334
8	Build Heap	343
9	Heapsort	344
10	Partition–Select	380
11	Select	380
12	Merge	393
13	Mergesort	394
14	Partition–Lomuto	406
15	Quicksort	411
16	Counting Sort	417
17	BFS	437
18	DFS	448
19	DFS-VISIT	448
20	Tsort	462
21	Tsort-rec	462
22	StrConn	472
23	SC-rec	472
24	Kosaraju	472
25	Find Cut Vertices	487
26	FCV-rec	487
27	МПД Prim2	500
28	МПД Kruskal	504

Списък на допълненията

1	За произхода на думата “алгоритъм”	4
2	Не може ли изч. задача да е функция $\mathbf{P} : \mathcal{J} \rightarrow 2^{\mathcal{S}}$?	7
3	Функциите са повече от алгоритмите!	15
4	Изчислителни модели	16
5	On-line и off-line алгоритми	19
6	Алгоритмичната нерешимост на HALTING PROBLEM	21
7	Рандомизирани алгоритми	23
8	За разликата между <i>ефективност</i> и <i>ефикасност</i>	33
9	Когато коректността не е стопроцентова.	34
10	Не-финитност и частични функции	35
11	Други ресурси освен времето и паметта	89
12	Принцип на Landauer	93
13	P_n : числата на Bell с наредба	97
14	Сл. по време ограничава сл. по памет отгоре и отдолу	102
15	Цикъл for на ниво машинни инструкции	104
16	Друга интерпретация на появата на Θ вляво	111
17	Нотациите O и o в анализа	116
18	За преднаредбите	167
19	Няколко рекурентни уравнения чрез развиване	206
20	Няколко рекурентни уравнения по индукция	226
21	Решение на $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$	234
22	Условието за регулярност в МТ влече Случай 3	237
23	Ограничена МТ следва от метода с хар-чното у-ние	238
24	Едно разширение на Теорема 41	241
25	Доказателство на теоремата на Akra-Bazzi	245
26	Извеждане на решението на линейните рек. у-ния	250
27	Когато има комплексни характеристични корени	270
28	Двоичното търсене в рекурсивен вариант	280
29	Коректността на COMPUTE MODE	284
30	Неформално за доказателствата за коректност	294
31	INSERTION SORT чрез суопове	300
32	За броя на пирамидите на n върха	322
33	Друг възможен псевдокод на ITERATIVE HEAPIFY	334
34	HEAPIFY в рекурсивен вариант	340
35	HEAPSORT има сложност $\Theta(n \lg n)$ в най-добрия случай	346
36	Таблица на Young: аналог на двоична пирамида	357
37	За произхода на термина “Разделяй-и-Владей”.	365
38	Как да изчисляваме индексите на елементите на B	368
39	Решение на $T(n) = T(n/5) + T(7n/10) + n$ чрез Akra-Bazzi	385

40	Когато константата в алгоритъма SELECT не е 5	386
41	Най-лош случай за първото викане в SELECT	388
42	Бързо намиране на броя на инверсиите в масив.	395
43	За избора на pivot в QUICKSORT.	398
44	Коректността на PARTITION-HOARE	403
45	Друго доказателство за коректността на BFS	442
46	Разликата между BFS и DFS, илюстрирана с шах	446
47	За произхода на “топологическо сортиране”.	460
48	Команда <code>tsort</code> .	461
49	Частична наредба $\sqsubseteq$ върху разбиванията	505
50	Историята на МПД: Borůvka, Prim, Kruskal, anti-Kruskal	515
51	Генерирането на всички най-къси пътища е неефикасно.	521
52	За структурата на най-дългите пътища	526
53	Намиране на арбитражи чрез отрицателни цикли	527
54	За подобие между МПД PRIM2 и SSSHP DIJKSTRA2.	532
55	Колко са булевите вектори без съседни единици	543
56	Най-бързото пресмятане на числата на Fibonacci	550
57	За произхода на “Динамично Програмиране”	577
58	Имплементацията на <code>binomial(n,k)</code> в Maple(TM)	581
59	За числата на Catalan	593
60	Броят на триангулациите на изпъкнал n -ъгълник	606
61	Още за псевдополиномиалната сложност	623
62	Алчността на алчните алгоритми: глупава или умна	647
63	МАКСИМАЛНА АНТИКЛИКА върху интервални графи	649
64	За алгоритъма на Needleman и Wunsch	684
65	EDIT DISTANCE срещу SEQUENCE ALIGNMENT	686
66	VERTEX COVER и EDGE COVER са различни задачи	705
67	Долни граници, задачи и изчислителни модели	736
68	Защо допуснахме, че a_1 , a_2 и a_3 са уникални?	744
69	Thinking outside the box – добро или лошо?	762
70	$f(n) \neq \Omega(g(n))$ не влече $f(n) = o(g(n))$	767
71	Морски шах с дявола	774
72	Безсмислено за макс&мин сравнение е смислено за сорт.	789
73	Една задача върху графи и Лема 50	800
74	Грешно д-во на долната гр. $n - 2 + \lceil \log_2 n \rceil$ за втори макс.	802
75	Накратко за историята на неподатливостта	839
76	Backtracking и дървото на конфигурациите на HMT	858
77	Защо $P \stackrel{?}{=} NP$ е толкова важен	870
78	За името “TRAVELING SALESMAN PROBLEM”	878
79	Удовлетворимост на всякакви формули с булев смисъл	892
80	BOUNDED HALTING PROBLEM е NP-пълна задача	909
81	$2SAT \in P$	930
82	Намиране на хроматичното число чрез Maple (TM)	974
83	Едно приложение на СЪЧЕТАНИЕ в общите графи	985
84	Теорема на König и Hall	987
85	За броя на графите и хиперграфите	993
86	Задачи-допълнения и клас на сложност coNP	1038
87	Point Visibility Graphs	1047

Списък на теоремите

1	HALTING PROBLEM е алгоритмично нерешима	21
2	Голяма Теорема на Fermat	36
3	Коректността на алгоритъма НАРАСТВАНЕ С ЕДИНИЦА	38
4	Коректността на алгоритъма MAX UNSORTED, REC	39
5	Коректността на алгоритъма MAX BITONIC	41
6	$\gcd(a, b)$ е или a , или $\gcd(b, a \bmod b)$	44
7	Коректността на алгоритъма EUCLID REC	45
8	Коректността на алгоритъм за сумата от елементите на масив	46
9	Коректността на алгоритъма MAX UNSORTED, ITER	49
10	Коректността на ALG 2SUM	56
11	Коректността на ALG 3SUMCOUNT	58
12	Коректността на алгоритъма ALG MAX SUM PLUS DIST	66
13	Коректността на SMALLEST POSITIVE MISSING	79
14	Коректността на BOYER-MOORE	83
15	Коректността на TRANSPORT	86
16	Свойства на нотацията Θ	110
17	Θ е сечението на O и Ω	114
18	Несъвместимости между асимптотичните нотации	114
19	Транспонирана симетрия между асимптотични нотации	115
20	$<$ влече $\leq$ и $>$ влече $\geq$	119
21	$\leq$ не влече $<$ и $\geq$ не влече $>$	120
22	$<$ е несъвместима с $\geq$ и $>$ е несъвместима с $\leq$	120
23	$<$ и $>$ са несъвместими с $\asymp$	120
24	Различните възможности при асимптотично сравняване на функции	121
25	Повдигането на константна степен запазва $\asymp$	124
26	Водещото събираемо определя асимптотиката	124
27	Експонирането запазва $<$	125
28	Логаритмуването запазва $\asymp$	126
29	Полилогаритмичното нарастване е по-бавно от полиномиалното	128
30	Полиномиалното нарастване е по-бавно от експоненциалното	128
31	Итерирането на логаритъма дава по-бавно асимпт. нарастване	130
32	Апроксимация на Stirling, слаба версия	130
33	Асимптотиката на $\lg n!$	135
34	Асимптотиката на $\sum_{k=1}^n k \lg k$	136
35	Асимптотиката на $\binom{n}{n/2}$	136
36	Асимптотиката на H_n	137
37	Сложността на двоичното търсене е логаритмична	183
38	$T(n) = \sum_{i=1}^{\ell} T(n_i) + \Theta(\ell)$ по индукция	224
39	$T(n) = \sum_{i=1}^k T(n/b_i) + n$ по индукция	225

40	Решенията на клас рекурентни уравнения (Theorem 2.1 в [3])	231
41	Мастър теорема (Theorem 4.1 от [34, стр. 94])	231
42	Разширение на МТ	241
43	Теорема на Akra-Bazzi (Theorem 1 от [105])	244
44	Решение на линейно хомогенно рекурентно уравнение	248
45	Решение на линейно нехомогенно рекурентно уравнение	250
46	М-во решения е фундаментално тстк дет. на Casorati е ненулева	258
47	Фундаментално множество от решения съществува	258
48	Принцип на суперпозицията	259
49	Фундаментално множество от решения и конкретно решение	260
50	Решенията на хомог. лин. рек. у-ние образуват линейно пр-ство	260
51	Фундаментално множество от решения при кратни корени	264
52	Разликата м-у две реш. на нехомогенното е реш. на хомогенното	266
53	Общото решение на нехомогенното уравнение	267
54	Коректността на ALG BSEARCH ITER	279
55	Коректността на ALG BSEARCH REC	280
56	Коректността на INSERTION SORT	299
57	Коректността на INSERTION SORT, SWAP	302
58	Коректността на SELECTION SORT	304
59	Височината на попълнено двоично дърво	309
60	Броят на листата на попълнено двоично дърво	311
61	$\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$ и $\lfloor f(\lceil x \rceil) \rfloor = \lfloor f(x) \rfloor$	312
62	Броят на върховете с височина k в попълнено дърво	314
63	Коректността на наивното построяване на пирамида	329
64	Коректността на ITERATIVE HEAPIFY	336
65	Коректността на RECURSIVE HEAPIFY	340
66	Коректността на бързото построяване на пирамида	343
67	Бързото построяване на пирамида става в линейно време	343
68	Коректността на HEAPSORT	344
69	Коректността на алгоритъма SELECT	382
70	SELECT е линеен алгоритъм	382
71	Коректността на MERGE	393
72	Коректността на MERGESORT	394
73	Коректността на PARTITION-HOARE	404
74	Коректността на PARTITION-LOMUTO	410
75	COUNTING SORT е стабилен сортиращ алгоритъм	421
76	Коректността на BFS за разстоянията	440
77	Коректността на BFS за дървото	441
78	Теоремата за скобовата структура на интервалите на DFS	450
79	Теоремата за белите пътища (Theorem 22.9 в [34, стр. 608])	451
80	Възможностите за $d[x]$, $d[y]$, $f[x]$ и $f[y]$ на $(x, y) \in E$ след DFS(G)	453
81	Theorem 22.10 в [34, стр. 610]	455
82	Граф е цикличен тстк DFS открива ребро назад	456
83	НДУ за съществуване на топологическо сортиране	458
84	Поне един източник и поне един сифон в даг	459
85	Коректността на топологическото сортиране на Tarjan	463
86	Коректността на топологическото сортиране на Kahn	465
87	Алгоритъм 20 индуцира топо-сортиране на фактор-графа	475
88	Коректността на Алгоритъм 24.	478

89	Намиране на срязващите върхове: листата на дървото	485
90	Намиране на срязващите върхове: коренът на дървото	485
91	Намиране на срязващите върхове: не-листа, не-корен	486
92	Намиране на мостовете	490
93	Theorem 23.1 от [34, стр. 627] (МПД теоремата)	494
94	Коректността на алгоритъма на Prim	501
95	Максимална височина на дърво след Union by rank	513
96	Най-къс път се състои от най-къси подпътища	522
97	Коректността на алгоритъма на Dijkstra	533
98	Коректността на алгоритъма DAG SSSHP	537
99	Числата на Fibonacci в редица от матрични степени	550
100	Едно твърдение върху числа на Fibonacci	552
101	НАЙ-РАННО ПРИКЛЮЧВАНЕ е оптимално за INTERVAL SCHEDULING	646
102	Обосновка на бързия алг. за LONGEST INCREASING SEQUENCE	666
103	GREEDY TYPESETTING дава опт. решение, ако ценовата функция е линейна	696
104	GREEDY TYPESETTING не дава винаги опт. решение, ако ценовата ф-я е кубична	697
105	VERTEX COVER и INDEPENDENT SET са дуални	704
106	В граф без изол. в-ве, всяко върхово покриване е доминиращо множество	710
107	Височината на сортиращо дърво за вземане на решение е $\Omega(n \lg n)$	752
108	Сортиращите дървета сравняват всяка двойка съседни по големина елементи	753
109	За дърветата на EU върху ДА-екземплярите	759
110	Дърветата за вземане на решение на EU различават пермутациите	760
111	Задачата DEGENERACY TESTING е 3SUM-hard	771
112	Намирането на максималния елемент иска поне $n - 1$ сравнения	782
113	Намирането на втория по големина елемент иска $\geq n - 2 + \lceil \log_2 n \rceil$ сравнения	801
114	01 е трудноуловим по отношение на четните булеви стрингове	825
115	Коректността на противника при трудноуловимостта на свързаността	832
116	Експоненциална долна граница за $RSQ(\Sigma)$ (Theorem 2.1, [114])	841
117	Машината на Turing M_{pal} решава задачата ПАЛИНДРОМИ върху $\{0, 1\}$	851
118	Верификатор може да симулира ефикасно машина с размножаване	865
119	Машина с размножаване може да симулира ефикасно верификатор	867
120	Полиномиалните редукции са транзитивни	875
121	Класът $\mathbf{P}$ и полиномиалните редукции	877
122	$\mathbf{P}$ е минималният клас на еквивалентност на $\leq_p$	878
123	HALTING PROBLEM $\in \mathbf{NP-h}$	889
124	Всяка булева формула може да се препише в ДНФ или КНФ	898
125	BOUNDED HALTING PROBLEM $\in \mathbf{NP-c}$	910
126	Теорема на Cook	911
127	$\mathbf{P} \in \mathbf{NP} \wedge \mathbf{P}' \in \mathbf{NP-c} \wedge \mathbf{P}' \leq_p \mathbf{P} \rightarrow \mathbf{P} \in \mathbf{NP-c}$	919
128	Теорема на Ladner	920
129	Екземпляр на 2SAT е удовлетворим тстк импл. граф е удачен	936
130	Теорема на Kőnig, 1931 г.	987
131	VERTEX COVER е податлива върху двуделните графи	988
132	Теорема на Hall, 1931 г.	989
133	Коректността на VERTEX COVER $\leq_p$ HAMILTONIAN CYCLE	1013
134	Коректността на 3DM $\leq_p$ 2-PARTITION	1023
135	$\star$ -FREE REGEXP INEQ $\in \mathbf{NP}$	1035
136	TAUTOLOGY $\in \mathbf{coNP-c}$	1042
137	$\mathbf{P} = \mathbf{NP}$ влече $\mathbf{NP} = \mathbf{coNP}$	1043

138	$P \in \text{coNP-c} \wedge P \in \text{NP}$ влече $\text{NP} = \text{coNP}$	1043
139	$\text{ART GALLERY} \in \text{NP}$	1048

Списък на нотациите

1	$\forall n \nrightarrow$	109
2	“ $\{x\}$ ” наместо “ $x - [x]$ ”	131
3	“ H_n ” означава n -тата парциална сума на хармоничния ред	137
4	Фактор-релацията $\mathbf{R}/\simeq$	170
5	$N(u)$ и $N(U)$: съседите на u и U	429
6	$u \rightsquigarrow v$, $u \xrightarrow{P} v$, $\text{Paths}(u, v)$	429
7	$\mathcal{SC}(G)$ и $\mathcal{SCC}(G)$: релацията на силна свързаност и нейните класове на екв.	429
8	“ $\text{adj}[x]$ ” означава списъка на съседство на връх x	431
9	$G/\mathcal{SC}(G)$ е фактор-графът на G	468
10	“ $\{^n_k\}$ ” означава числото на Stirling от втори род n -подмножество- k	583
11	“ $[^n_k]$ ” означава числото на Stirling от първи род n -цикъл- k	584
12	“ $ ^n_k $ ” означава броя на целочислените разбивания на n на k части	587
13	Ψ	711
14	Бележим полиномиалните редукции с “ $\leq_p$ ”	874
15	$\text{Var}(\phi)$	890
16	$\text{Val}(\mathcal{Y})$	892
17	$t \models \phi$	893

Списък на изчислителните задачи

1	СВЪРЗАНОСТ НА ГРАФИ	11
2	HAMILTONIAN CYCLE	11
3	НАЙ-КЪС ПЪТ В ГРАФ, ПЕДАНТИЧНО ОПИСАНИЕ	12
4	НАЙ-КЪС ПЪТ В ГРАФ, ТРАДИЦИОННО ОПИСАНИЕ	12
5	НАЙ-КЪС ПЪТ В ГРАФ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	12
6	СОРТИРАНЕ НА ЧИСЛА	13
7	ПЪТ В ГРАФ	13
8	HALTING PROBLEM	21
9	2SUM	56
10	3SUMCOUNT, ВЕРСИЯ ЗА ТЪРСЕНЕ	57
11	MAX SUM PLUS DIST	64
12	ТРАНСПОРТИРАНЕ СЪС ЗАГУБИ	84
13	СОРТИРАНЕ	276
14	СОРТИРАНЕ, АБСТРАКТНА ДЕФИНИЦИЯ	277
15	ТЪРСЕНЕ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	278
16	ТЪРСЕНЕ, ВЕРСИЯ ЗА ТЪРСЕНЕ	278
17	НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ОЛЕКОТЕНИЯТ ВАРИАНТ	281
18	НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ТЕЖКИЯТ ВАРИАНТ	281
19	УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ (ELEMENT UNIQUENESS / EU)	282
20	МЕДИАНА	289
21	kSUM, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	289
22	kSUM, ВЕРСИЯ ЗА ТЪРСЕНЕ	290
23	НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D, ОЛЕКОТЕНИЯТ ВАРИАНТ	366
24	НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D, ТЕЖКИЯТ ВАРИАНТ	366
25	ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА	379
26	МИНИМАЛНО ПОКРИВАЩО ДЪРВО (MINIMUM SPANNING TREE)	493
27	SINGLE PAIR SHORTEST PATH	520
28	SINGLE SOURCE SHORTEST PATHS	520
29	SINGLE DESTINATION SHORTEST PATHS	520
30	ALL PAIRS SHORTEST PATHS	521
31	LONGEST PATH МЕЖДУ ДВА ВЪРХА, ОПТИМИЗАЦИОННА ВЕРСИЯ	526

32	MATRIX-CHAIN MULTIPLICATION	596
33	MINIMUM WEIGHT TRIANGULATION	606
34	NON-ASSOCIATIVE OPERATION	613
35	CONTEXT-FREE GRAMMAR DERIVATION, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	615
36	2-PARTITION, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	617
37	2-PARTITION, ОПТИМИЗАЦИОННА ВЕРСИЯ	617
38	2-ESS, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	626
39	2-ESS, ОПТИМИЗАЦИОННА ВЕРСИЯ	626
40	UNBOUNDED KNAPSACK	632
41	0-1 KNAPSACK	638
42	BOUNDED KNAPSACK	642
43	INTERVAL SCHEDULING	644
44	INTERVAL SCHEDULINGQ, WEIGHTED	651
45	LONGEST INCREASING SEQUENCE	660
46	LONGEST COMMON SUBSEQUENCE	669
47	SEQUENCE ALIGNMENT	676
48	EDIT DISTANCE	687
49	PALINDROME PARTITIONING	694
50	PRETTY TYPESETTING	696
51	VERTEX COVER	703
52	INDEPENDENT SET	704
53	EDGE COVER	705
54	INDEPENDENT SET, WEIGTHED	708
55	DOMINATING SET	709
56	DOMINATING SET, WEIGHTED	718
57	LONGEST DIRECTED PATH	732
58	МОДА	768
59	DEGENERACY TESTING	771
60	ПАЛИНДРОМИ	843
61	TSP, ОПТИМИЗАЦИОННА ВЕРСИЯ	878
62	TSP, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	878
63	HAMILTONIAN PATH	884
64	SATISFIABILITY (SAT)	899
65	СУДОКУ	900
66	RESTRICTED LABEL PLACEMENT	907
67	BOUNDED HALTING PROBLEM	910
68	GRAPH ISOMORPHISM	922
69	FACTORING, ВЕРСИЯ ЗА ТЪРСЕНЕ	922
70	FACTORING, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	922

71	3SAT	926
72	2SAT	930
73	Max2SAT, ОПТИМИЗАЦИОННА ВЕРСИЯ	941
74	Max2SAT, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	941
75	NAESAT	944
76	MaxCut, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	946
77	VERTEX COVER, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	950
78	INDEPENDENT SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	956
79	CLIQUE, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	956
80	DOMINATING SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	956
81	0-1 INTEGER PROGRAMMING	965
82	3-COLORABILITY	967
83	k-COLORABILITY	972
84	PLANAR 3-COLORABILITY	972
85	3-DIMENSIONAL MATCHING (3DM)	995
86	LONGEST CYCLE, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	1018
87	LONGEST PATH, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	1018
88	SUBSET SUM	1024
89	UNBOUNDED KNAPSACK, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	1025
90	0-1 KNAPSACK, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	1030
91	BIN PACKING	1031
92	MULTIPROCESSOR SCHEDULING, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ	1033
93	★-FREE REGEXP INEQ	1035
94	<u>СВЪРЗАНОСТ НА ГРАФИ</u>	1039
95	<u>HAMILTONIAN CYCLE</u>	1039
96	TAUTOLOGY	1042
97	ART GALLERY	1048

Списък на редукциите

1	HAMILTONIAN CYCLE $\leq_p$ TSP	879
2	ЧЕТНОСТ $\leq_p$ HALTING PROBLEM	880
3	HAMILTONIAN CYCLE $\leq_p$ HAMILTONIAN PATH	885
4	HAMILTONIAN PATH $\leq_p$ HAMILTONIAN CYCLE	887
5	СУДОКУ $\leq_p$ SAT	901
6	RESTRICTED LABEL PLACEMENT $\leq_p$ SAT	907
7	HAMILTONIAN PATH $\leq_p$ SAT	908
8	$\forall \Pi \in \mathbf{NP} : \Pi \leq_p$ BOUNDED HALTING PROBLEM	911
9	$\forall \Pi \in \mathbf{NP} : \Pi \leq_p$ SAT	912
10	SAT $\leq_p$ 3SAT	926
11	3SAT $\leq_p$ MAX2SAT	941
12	3SAT $\leq_p$ NAESAT	944
13	NAESAT $\leq_p$ MAXCUT	946
14	3SAT $\leq_p$ VERTEX COVER	950
15	VERTEX COVER $\leq_p$ INDEPENDENT SET	956
16	INDEPENDENT SET $\leq_p$ CLIQUE	956
17	VERTEX COVER $\leq_p$ DOMINATING SET	956
18	3SAT $\leq_p$ 0-1 INTEGER PROGRAMMING	965
19	3SAT $\leq_p$ 3-COLORABILITY	967
20	3-COLORABILITY $\leq_p$ k-COLORABILITY	972
21	3-COLORABILITY $\leq_p$ PLANAR 3-COLORABILITY	972
22	3SAT $\leq_p$ 3DM	995
23	VERTEX COVER $\leq_p$ HAMILTONIAN CYCLE	1006
24	HAMILTONIAN CYCLE $\leq_p$ LONGEST CYCLE	1018
25	HAMILTONIAN PATH $\leq_p$ LONGEST PATH	1018
26	3DM $\leq_p$ 2-PARTITION	1018
27	2-PARTITION $\leq_p$ SUBSET SUM	1024
28	SUBSET SUM $\leq_p$ UNBOUNDED KNAPSACK, РАЗПОЗНАВАНЕ	1025
29	2-PARTITION $\leq_p$ 0-1 KNAPSACK, РАЗПОЗНАВАНЕ	1030
30	2-PARTITION $\leq_p$ BIN PACKING	1031
31	2-PARTITION $\leq_p$ MULTIPROCESSOR SCHEDULING	1033
32	3SAT $\leq_p$ $\star$ -FREE REGEXP INEQ	1037
33	VERTEX COVER $\leq_p$ ART GALLERY	1048

На Елица Златева

Част I

Фундамент

Лекция 1

Изчислителни задачи и алгоритми.

Резюме: Въвеждаме понятията алгоритъм и изчислителна задача. Разглеждаме три вида изчислителни задачи: за разпознаване, оптимизационни и за търсене. Дискутираме инструкциите, от които са съставени алгоритмите. Даваме като пример Евклидовия алгоритъм: най-старият известен истински алгоритъм. Разглеждаме различните начини да бъдат описвани алгоритми.

1.1 Алгоритъм – що е то?

1.1.1 Опит за дефиниция

Понятието *алгоритъм* е първично. За него няма общоприета прецизна формална дефиниция, също както няма дефиниция на *множество*. Задълбочена дискусия на тема дефиниция на алгоритъм има в статията на Blass и Gurevich [21]. Steve Skiena [144, стр. 3] казва:

What is an algorithm? An algorithm is a procedure to accomplish a specific task. An algorithm is the idea behind any reasonable computer program.

To be interesting, an algorithm must solve a general, well-specified problem. An algorithmic problem is specified by describing the complete set of instances it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental.

Jack Edmonds [42] казва:

Here “algorithm” is used in the strict sense to mean the idealization of some physical machinery which gives a definite output, consisting of cost plus the desired result, for each member of a specified domain of inputs, the individual problems.

Често в литературата на тема алгоритми се казва, че алгоритъм е *механична процедура*. Добре известен е следният цитат от статия на Edward Forrest Moore [117]:

The methods given in this paper require no foresight or ingenuity, and hence deserve to be called algorithms. They would be especially suited for use in machine, either a special-purpose or a general-purpose digital computer.

“Механична процедура” е, точно както в цитата от Moore, действия, чието извършване не изисква човешка досетливост и могат да бъдат извършени от машина. Преди ерата на цифровите компютри такива машини са били механични и хората са си представяли (нещо като)

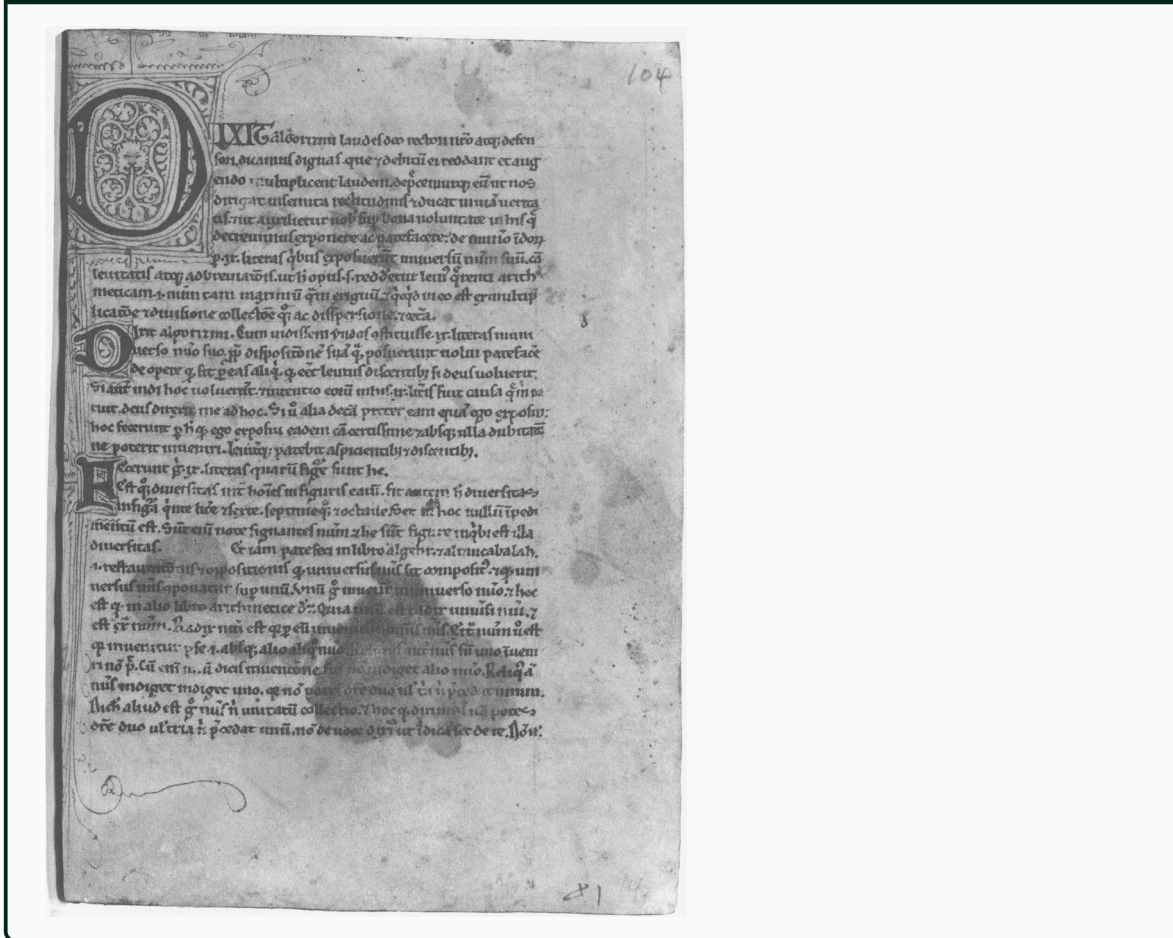
часовникови механизми, които ги извършват. Днес вместо “механична процедура” бихме казали “алгоритъм”[†], защото механичната имплементация не е съществена – процедурата може да се имплементира върху електрическа, електронна или квантово-фотонна машина [152]. Същественото е, че процедурата в никой момент не използва човешкия талант да “вижда” решение чрез прозрение, а работата ѝ се реализира от машина, базирана единствено върху законите на физиката.

Допълнение 1: За произхода на думата “алгоритъм”

Подробно разглеждане на историята ключовия термин *алгоритъм* има в първия том от култовата книга *The Art of Computer Programming* на Donald Knuth [92, стр. 1–2]. Терминът е поевропейчен вариант на името на великия персийски учен Мохамад ибн Муса ал-Хорезми (около 780 г. – около 850 г. сл. Хр.) В англезичната литература името се пише—когато целта е записът да отразява максимално точно оригиналното произнасяне—Muḥammad ibn Mūsā al-Khwārizmī. Буквално, името означава “Мухамад, син на Муса, Хорезмиецът”. Частицата “ал-” на арабски е определителен член. Хорезм е древен град в Централна Азия, който днес се намира в Узбекистан и се нарича *Хива*. “Ал-Хорезми” означава “който е от Хорезм”, накратко “хорезмиецът”. Името предполага, че Мухамад е роден в Хорезм, макар че това не е сигурно. Това, което днес се смята от историците за сигурна истина е, че ал-Хорезми живее и работи в двора на халифа в Багдад по време на златния период на арабската култура, в периода около 813–830 г. сл. Хр. Ал-Хорезми е автор на трактат, който се смята за основоположен за алгебрата, на трактати по география и астрономия, но от алгоритмична гледна точка, важният му принос е трактат върху създадената няколко века по-рано в Индия десетична позиционна бройна система, използваща нулата като пълноправна цифра наред с другите девет цифри. За съжаление, оригиналът на този текст изглежда загубен във времето. Предполага се, че се е казвал “За изчислението с индийски цифри”. До нас е достигнал превод на латински от 12 век, който по всяка вероятност е правен от друг превод на латински, който е правен от копие на арабски на оригиналния трактат на ал-Хорезми. Черно-бяло факсимиле на първата страница от въпросния превод на латински от 12 век (който се съхранява в библиотеката на Кеймбриджкия Университет) е показано на Фигура 1.1. Много подробна информация за този превод на трактата ал-Хорезми, както и негов превод на английски, има в статията на Crossley и Henry [83], свободно достъпна онлайн на [sciencedirect.com](https://www.sciencedirect.com).

[†]Естествено, няма да се опитваме да дефинираме “алгоритъм” чрез “алгоритъм”.

Фигура 1.1 : Първата страница на превода на латински на трактата на ал-Хорезми за изчислението с цифри. Първите думи са “Dixit algorismi”.



Както се вижда, трактатът започва с думите “Dixit algorismi”^a. На латински това означава, “Така каза Алгоризми”, където “Алгоризми” е неправилно написаното име на ал-Хорезми. “Dixit algorismi” е и началото на втория параграф.

Оттук в западноевропейските езици се появява думата “algorism”, която означава процедура за работа с числа, написани в индо-арабската бройна система. По онова време, а именно Късното средновековие, в Европа все още използват римската непозиционна бройна система. Предимствата на индо-арабската позиционна бройна система са очевидни за всеки непредубеден човек, но минават няколко века, преди тя да се наложи. По времето, когато двете бройни системи се ползват едновременно, смятащите хора в Европа са разделени на две: абакистите, които ползват римската бройна система и *абака* (вид устройство за смятане), и алгористите, които ползват индо-арабската бройна система. Правилата за работа с индо-арабската система са се казвали “алгоризми”. Имало е нескрит антагонизъм между тези две групировки [146], докато индо-арабската система не се е наложила напълно.

Много по-късно думата *algorism* се трансформира в *algorithm* и смисълът ѝ става това, което днес разбираме под “алгоритъм”.

^aНачалото на текста е “Dixit algorizmi: laudes deo rectori nostro atque defensori dicamus dignas, que et debitum ei reddant, et augendo multiplicent laudem.”. Снопед google translate, това означава “Algorizmi said: let us give praise to God our ruler and defender worthy, that they may repay the debt to him, and multiply the praise by increasing it”.

1.1.2 Изчислителни задачи

1.1.2.1 Дефиниция

Понятието *изчислителна задача*, на английски *computational problem*, е ключово в теорията на алгоритмите. Авторът на тези лекционни записки е на мнение, че често се бърка “алгоритъм” и “изчислителна задача”, а това са фундаментално различни неща и разликата между тях трябва да е кристално ясна. Да си припомним цитата на Skiena на стр. 3: “... an algorithm must solve a general, well-specified *problem*.”.

Всяка изчислителна задача се характеризира със своите *екземпляри*, на английски *instances*, като на всеки екземпляр съответстват нула, едно или повече *решения*, на английски *solutions*. Ето какво казват за изчислителните задачи Garey и Johnson [56, стр. 4]:

For our purposes, a problem will be a general question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified. A problem is described by giving: (1) A general description of all its parameters, and (2) a statement of what properties the answer, or solution, is required to satisfy. An instance of a problem is obtained by specifying particular values for all the problem parameters.

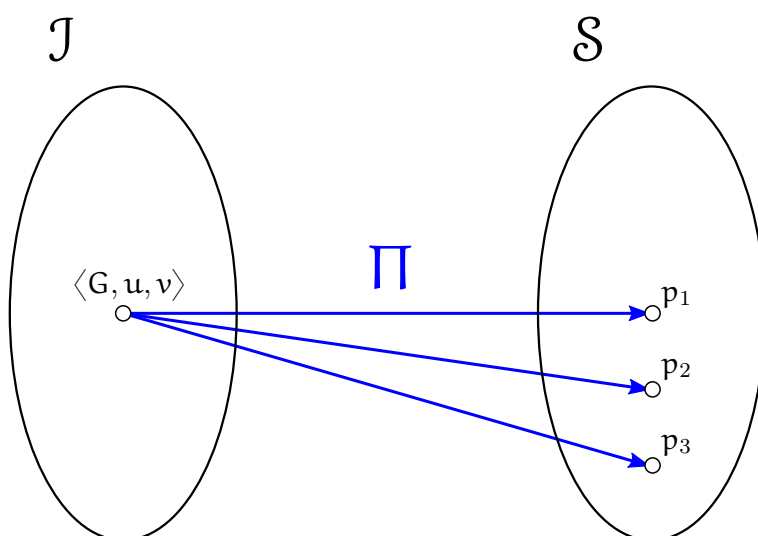
По правило, всяка интересна изчислителна задача има безброй много екземпляри. Освен това, както ще видим в Лекция 2, общоприетата теория за анализ на алгоритми не е приложима, ако задачата има краен брой екземпляри. И така, разглеждаме само задачи с безкрайно много екземпляри. Обаче **изброимо** безкрайно. Дори ако казваме, че екземплярите на задачата съдържат като компоненти реални числа, което—строго формално—влече неизброима безкрайност, ние знаем, че всъщност числата са рационални, а “реални” е просто начин на изразяване.

За разлика от “алгоритъм”, което понятие няма формална дефиниция, “изчислителна задача” има формална дефиниция – изчислителна задача е вид релация. Това е по-абстрактно от определението “general question” на Garey и Johnson, но е по-прецизно.

Определение 1: Изчислителна задача

Изчислителна задача P е двуместна релация $P \subseteq \mathcal{I} \times \mathcal{S}$, където $\mathcal{I}$ е безкрайното множество от екземплярите, а $\mathcal{S}$ е безкрайното или крайно множество от решенията. За всеки екземпляр $x \in \mathcal{I}$, за всяко решение $y \in \mathcal{S}$, такова че $(x, y) \in P$, казваме, че y е *решение за x* .

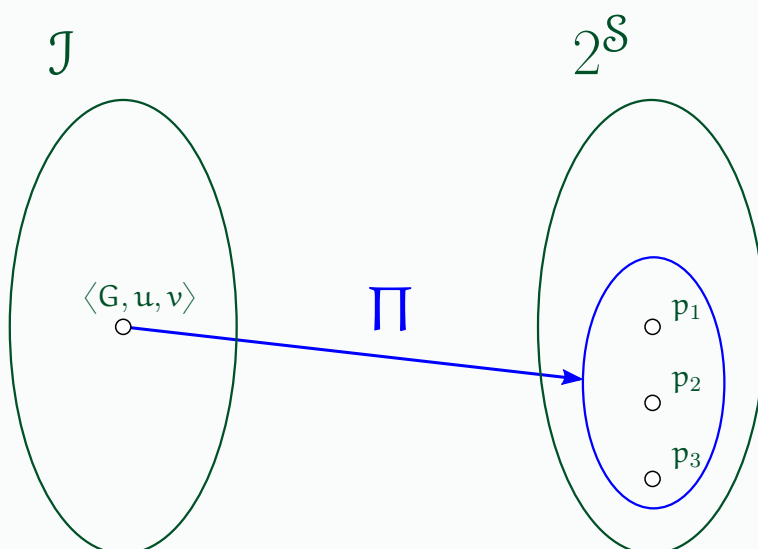
Следната фигура илюстрира нашата представа за изчислителната задача ПЪТ В ГРАФ като релация: първият домейн (екземплярите) е множеството от всички наредени тройки от неориентиран граф, един връх в него и друг връх в него, а вторият домейн (решенията) е множеството от пътищата (във всички графи). За всеки екземпляр $\langle G, u, v \rangle$, той е в релация с, и само с, всеки път p в G , чиито крайни върхове са именно u и v . Това, че в G има точно три пътя p_1 , p_2 и p_3 от u до v изразяваме, казвайки, че $(\langle G, u, v \rangle, p_1) \in P$, $(\langle G, u, v \rangle, p_2) \in P$, $(\langle G, u, v \rangle, p_3) \in P$ и в P няма други наредени двойки с първи елемент $\langle G, u, v \rangle$. Ако си представим P като изображение, по отношение на $\langle G, u, v \rangle$ нещата изглеждат така:



Очевидно е, че в общия случай изображението не е функция. Само в случай, че за всеки екземпляр има точно едно решение можем да кажем, че P е функция $P : J \rightarrow S$.

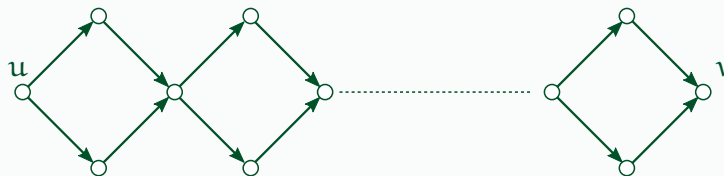
Допълнение 2: Не може ли изч. задача да е функция $P : J \rightarrow 2^S$?

Човек може да се изкуши да дефинира “изчислителна задача” не като релация с първи домейн екземплярите и втори домейн, решенията, а като функция с домейн екземплярите и кодомейн степенното множество на множеството от решенията. По този начин може успешно да изрази това, че даден екземпляр има множество от решения. Екземпляр да няма решение е същото като да го изобразим в празното множество. Продължавайки с примера горе, със същия успех може да кажем, че функцията P изобразява $\langle G, u, v \rangle$ в множеството $\{p_1, p_2, p_3\}$.



Но това е само на пръв поглед. Има две причини да не искаме да правим това.

1. Може да искаме да смятаме за решение **кой да е** от пътищата p_1 , p_2 и p_3 , а **не непременно множеството** от всички тях.
2. Решенията може да са прекалено много. В примера с пътищата в даден граф, броят на пътищата може да е (поне) експоненциален в размера на графа. Вижте този (фрагмент от) граф^a.



Броят на пътищата от u до v в него е 2^k , където k е броят на “ромбчетата”. Ако настояваме решението да е множеството от всички пътища, а индивидуалните пътища не са решения, то никой алгоритъм няма да може да генерира решението дори за екземпляри със скромни размери по простата причина, че големината на решението е фантастично голяма и е напълно извън възможностите на всеки реален компютър (вижте Допълнение 12).

Поради това е много по-разумно да дефинираме “изчислителна задача” като релация, а не като функция.

^aСъщата конструкция се ползва и Допълнение 51.

Определение 2: Функция, съответна на изчислителна задача

Нека е дадена изчислителна задача $\Pi \subseteq \mathcal{I} \times \mathcal{S}$. Нека $\star$ е елемент, не принадлежащ на $\mathcal{S}$. Нека $\mathcal{S}' = \mathcal{S} \cup \{\star\}$. *Функция, съответна на Π* , е всяка функция $f : \mathcal{I} \rightarrow \mathcal{S}'$, такава че $\forall x \in \mathcal{I}$:

- ако съществува решение за x , то $f(x)$ е **някое** решение за x ;
- ако не съществува решение за x , то $f(x)$ е $\star$.

1.1.2.2 Видове изчислителни задачи

В този курс ще разгледаме три вида изчислителни задачи: задачи за разпознаване, оптимизационни задачи и задачи за търсене.

Определение 3: Задачи за разпознаване

Задача за разпознаване е всяка изчислителна задача, за която множеството $\mathcal{S}$ от решенията е $\{\text{TRUE}, \text{FALSE}\}$. Още може да кажем, че $\mathcal{S} = \{1, 0\}$ или $\mathcal{S} = \{Y, N\}$.

Ако P е задача за разпознаване, множеството от нейните екземпляри се разбива на две подмножества P_Y и P_N . P_Y се състои точно от екземплярите, за които отговорът е TRUE; това са *ДА-екземплярите*. P_N се състои точно от екземплярите, за които отговорът е FALSE; това са *НЕ-екземплярите*.

Задачите за разпознаване са задачите с еднобитово решение. Вместо “решение” може да казваме “отговор”.

На “английски задача за разпознаване” е *decision problem*. “ДА-екземпляр” е “YES-instance”, а “НЕ-екземпляр” е “NO-instance”. Ето няколко такива задачи.

- Дали дадено число е просто. Екземплярите са числата от $\mathbb{N} \setminus \{0, 1\}$. ДА-екземплярите са простите числа. НЕ-екземплярите са съставните числа.
- Дали дадена програма, написана на даден език за програмиране, е синтактично коректна. Екземплярите са стринговете над някаква азбука, която е адекватна за целта. ДА-екземплярите са стринговете, които са синтактично коректни програми. НЕ-екземплярите са стринговете, които не са синтактично коректни програми.
- Дали даден граф е свързан. Екземплярите са неориентираните графи. ДА-екземплярите са свързаните графи. НЕ-екземплярите са несвързаните графи.

Очевидно всяка задача за разпознаване е (частична) функция, понеже няма смисъл от задача, в която отговорът за даден екземпляр е хем истина, хем лъжа.

Определение 4: Оптимизационни задачи

Оптимизационна задача е всяка задача, в която всеки екземпляр се асоциира с множество от *допустими решения* (на английски, *feasible solutions*), което може да е празно, непразно крайно или дори безкрайно. Дефинирана е *целева функция* (на английски, *objective function*), която изобразява всяко допустимо решение в число, наречено *цена* или *тегло*. В общия случай, това число е реално^a. Дефинирана е и *цел*, която е или минимизация, или максимизация. Множеството от *оптималните решения* е подмножеството на допустимите решения, върху чиито елементи целевата функция има минимална стойност, ако целта е минимизация, или максимална стойност, ако целта е максимизация.

При дадена оптимизационна задача може да се искат различни неща.

- Може да се иска цената на кое да е оптимално решение.
- Може да се иска кое да е оптимално решение.
- Може да се иска броят на оптималните решения.
- Може да се искат всички оптимални решения, или индикация, че оптимални решения няма. Иначе казано, може да се иска множеството от оптималните решения.

Забележете, че, ако множеството от допустимите решения е безкрайно, оптимално решение може да не съществува; примерно, в граф с отрицателни цикли може да няма най-къс път от даден връх до даден връх, ако разглеждаме пътища, които не са непременно прости (Конвенция 12 и Наблюдение 68 в Лекция 11).

^aКато имаме предвид, че всъщност числата са рационални, а “реални” е само начин на изразяване.

Да се иска броят на оптималните решения е рядкост, а в тези лекции никога не искаме всички оптимални решения, защото те може да са прекалено много (вижте Допълнение 51).

Пример за оптимизационна задача е добре познатата ни задача за намиране на най-къс път в граф: при даден ориентиран граф G , тегловна функция w върху ребрата и два върха s и t , да се намери път с минимална цена от s до t . Всеки екземпляр е наредена четворка $\langle G, w, s, t \rangle$, множеството от допустимите решения е множеството от всички пътища от s до t , цената на решение е сумата от теглата на пътя, а целта е минимизация[‡]. Съгласно Определение 4, може да се иска само минималната цена, може да се иска един от пътищата с минимална цена, може да се иска броя на пътищата с минимална цена, може да се иска множеството от пътищата с минимална цена (което множество обаче може да е толкова голямо, че да не може да се генерира на практика).

[‡]В учебника Complexity and Approximations [8, стр. 22] се казва, че цената не е само върху решенията u , а върху наредените двойки (x, u) , където x е екземпляр, а u е решение за него. Ако приемем това, то приемаме, че едно и също решение u може да има различна цена в контекста на различни екземпляри, чието решение се явява. В тези лекции никога няма да е така. Примерно, за нас цената на най-къс път не зависи от контекста; претеглената дължина на даден път с дадени тегла е една и съща за всеки контекст, тоест, за всеки граф, в който има такъв път.

Определение 5: Задачи за търсене

Задача за търсене е изчислителна задача, в която за даден екземпляр има нула, едно или повече решения. Иначе казано, за всеки екземпляр $x \in \mathcal{I}$ има множество решения $Y \subseteq \mathcal{S}$, такива че $\forall y \in Y : (x, y) \in \mathcal{P}$, където $\mathcal{P}$ е задачата; множеството Y може и да е празното множество. Може да се иска кой да е елемент от Y или индикация, че Y е празното множество; а може да се иска самото Y .

Терминът на английски е *search problem*.

Наблюдение 1: Задачите за търсене са най-общите задачи

Предвид това, че при задачите за търсене се говори просто за решения, това са възможно най-общият вид задачи. Определение 5 практически повтаря Определение 1. Ерго, всяка изчислителна задача е, в някакъв смисъл, задача за търсене. Оптимизационните задачи са строго подмножество на задачите за търсене, и също така задачите за разпознаване са строго подмножество на задачите за търсене.

1.1.2.3 Как описваме изчислителните задачи

Имената на изчислителните задачи се пишат с малки главни букви. Забележете огромната разлика между, да кажем, “СОРТИРАНЕ” (Задача 6) и “сортиране”:

- “СОРТИРАНЕ” е множество от наредени двойки от наредени n -орки.
- “сортиране” е отглаголно съществително, означаващо вид действие.

В никакъв случай не може да кажем, че “СОРТИРАНЕ” е отглаголно съществително или “сортиране” е множество от наредени двойки.

Как описваме задачите за разпознаване. Въвеждаме понятието *общ екземпляр* на изчислителна задача, на английски *generic instance*, като общият екземпляр е описание, от което получаваме представа за всеки конкретен екземпляр. Ще казваме накратко “екземпляр” вместо “общ екземпляр”. При задачите за разпознаване говорим не за решение, а за *въпрос* с отговор ДА или НЕ. Ето как описваме формално задачата СВЪРЗАНОСТ НА ГРАФИ.

Изн. Задача 1: СВЪРЗАНОСТ НА ГРАФИ

екземпляр: Неориентиран граф G .

въпрос: Дали G е свързан?

Задача за разпознаване, която ще ползваме често за илюстрация, е HAMILTONIAN CYCLE.

Изн. Задача 2: HAMILTONIAN CYCLE

екземпляр: Неориентиран граф G .

въпрос: Дали има Хамилтонов цикъл в G ?

Как описваме оптимизационните задачи. Ако следваме Определение 4, трябва да описваме оптимизационните задачи така. Да кажем, че задачата е **НАЙ-КЪС ПЪТ В ГРАФ**.

Изч. Задача 3: НАЙ-КЪС ПЪТ В ГРАФ, ПЕДАНТИЧНО ОПИСАНИЕ

екземпляр: Наредена четворка $\langle G, w, s, t \rangle$, където G е ориентиран граф, w е тегловна функция върху ребрата му, а s и t са върхове в него.

множество от допустимите решения: Множеството P от пътищата от s до t в G .

целева функция: За всеки път $p \in P$, теглото $w(p)$ е $\sum_{e \in E(p)} w(e)$.

цел: Минимизация.

От това описание не става ясно дали искаме само теглото на оптимален път или самия път. По-съществено е, че рядко ще описваме задачи толкова педантично. По-скоро бихме се изразили така.

Изч. Задача 4: НАЙ-КЪС ПЪТ В ГРАФ, ТРАДИЦИОННО ОПИСАНИЕ

екземпляр: Наредена четворка $\langle G, w, s, t \rangle$, където G е ориентиран граф, w е тегловна функция върху ребрата му, а s и t са върхове в него.

решение: Път p от s до t в G , такъв че $w(p)$ е минимално, или индикация, че път от s до t няма.

Наблюдение 2: Задача за разпознаване, съответна на оптимизационна задача

Всяка оптимизационна задача P може да бъде превърната изкуствено в задача за разпознаване P' , като

1. към екземпляра добавим число k ,
2. заменим решението с въпроса “Дали за обекта от екземпляра има решение с цена, не по-малка от k ?”, ако задачата е максимизационна, или “Дали за обекта от екземпляра има решение с цена, не по-голяма от k ?”, ако задачата е минимизационна.

Казваме, че P' е *задачата за разпознаване, съответна на P* . Или че P' е *задачата P във версия за разпознаване*.

Ето как изглежда задачата за най-къс път във версия за разпознаване.

Изч. Задача 5: НАЙ-КЪС ПЪТ В ГРАФ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Наредена петорка $\langle G, w, s, t, k \rangle$ от ориентиран граф G , тегловна функция w върху ребрата му, върхове s и t от G , число k .

въпрос: Има ли в G път p от s до t , такъв че $w(p) \leq k$?

На пръв поглед, този вариант на задачата за най-къс път е безполезен на практика. Но, както ще видим в Лекция 14, задачите за разпознаване имат голямо значение в теорията на изчислителната сложност и има смисъл да разглеждаме оптимизационни задачи, изкуствено модифицирани като задачи за разпознаване.

Как описваме задачите за търсене. По определение, те имат екземпляри и на всеки екземпляр съответства множество, може и празно, от решения. Трябва да кажем какъв е общият екземпляр и какво смятаме за решение на екземпляра. Ето как описваме СОРТИРАНЕ НА ЧИСЛА[†].

Изч. Задача 6: СОРТИРАНЕ НА ЧИСЛА

екземпляр: Редица $A = (a_1, a_2, \dots, a_n)$ от числа.

решение: Пермутация $(a'_1, a'_2, \dots, a'_n)$ на числата от A , такава че $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

СОРТИРАНЕ НА ЧИСЛА винаги има решение, така че не се налага да индикираме липса на решение.

Според Наблюдение 1, задачите за търсене са най-общият вид задачи, ако говорим формално. На практика обаче ще разглеждаме задачите за търсене като опростени версии на оптимизационни задачи: без целеви функции и цели. Като пример за това, ПЪТ В ГРАФ, която е задача за търсене, изглежда като “орязан” вариант на НАЙ-КЪС ПЪТ В ГРАФ.

Изч. Задача 7: ПЪТ В ГРАФ

екземпляр: Наредена тройка $\langle G, s, t \rangle$, където G е неориентиран граф, а s и t са върхове в него.

решение: или път между s и t в G , или индикация, че такъв път няма.

Дали става дума за ориентиран или неориентиран граф почти не променя дефиницията на задачата – разликата е само в предлозите (“между” за неориентирани графи, “от ... до” за ориентирани). Същественото е, че път може да няма, така че трябва да предвидим възможност да се индикира това.

Олекотеният и тежкият вариант на оптимизационна задача. В Определение 4 се казва, че за дадена оптимизационна задача може да искаме или само цената на някое оптимално решение, или едно оптимално решение, или броят на оптималните решения, или всички оптимални решения, като на практика никога не се интересуваме от броя, а всички оптимални решения може да са фантастично много.

На практика искаме (едно/някое/кое да е) оптимално решение, понеже само оптималната цена е прекалено малко информативна. Примерно, ако искате да пътувате най-евтино от град X до град Y , Вие решавате задачата намиране на най-къс път, където дължините са цени. Това, което Ви интересува на практика е самото (най-евтино) пътуване, а не просто цената му. Защо тогава изобщо разглеждаме възможността да решаваме оптимизационни задачи, при които изчисляваме само цената на оптимално решение? Отговорът е, че го правим с учебна цел. По правило алгоритъм, който намира само оптималната цена, е по-прост и лесен за възприемане от алгоритъм, който намира някое оптимално решение[‡].

И така, често алгоритмите, решаващи оптимизационни задачи, се конструират на два етапа. В първия етап конструираме алгоритъм, който намира само оптималната цена. След като се убедим в коректността му и намерим сложността му, от него правим с неголеми модификации алгоритъм, който намира някое оптимално решение. В Лекция 11 и Лекция 12 подходът почти винаги е такъв.

[†]В Лекция 4 ще видим по-обща дефиниция на задачата.

[‡]Ако разполагаме с решение, да намерим цената на това решение е тривиално.

Има смисъл да кажем, че оптимизационната задача съществува в два варианта: *олекотеният вариант*, в който ни интересува само оптималната цена, и *тежкият вариант*, в който искаме някое оптимално решение. Като пример, олекотеният вариант на НАЙ-КЪС ПЪТ В ГРАФ е, при даден екземпляр $\langle G, w, s, t \rangle$, да се намери минималната цена на път от s до t , а тежкият вариант е да се намери един път с минимална цена от s до t (или индикация, че такъв път няма).

1.1.3 Изчислителни задачи и алгоритми

Както вече казахме, изчислителна задача е нещо съвсем различно от алгоритъм. Изчислителната задача е релацията $P \subseteq \mathcal{I} \times \mathcal{S}$ от Определение 1. Съществуват безброй много алгоритми, които решават задачата P . Всеки от тези алгоритми е реализация на някоя функция, съответна (припомнете си Определение 2) на P . На прост български, алгоритъм за P е механизъм, който при даден екземпляр $x \in \mathcal{I}$ или конструира решение $y \in \mathcal{S}$, такова че y е решение за x , или индикира, че такова решение не съществува. x наричаме *вход на алгоритъма*, а y , или индикацията, че такова y не съществува, наричаме *изход на алгоритъма*.

Използвайки термините “вход” вместо “екземпляр” и “изход” вместо “решение”, подчертаваме, че гледаме на алгоритъма като на **процес**, а не като на статично съответствие между екземпляри и решения. Фигура 1.2 илюстрира най-общото ни разбиране за алгоритъм.



И така, алгоритъм е **реализацията** на някаква функция. Това е от ключово значение: за нас алгоритъмът не е черна кутия! Той е **механизмът в кутията**, който реализира функцията. Повече за въпросната реализация ще кажем в Подсекция 1.1.4. Следното не може да се нарече “определение”, а е просто обобщение на казаното досега.

Разяснение 3: Задачи и алгоритми

При дадена изчислителна задача, която е релация, изобразяваща екземпляри в решения, *алгоритъм за тази задача* е всяка реализация на функция, съответна на тази задача-релация, на някакво приемливо ниво на детайлност.

Забележете, че всеки детерминиран алгоритъм реализира именно функция, а не релация, защото при даден вход, изходът е строго определен. Затова има смисъл да се говори за “реализация на функция, съответна на P ”, а не за “реализация на P ”.

Както ще видим нататък, съществува възможност даден алгоритъм никога да не завършва работата си върху някой вход. Поради това е по-удачно да кажем, че даден алгоритъм реализира частична функция с домейн J и кодомейн S , като елементите от J , върху които алгоритъмът не завършва, са точно елементите, които нямат образ. Ако алгоритъмът със сигурност завършва работата си върху всеки вход, то въпросната функция е тотална.

Допълнение 3: Функциите са повече от алгоритмите!

Заслужава да се отбележи, че въпросните функции са повече от алгоритмите в теоретико-множествения смисъл на по-голяма безкрайна кардиналност. Алгоритмите, както ще видим, са безкрайно много, но *изброимо* безкрайно. Функциите, от друга страна, са *неизброимо* безкрайно много.

Сега ще покажем с диагоналния метод, че дори само множеството от функции от вида $f: \mathbb{N} \rightarrow \{0, 1\}$ е неизброимо безкрайно. Да допуснем, че функциите от този вид може да бъдат изброени като $f_0, f_1, f_2, \dots$.

Да разгледаме редицата $f_0(0), f_1(1), f_2(2), \dots$. Това е някаква функция с домейн $\mathbb{N}$ и кодомейн $\{0, 1\}$. Да дефинираме нотацията $\overline{f_k(k)}$ по следния начин: за всяко $k \in \mathbb{N}$:

$$\overline{f_k(k)} = \begin{cases} 1, & \text{ако } f_k(k) = 0 \\ 0, & \text{ако } f_k(k) = 1 \end{cases}$$

Сега да разгледаме редицата $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \dots$. Това е редица от нули и единици, което означава, че е функция $\phi: \mathbb{N} \rightarrow \{0, 1\}$. Но **за всяко** $j \in \mathbb{N}$, $\phi(j) \neq f_j(j)$. Следователно, ϕ не е нито една от функциите в наредбата $f_0, f_1, f_2, \dots$.

Щом функциите от вида $f: \mathbb{N} \rightarrow \{0, 1\}$ са неизброимо безкрайно много, то и функциите от вида $f: \mathbb{N} \rightarrow \mathbb{N}$ също са неизброимо безкрайно много. Следва, че възможните функции са повече от алгоритмите в същия смисъл, в който реалните числа са повече от естествените.

Заклучаваме, че само безкрайно малко подмножество на функциите от $\mathbb{N}$ в $\mathbb{N}$ биват реализирани от алгоритми! Останалите “си нямат” алгоритми.

Наблюдение 4

Алгоритмите имат физически аспект. Авторът не твърди, че теорията на алгоритмите е дял на физиката, но твърди, че осмислянето на алгоритмите не е чисто абстрактно. Алгоритъм, както казахме, е реализация на дадена частична функция. Тази реализация трябва да е **физически постижима**. Тя трябва да може да се изпълни от устройство, работещо по **законите на физиката**. Причината някои алгоритми да се смятат за практически безполезни е, че **физическото време**, което би отнело изпълнението им, е фантастично голямо. Ерго, за да твърдим, че даден алгоритъм е практически безполезен в този смисъл, трябва да знаем неща за реалния свят. Практическата безполезност в този смисъл не може да бъде разбрана от чисто математически съображение. Вижте Допълнение 12 за повече разяснения на физическите аспекти на въпросната практическа безполезност.

В контраст с това, изчислителните задачи са чисто математически обекти-релации. Можем да разглеждаме изчислителни задачи, без да се интересуваме от реалния свят.

Допълнение 4: Изчислителни модели

Подробностите по реализацията, за която стана дума в Разяснение 3, са предмет на *изчислителния модел* (на английски, *computational model* или *model of computation*), който сме приели. В тези лекционни записки няма да разглеждаме изчислителни модели експлицитно. Изчислителният модел, който ние възприемаме, е близък до машина с произволен достъп (на английски е RAM, идва от *Random Access Machine*, да не се бърка с *Random Access Memory*!). За повече информация за изчислителните модели вижте [136]. RAM моделът там се въвежда на страница 19.

1.1.4 Инструкции

Всеки алгоритъм е реализация на някаква функция. Тази реализация е **крайна**[†] редица от *инструкции* (още се наричат *команди*). Това е съдържанието на кутията на Фигура 1.2 – крайна редица от инструкции. За удобство ще записваме алгоритмите като номерирани редици от инструкции, като всяка следваща инструкция се разполага на нов ред.

Всеки алгоритъм съществува в *дискретно време*, което се състои от *моменти* (още казваме *стъпки*, или *тактове*). Във всеки момент от дискретното време се изпълнява точно една инструкция. От това следва, че не допускаме паралелизъм в изчислението.

Какви инструкции са допустими? Няма универсално общоприето правило допустимите инструкции, но е очевидно, че ако допуснем произволни инструкции, ще обезсмислим конструирането на алгоритми. Например, ако допуснем инструкция, която сортира n числа за произволно n , въпросът за конструиране на сортиращ алгоритъм е “решен” – алгоритъмът се състои от тази единствена инструкция. Такъв алгоритъм би бил напълно безполезен на практика. Полезните, смислени инструкции са тези, които могат да бъдат реализирани **физически** така, че да се изпълняват за (време, близко до) един машинен такт върху реален компютърен процесор. Ето примери за смислени инструкции:

събери числата, намиращи се в променливите a и b , и запиши резултата в a

и

сравни числените променливи a и b и ако a е по-малко или равно на b , премини към изпълнение на инструкция номер 45, в противен случай премини към изпълнение на инструкция номер 99

С още много примери нататък ще илюстрираме ясно какви инструкции и какво ниво на детайлизация имаме предвид. Засега казваме, че инструкциите се делят на *императивни*, *условни*, *инструкции за извиквания на други алгоритми* и *входно-изходни*.

Променливи. Смятаме, че информацията на нашия алгоритъм—а именно информацията на входа, информацията на изхода и междинната информация по време на работа на алгоритъма—е структурирана в променливи. Всяко късче информация е част от някаква променлива. Има атомарни (елементарни) променливи, например целите числа, и съставни променливи, например масивите.

[†]На Stephen Kleene се приписва следното твърдение “An algorithm is a finite answer to an infinite number of questions”.

Наблюдение 5: Кодирания

Всеки истински компютър работи не директно с числа или букви от естествената азбука, а с *представяния*, или *кодирания*, на числата или буквите от естествената азбука. Кодирането е биекция от множеството на числата или буквите от естествената азбука в множеството от стринговете над някаква машинна азбука, която най-често е $\{0, 1\}$.

За целите на този курс е много удобно да избегнем разглеждането на кодиранията и да смятаме, че на най-ниското ниво имаме обекти като числа, с които се работи директно. Атомарните променливи са от тези типове: `boolean`, `integer`, `real`, `char`, `pointer` и `vertex` (на граф). В това разделение има голяма доза условност, например върховете на графа може да се идентифицират с естествени числа, булевите стойности да са нула и не-нула и т. н.

Конвенция 1: Атомарните променливи: в единица памет и в единица време

Приемаме, че всяка атомарна променлива се побира в единица памет, независимо от големината на своето съдържание, и че всяка инструкция се изпълнява в една стъпка от дискретното време, независимо от големината на операндите.

Примерно, приемаме, че всяко число се побира в единица памет, колкото и голямо да е, и че две числа се събират или умножават за единица време, колкото и големи да са.

От практическа гледна точка, разбира се, Конвенция 1 е нереалистична предвид факта, че множеството от числата е безкрайно. За всеки реален компютър, множеството от числата, чието представяне може да се побере в една машинна дума, е крайно, независимо от избраното кодиране. А числата, цели, дробни или реални, са безброй много. Ерго, всеки реален компютър може да работи само с безкрайно малка част от числата от даден вид. Независимо от избраното кодиране, само безкрайно малка част от всички тях са достъпни за него.

Следствие от това е, че всеки реален компютър може да изпълнява действия върху числа—събиране, умножение, записване в паметта и така нататък—за (близо до) един такт само при положение, че тези числа от ограничено (крайно) множество. За произволни числа това няма как да стане.

Но, за целите на курса, възприемаме Конвенция 1. Това ни позволява да се съсредоточим върху основните аспекти на алгоритмите, а и днешните компютри са толкова мощни, че допускането не е нереалистично за **типично възникващите на практика** данни.

В какъв ред се изпълняват инструкциите? Както казахме, инструкциите на алгоритъма са краен брой, подредени и номерирани. Започвайки от първата инструкция на алгоритъма, във всеки следващ момент се изпълнява точно една инструкция. Коя инструкция следва инструкция i се определя така.

- Ако инструкция i е императивна или входно-изходна, но не е инструкция за край и не е последната инструкция, то след нея се изпълнява инструкция $i + 1$.
- Ако инструкция i е условна, например

`if <някакъв израз с булева интерпретация> then`

или е начало на цикъл, например

`while <някакъв израз с булева интерпретация> do`

или

for **〈някакъв израз с булева интерпретация〉** do

то следващата инструкция, която ще се изпълни, зависи от конкретния алгоритъм и стойностите на променливите в момента.

- Ако инструкция i е безусловна инструкция за преход

goto k

то след нея се изпълнява инструкция k (допускаме, че k е валиден номер на инструкция).

- Ако инструкция i е инструкция за викане на друг алгоритъм, да го наречем $ALGX$, има две възможности. Ще ги наречем условно “прагматичната” и “теоретичната”.
 - ♦ Прагматичната възможност е да мислим, че $ALGX$ присъства като редица от номерирани инструкции и следващата инструкция, която ще се изпълни, е първата инструкция на $ALGX$. Това е програмистката гледна точка: ако от някаква функция викаме друга функция, кодът на другата функция трябва да присъства и да е достъпен, и след извикването започваме да изпълняваме нея. В такъв случай, последната изпълнена инструкция на $ALGX$ няма да е последната изпълнена инструкция на алгоритъма, с който започнахме – след последната инструкция на $ALGX$ ще се изпълни инструкция $i + 1$ от викация алгоритъм (а ако няма такава, то това ще е краят на изпълнението).
 - ♦ Теоретичната възможност е точно обратното нещо: мислим, че $ALGX$ не присъства като конкретни инструкции. Извиквайки $ALGX$, **някак си, няма значение как**, получаваме ефекта, който очакваме от него. Например, ако $ALGX$ е сортиращ алгоритъм, то ефектът от викането на $ALGX(a_1, \dots, a_n)$ е, че числата се оказват сортирани и толкова. С други думи, гледаме на $ALGX$ като на черна кутия. Това е абстракция и като всяка абстракция позволява да игнорираме детайли, които смятаме за маловажни (конкретиката на $ALGX$), и да се фокусираме върху това, което смятаме за важно (конкретиката на викация алгоритъм).

При това можем да отчитаме времето за изпълнение на $ALGX^\dagger$, но може да отидем още по-далече в абстракцията и да мислим, че $ALGX$ се изпълнява **мигновено**, тоест за една единица от дискретното време. Ако дори не отчитаме времето за изпълнение на $ALGX$, казваме, че $ALGX$ е **оракул**. В реалния свят оракули няма, но оракулите са полезна абстракция.

След всички тези обяснения: ако гледаме на $ALGX$ като на черна кутия, то следващата инструкция след инструкция i от викация алгоритъм ще бъде, разбира се, инструкция $i + 1$ от викация алгоритъм, ако има такава; а ако няма такава, целият алгоритъм ще приключи.

- Ако инструкция i е за край на алгоритъма, след нея не се изпълнява нищо. Има две възможности за това: тази инструкция да е последната в алгоритъма, без да е част от цикъл, или да бъде специална инструкция за прекратяване на работата, наречена например **exit** или **halt**.

[†]Въпреки че в тази възможност не гледаме на $ALGX$ като на редица от инструкции, ние може да знаем някакви долни граници за задачата, която $ALGX$ решава, и да искаме да отчитаме това време като част от общото време за изпълнение на целия алгоритъм.

Наблюдение 6: Алгоритмите са изолирани от “външния свят”

От всичко казано дотук за изпълнението на инструкциите следва, че всеки алгоритъм, поне според разбирането в този курс, е “капсулован” по време на работата си. Алгоритъмът получава своя вход от външния свят и връща своя изход на външния свят, но по време на работата му, за него външен свят не съществува.

Допълнение 5: On-line и off-line алгоритми

Някои автори, например Karp [90], категоризират алгоритмите на

- on-line алгоритми (още се пише “online”), при които целият вход може да не е наличен при стартиране на алгоритъма и съответно има възможност алгоритъмът да получава още и още входни данни, докато работи, и
- off-line алгоритми (още се пише “offline”), при които целият вход е наличен в момента на стартиране на алгоритъма, като по време на работата входът не се мени.

Очевидно off-line алгоритмите са частен случай на on-line алгоритмите. Както видяхме, разбиранията на Knuth позволяват on-line алгоритми.

Off-line алгоритмите са точно тези, които са описани в Наблюдение 6, понеже за тях, по време на работата им, не съществува външен свят. В контраст на това, on-line алгоритмите (може да) получават информация от външния свят, докато работят, така че те не са капсуловани в този смисъл.

Алгоритмите в този курс са off-line без изключение. В учебника [34] има задачи за конструиране на алгоритми, които получават *заявка* (*query* на английски), връщат резултат, пак получават заявка, пак връщат резултат, и така нататък – това са примери за on-line алгоритми.

1.1.5 Knuth за алгоритмите

Knuth [92, стр. 6] изброява пет свойства, които трябва да има всеки алгоритъм.

Финитност. Всеки алгоритъм, за всеки свой вход, трябва да приключи работата си за краен брой стъпки. Процедура, която има всички свойства на алгоритъм, може би с изключение на свойството финитност, се нарича *изчислителен метод*. Има два начина алгоритъм да не завърши работата си.

- Може да “зацikli”, което означава да влезе повторно в конфигурация, в която вече е бил. *Конфигурация* е наредената двойка от състоянието на паметта (стойностите на променливите) и номера на текущата инструкция. Прост пример за фрагмент от алгоритъм, написан на синтаксиса на C, който гарантира зацикляне, е

```
for (;;) ;
```

- Може, без да влиза никога повторно в конфигурация, в която вече е бил, да мени някои променливи неограничено, например

```
for (i = 0, j = 1; i < 2; j ++);
```

Теоретично говорейки, върху истински компютър вторият сценарий не може да се реализира, тъй като всеки истински компютър има само крайна памет и краен, макар и невъобразимо голям, брой възможни състояния[†], така че ако бъде оставен да работи достатъчно дълго, той неизбежно ще влезе в конфигурация, в която вече е бил. В посочения пример, паметта, която реализира променливата *j*, рано или късно ще се препълни. Ако *j* е тип `unsigned int` и архитектурата е 32-битова, в някакъв момент *j* ще стане 4 294 967 295[‡], после 0, после пак 1, и така нататък.

От практическа гледна точка вторият сценарий лесно може да се реализира върху истински компютър, защото, ако времето за повторно влизане в дадена конфигурация е по-голямо от човешки живот или дори живота на звездите във Вселената, то спокойно можем да смятаме, че до повторение на конфигурация няма да се стигне.

Дефинитност. Всяка стъпка на алгоритъма трябва да бъде дефинирана прецизно и недвусмислено. Такава недвусмисленост имат алгоритмите, описани на някакъв конкретен език за програмиране. Ако описваме алгоритъм на естествен език, има риск от двусмислие, понеже различни читатели може да тълкуват по различен начин това, което е описано на естествен език.

Вход. Всеки алгоритъм има нула или повече *входни данни*, които са зададени преди началото на изпълнението или се задават динамично по време на изпълнението. Тези данни се вземат от някакво определено множество[§].

Изход. Всеки алгоритъм има една или повече *изходни данни*, които са в определена релация[¶] с входните данни.

Ефективност. Всеки алгоритъм трябва да е ефективен в смисъл, че всяка от неговите операции^{||} трябва да може принципно да бъде извършена в *крайно време* от човек с лист и молив. Моделът с човека с лист и молив е именно модел. Това е идеализация, която не отчита ресурсите, необходими за целта: време, количество хартия (памет), количество моливи и така нататък. По отношение на ефективността, игнорираме ресурсите и допускаме, че човекът може да се занимава произволно дълго (но крайно!) време с тази операция без умора, без грешки, без да остарява и умира, без да му свършва хартията или молива, и се интересуваме дали при тези допускания операцията може да бъде завършена за крайно време.

Не всяка операция е ефективна. Добре известно е, че има алгоритмично нерешими задачи. Всяка от тези задачи може да се използва за конструирането на операция, която не е ефективна. Пример за не-ефективна операция е:

Ако $P(x)$ терминира, то ...

[†]Тук се твърди имплицитно, че всеки реален компютър е детерминиран краен автомат – броят на състоянията, макар и фантастично голям, е ограничен от две-на-степен-обема-на-цялата-памет, и преходът от едно състояние към друго е напълно детерминиран. Това е така, само ако компютърът е изолиран от околния свят. Ако обаче компютърът ползва `input` от околния свят, който `input` е случаен, то не може да твърдим, че е *детерминиран* краен автомат. `Input` от околния свят може да е нещо, което идва по мрежата, или нещо, въведено от потребителя.

[‡] $2^{32} = 4\,294\,967\,296$.

[§]Това множество е множеството от екземплярите на съответната изчислителна задача.

[¶]Става дума за релацията, която е изчислителната задача.

^{||}Това, което Knuth нарича “операции”, е нещо различно това, което ние нарекохме “инструкции”. Нашите инструкции са елементарни в смисъл, че трябва да се изпълняват за единица време. При операциите, за които говори Knuth, такова изискване няма. Те трябва да се изпълняват за крайно време и толкова.

където P е някаква програма, а x е неин вход. Тази операция не е ефективна, защото задачата дали дадена програма терминира върху даден вход е алгоритмично нерешима. Това е известната СТОП ЗАДАЧА, на английски HALTING PROBLEM (вж. Допълнение 6).

Изч. Задача 8: HALTING PROBLEM

екземпляр: Компютърна програма $\mathcal{P}$ и неин вход $\mathcal{I}$.

въпрос: Дали $\mathcal{P}$ с вход $\mathcal{I}$ терминира?

Knuth дава друг пример за не-ефективна инструкция, слагайки в условна инструкция булево условие, чийто отговор е същият като на някаква нерешена от никого до момента математическа хипотеза. Ето пример за такава не-ефективна инструкция (примерът на Knuth е друг, но идеята е същата):

Ако съществува четно число, по-голямо от 2, което не е сума на две прости числа, то ...

Не-ефективността тук идва от това, че досега никой не е доказал или опровергал хипотезата на Goldbach[†]. В съвременната формулировка, тя звучи по следния начин.

Хипотеза 1: Goldbach

Всяко четно число, по-голямо от 2, е сума на две прости числа.

Докато хипотезата на Goldbach остава недоказана и неопровергана, никоя условна инструкция, чието булево условие зависи от истинността на тази хипотеза, не може да бъде смятана за ефективна.

Допълнение 6: Алгоритмичната нерешимост на HALTING PROBLEM

Да си припомним задачата.

Изч. Задача: HALTING PROBLEM

екземпляр: Компютърна програма $\mathcal{P}$ и неин вход $\mathcal{I}$.

въпрос: Дали $\mathcal{P}$ с вход $\mathcal{I}$ терминира?

Alan Turing е доказал [151], че няма алгоритъм, който решава HALTING PROBLEM. Следва опростена версия на доказателството. За по-задълбочено разглеждане, добър източник е, например, книгата на Sipser [143].

Теорема 1: HALTING PROBLEM е алгоритмично нерешима

HALTING PROBLEM е алгоритмично нерешима.

Доказателство: Да допуснем противното. Тогава съществува програма $\mathcal{Q}$, чийто вход е наредена двойка от компютърна програма $\mathcal{P}$ и неин вход $\mathcal{I}^a$, като $\mathcal{Q}(\mathcal{P}, \mathcal{I})$ връща TRUE, ако $\mathcal{P}$ с вход $\mathcal{I}$ терминира, и FALSE в противен случай. Да дефинираме програма

[†]Повече информация за хипотезата на Goldbach има в [153]. Оригинално писмо на Goldbach до Euler, изказващо хипотезата, може да бъде видяно на [този сайт](#).

ма $\mathcal{S}(\mathcal{P})$ по следния начин: $\mathcal{S}(\mathcal{P}) = \mathcal{Q}(\mathcal{P}, \mathcal{P})$. Забележете, че $\mathcal{S}(\mathcal{P})$ се състои от един единствен ред:

```
return  $\mathcal{Q}(\mathcal{P}, \mathcal{P})$ 
```

Дефинираме още една програма $\mathcal{I}(\mathcal{P})$ по следния начин:

```
if  $\mathcal{S}(\mathcal{P})$  then loop forever  
else return TRUE
```

Да анализираме $\mathcal{I}(\mathcal{I})$. Ако $\mathcal{I}(\mathcal{I})$ не терминира, трябва да е вярно, че $\mathcal{S}(\mathcal{I})$ връща TRUE. Но тогава трябва да е вярно, че $\mathcal{Q}(\mathcal{I}, \mathcal{I})$ връща TRUE. Тогава трябва да е вярно, че $\mathcal{I}$ терминира с вход $\mathcal{I}$. Това е противоречие.

Тогава да допуснем, че $\mathcal{I}(\mathcal{I})$ терминира. Тогава тя връща TRUE. Тогава трябва да е вярно, че $\mathcal{S}(\mathcal{I})$ връща FALSE. Тогава $\mathcal{Q}(\mathcal{I}, \mathcal{I})$ връща FALSE. Тогава $\mathcal{I}$ не терминира с вход $\mathcal{I}$. Но това също е противоречие. $\square$

^aВсъщност, входът на $\mathcal{Q}$ е **кодирането** на $\mathcal{P}$ и **кодирането** на нейния вход $\mathcal{I}$.

Базирайки се на казаното от Knuth, има два принципно различни начини дадена инструкция да е неподходяща: тя може да е лошо дефинирана или неефективна. Примери за лошо дефинирани инструкции са

Ако красотата ще спаси света, то ...

и

Ако Моцарт е по-добър композитор от Бах, то ...

Това са субективни твърдения и мнения, които нямат нищо общо с формалната математическа прецизност на алгоритмичните инструкции. Примери за не-ефективни инструкции вече видяхме: условни инструкции, базирани на булево условие, съдържащо алгоритмично нерешима задача, или недоказана и неопровергана хипотеза.

Към петте свойства на алгоритмите, посочени от Knuth, ще добавяме следното.

Детерминираност. За всеки алгоритъм A и за всеки негов вход x , работата на A с вход x е напълно детерминирана, тоест, определена, от x . Това означава, че изходът зависи единствено от x и от нищо друго. Нещо повече: ако пуснем A върху x няколко пъти, всяко изпълнение ще се случи в един и същи брой стъпки, да кажем k стъпки, като за всяко $i \in \{1, \dots, k\}$, текущата инструкция ще е една и съща във всяко изпълнение. В този смисъл, различните изпълнения на един и същи алгоритъм върху един и същи вход са напълно идентични.

В алгоритмите в тези лекции понякога използваме термина “произволен”. Примерно, има инструкция от вида “нека u е произволен връх на графа G ”. Това привидно внася недетерминизъм, защото коренът на “произволен” е “воля”. Истински произволен връх може да бъде избран само от агент със свободна воля, тоест човек. Компютърът, естествено, няма свободна воля, бидейки напълно детерминирана машина. Работата е там, че такова използване на “произволен” е формално непрецизно. Този термин може да се съдържа само в описание на алгоритъма на високо ниво, като е предназначен за човек. Ако мислим за описание на ниско ниво или директно за софтуер, произволност няма – когато нареждаме на компютъра какво да прави, не можем да оставим избора на волята на компютъра, понеже той няма такава.

Допълнение 7: Рандомизирани алгоритми

Съществуват и *рандомизирани алгоритми*, които имат достъп до източник на случайни числа и ползват случайни числа по време на работата си. При тях както изходът, така и времето за работа върху зависят не само от входа, а и от използваните случайни числа. Рандомизираните алгоритми очевидно не притежават свойството детерминираност. В този курс няма да разглеждаме рандомизирани алгоритми, така че всички алгоритми ще са детерминирани.

1.1.6 Изчислителни методи. Итератори.

Както видяхме в Подсекция 1.1.5, процедура, която може да не терминира, е изчислителен метод. Въпреки че такава процедура не може да е алгоритъм и оттам не може да е обект на изучаване в този курс, понякога се налага да разглеждаме изчислителни методи. Самият Knuth [92, стр. 6] посочва като важен пример на нетерминиращ алгоритъм *реактивен процес*, който (изчислителен процес) непрекъснато реагира на външната среда.

Важен пример за изчислителен метод е *итератор*. Авторът на лекционните записки заема термина “итератор” от книгата “Chaos and Fractals: New Frontiers of Science” [128]. Обикновено в компютърните науки под “итератор” се разбира *средство за обхождане на някакви агрегирани данни*, примерно списъци. В [128, стр. 17] под “итератор” разбират нещо съвсем различно:

We will use the terms iterator, feedback and dynamic law synonymously. ... The same operation is carried out repeatedly, the output of one iteration being the input for the next one.

Итератор, в този смисъл, е изчислителен метод с обратна връзка (feedback), който има вход и изход, като типът данни на входа и на изхода е еднакъв, най-често реално число. Този метод стартира от даден вход, после изходът се подава като нов вход, изходът от него се подава като нов вход, и така нататък до безкрай.

Ето формално определение.

Определение 6: Итератор

Итератор наричаме изчислителен метод от вида

$$n \mapsto f(n)$$

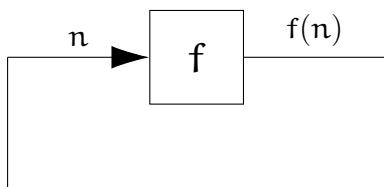
където n е реално число, а f е реална функция. Това описва безкраен изчислителен метод, който започва с n , от него получава $f(n)$, от него получава $f(f(n))$, и така нататък, без край.

$n \mapsto f(n)$ се чете “ n се изобразява във $f(n)$ ”, но тук се има предвид нещо повече от функцията f . Тук функцията е участник в процес с обратна връзка, в който нейният изход y се подава като вход, отново и отново.

Въпросният итератор може да се опише и така:

$$n \mapsto f(n) \mapsto f(f(n)) \mapsto f(f(f(n))) \mapsto \dots$$

А може да се опише картинно и така, при което става съвсем ясно защо говорим за обратна връзка:



Естествено, тази рисунка има смисъл само при допускане за наличие на време (дискретно или континуално), инак всички стойности, генерирани от итератора, трябва да са заедно върху “жицата”.

Въпреки че така описаният изчислителен процес е безкраен, ако итераторът веднъж повтори стойност, тоест достигне n , за което $f(n) = n$, той ще “залепне” в тази стойност и няма да се промени повече. Такава стойност се нарича *фиксирана точка*, на английски *fixed point*, на функцията (и на итератора).

И изобщо, ако итераторът повтори стойност, той ще я повтаря отново и отново, без край. По-подробно казано, ако за някакво n' итераторът генерира редица

$$n' \mapsto n_1 \mapsto n_2 \mapsto \dots \mapsto n_k \mapsto n'$$

то той ще генерира отново и отново тази редица, без край. Ако n' е хронологически първата повторена стойност и тя не се съдържа в $\{n_1, n_2, \dots, n_k\}$, казваме, че редицата

$$\langle n', n_1, n_2, \dots, n_k, n' \rangle$$

е *орбита* за итератора, като дължината на орбитата е $k + 1$. Очевидно фиксирана точка задава орбита с дължина 1.

Ако си представим итератора като граф—само че безкраен граф—върху множеството, върху което е дефиниран, то орбита отговаря на прост цикъл в този граф, а фиксирана точка отговаря на примка в него.

1.1.7 Итеративни и рекурсивни алгоритми

По принцип не е невъзможно алгоритъм да има само императивни инструкции плюс една входно-изходна инструкция. В такъв случай неговата работа се състои от изпълнението на първата инструкция, последвано от изпълнението на втората инструкция, последвано от изпълнението на третата инструкция и така нататък до последната инструкция, която е входно-изходната и връща изхода. Това обаче би бил един твърде прост и скучен алгоритъм. Интересните алгоритми имат разклонения в изпълнението и цикли[†] и/или са рекурсивни.

Стандартната мярка за това дали даден изчислителен модел е “всеомогъщ” е дали може да симулира произволна машина на Turing[‡]. Подробно изложение за машини на Turing има в Секция 14.4. “Всеомогъщ” е доста условно казано заради съществуването на алгоритмично нерешими задачи; имаме предвид [тезиса на Church](#), според който всичко, което е алгоритмично изчислимо, е изчислимо от машина на Turing. Изчислителен модел, който може да симулира произволна машина на Turing, се нарича *пълен по Turing*, на английски [Turing complete](#). Добре известен факт е, че език за програмиране с конструкции `if` и `goto` е пълен по Turing. Ерго, ако в нашите алгоритми допускаме инструкции за условен преход (**if**) и безусловен преход (**goto**), ще можем да конструираме алгоритъм за каквато искаме (алгоритмично решаема) задача, стига да знаем как да го направим.

[†]На английски алгоритмичен или програмен цикъл е *loop*, а не *cycle*!

[‡]Или, алтернативно, [универсалната машина на Turing](#)

На практика нещата са по-сложни. След експлозивното нарастване на софтуера през 50-те години на XX век е забелязана отчетлива тенденция да се създават ужасно трудни за четене, поддръжка и надграждане програми, като голяма част от проблема е безразборно ползване на `goto` инструкции. Затова се появява понятието **структурирано програмиране**, което означава добри практики за писане на ясен за разбиране, поддръжка и надграждане код. Тези добри практики, между другото, настояват циклите да се реализират с `for` или `while` конструкции, като няма излизания от цикъл чрез `goto` от някакво място в тялото на цикъла и няма влизания в тялото на цикъл чрез `goto`, което `goto` е извън цикъла.

Тези добри практики се пренасят при дизайна на алгоритми. Добре известно е, че един и същи алгоритъм може да съществува в различни версии, едната от които е значително по-лесна за верификация от другата, въпреки че двете са напълно еквивалентни в смисъл, че връщат един и същи изход върху всеки вход. Едва ли е необходимо да се каже, че по-трудната за верификация версия не е конструирана съгласно добрите практики, а съдържа микро-оптимизации, които може би малко подобряват бързодействието и/или правят кода по-къс, но затрудняват верификацията. Авторът на записките признава, че някои алгоритми в тези записки съдържат такива затрудняващи верификацията микро-оптимизации, примерно итеративният вариант на двоичното търсене (Алгоритъм 4), който позволява излизане от средата на цикъла (ред 6) с директно връщане на резултата.

Въпреки наличието на малки отклонения от добрите практики на структурираното програмиране, като цяло алгоритмите в тези записки са писани съгласно тях и се разделят, грубо казано, на *итеративни* и *рекурсивни*.

Итеративни алгоритми. “Сърцевината” на итеративните алгоритми е `for` или `while` или `do while` цикъл, като често има вложеност на цикли.

Рекурсивни алгоритми. *Рекурсивен алгоритъм*[†] е алгоритъм, който вика себе си. Очевидно, ако детерминиран алгоритъм извика себе си с точно същия вход, с който е бил извикан, той ще неизбежно ще вика себе си отново и отново със същия вход до безкрай, без да терминира никога. Поради това викане на себе си със същия вход е недопустимо.

Типичен пример за рекурсивен алгоритъм е следният алгоритъм, пресмятащ факториел на число.

```

FACTORIAL( $n \in \mathbb{N}$ )
1  if  $n = 0$ 
2      return 1
3  else
4      return  $n \times \text{FACTORIAL}(n - 1)$ 

```

Да разгледаме работата му върху някакъв малък вход, например 5.

- `FACTORIAL(5)` проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява `FACTORIAL(4)`.
- `FACTORIAL(4)` проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява `FACTORIAL(3)`.

[†]Думата *рекурсия* идва от латинския глагол *currere*—откъдето идва, например, и думата “куриер”—което означава “тичам”, и префикса *re-*, който означава “назад” или “отново”. *Recurre* означава “тичам назад”, откъдето *рекурсивен* буквално означава “тичащ назад”.

- $\text{FACTORIAL}(3)$ проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява $\text{FACTORIAL}(2)$.
- $\text{FACTORIAL}(2)$ проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява $\text{FACTORIAL}(1)$.
- $\text{FACTORIAL}(1)$ проверява условието на ред 1 и понеже то е ЛЪЖА, на ред 4 изпълнява $\text{FACTORIAL}(0)$.
- $\text{FACTORIAL}(0)$ проверява условието на ред 1 и понеже то е ИСТИНА, на ред 2 връща стойност 1 на $\text{FACTORIAL}(1)$.
- $\text{FACTORIAL}(1)$ получава стойност 1 от викането на $\text{FACTORIAL}(0)$, след което умножава 1×1 и връща резултата, който е 1, на $\text{FACTORIAL}(2)$.
- $\text{FACTORIAL}(2)$ получава стойност 1 от викането на $\text{FACTORIAL}(1)$, след което умножава 2×1 и връща резултата, който е 2, на $\text{FACTORIAL}(3)$.
- $\text{FACTORIAL}(3)$ получава стойност 2 от викането на $\text{FACTORIAL}(2)$, след което умножава 3×2 и връща резултата, който е 6, на $\text{FACTORIAL}(4)$.
- $\text{FACTORIAL}(4)$ получава стойност 6 от викането на $\text{FACTORIAL}(3)$, след което умножава 4×6 и връща резултата, който е 24, на $\text{FACTORIAL}(5)$.
- $\text{FACTORIAL}(5)$ получава стойност 24 от викането на $\text{FACTORIAL}(4)$, след което умножава 5×24 и връща 120.

Работата на $\text{FACTORIAL}(5)$ се състои от два етапа. Първият етап можем да наречем *движение надолу*. През него входът намалява, докато не бъде достигната стойността 0, определена от ред 1. Тази стойност ще наречем *спирачка на рекурсията*. Вторият етап да наречем, *движение нагоре*. Същинското изчисляване на факториела става във втория етап. По отношение на софтуерна имплементация на алгоритъма, първият етап е само попълване на подходяща структура данни, а именно стек, с подходящи стойности. Името “рекурсия”, което, както казахме, буквално означава “тичане назад”, без съмнение се дължи на втория етап – движението нагоре.

1.2 Евклидовият алгоритъм

Въпреки че терминът “алгоритъм” идва от името на човек, живял през IX век, алгоритми са били известни и използвани много преди това. Най-старият известен нетривиален алгоритъм е *Евклидовият алгоритъм*. Евклид е един от най-великите умове на античния свят, математик, известен като *баща на геометрията*. Неговата поредица от книги, наречена *Елементи*, е в основата на курсовете по класическа геометрия и до днес. В седмата книга от *Елементи* [69] е описана процедура за изчисляване на най-големия общ делител на две цели положителни числа. Тази процедура има всички характеристики на алгоритъм според Knuth (вж. Подсекция 1.1.5), поради което я наричаме “алгоритъм”. Въпросният алгоритъм се съдържа в Твърдение 1 [69, стр. 296] и Твърдение 2 [69, стр. 298].

1.2.1 Оригиналният Евклидов алгоритъм

Твърдение 1: Кога две числа са взаимно прости

Дадени са две числа, като по-малкото бива изваждано последователно от по-голямото. Ако числото, което се получава накрая, никога не дели това преди него^a, и най-накрая остане единица, оригиналните числа са взаимно прости.

^aИма се предвид, в процеса на изваждане.

Аргументация

Нека по-малкото от две неравни числа AB и CD [†] бива изваждано последователно от по-голямото и нека получаваното число никога не дели това преди него, докато накрая се получи единица; твърдя, че AB и CD са взаимно прости, тоест, само единицата ги дели и двете.

Защото, ако AB и CD не са взаимно прости, някакво число ги дели[‡]. Нека ги дели някакво число и нека то да е E ; нека CD , което дели BF , оставя някакво FA , по-малко от себе си.

Нека AF , което дели DG , оставя по-малко от себе си GC , и нека GC , което дели FH , оставя единица HA .

Но тъй като E дели CD , и CD дели BF , то E дели BF .

Но то също така дели цялото BA ; следователно, то дели и остатъка AF .

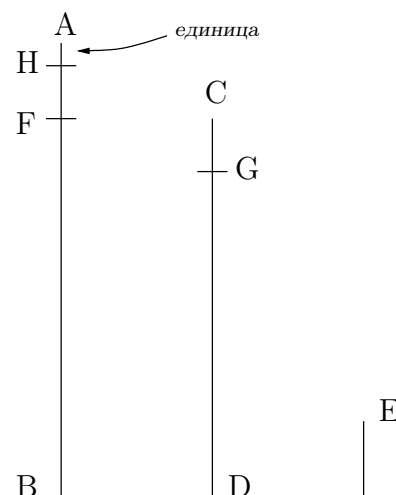
Но AF дели DG ; затова E дели и DG .

Но то също така дели и цялото DC , затова дели и остатъка CG .

Но CG дели FH ; затова E също дели и FH .

Но то дели и цялото FA , затова то още дели и остатъка, а именно единицата AH , въпреки че то[§] е число. А това е невъзможно.

Затова никое число не дели числата AB и CD ; затова те са взаимно прости. $\square$



[†]Древните гърци са мислели за числата като за дължини на отсечки.

[‡]Древните гърци не са смятали единицата за число, а за нулата дори не са имали понятие.

[§]Става дума отново за E .

Твърдение 2: Как се намира най-голям общ делител

Дадени са две числа, които не са взаимно прости, да се намери техният най-голям общ делител.

Аргументация

Нека AB и CD са две дадени числа, които не са взаимно прости.

Иска се да се намери най-големият общ делител на AB , CD .

Ако CD дели AB —тогава то дели и себе си— CD е общ делител на AB и CD .

И е очевидно, че то освен това е най-големият общ делител; защото никое число, по-голямо от CD , не дели CD .

Но ако CD не дели AB , тогава по-малкото от числата AB , CD , ако бъде изваждано последователно от по-голямото, то ще остане число, което дели това преди себе си.

Единица няма да остане; инак, AB , CD ще се окажат взаимно прости (виж Твърдение 1), което противоречи на хипотезата.

Затова ще остане число, което дели това преди него.

Нека CD , което дели BE , оставя EA , което е по-малко от него самото, което дели DF , оставяйки FC , което е по-малко от него самото[†], и нека CF дели AE .

Тъй като CF дели AE , и AE дели DF , следва, че CF дели DF .

Но то дели освен това себе си, затова дели цялото CD .

Но CD дели BE , следователно CF също така дели BE .

Но то[‡] също така дели EA , затова то също така дели цялото BA .

Но то също така дели CD , затова CF дели AB , CD .

Затова CF е общ делител на AB , CD .

Сега аз казвам, че това число е най-голямото такова.

Защото, ако CF не е най-големият общ делител на AB , CD , някое число, по-голямо от CF ще дели числата AB , CD .

Нека едно такова число е G .

Понеже G дели CD , тъй като CD дели BE , G също така дели и BE .

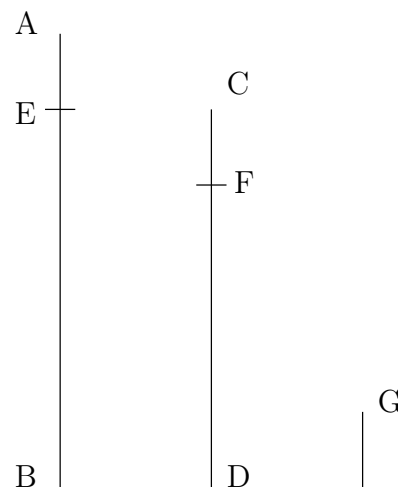
Но то също дели и цялото BA ; затова то дели и остатъка AE .

Но AE дели DF ; затова и G дели DF .

Но то също дели и цялото DC ; затова то дели и остатъка CF , тоест, по-голямото дели по-малкото: което е невъзможно.

Затова никое число, което е по-голямо от CF , не дели числата AB , CD .

Затова CF е най-големият общ делител на AB , CD . □



Този текст ни звучи непривично заради особената нотация и конвенции. Очевидно по Евклидово време не са имали съвременната идея за индукция; той дори не ползва никаква форма на “и така нататък” при аргументацията, а разглежда трикратно прилагане на конструкцията на вземане на (това, което днес бихме нарекли) остатъци при деление и смята, че това е

[†]Вече се има предвид от EA .

[‡] CF

достатъчно убедително. В Твърдение 1, трикратното прилагане е:

AB и CD дават остатък $FA \rightarrow \text{остатък } GC \rightarrow \text{остатък } HA$, равен на единица

Въпреки необичайното за нас изразяване, тези две задачи от седмата книга на Евклид съдържат истинско описание на алгоритъм, напълно недвусмислено, с аргументация за привършване на процедурата и за коректност на получения резултат. Анализ на сложността по време, естествено, няма. В десетата книга на Евклид се дискутира същата идея, но върху числа-отсечки, които днес бихме нарекли “реални”; в Евклидовата терминология, това са отсечки, чието отношение на дължините не е рационално число (*incommensurables* в английския превод). Изключително задълбочена дискусия за Евклидовия алгоритъм има във втория том от поредицата на Knuth [93].

1.2.2 Съвременни варианти на Евклидовия алгоритъм.

Съвременната формулировка на Евклидовия алгоритъм ползва не последователни изваждания, а значително по-бързото изчисляване на остатък при деление (което може да се имплементира чрез изваждания-докато-е-възможно, но това би била много тромава от днешна гледна точка имплементация). Ето две възможни реализации на този алгоритъм: итеративна и рекурсивна.

```

EUCLID ITER( $a, b \in \mathbb{N}^+, a \geq b$ )
1  while  $a \bmod b > 0$  do
2       $r \leftarrow a \bmod b$ 
3       $a \leftarrow b$ 
4       $b \leftarrow r$ 
5  return  $b$ 

```

```

EUCLID REC( $a, b \in \mathbb{N}, a \geq b, a > 0$ )
1  if  $b = 0$ 
2      return  $a$ 
3  else
4      return EUCLID REC( $b, a \bmod b$ )

```

В Лекция 2 ще докажем коректността на Евклидовия алгоритъм.

1.3 Как описваме алгоритми

1.3.1 Псевдокод

Реализациите на Евклидовия алгоритъм от Подсекция 1.2.2 са добра илюстрация за използването на *псевдокод*. Псевдокод е езикът, който ползваме за описание на алгоритми. Това не е истински език за програмиране, а измислен от нас език, който е пределно изчистен от ненужни детайли. С псевдокод не можем да направим синтактична грешка – понеже няма общоприети правила за псевдокод, синтаксисът си го измисляме ние, пишейки псевдокода. Псевдокодът, който ще използваме в тези лекции, е почти същият като псевдокода в учебника на Cormen, Leiserson, Rivest и Stein [34]. Той прилича далечно на синтаксиса на езика за програмиране Pascal. Читател, свикнал със синтаксиса на езиците C или Java лесно ще чете псевдокода, ако има предвид няколко негови особености.

- Ползваме стрелка наляво “ $\leftarrow$ ” за присвояване (на английски, *assignment*), а не “ $:=$ ” като на Pascal или “ $=$ ” като на C.
- Ползваме знак за равенство “ $=$ ” за сравняване на обекти, също като на Pascal, а не “ $==$ ” като на C.

- Също като на Pascal, обхватът на масив се записва с “..”, примерно $A[1 .. n]$.
- Също като на Pascal, двумерните масиви са записват с един чифт прави скоби, а запетаята е разделителят между обхватите, примерно $A[1 .. k, 1 .. n]$.
- Също като на Pascal, масивите се индексират от едно, а не от нула. С други думи, по подразбиране, началният елемент на масива A е $A[1]$, а не $A[0]$. $A[0]$ може да се използва само ако масивът е дефиниран като, примерно, $A[0 .. n]$.
- Цикъл с постусловие **repeat–until** се “върти”, докато постусловието е лъжа; при първото достигане на постусловието, в което то е TRUE, изпълнението на цикъла се прекратява и управлението минава към следващата инструкция, ако има такава. Pascal ползва **repeat–until** конструкции.

В C и Java, циклите с постусловия са **do–while** и са изградени по обратната конвенция: следващо изпълнение има ако и само ако постусловието е TRUE. В тези записки ще се стремим да ползваме **do–while** конструкции, ако искаме да ползваме цикли с постусловие.

- Не се използват ограничители като “{” и “}” за означаване на блокове от код, било в цикъл, било в условна инструкция. Блоковете код се маркират с различен отстъп от лявата страна. Това е като в езика за програмиране Python. Само ако алгоритъмът е много дълъг и окото на читателя може да се обърка, ще ползваме “{” и “}” за означаване на блокове от код.
- В булевите изрази конюнкцията пишем с “**and**”, а дизюнкцията с “**or**”, като в Pascal, а не със съответно “&&” и “||”, като в C. Обаче ползваме една конвенция от C: всеки булев израз-конюнкция се оценява отляво надясно (left to right logical-and evaluation), като при това, ако един от операндите се окаже FALSE, оценяването на останалите операнди не се извършва. В Pascal такава конвенция няма. Тази конвенция ползваме, примерно, в алгоритъма INSERTION SORT (Подсекция 4.4.1), във вложения **while**-цикъл, в който проверката е записана като

$$j > 0 \text{ and } A[j] > key$$

Ако се окаже, че $j \leq 0$, не се проверява дали $A[j] > key$; това е смислено, понеже $A[0]$ не е дефиниран.

Дуално, всеки булев израз-дизюнкция също се оценява отляво надясно, като, ако единият от операндите се окаже TRUE, оценяването на останалите операнди не се извършва.

Писането на псевдокод има потенциален недостатък. Свободата при писането на псевдокода създава опасност да започнем да пишем инструкции, които:

- не са ефикасни
- или не са ефективни,
- или дори не са добре дефинирани.

1.3.2 Други видове описания на алгоритми

Вече видяхме един начин за описване на алгоритми: в псевдокод. Алгоритми може да се описват и на естествен човешки език, например описанията на Евклид в аргументациите на Твърдение 1 и Твърдение 2. Но естественият език е подходящ само за описание на прости алгоритми или за описание на високо ниво, без детайлите. Ако трябва да опишем сложен алгоритъм на човешки език, и то в детайли, неизбежно ще започнем да използваме някаква смесица от естествен език и псевдокод. Колкото по-сложно и детайлно е описанието, толкова по-вероятно е това да бъде неясно[†]. Защо тогава да не ползваме пределно изчистения и недвусмислен псевдокод поначало?

Алгоритми могат да се описват и на истински езици за програмиране като C или Java. В този случай не правим разлика между описание на алгоритъм и неговата програмна реализация – програмната реализация е алгоритъмът. Този начин за описание на алгоритми има съществен недостатък: колкото и да е изчистен езикът за програмиране, програмата, написана на него, съдържа детайли, които са ирелевантни за алгоритъма и пречат да се разбере идеята. Да си припомним цитата на Skiena на стр. 3: “An algorithm is the idea behind any reasonable computer program”. Ако вярваме в това, то има смисъл да правим разлика между описанието на идеята и на конкретна програма, имплементираща идеята. Разбира се, това не трябва да се абсолютизира. Прост алгоритъм като Евклидовият можеше да бъде описан на чисто C и описанието щеше да е толкова ясно, колкото е псевдокодът. Но колкото по-изтънчен и труден за възприемане е алгоритъмът, толкова повече се виждат предимствата на описанието в псевдокод пред описанието на C. Най-малкото, ако пишем на псевдокод, винаги можем да изберем на какво ниво на детайлизация да е описанието и можем да опишем практически всеки алгоритъм съвсем кратко.

И накрая, алгоритми може да се описват чрез *блок-схеми*[‡]. Фигура 1.3 показва блок-схема на Евклидовия алгоритъм (алгоритъм EUCLID ITER от Подсекция 1.2.2). Блок-схемата е диаграма, изобразяваща ориентиран граф с четири вида върхове:

- връх-начало и връх-край, които се рисуват със заоблени правоъгълници,
- върхове-условия, които се рисуват с ромбове, в които са написани условията,
- върхове-императивни инструкции, в които са написани императивните инструкции,
- може да има и четвърти вид върхове, да ги наречем *ветрила*, които не съответстват на инструкции, а са нещо като точки, в които може да дойде изпълнението от повече от едно места.

Стрелките показват как върви изпълнението, което по правило се изобразява отгоре надолу. Върхът-начало има степен на входа нула, защото от него започва изпълнението, и степен на изхода единица. Върховете-условия и върховете-императивни инструкции, както и върхът-край, имат степен на входа единица. Върхът-край има степен на изхода нула, върховете-условия има степен на изхода две, а върховете-императивни инструкции имат степен на изхода едно. Ако в някой връх може да се “влезе” по повече от един начин[§], преди него слагаме връх от тип ветрило, който има повече от един входове и само един изход. Връх-ветрило рисуваме с дебела точка.

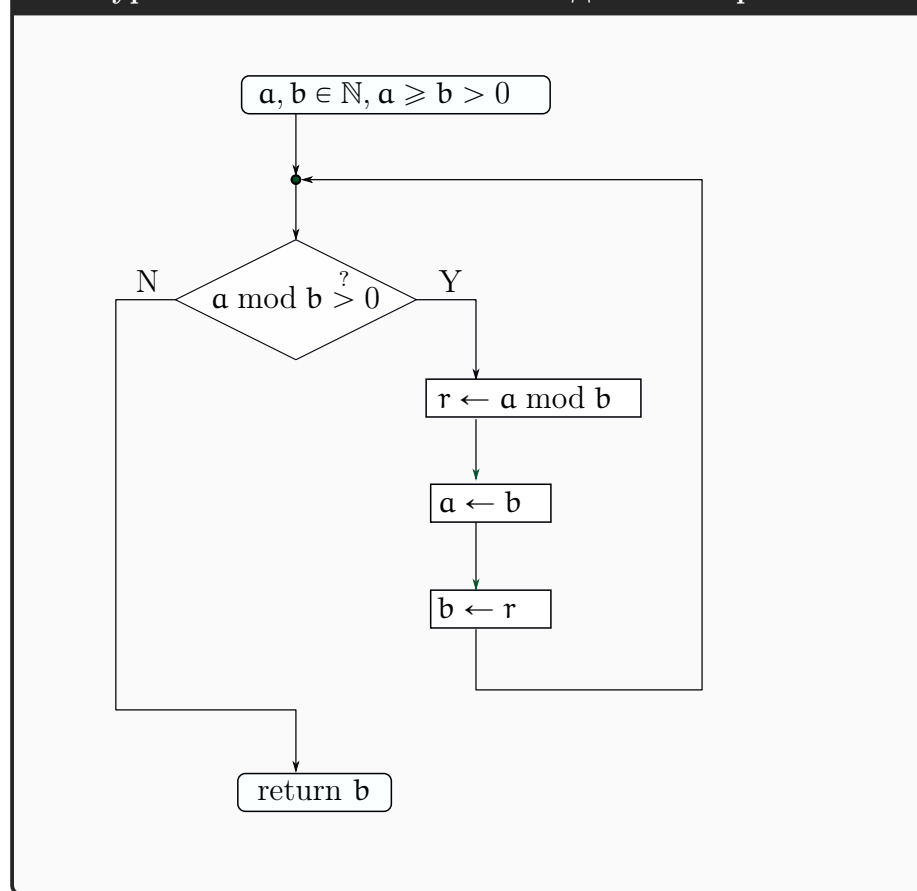
[†]Опитайте се да опишете на чист български език алгоритъм, в който има сложни if-then-else условия с многократна вложеност.

[‡]На английски “блок-схема” е *flowchart*.

[§]Например на Фигура 1.3 в условието “ $a \bmod b \stackrel{?}{>} 0$ ” може да се влезе по два начина: “отгоре” и “отдолу”.

Ориентираните ребра показват изпълнението на алгоритъма. Както казахме, върховете-условия имат степен на изхода две, като едното от излизащите ребра е маркирано с Y, а другото, с N, които очевидно идват от Yes и No и показват къде отива изпълнението на алгоритъма, ако условието е съответно истина или лъжа.

Фигура 1.3 : Блок-схема на Евклидовия алгоритъм.



Описанието с блок-схеми е удачно за прости алгоритми. На практика много често алгоритми се описват с блок-схеми, ако описанието е предназначено за хора, които не са специалисти по алгоритми (съответно е безсмислено да им се дава псевдокод) и освен това алгоритмите са прости, като често дори нямат цикли[†]. Пример за прост алгоритъм без цикли, който трябва да бъде разчетен от човек, който не е специалист по алгоритми, е алгоритъм за това какво да правим, ако автомобилът ни отказва да запали. Ето един *такъв алгоритъм*. Но за сложни алгоритми, особено ако описанието е върху няколко страници, блок-схемите не са подходящи.

[†]Алгоритъмът да няма цикли е същото като графът на блок-схемата да е ацикличен, тоест стрелките да вървят само отгоре надолу.

Лекция 2

Анализ на алгоритми.

Резюме: Разглеждаме различните аспекти на анализа на алгоритмите: анализ на коректността и анализ на сложността, който на свой ред се разбива на анализ по време и анализ по памет. Въвеждаме големина на входа на алгоритъм и видовете сложност като функции от големината на входа. Извеждаме точни изрази за сложността на два много прости алгоритъма. Показваме необходимостта от по-малко прецизна мярка за сложността на алгоритмите. Въвеждаме петте общоприети асимптотични нотации и съответните им релации. Разглеждаме асимптотичната сложност на често срещани функции: факториел, средният биномен коефициент и парциалната сума на хармоничния ред. Разискваме дали високата сложност е непременно лошо нещо.

Да бъде *анализиран* даден алгоритъм означава да бъде доказано, че той е коректен, и ако е коректен, да бъде изчислено колко ресурси ползва—за всеки възможен вход—като ресурсите са основно време и памет. Това са различни аспекти на анализа. Първият аспект—коректността—е свързан с понятието *ефективност*, а вторият аспект е свързан с понятието *ефикасност*.

Допълнение 8: За разликата между *ефективност* и *ефикасност*

В стандартния български език е прието *ефективност* и *ефикасност* да се смятат за синоними, но на английски има значителна разлика между съответните *effective* и *efficient*. Въпросът дали даден алгоритъм е *effective* има отговор или ДА, или НЕ; ако алгоритъмът върши работата, която е определена от заданието (изчислителната задача), то той е **ефективен**, понеже неговата работа дава желанния **ефект**; в противен случай той не е ефективен. При ефикасността въпросът не е дали, а до колко. Ефикасността е свързана с ресурсите, които алгоритъмът ползва.

При тази езикова конвенция, “ефективен” е свързан с резултата, а “ефикасен”, с пътя, по който достига до резултата. Ако зададем въпроса дали даден алгоритъм е ефикасен и очакваме отговор или ДА, или НЕ, то ние сме задали един безсмислен въпрос^a. Обаче има смисъл да твърдим, че един алгоритъм е по-ефикасен от друг алгоритъм по отношение на някакъв ресурс, ако първият ползва по-малко от този ресурс от втория за всички входове; например, ако винаги работи по-бързо. От друга страна, сравняването на ефективността на два алгоритъма е безсмислено, ако и двата са коректни – в такъв случай и двата са ефективни, и това е всичко по отношение на ефективността.

Терминологичната разлика между “ефективен” и “ефикасен” е много полезна на практика и ние ще я спазваме.

^aВ теорията на изчислителната сложност съществува конвенция, според която “ефикасен” означава “работещ в полиномиално време” (Конвенция 15). При наличие на такава конвенция, въпросът дали даден алгоритъм е ефикасен с отговор или ДА, или НЕ, е смислен.

Допълнение 9: Когато коректността не е стопроцентова.

Съществуват сложни изчислителни задачи, за които са приемливи алгоритми, които не винаги работят коректно, но все пак са полезни при условие, че можем да ограничим количествено броя на входовете, върху които алгоритъмът греша. Ако броят на входовете, върху които алгоритъмът не работи коректно, е пренебрежимо малък спрямо броя на всички входове, то този алгоритъм може да е интересен и полезен при условие, че не разполагаме с други алгоритми за тази задача или разполагаме, но те са прекалено бавни. В такъв случай, ефективността не е булева величина ДА/НЕ, а нещо по-сложно, например положително реално число.

В този курс такива задачи и алгоритми няма да разглеждаме, така че ефективността за нас ще е именно булева. Ако някакъв алгоритъм не работи коректно дори за един единствен вход, ще казваме, че не е коректен.

И така, анализирайки един алгоритъм, първо ще доказваме, че е ефективен, тоест коректен, и след това ще изследваме колко е ефикасен, тоест какви ресурси ползва. Когато разглеждаме сложността на **даден алгоритъм за дадена задача**, искаме алгоритъмът да е колко е възможно по-ефикасен (естествено, бивайки ефективен). Когато разглеждаме сложността **на задачата**, фактът, че за нея няма бърз алгоритъм може да бъде от голям теоретичен интерес, но може и да е полезен на практика, както ще стане ясно в Подсекция 2.5.

2.1 Анализ на коректността

Анализът на коректността е по-важният вид анализ в смисъл, че ако се окаже, че даден алгоритъм не е коректен, това е край на анализа му. Колко бързо работи и колко памет ползва некоректен алгоритъм, не ни интересува.

Нека е даден алгоритъм A , който решава някаква изчислителна задача P . Както казахме вече, можем да мислим за P като за релация от множеството на конкретните екземпляри в множеството на конкретните решения, а алгоритъмът A е една възможна реализация на тази релация чрез някакви основни елементи-инструкции. Анализът на коректността е доказателство, че наистина A реализира именно **тази** релация, а не някаква друга. Ще ползваме главно две техники, за да доказваме коректност на алгоритми: чрез индукция (виж страница 38), когато става дума за рекурсивни алгоритми, и чрез *инвариант на цикъл* (виж страница 45), когато става дума за итеративни алгоритми.

Определение 7: Верификация

Верификация, на английски *verification*, означава формално доказване на коректност.

Произходът на термина е от латинските думи *veritas*, което означава “истина”, и *facio*, което означава “правя”. За целите на тези лекции имаме предвид верификации на алгоритми. “Ще верифицираме алгоритъма A ” означава “ще докажем формално коректността на алгоритъма A ”.

2.1.1 Доказателство за финитност

Важна част от доказателството за коректност е доказателството за финитност (вж. Подсекция 1.1.5). С други думи, че алгоритъмът завършва работата си върху всеки вход.

Допълнение 10: Не-финитност и частични функции

Както вече казахме много пъти, алгоритъмът е реализация на някаква релация с домейн екземплярите и кодомейн, решенията, която релация отговаря еднозначно на изчислителната задача. За простота тук ще смятаме, че релацията е функция. При липса на финитност, според терминологията на Knuth (вж. Подсекция 1.1.5), казваме “изчислителен метод” вместо “алгоритъм”.

Свойствата финитност и нефинитност може да бъдат описани елегантно в терминологията на функциите: при наличие на финитност говорим за алгоритъм, който реализира **тотална** функция, а при липса на финитност говорим за изчислителен метод, който реализира **частична** функция^a. В контекста на теорията на алгоритмите ние се интересуваме от реализацията на изображението (частична или тотална функция) чрез елементарни инструкции. Има смисъл да кажем, че даден елемент x от домейна се изобразява в елемент от кодомейна ако и само ако реализацията от елементарни инструкции, стартирайки с вход x , терминира; а ако не терминира, то x не се изобразява в кодомейна. Тогава можем да кажем, че алгоритмите реализират тотални функции, докато изчислителните методи реализират по-общите частични функции.

В този контекст, да докажем финитността на алгоритъма е да докажем, че той е реализация на тотална функция. Повече информация за съответствието между терминиращи и не непременно терминиращи процедури, от една страна, и тотални функции и частични функции, от друга страна, има в книгата на Manna [112].

^aДа си припомним, че “частична функция” е обобщение на “тотална функция”, при което може да има елементи от домейна, на които не съответстват елементи-изображения от кодомейна.

Доказателството за финитност често се пренебрегва, защото бива считано за очевидно. В много случаи наистина е очевидно, но не винаги е така. Може да бъде изключително трудно да се отговори дали даден алгоритъм винаги завършва. Както е показано в Допълнение 6, **не съществува** алгоритъм, който по дадено кодиране на произволен друг алгоритъм плюс вход да определи дали той (другият) ще завърши работата си върху този вход, или не. Алгоритмичната нерешимост на HALTING PROBLEM обрича на провал всеки опит да подходим алгоритмично към анализа на финитността на произволен алгоритъм. Можем да докажем—ако успеем!—финитността на **даден** алгоритъм, но нямаме алгоритмичен начин да правим това за **всеки** алгоритъм.

Нещо повече. Дори за конкретен даден алгоритъм, да анализираме неговата финитност е същото като да решим някаква математическа задача. Верността на математическо твърдение е **кодирана** в това, дали алгоритъмът терминира винаги, или не! Ще разгледаме три примера за това.

Първи пример: хипотезата на Goldbach. Следният пример беше предложен на автора от Георги Георгиев. Хипотезата на Goldbach (вж. Хипотеза 1) може да бъде използвана, за да се конструира алгоритъм, чиято финитност зависи директно от това, дали хипотезата е истина.

DUMMY-GOLDBACH($\emptyset$)

```

1  n ← 2
2  do
3      n ← n + 2
4      more ← FALSE
5      for i ← 2 to  $\frac{n}{2}$ 
6          if isprime(i) and isprime(n - i)
7              more ← TRUE
8  while more

```

Очевидно DUMMY-GOLDBACH терминира ако и само ако Хипотезата на Goldbach е невярна. Тъй като засега не знаем дали хипотезата е вярна, не можем да кажем и дали този алгоритъм терминира, или не.

Втори пример: голямата теорема на Fermat.

Теорема 2: Голяма Теорема на Fermat

Уравнението $a^n + b^n = c^n$ няма решения в цели положителни числа за $n \geq 3$.

Голяма Теорема на Fermat има свой алгоритмичен аналог – да докажем, че следният код, написан на C, не терминира, е същото като да докажем, че тя е вярна:

```

k = 3;
for (;;) {
    for(a = 1; a <= k; a++)
        for(b = 1; b <= k; b++)
            for(c = 1; c <= k; c++)
                for(n = 3; n <= k; n++)
                    if (pow(a,n) + pow(b,n) == pow(c,n))
                        exit();
    k++; }

```

Голямата теорема на Fermat е доказана—за разлика от Хипотезата на Goldbach—така че ние знаем, че тази процедура не терминира и поради това не е алгоритъм. Но тази теорема е **изключително трудна** за доказване. Доказателството ѝ е било непреодолимо предизвикателство в продължение на стотици години. През 90те години на 20 век най-накрая Andrew Wiles направи доказателство [156], чието прочитане (с разбиране ...) остава разбираемо за сравнително малко хора. За подробности вижте [тази онлайн, свободно достъпна, книга на Nigel Boston](#).

Трети пример: хипотезата на Collatz.

Хипотеза 2: Collatz

Нека $C : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ е дефинирана така:

$$C(n) = \begin{cases} \frac{n}{2}, & \text{ако } n \text{ е четно,} \\ 3n + 1, & \text{ако } n \text{ е нечетно.} \end{cases}$$

Нека $C^{(0)}(n) = n$ и $C^{(k+1)}(n) = C(C^{(k)}(n))$, за всяко $n \in \mathbb{N}^+$ и всяко $k \in \mathbb{N}$. Тогава,

$$\forall n \exists m : C^{(m)}(n) = 1$$

Хипотезата казва, че за итераторът (вижте Подсекция 1.1.6) $n \mapsto C(n)$ достига 1 от всяко цяло положително n . Например, ако започнем с 14, ще минем през следните стойности, докато стигнем до единицата:

$$14 \mapsto 7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto 1$$

Въпреки значителните усилия на много математици в продължение на много години, хипотезата остава именно хипотеза – нито е намерен контрапример, нито има доказателство. За повече информация вижте статията на Belaga и Mignotte [11] и огромната библиография от статии за хипотезата на Collatz [101], [102].

Това, че $\langle 1, 4, 2, 1 \rangle$ е орбита за този итератор, е очевидно. Ако игнорираме четните числа, което спокойно можем да направим тук—всяко четно число при достатъчно много деления на 2 дава нечетно число—имаме право да кажем, че 1 е фиксирана точка за итератора на Collatz. Ерго, ако хипотезата е вярна, то итераторът на Collatz влиза в орбита $\langle 1, 4, 2, 1 \rangle$, стартирайки от кое да е цяло положително число. Ако хипотезата е вярна и игнорираме четните числа, то итераторът влиза във фиксирана точка 1, стартирайки от кое да е цяло положително число.

Следният алгоритъм имплементира идеята за финитност, необходимо и достатъчно условие за която е хипотезата на Collatz да е истина.

DUMMY-COLLATZ($n \in \mathbb{N}^+$)

```

1  while  $n \neq 1$  do
2      if iseven( $n$ )
3           $n \leftarrow \frac{n}{2}$ 
4      else
5           $n \leftarrow 3n + 1$ 
```

Очевидно DUMMY-COLLATZ терминира за всяко n ако и само ако Хипотеза 2 е истина.

Наблюдение 7

Доказателството за финитност, което е само един аспект от доказателството за коректност на даден алгоритъм, може да бъде изключително трудно и да изисква дълбоки познания по математика.

Забележка

Алгоритмите, които разглеждаме в този курс, са значително по-прости от посочените в тази подсекция и доказателствата за тяхното терминиране по правило са тривиални. За типичния алгоритъм в курса ние дори не споменаваме това, че той терминира, когато правим анализ на коректността му – няма смисъл да подчертаваме очевидното.

2.1.2 Верификация на рекурсивни алгоритми: по индукция

Формално и прецизно доказателство за коректност на рекурсивен алгоритъм A се прави по индукция по големината на входа. Базата на доказателството отговаря на спирачките на рекурсията – проверяваме, че за всеки вход x с големина, която задейства “спирачка”, $A(x)$ връща точно това, което очакваме. Следва същинската част на доказателството. Разглеждаме вход w с големина, която не задейства “спирачка”. Това значи, че $A(w)$ прави едно или повече викания върху входове с по-малки големина. Допускаме, че всяко от тези викания връща точно това, което очакваме—това е индуктивното предположение—и въз основа на допускането доказваме, че $A(w)$ връща точно това, което очакваме.

Да разгледаме няколко примера.

2.1.2.1 НАРАСТВАНЕ С ЕДИНИЦА

Това е елементарен алгоритъм, който получава число на входа и връща това число, инкрементирано с единица. На практика никой не инкрементира числа по този начин; би било пълна безсмислица да го изпълняваме, при положение, че разполагаме с механизъм да добавяме единица (ред 5), така че може да заменим целия алгоритъм с `return n + 1`. Нещо повече, всеки процесор има инструкции за събиране, които са изключително бързи, така че и викане на процедура за добавяне на единица е безсмислено на практика. Алгоритъмът е стриктно с учебна цел.

Забележете, че индукцията тук не е по големината на входа—както е в почти всички примери, които ще видим в тези лекции—а по стойността на входа. Причината за това отклонение от правилото е, че входът е число (вижте дискусията в Подсекция 2.2.1 за входове, състоящи се от едно число или константен брой числа).

НАРАСТВАНЕ С ЕДИНИЦА(n : естествено число)

```

1  if n = 0
2      return 1
3  else
4      if n mod 2 = 0
5          return n + 1
6      else
7          return 2 × НАРАСТВАНЕ С ЕДИНИЦА( $\lfloor \frac{n}{2} \rfloor$ )

```

Теорема 3: Коректността на алгоритъма НАРАСТВАНЕ С ЕДИНИЦА

Алгоритъмът НАРАСТВАНЕ С ЕДИНИЦА връща $n + 1$ за всеки вход n .

Доказателство:

Със силна индукция по n .

База. $n = 0$. В такъв случай булевият израз на ред 1 е истина и алгоритъмът връща $0 + 1 = 1$ чрез присвояването на ред 2. ✓

Индуктивно Предположение. Нека за всяко $m < n$ алгоритъмът връща $m + 1$.

Индуктивна стъпка. Да разгледаме работата на НАРАСТВАНЕ С ЕДИНИЦА върху вход $n \geq 1$. Първо да допуснем, че n е четно. Тогава булевият израз на ред 4 е истина, следователно ред 5 се изпълнява и алгоритъмът връща $n + 1$. ✓

Сега да допуснем, че n е нечетно. Булевият израз на ред 4 е лъжа, следователно ред 7 се изпълнява и алгоритъмът връща $2 \times \text{НАРАСТВАНЕ С ЕДИНИЦА} \left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Тъй като $\left\lfloor \frac{n}{2} \right\rfloor < n$, може да приложим индуктивното предположение. Съгласно него, НАРАСТВАНЕ С ЕДИНИЦА $\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ връща $\left\lfloor \frac{n}{2} \right\rfloor + 1$. Тъй като n е нечетно, $n = 2k + 1$ за някое $k \in \mathbb{N}$, и изходът е

$$2 \times \left(\left\lfloor \frac{2k+1}{2} \right\rfloor + 1 \right) = 2 \times \left(\left\lfloor k + \frac{1}{2} \right\rfloor + 1 \right) = 2 \times (k + 1) = 2k + 2 = n + 1 \quad \checkmark$$

Това е и краят на доказателството. □

В това доказателство се използва едно неявно допускане: итераторът (вижте Подсекция 1.1.6) $n \mapsto \left\lfloor \frac{n}{2} \right\rfloor$ достига 0, започвайки от произволно естествено число. Това е вярно и може лесно да се докаже по индукция, ако разбием $\mathbb{N}$ на подходящи подмножества $S_0 = \{0\}$, $S_1 = \{1\}$, $S_2 = \{2, 3\}$, $S_3 = \{4, 5, 6, 7\}$, и така нататък, такива че въпросният итератор изобразява S_k в S_{k-1} , и извършим индукцията по индекса на тези множества, но отгоре надолу. Този факт ни позволява да сме убедени, че базата $n = 0$ е достатъчна за рекурсията в НАРАСТВАНЕ С ЕДИНИЦА.

2.1.2.2 МАКСИМУМ НА НЕСОРТИРАН МАСИВ, РЕКУРСИВЕН

Даден е произволен масив от n числа. Търси се максималният елемент на масива. Има елементарен итеративен алгоритъм за тази задача, описан и верифициран в Подподсекция 2.1.3.1. Тук ще разгледаме рекурсивен алгоритъм за задачата. С учебна цел, базата е за $n = 0$ и съответният максимум е $-\infty$; на практика базата би била за $n = 1$ и съответният максимум би бил $A[1]$.

```

MAX UNSORTED, REC(A[1 .. n]: array of integers)
1  if n = 0
2    return  $-\infty$ 
3  else
4    max  $\leftarrow$  MAX UNSORTED, REC(A[1 .. n - 1])
5    if A[n] > max
6      max  $\leftarrow$  A[n]
7    return max

```

Теорема 4: Коректността на алгоритъма MAX UNSORTED, REC

Ако $A[1 .. n]$ е произволен масив от числа, то MAX UNSORTED, REC($A[1 .. n]$) връща неговия максимален елемент.

Доказателство: По индукция по n . Забележете, че това е и индукция по големината на входа, защото в тази задача n е и големината на входа, за разлика от алгоритъма НАРАСТВАНЕ С ЕДИНИЦА на предишната страница, в който индукцията пак е по n , но големината на входа винаги е единица.

Базовият случай е $n = 0$. От една страна, масивът е празен, така че максимумът му е $-\infty$, която е неутралният елемент на операцията максимум. От друга страна, условието на ред 1 е истина и алгоритъмът връща $-\infty$ на ред 2. ✓

Да допуснем, че за някое $n - 1$, такова че $n - 1 \geq 0$, MAX UNSORTED, REC($A[1 \dots n - 1]$) връща максималния елемент на $A[1 \dots n - 1]$. Разглеждаме викането MAX UNSORTED, REC($A[1 \dots n]$). Тъй като $n > 0$, условието на ред 1 е лъжа и изпълнението отива на ред 4. Съгласно индуктивното предположение, променливата *max* получава стойността на максимума на $A[1 \dots n - 1]$. Следните две възможности са изчерпателни:

- $A[n]$ е по-голям от всеки елемент на $A[1 \dots n - 1]$. Прилагаме индуктивното предположение и заключаваме, че $A[n] > \text{max}$. Условието на ред 5 е истина и изпълнението отива на ред 6. Там на *max* се присвоява $A[n]$, което, при текущите допускания, е максимумът на $A[1 \dots n]$. После изпълнението отива на ред 7.
- Не е вярно, че $A[n]$ е по-голям от всеки елемент на $A[1 \dots n - 1]$. Тогава максимумът на $A[1 \dots n]$ е максимумът на $A[1 \dots n - 1]$. Прилагаме индуктивното предположение и заключаваме, че на ред 4, на *max* се присвоява максимума на $A[1 \dots n]$. Условието на ред 5 е лъжа и изпълнението отива на ред 7.

Във всеки от тези случаи, на ред 7, променливата *max* съдържа максимума на $A[1 \dots n]$. Заключаваме, че алгоритъмът връща максимума на $A[1 \dots n]$. □

2.1.2.3 МАКСИМУМ НА БИТОНИЧЕН МАСИВ

Неформално, масив се нарича *битоничен*, на английски *bitonic*, ако започва като строго растящ, след което става строго намаляващ. Формално, ако $A[1 \dots n]$ е числен масив, то A е битоничен, ако съществува индекс $k \in \{1, \dots, n\}$, такъв че

$$\begin{aligned} A[1] &< A[2] < \dots < A[k] \\ A[k+1] &> A[k+2] > \dots > A[n] \end{aligned}$$

Забележете, че нарастващата част е празна, в случай, че $k = 1$; тогава първата верига от неравенства съдържа нула неравенства и е тривиално истина. Аналогично, намаляващата част е празна, ако $k = n$.

Дефиницията не казва нищо за взаимните големина на $A[k]$ и $A[k + 1]$.

- Може $A[k] < A[k + 1]$. В този случай нарастващата част е $A[1 \dots k + 1]$, а намаляващата е $A[k + 1 \dots n]$, като те имат общ елемент $A[k + 1]$.
- Може $A[k] > A[k + 1]$. В този случай нарастващата част е $A[1 \dots k]$, а намаляващата е $A[k \dots n]$, като те имат общ елемент $A[k]$.
- Може $A[k] = A[k + 1]$. В този случай нарастващата част е $A[1 \dots k]$, а намаляващата е $A[k + 1 \dots n]$, като те нямат общ елемент.

Следователно, има или точно един максимален елемент, или точно два максимални елемента, които са съседни в масива. Всеки от следните масиви е битоничен:

[1, 2, 3, 4, 9]
 [9, 4, 3, 2, 1]
 [1, 2, 3, 9, 1]
 [1, 2, 3, 9, 9, 1]

Повече от два максимални елемента обаче не може да има. Освен това, ако $A[k] = A[k + 1]$, със сигурност това е максималният елемент.

Да разгледаме следния рекурсивен алгоритъм. Ще покажем, че той връща максимума на $A[1..n]$, ако масивът е битоничен и началното викане е $\text{MAX BITONIC}(A, 1, n)$.

Алгоритъм 1: Max Bitonic

```

MAX BITONIC( $A[1..n]$ ): битоничен масив;  $\ell, h$ : цели числа, такива че  $1 \leq \ell \leq h \leq n$ )
1  if  $\ell = h$ 
2      return  $A[\ell]$ 
3  mid  $\leftarrow \lfloor \frac{\ell+h}{2} \rfloor$ 
4  if  $A[\text{mid}] < A[\text{mid} + 1]$ 
5      return MAX BITONIC( $A, \text{mid} + 1, h$ )
6  else if  $A[\text{mid}] > A[\text{mid} + 1]$ 
7      return MAX BITONIC( $A, \ell, \text{mid}$ )
8  else
9      return  $A[\text{mid}]$ 

```

Теорема 5: Коректността на алгоритъма MAX BITONIC

Ако $A[1..n]$ е битоничен и $1 \leq \ell \leq h \leq n$, то $\text{MAX BITONIC}(A, \ell, h)$ връща максималния елемент на подмасива $A[\ell..h]$.

Доказателство: Управляващ параметър за рекурсията е разликата $h - \ell$. Това си заслужава да се повтори: въпреки че има два числени аргумента, а именно ℓ и h , рекурсията е едномерна[†], понеже е по разликата $h - \ell$. Рекурсивно викане се прави с по-малка такава разлика и “спирачката” на рекурсията се “задейства” от тази разлика.

Преди да продължим с доказателството, да се убедим, че индексът mid , равен на $\lfloor \frac{\ell+h}{2} \rfloor$, притежава свойства, заради които рекурсията е добре дефинирана. Да е добре дефинирана означава три неща.

- Първо, разликата горен индекс минус долен индекс във всяко от двете рекурсивни викания (редове 5 и 7) е по-малка от $h - \ell$. Това е важно, понеже, както вече казахме, рекурсията е по разликата горен индекс минус долен индекс.
- Второ, разликата горен индекс минус долен индекс във всяко от двете рекурсивни викания (редове 5 и 7) е неотрицателна. Това е важно, понеже спирачката на рекурсията (ред 2) е върху само една стойност на тази разлика, а именно нула. Ерго, нулата трябва да е достижима – от каквито и ℓ и h да тръгнем, какъвто и път да измине рекурсията надолу, тоест, което от двете рекурсивни викания да се изпълни във всяко не-листо на дървото на рекурсията[‡], трябва в крайна сметка разликата горен индекс минус долен индекс да стане нула, а не отрицателно число. Разликата да стане отрицателна означава спирачката да бъде прескочена.

[†]Двумерна рекурсия е, примерно, пресмятането на биномния коефициент съгласно $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

[‡]Не се твърди, че непременно ще се изпълни едното от двете рекурсивни викания! Може алгоритъмът да терминира в ред 9 в някое не-листо на дървото на рекурсията. Тук обаче разглеждаме възможността ред 9 да не бъде достигнат, за да се убедим, че всяка редица от рекурсивни викания завършва с разлика горен индекс минус долен индекс равна на нула.

Забележете, че ако булевото условие на ред 1 беше $\ell \geq h$, нямаше да има нужда да се притесняваме за всичко това – тогава не би имало начин спирачката да се окаже прескочена.

- Трето, елементите $A[\text{mid}]$ и $A[\text{mid} + 1]$, за които става дума на ред 4 и на ред 6, се намират в подмасива $A[\ell \dots h]$. С други думи, че няма опит за достъп извън подмасива от входа.

Лема 1

Ако е ℓ и h са неотрицателни числа, такива че $\ell < h$, и ако $\text{mid} = \lfloor \frac{\ell+h}{2} \rfloor$, в сила е

$$0 \leq h - (\text{mid} + 1) < h - \ell \quad (2.1)$$

$$0 \leq \text{mid} - \ell < h - \ell \quad (2.2)$$

Доказателство: Ще покажем $0 \leq h - (\text{mid} + 1)$ от (2.1), което е същото като $\text{mid} + 1 \leq h$, което е същото като $\text{mid} < h$. В сила е:

$$\left\lfloor \frac{\ell + h}{2} \right\rfloor < h \quad \leftrightarrow \quad \left\lfloor \frac{\ell + h}{2} \right\rfloor - h < 0 \quad \leftrightarrow \quad \left\lceil \frac{\ell - h}{2} \right\rceil < 0$$

като последното е вярно (почти) директно от дефиницията на $\lfloor \cdot \rfloor$ и факта, че $\ell - h < 0$. ✓

Ще покажем $h - (\text{mid} + 1) < h - \ell$ от (2.1), което е същото като $\text{mid} + 1 > \ell$, което е същото като $\text{mid} \geq \ell$. В сила е

$$\left\lfloor \frac{\ell + h}{2} \right\rfloor \geq \ell \quad \leftrightarrow \quad \left\lfloor \frac{\ell + h}{2} \right\rfloor - \ell \geq 0 \quad \leftrightarrow \quad \left\lfloor \frac{\ell + h}{2} - \ell \right\rfloor \geq 0 \quad \leftrightarrow \quad \left\lfloor \frac{h - \ell}{2} \right\rfloor \geq 0$$

като последното е вярно (почти) директно от дефиницията на $\lfloor \cdot \rfloor$ и факта, че $h - \ell > 0$. ✓

Ще покажем $0 \leq \text{mid} - \ell$ от (2.2). Но ние вече показахме, че $\text{mid} \geq \ell$. ✓

Ще покажем $\text{mid} - \ell < h - \ell$ от (2.2), което е същото като $\text{mid} < h$. Но ние вече показахме точно това. ✓ □

Следствие 1

Щом $0 \leq \text{mid} + 1 \leq h$, рекурсивното викане на ред 5 е добре дефинирано. Щом $0 \leq \ell \leq \text{mid} \leq h$, рекурсивното викане на ред 7 е добре дефинирано.

Базовият случай е $h - \ell = 0$, тоест, $h = \ell$. От една страна, в такъв случай $A[\ell \dots h]$ е $A[\ell \dots \ell]$, тоест, $A[\ell]$. Очевидно максимумът е $A[\ell]$. От друга страна, ако четем псевдокода, виждаме, че този случай “задейства спирачката” на рекурсията, като условието на ред 1 е истина и изпълнението достига до ред 2. Наистина, алгоритъмът връща $A[\ell]$ на ред 2. ✓

Да разгледаме стойности на ℓ и h , които не “задействат спирачката” на рекурсията. С други думи, $\ell < h$. Индуктивното предположение е следното: за всички ℓ' и h' , такива че $\ell \leq \ell' \leq h' \leq h$ и $h' - \ell' < h - \ell$, $\text{MAX BITONIC}(A, \ell', h')$ връща максимума на $A[\ell' \dots h']$.

Да разгледаме работата на $\text{MAX BITONIC}(A, \ell, h)$. Щом $\ell \neq h$, изпълнението отива на ред 3, където на mid се присвоява $\lfloor \frac{\ell+h}{2} \rfloor$.

Следните възможности са взаимно изключващи се и изчерпателни.

- $A[\text{mid}] < A[\text{mid} + 1]$. В този случай нито един от $A[\ell], \dots, A[\text{mid}]$ не е максимален за $A[\ell \dots h]$, защото входният масив е битоничен. Тогава максимумът на $A[\ell \dots h]$ се намира в подмасива $A[\text{mid} + 1 \dots h]$. Тоест, максимумът на $A[\ell \dots h]$ е максимумът на $A[\text{mid} + 1 \dots h]$.

От друга страна, в този случай условието на ред 4 е истина и се изпълнява ред 5. Но $h - (\text{mid} + 1) < h - \ell$ (Лема 1) и можем да приложим индуктивното предположение. От него следва, че рекурсивното викане на ред 5 връща максимума на $A[\text{mid} + 1 \dots h]$. Както вече установихме, той е и максимумът на $A[\ell \dots h]$. Заклучаваме, че в този случай $\text{MAX BITONIC}(A, \ell, h)$ връща максимума на $A[\ell \dots h]$. ✓

- $A[\text{mid}] > A[\text{mid} + 1]$. В този случай нито един от $A[\text{mid} + 1], \dots, A[h]$ не е максимален, защото входният масив е битоничен. Тогава максимумът на $A[\ell \dots h]$ се намира в подмасива $A[\ell \dots \text{mid}]$. Тоест, максимумът на $A[\ell \dots h]$ е максимумът на $A[\ell \dots \text{mid}]$.

От друга страна, в този случай условието на ред 4 е лъжа. Изпълнението отива на ред 6. Условието там е истина и се изпълнява ред 7. Но $\text{mid} - \ell < h - \ell$ (Лема 1) и можем да приложим индуктивното предположение. От него следва, че рекурсивното викане на ред 7 връща максимума на $A[\ell \dots \text{mid}]$. Както вече установихме, той е и максимумът на $A[\ell \dots h]$. Заклучаваме, че в този случай $\text{MAX BITONIC}(A, \ell, h)$ връща максимума на $A[\ell \dots h]$. ✓

- $A[\text{mid}] = A[\text{mid} + 1]$. В този случай имаме двоен максимум $A[\text{mid}] = A[\text{mid} + 1]$, защото входният масив е битоничен.

От друга страна, изпълнението достига ред 9, където алгоритъмът връща $A[\text{mid}]$. Заклучаваме, че в този случай $\text{MAX BITONIC}(A, \ell, h)$ връща максимума на $A[\ell \dots h]$. ✓ □

За пълнота на доказателството е добре да споменем изрично следния факт, който е тривиално следствие на Теорема 5.

Следствие 2: Коректността на началното викане на MAX BITONIC

$\text{MAX BITONIC}(A, 1, n)$ връща максималния елемент на масива A .

2.1.2.4 Евклидовият алгоритъм в рекурсивен вариант

Тъй като става дума за *най-голям общ делител*, на английски *greatest common divisor*, накратко *gcd*, нека за пълнота дефинираме това добре познато понятие.

Определение 8: $\text{gcd}(a, b)$

Нека a и b са естествени числа, които не са и двете нули. Тогава $\text{gcd}(a, b)$ е най-голямото естествено число, което дели и a , и b .

Следствие 3

Ако $a \in \mathbb{N}^+$, то $\text{gcd}(a, 0) = a$.

Съгласно [61, (3.21) на стр. 82]:

Определение 9: mod: остатък при целочислено деление

Нека $a \in \mathbb{N}$ и $b \in \mathbb{N}^+$. Тогава

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Лема 2

Нека $a \in \mathbb{N}$ и $b \in \mathbb{N}^+$. Всеки общ делител на a и b също така дели и $a \bmod b$.

Доказателство: Нека c е общ делител на a и b . Тогава съществуват цели положителни p и q , такива че $a = cp$ и $b = cq$. Но $a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$ от Определение 9, така че $a \bmod b = cp - cq \left\lfloor \frac{a}{b} \right\rfloor = c(p - q \left\lfloor \frac{a}{b} \right\rfloor)$. Но $p - q \left\lfloor \frac{a}{b} \right\rfloor$ е цяло число. Ерго, c дели $a \bmod b$. $\square$

Лема 3

Нека $a, b \in \mathbb{N}$ и $a \geq b > 0$. Тогава $a \bmod b < b$.

Доказателство: Ако b дели a , то твърдението е вярно, понеже $a \bmod b = 0$, а $b > 0$ по условие. Нека b не дели a . Тогава $\frac{a}{b} = k + x$, където $k \in \mathbb{N}^+$, а $0 < x < 1$. Тогава $a = bk + bx$ и $\left\lfloor \frac{a}{b} \right\rfloor = k$, откъдето

$$b \left\lfloor \frac{a}{b} \right\rfloor = bk \tag{2.3}$$

Трябва да покажем, че $a \bmod b < b$, което е по дефиниция $a - b \left\lfloor \frac{a}{b} \right\rfloor < b$, което е еквивалентно на $a < b \left\lfloor \frac{a}{b} \right\rfloor + b$, което е еквивалентно на $a < bk + b$ предвид (2.3). Но вече знаем, че $a = bk + bx$, където $0 < x < 1$. Заклучаваме, че наистина $a < bk + b$. $\square$

Следната теорема е ключова за верификацията на Евклидовия алгоритъм.

Теорема 6: $\gcd(a, b)$ е или a , или $\gcd(b, a \bmod b)$

Нека $a, b \in \mathbb{N}$ и $a \geq b$, $a > 0$. Тогава

$$\gcd(a, b) = \begin{cases} a, & \text{ако } b = 0 \\ \gcd(b, a \bmod b), & \text{в противен случай} \end{cases}$$

Доказателство: Ако $a > 0$ и $b = 0$, то $\gcd(a, b) = a$ от Следствие 3. Нека $a \geq b > 0$.

- Ако b дели a , то $a \bmod b = 0$, така че “ $\gcd(b, a \bmod b)$ ” става “ $\gcd(b, 0)$ ”, а $\gcd(b, 0) = b$ от Следствие 3. От друга страна, $\gcd(a, b) = b$ в този случай, така че твърдението пак е вярно.
- Нека b не дели a . Тогава $a \bmod b > 0$.
 - ♦ Съгласно Лема 2, ако d е общ делител на a и b , то d е делител и на $a \bmod b$.
 - ♦ От друга страна, ако някое цяло d дели както b , така и $a \bmod b$, то d дели и $b \left\lfloor \frac{a}{b} \right\rfloor + a \bmod b$. Но $b \left\lfloor \frac{a}{b} \right\rfloor + a \bmod b = a$. Следователно, ако някое цяло d дели както b , така и $a \bmod b$, то d дели и a .

Заклучаваме, че d е общ делител на $a \bmod b$ и b т.е. d е общ делител на a и b . Ерго, множеството от делителите на $a \bmod b$ и b точно съвпада с множеството от делителите на a и b . Щом имат едни и същи делители, в частност най-големият им делител е един и същи. $\square$

Ето отново рекурсивната версия на Евклидовия алгоритъм от Подсекция 1.2.2.

```

EUCLID_REC( $a, b \in \mathbb{N}, a \geq b, a > 0$ )
1  if  $b = 0$ 
2    return  $a$ 
3  else
4    return EUCLID_REC( $b, a \bmod b$ )

```

Теорема 7: Коректността на алгоритъма EUCLID_REC

Алгоритъмът EUCLID_REC(a, b) връща $\gcd(a, b)$ за всички естествени a и b , такива че $a \geq b$.

Доказателство: Рекурсията е двумерна, в смисъл, че има два аргумента, а спирачката (началното условие) е едномерното $b = 0$. Съгласно Лема 3, ако $a \geq b > 0$, то $a \bmod b < b$, откъдето заключаваме, че итераторът $\langle a, b \rangle \mapsto \langle b, a \bmod b \rangle$ винаги достига до $\langle c, 0 \rangle$ за някое $c \in \mathbb{N}^+$. Това означава, че алгоритъмът терминира винаги. Остава да докажем коректността, което ще направим по индукция по втория аргумент b .

База. В базовия случай $b = 0$, т.е. разглеждаме входовете на алгоритъма $\langle a, 0 \rangle$, където $a > 0$. Тогава условието на ред 1 е истина и изпълнението отива на ред 2 и връща a . Но съгласно Следствие 3, $\gcd(a, 0) = a$, ако $a > 0$. ✓

Индуктивно Предположение. Нека за някакви естествени a и b , такива че $a \geq b > 0$, за всяка наредена двойка $\langle p, q \rangle$, такава че $p \geq q \geq 0$, $p > 0$ и $q < b$ е вярно, че EUCLID_REC(p, q) връща $\gcd(p, q)$.

Индуктивна стъпка. Разглеждаме работата на EUCLID_REC(a, b). Щом $b > 0$, условието на ред 1 е лъжа и изпълнението отива на ред 4, където се вика EUCLID_REC($b, a \bmod b$). Но $a \bmod b < b$ съгласно Лема 3. Щом $a \bmod b < b$, съгласно индуктивното предположение, EUCLID_REC($b, a \bmod b$) връща $\gcd(b, a \bmod b)$.

От Теорема 6 знаем, че $\gcd(a, b) = \gcd(b, a \bmod b)$, ако $b > 0$. Следователно, това, което EUCLID_REC(a, b) връща на ред 4, е $\gcd(a, b)$, което и трябваше да покажем. $\square$

2.1.3 Верификация на итеративни алгоритми: с инвариант

Доказателствата за коректност с инвариант на цикъла по същество също са доказателства по индукция, но с индукцията не е по големината на входа, а по броя на “завъртанията” на цикъла.

Инвариантът е **едноместен предикат**, свързан с цикъла. Променливата на предиката е броят на достиганията на дадена инструкция на алгоритъма. По правило, тази инструкция е инструкцията, която съдържа условието за край на цикъла. Да кажем, че условието за край на цикъла е на ред ℓ . Въпросният предикат трябва да е:

- верен първия път, когато изпълнението на алгоритъма е на ред ℓ ,

- верността му трябва да се запазва по време на всяко изпълнение, откъдето идва и името “инвариант”, и
- в момента на напускане на цикъла (такъв момент настъпва неизбежно, щом алгоритъмът завършва) неговата вярност трябва да влече директно това, което искаме да докажем за алгоритъма.

Тъй като алгоритъмът завършва работата си рано или късно, то броят на пътните, в които изпълнението е на ред ℓ , е **краен**. Ако ред ℓ се изпълнява точно m пъти, то индукцията е върху крайното множество $\{1, 2, \dots, m\}$

Това, какъв предикат използваме в доказателството, е ключово. Ако не успеем да измислим адекватен предикат, няма да имаме доказателство, колкото и добре да разбираме техниката с инвариант по принцип. Най-общо, предикатът се формулира така:

Инвариант

При k -тото достигане на ред ℓ на алгоритъм **НЯКАКЪВ АЛГОРИТЪМ** е изпълнено «*някакво-твърдение-формулирано-чрез- k* ».

Ако предикатът е $P(k)$, то доказателството следва позната схема на доказателства по индукция върху **краен** домейн (който е индуктивно дефинирано множество):

$$P(1) \wedge \forall k_{1 \leq k < m} (P(k) \rightarrow P(k+1)) \quad \vdash \quad \forall k_{1 \leq k \leq m} (P(k))$$

Пример за верификация на итеративен алгоритъм с инвариант. Това е може би най-простият итеративен алгоритъм. Ще докажем коректността му с инвариант.

FIND SUM($A[1 \dots n]$: array of integers)

```

1  sum ← 0
2  i ← 1
3  while i ≤ n do
4      sum ← sum + A[i]
5      i ← i + 1
6  return sum
```

Теорема 8: Коректността на алгоритъм за сумата от елементите на масив

FIND SUM връща сумата от елементите на входния масив $A[1 \dots n]$.

Доказателство: Доказателството на теоремата се основава на следното помощно твърдение, което ще докажем първо.

Инвариант

Всеки път, когато изпълнението на FIND SUM е на ред 3, променливата *sum* съдържа сумата от елементите на подмасива $A[1 \dots i - 1]$.

Доказателството на инварианта е по индукция по броя на достиганията на ред 3.

База. Да разгледаме първото достигане на ред 3. От една страна, подмасивът $A[1 \dots i-1]$ в този момент е празен, защото i съдържа стойност 1 заради присвояването на ред 2, следователно $A[1 \dots i-1]$ всъщност е $A[1 \dots 0]$, а това е празният подмасив. Сумата от елементите на празния масив е 0, защото 0 е неутралният елемент на операцията събиране. Следователно, математически погледнато, сумата от елементите на $A[1 \dots i-1]$ е 0. От друга страна, променливата sum съдържа 0 заради присвояването на ред 2. ✓

Индуктивно предположение. Да допуснем, че твърдението е вярно при някое достигане на ред 3, което не е последното. А именно, допускането е, че sum съдържа сумата от елементите на подмасива $A[1 \dots i-1]$.

Преди да продължим с доказателството, подчертаваме, че би било грешка да изпуснем уточнението “което не е последното”, защото искаме да се убедим, че инвариантът се запазва при преминаване през тялото на цикъла, а след последното достигане на ред 3, повече преминаване през тялото цикъла няма.

Индуктивна стъпка. След присвояването на ред 4, прилагайки и индуктивното предположение, заключаваме, че sum съдържа сумата от елементите на подмасива $A[1 \dots i]$. Но след това на ред 5, променливата i се инкрементира. Изразено чрез новото i , заключението става “ sum съдържа сумата от елементите на подмасива $A[1 \dots i-1]$ ”. Но това е точно същото нещо, което инвариантът казва. Ерго, инвариантът се запазва.

С това доказателството на инварианта приключи. Да се върнем към доказателството на теоремата.

Следствие от инварианта. FIND SUM очевидно терминира, така че някое достигане на ред 3 е последно. Да разгледаме моментът, в който ред 3 се изпълнява за последен път. Тогава i съдържа стойността $n+1$. Да заместим i с $n+1$ в инварианта. Получаваме буквално “променливата sum съдържа сумата от елементите на подмасива $A[1 \dots (n+1)-1]$ ”. По-просто казано, sum съдържа сумата от елементите на масива $A[1 \dots n]$.

Доказателството е почти готово. Забелязваме, че веднага след това се изпълнява ред 6. FIND SUM връща sum , която променлива съдържа сумата от елементите на входния масив. Следователно, FIND SUM връща сумата от елементите на входния масив, което и трябваше да покажем. □

В следващите доказателства за коректност с инвариант няма да правим следствие от инварианта, както сторихме току-що, а в доказателството ще добавяме още една фаза, наречена **Терминация**. Освен това, за краткост ще сливаме фазите **Индуктивно предположение** и **Индуктивна стъпка** в една единствена фаза, наречена **Поддръжка**. Въпросните доказателства имат следната структура съгласно учебника CLR [34]. Инвариантът е едноместен предикат, да го наречем $P(k)$, който е свързан с този ред от алгоритъма, който съдържа условието, управляващо цикъла. В най-общия случай, променливата k е броят на достиганията на въпросния ред от началото на изпълнението, включително и сегашното достигане. Общата схема на доказателството, както стана ясно от изложението досега, е:

- **База** Показваме, че $P(1)$ е истина при първото достигане на въпросния ред.
- **Поддръжка** Допускайки, че $P(k)$ е в сила за някое достигане, което не е последното, показваме, че $P(k+1)$ е в сила при следващото достигане.
- **Терминация** Разглеждаме момента, в който изпълнението е на въпросния ред от алгоритъма за последен път. Нека k' е броят на всички достигания до въпросния ред, включително и последното (текущото). Замествайки k с k' в инварианта, получаваме точно това твърдение за алгоритъма, което ни трябва.

Важен частен случай е следният. Да кажем, че имаме **for**-цикъл, като ℓ е номерът на реда на **for**-а в алгоритъма:

ℓ : **for** $i \leftarrow a$ to b

a и b са константи, а променливата i не се променя в тялото на цикъла. Тогава за простота променливата на предиката е управляващата променлива на цикъла, тоест предикатът е $P(i)$. Общата схема на доказателството в този частен случай е:

- Показваме, че $P(a)$ е вярно при първото достигане на ред ℓ .
- Допускайки, че $P(i)$ е вярно при някое достигане на ред ℓ , което не е последното, (което значи, че $i \leq b$), доказваме $P(i + 1)$ при следващото достигане на ред ℓ .
- Разглеждаме момента, в който доказателството е на ред ℓ за последен път. Тогава трябва i да съдържа $b + 1$. Замествайки i с $b + 1$, получаваме получаваме точно това твърдение за алгоритъма, което ни трябва.

Дебело подчертаваме следните две неща.

- Първо, може да има много начини да се мине през тялото на цикъла в зависимост от стойностите както на контролната променлива, така и на останалите променливи. Например, може тялото на цикъла да има сложна вложена **if-else-if** структура. Доказателството трябва да следва всеки възможен път на изпълнение, тоест всяка възможност за преминаване през тази структура. Примерно, ако тялото на цикъла има k на брой последователни **if-else** блока:

```

if «булево условие 1» then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»
if «булево условие 2» then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»
...
if «булево условие  $k$ » then
    «някакви императивни инструкции»
else
    «някакви императивни инструкции»

```

то от най-общи комбинаторни съображения има 2^k начина, по които изпълнението може да мине през всички тези **if-else**. Доказателството, което разглежда всеки от тези 2^k начина, може да е изключително дълго.

- Въпросният предикат P трябва не просто да е верен, а освен това да ни върши работа за доказателството. Не всеки верен предикат ни върши работа. Като пример, да разгледаме следния алгоритъм.


```

ALGY( $A[1 \dots n]$ : масив от цели числа)
1   for  $i \leftarrow 1$  to  $n$ 
2        $A[i] \leftarrow i$ 

```

Да разгледаме невалидно доказателство, че това е сортиращ алгоритъм (какъвто той не е), което се основава на следния инвариант.

Инвариант

Всеки път, когато изпълнението на ALGY е на ред 1, подмасивът $A[1 \dots i-1]$ е сортиран.

Този инвариант верен ли е? Очевидно да, и доказването му е тривиално. ALGY сортиращ алгоритъм ли е? Очевидно не, защото той унищожава входните данни; това, че реализира сортиран масив, а именно $[1, 2, \dots, n-1, n]$, не е достатъчно, за да е сортиращ. Защо верният инвариант не доказва коректността на ALGY? Защото, за да бъде един алгоритъм сортиращ, той трябва не просто да конструира сортиран масив, а трябва да конструира сортиран масив, който **се състои точно от входните елементи**. Още веднъж: инвариантът трябва да е не просто верен, а и полезен за целта на доказателството.

Наблюдение 8

Не всеки верен инвариант е полезен за това, което искаме да докажем.

Да разгледаме още няколко примера за доказателство на коректност чрез инвариант.

2.1.3.1 МАКСИМУМ НА НЕСОРТИРАН МАСИВ, ИТЕРАТИВЕН

На практика никой не би инициализирал *max* с $-\infty$; на практика човек би инициализирал *max* с $A[1]$ и i с 2. Следната имплементация е с учебна цел.

```

MAX UNSORTED, ITER( $A[1 \dots n]$ : array of integers)
1    $max \leftarrow -\infty$ 
2    $i \leftarrow 1$ 
3   while  $i \leq n$  do
4       if  $A[i] > max$ 
5            $max \leftarrow A[i]$ 
6        $i \leftarrow i + 1$ 
7   return  $max$ 

```

Теорема 9: Коректността на алгоритъма MAX UNSORTED, ITER

Нека $A[1 \dots n]$ е произволен масив от числа. MAX UNSORTED, ITER(A) връща максимума на A .

Преди да започнем да доказваме теоремата да отбележим нещо. Нейното доказателство ползва неявно факта, че MAX UNSORTED, ITER не променя входния масив. Това е напълно

очевидно: в нито една от четирите инструкции за присвояване (редове 1, 2, 5 и 6) не се присвоява стойност на елемент на A . Този факт облекчава доказателството, защото позволява да говорим просто за “масива A ”; независимо от това дали става дума за входа, или за A по време на работата, или за A в края на алгоритъма, това е **един и същи масив**.

В контраст на това, итеративната версия на Евклидовия алгоритъм от Подподсекция 2.1.3.3, INSERTION SORT от Подсекция 4.4.1 и много други алгоритми в тези лекции менят входа и при тях се налага да въвеждаме специални означения за входните данни за даден момент от изпълнението. Примерно,

- в доказателството за коректност на Евклидовия алгоритъм използваме a' и b' , за да означим съдържанието съответно на a и b в самото начало на алгоритъма,
- в доказателството за коректност на INSERTION SORT, в Лема 23 използваме означението A' за съдържанието на масива A в самото начало на едно изпълнение на външния цикъл, а в Теорема 56 използваме означението A'' за съдържанието на A в самото начало на алгоритъма.

Доказателство: Следното твърдение е инвариант за **while**-цикъла на редове 3–6.

Инвариант

Всеки път, когато изпълнението на MAX UNSORTED, ITER е на ред 3, променливата *max* съдържа стойността на максималния елемент в подмасива $A[1..i-1]$.

База. Да разгледаме първото достигане на ред 3. От една страна, подмасивът $A[1..i-1]$ е празен, защото i съдържа стойност 1 заради присвояването на ред 1, следователно $A[1..i-1]$ всъщност е $A[1..0]$, а това е празният подмасив. Максимумът на празния масив е $-\infty$, защото $-\infty$ е неутралният елемент на операцията максимум. От друга страна, променливата *max* съдържа $-\infty$ заради присвояването на ред 2. ✓

Поддръжка. Да допуснем, че твърдението е вярно при някое достигане на ред 3, което не е последното. Следните две възможности са изчерпателни по отношение на текущото i .

- $A[i]$ е по-голям от всеки елемент в $A[1..i-1]$. Съгласно индуктивното предположение, *max* съдържа стойността на максимален елемент в $A[1..i-1]$. Тогава $max < A[i]$.

От друга страна, условието на ред 4 е истина и изпълнението отива на ред 5, където *max* получава стойността на $A[i]$. И така, *max* съдържа максимума на $A[1..i]$. Тогава обаче i се инкрементира с единица (ред 6). Написано чрез новото i , твърдението става “*max* съдържа максимума на $A[1..i-1]$ ”. Виждаме, че инвариантът се запазва при следващото достигане на ред 3.

- Не е вярно, че $A[i]$ е по-голям от всеки елемент в $A[1..i-1]$. Тогава $A[i]$ е по-малък или равен на максимума на $A[1..i-1]$, който на свой ред е равен на *max* съгласно индуктивното предположение. Заключаваме, че *max* съдържа максимума на $A[1..i]$. В този случай, $max \nless A[i]$.

От друга страна, условието на ред 4 е лъжа и присвояването на ред 5 не се случва. После i се инкрементира с единица (ред 6). Написано чрез новото i , твърдението става “*max* съдържа максимума на $A[1..i-1]$ ”. Виждаме, че инвариантът се запазва при следващото достигане на ред 3.

Терминация. При последното достигане на ред 3 е вярно, че $i = n + 1$. Заместваме i с $n + 1$ в инварианта и получаваме “променливата *max* съдържа стойността на максималния елемент в подмасива $A[1 \dots (n + 1) - 1]$ ”. По-просто казано, “променливата *max* съдържа стойността на максималния елемент в подмасива $A[1 \dots n]$ ”. Но на ред 7, алгоритъмът връща именно *max*. Заклучаваме, че MAX UNSORTED, ITER(A) връща максимума на A. $\square$

2.1.3.2 Странен алгоритъм за сумиране на елементите на масив

Този алгоритъм нарочно е описан на C, а не на псевдокод.

```

1 int A[n];
2 int strangesum(int n) {
3     int i, s = 0;
4     for(i = 0; i < n; i++) {
5         if (i%2 == 0)
6             s += A[i/2];
7         else
8             s += A[n - 1 - i/2]; }
9     return s; }

```

Трябва да се докаже, че функцията `strangesum` връща сумата на елементите на масива $A[0 \dots n - 1]$. Известно е, че целочисленото деление $i/2$ връща $\lfloor \frac{i}{2} \rfloor$.

Инвариант

При всяко достигане на ред 4 е в сила

$$s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \quad (2.4)$$

База. Разглеждаме първото достигане на ред 4. От една страна, s съдържа 0 заради присвояването на ред 3. От друга страна,

- $\sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] = 0$, понеже $i = 0$, така че сумата е $\sum_{j=0}^{\lfloor \frac{0-1}{2} \rfloor} A[j]$, което е $\sum_{j=0}^{-1} A[j]$, което е 0;
- $\sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] = 0$, понеже $i = 0$, така че сумата е $\sum_{j=n-\lfloor \frac{0}{2} \rfloor}^{n-1} A[j]$, което е $\sum_{j=n}^{n-1} A[j]$, което е 0.

Виждаме, че и двете суми в (2.4) са нули, така че базата “излезе”. ✓

Поддръжка. Нека твърдението е вярно при някое достигане на ред 4, което не е последното.

Случай 1: i е четно. Условието на ред 5 е истина и изпълнението достига до ред 6.

Преди присвояването на ред 6, имаме $s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$ от предположението. След

присвояването, имаме

$$\begin{aligned}
 s &= \left(\sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \right) + A \left[\left\lfloor \frac{i}{2} \right\rfloor \right] \\
 &= \left(\sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + A \left[\left\lfloor \frac{i}{2} \right\rfloor \right] \right) + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]
 \end{aligned} \tag{2.5}$$

Тъй като i е четно, $i-1$ е нечетно. Лесно се вижда, че $\lfloor \frac{i-1}{2} \rfloor = \lfloor \frac{i}{2} \rfloor - 1$, следователно (2.5) е

$$\begin{aligned}
 &\left(\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor - 1} A[j] + A \left[\left\lfloor \frac{i}{2} \right\rfloor \right] \right) + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] = \\
 &\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]
 \end{aligned} \tag{2.6}$$

Изведохме, че след присвояването на ред 6 е изпълнено

$$s = \sum_{j=0}^{\lfloor \frac{i}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \tag{2.7}$$

Тъй като i е четно, $i+1$ е нечетно и лесно виждаме, че $\lfloor \frac{i}{2} \rfloor = \lfloor \frac{i+1}{2} \rfloor$. Имаме право да препишем (2.7) така:

$$s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j]$$

При следващото достигане на ред 4, i се инкрементира с единица. Спрямо новата стойност на i , в сила е:

$$s = \sum_{j=0}^{\lfloor \frac{i}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \tag{2.8}$$

Обаче дясната страна на (2.8) е различна от дясната страна на (2.4). Единствената разлика е в горната граница на сумата вляво: в инварианта (2.4) тя е $\lfloor \frac{i-1}{2} \rfloor$, докато в (2.8) тя е $\lfloor \frac{i}{2} \rfloor$. Поради това, инвариантът все още не е доказан формално.

Припомняме си, че преди инкрементирането, i беше четно, така че след инкрементирането, i е нечетно. Но за нечетно i е вярно, че $\lfloor \frac{i}{2} \rfloor = \lfloor \frac{i-1}{2} \rfloor$. Поради това имаме право да препишем (2.8) така:

$$s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

Доказахме, че инвариантът се запазва.

Случай 2: i е нечетно. Условието на ред 5 е лъжа и изпълнението достига до ред 8.

Преди присвояването на ред 8, имаме $s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$ от предположението. След присвояването, имаме

$$\begin{aligned} s &= \left(\sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \right) + A \left[n - 1 - \left\lfloor \frac{i}{2} \right\rfloor \right] \\ &= \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \left(\sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] + A \left[n - \left\lfloor \frac{i}{2} \right\rfloor - 1 \right] \right) \\ &= \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor - 1}^{n-1} A[j] \end{aligned} \quad (2.9)$$

За всяко i е вярно, че $n - \lfloor \frac{i}{2} \rfloor - 1 = n + \lceil -\frac{i}{2} \rceil - 1 = n + \lceil -\frac{i}{2} - 1 \rceil = n - \lfloor \frac{i}{2} + 1 \rfloor = n - \lfloor \frac{i+2}{2} \rfloor$. В случая i е нечетно. Щом i е нечетно, в сила е равенството $\lfloor \frac{i+2}{2} \rfloor = \lfloor \frac{i+1}{2} \rfloor$. Заключаваме, че в случая $n - \lfloor \frac{i}{2} \rfloor - 1 = n - \lfloor \frac{i+1}{2} \rfloor$. Следователно, можем да препишем (2.9) така:

$$\sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j] \quad (2.10)$$

Изведохме, че след придвояването на ред 8 е изпълнено

$$s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j] \quad (2.11)$$

Имаме право да препишем (2.11) така

$$s = \sum_{j=0}^{\lfloor \frac{(i+1)-2}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j]$$

При следващото достигане на ред 4, i се инкрементира с единица. Спрямо новата стойност на i , в сила е:

$$s = \sum_{j=0}^{\lfloor \frac{i-2}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \quad (2.12)$$

Припомняме си, че преди инкрементирането, i беше нечетно, така че след инкрементирането, i е четно. Но за четно i е вярно, че $\lfloor \frac{i-2}{2} \rfloor = \lfloor \frac{i-1}{2} \rfloor$. Поради това имаме право да препишем

(2.12) така:

$$s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

Виждаме, че инвариантът се запазва и в този случай.

Терминация. При последното достигане на ред 4, в сила е

$$i = n$$

$$s = \sum_{j=0}^{\lfloor \frac{i-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

Следователно,

$$s = \sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{n}{2} \rfloor}^{n-1} A[j]$$

Но $n - \lfloor \frac{n}{2} \rfloor = n + \lceil -\frac{n}{2} \rceil = \lceil n - \frac{n}{2} \rceil = \lceil \frac{n}{2} \rceil$. Следователно,

$$s = \sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} A[j] + \sum_{j=\lceil \frac{n}{2} \rceil}^{n-1} A[j] \quad (2.13)$$

Ще покажем, че $\lfloor \frac{n-1}{2} \rfloor$ и $\lceil \frac{n}{2} \rceil$ са съседни стойности, нарастващи в този ред, за всяко естествено число n .

- Първо да допуснем, че n е четно, тоест $n = 2k$ за някое естествено k . Тогава наистина

$$\left\lfloor \frac{n-1}{2} \right\rfloor = \left\lfloor \frac{2k-1}{2} \right\rfloor = k-1$$

а

$$\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{2k}{2} \right\rceil = k$$

- Сега да допуснем, че n е нечетно, тоест $n = 2k+1$ за някое естествено k . Тогава

$$\left\lfloor \frac{n-1}{2} \right\rfloor = \left\lfloor \frac{2k+1-1}{2} \right\rfloor = k$$

а

$$\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{2k+1}{2} \right\rceil = k+1$$

Щом $\lfloor \frac{n-1}{2} \rfloor$ и $\lceil \frac{n}{2} \rceil$ са съседни стойности, нарастващи в този ред, то множествата

$$\left\{0, 1, \dots, \left\lfloor \frac{n-1}{2} \right\rfloor\right\} \text{ и } \left\{\left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{n}{2} \right\rceil + 1, \dots, n-1\right\}$$

са разбиване на множеството $\{0, 1, \dots, n-1\}$. Следователно, (2.13) може да се препише просто така

$$s = \sum_{j=0}^{n-1} A[j]$$

□

2.1.3.3 Евклидовият алгоритъм в итеративен вариант

Ето отново итеративната версия на Евклидовия алгоритъм от Подсекция 1.2.2.

EUCLID ITER($a, b \in \mathbb{N}^+, a \geq b$)

```

1  while  $a \bmod b > 0$  do
2       $r \leftarrow a \bmod b$ 
3       $a \leftarrow b$ 
4       $b \leftarrow r$ 
5  return  $b$ 
```

И неговото доказателство се основава на Теорема 6.

Инвариант

Нека a' и b' означават съответно входните a и b в алгоритъм EUCLID ITER. При всяко достигане на ред 1 е вярно, че $\gcd(a, b) = \gcd(a', b')$. Освен това, $b > 0$.

База. При първото достигане на ред 1, $a' = a$ и $b' = b$, така че $\gcd(a, b) = \gcd(a', b')$ е тривиално вярно. Това, че $b > 0$ следва от факта, че $b = b'$, а b' е положително число. ✓

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 1, което не е последното. Да означим с a_1 и b_1 текущите стойности на a и b . Да означим с a_2 и b_2 стойностите на a и b при следващото достигане на ред 1. Допускането е, че $\gcd(a_1, b_1) = \gcd(a', b')$ и $b_1 > 0$.

Щом достигането на реда не е последното, то $a_1 \bmod b_1 > 0$; тоест, b_1 не дели a_1 . От Теорема 6 и факта, че $b_1 > 0$ следва, че $\gcd(a_1, b_1) = \gcd(b_1, a_1 \bmod b_1)$. Но очевидно присвояванията на редове 2, 3 и 4 имат следния ефект: $a_2 = b_1$ и $b_2 = a_1 \bmod b_1$. Покажем, че $\gcd(a_1, b_1) = \gcd(a_2, b_2)$. Щом $\gcd(a_1, b_1) = \gcd(a', b')$ (от допускането) и $\gcd(a_1, b_1) = \gcd(a_2, b_2)$, то $\gcd(a', b') = \gcd(a_2, b_2)$. ✓

Терминирание. Да разгледаме последното достигане на ред 1. Да наречем съответно $\hat{a}$ и $\hat{b}$ текущите стойности на a и b . От инварианта знаем, че $\hat{b} > 0$. Щом условието на ред 1 е лъжа, очевидно $\hat{a} \bmod \hat{b} = 0$. От Определение 9 следва, че $\hat{a} = \hat{b} \left\lfloor \frac{\hat{a}}{\hat{b}} \right\rfloor$. Тогава $\hat{b}$ дели $\hat{a}$. Освен това $\hat{a} > 0$, така че $\gcd(\hat{a}, \hat{b}) = \hat{b}$. От инварианта знаем, че $\gcd(a', b') = \gcd(\hat{a}, \hat{b})$. Веднага следва, че $\gcd(a', b') = \hat{b}$. Ерго, която алгоритъмът на ред 5 връща $\gcd(a', b')$. □

2.1.3.4 ALG 2SUM

Задачата е частен случай на Задача 21 при $k = 2$.

Изч. Задача 9: 2SUM

екземпляр: Мултимножество от реални числа S , реално число m .

въпрос: Дали има подмултимножество $\{a_1, a_2\}_M \subseteq S$, такова че $a_1 + a_2 = m$?

Има смисъл S да е **мултимножество**, а не обикновено множество. Ако S е обикновено множество, то не може да съдържа различни елементи с една и съща стойност, което би ограничило общността. На практика, S се представя с масив, който реализира именно мултимножество, понеже допуска различни елементи с една и съща стойност. Като пример: ако $m = 7$ и масивът, реализиращ S , е $[3.5, 3.5, 1]$, бихме искали това да бъде ДА-екземпляр, понеже има два различни елемента със сума 7.

Алгоритъм 2: Alg 2SUM

```

ALG 2SUM( $A[1..n]$ : масив от цели числа, такъв че  $A[1] \leq A[2] \leq \dots \leq A[n]$ ;  $m \in \mathbb{Z}$ )
1   $i \leftarrow 1, j \leftarrow n$ 
2  while  $i < j$  do
3      if  $A[i] + A[j] = m$ 
4          return 1
5      else if  $A[i] + A[j] > m$ 
6           $j--$ 
7      else
8           $i++$ 
9  return 0

```

За удобство да дефинираме *диада* като всяка двуелементна поредица[†] $\langle A[p], A[q] \rangle$ в $A[1..n]$, такава че $1 \leq p < q \leq n$ и $A[p] + A[q] = m$.

Теорема 10: Коректността на ALG 2SUM

Нека входният масив на ALG 2SUM е сортиран ненамаляващо. Алгоритъмът ALG 2SUM връща или 1, ако във входния масив има поне една диада, или 0, ако във входния масив няма нито една диада.

Доказателство: Следното твърдение е инвариант за **while**-цикъла (редове 2–8).

Инвариант

Всеки път, когато изпълнението е на ред 2, всички диади в $A[1..n]$ се намират в подмасива $A[i..j]$.

База. Да разгледаме първото достигане на ред 2. Тогава i е 1, j е n , така че твърдението става “всички диади в $A[1..n]$ се намират в подмасива $A[1..n]$ ”, което е тривиално вярно. ✓

Поддръжка. Да допуснем, че твърдението е вярно при някое достигане на ред 2, което не е последното. Щом не е последното, вярно е, че $A[i] + A[j] \neq m$, иначе следващо достигане на ред 2 няма да има, защото условието на ред 3 би било истина и алгоритъмът би терминирал през ред 4. И така, $A[i] + A[j] \neq m$. Следните случаи са взаимно изключващи се и изчерпателни.

[†]Поредица не е същото като подредица! Елементите на поредицата не са непременно съседни в масива, докато елементите на подредицата са съседни в масива.

- $A[i] + A[j] > m$. От една страна е вярно, че $A[i'] + A[j] > m$ за всяко $i' > i$; това следва веднага от факта, че масивът е сортиран и релацията $\geq$ е транзитивна. С други думи, $A[j]$ не може да е участник в диада нито с $A[i]$, нито с $A[i + 1]$, нито с $A[i + 2]$, и така нататък. Но тогава всички диади в $A[i \dots j]$ се намират в $A[i \dots j - 1]$. Тъй като по допускане всички диади в целия $A[1 \dots n]$ са в $A[i \dots j]$, следва, че всички диади в $A[1 \dots n]$ се намират в $A[i \dots j - 1]$.

От друга страна, изпълнението отива на ред 6, където j бива декрементирано. Спрямо новото j , твърдението става “всички диади в $A[1 \dots n]$ се намират в $A[i \dots j]$ ”. Виждаме, че инвариантът се запазва. ✓

- $A[i] + A[j] < m$. От една страна е вярно, че $A[i] + A[j'] < m$ за всяко $j' < j$; това следва веднага от факта, че масивът е сортиран и релацията $\leq$ е транзитивна. С други думи, $A[i]$ не може да е участник в диада нито с $A[j]$, нито с $A[j - 1]$, нито с $A[j - 2]$, и така нататък. Но тогава всички диади в $A[i \dots j]$ се намират в $A[i + 1 \dots j]$. Тъй като по допускане всички диади в целия $A[1 \dots n]$ са в $A[i \dots j]$, следва, че всички диади в $A[1 \dots n]$ се намират в $A[i + 1 \dots j]$.

От друга страна, изпълнението отива на ред 8, където i бива инкрементирано. Спрямо новото i , твърдението става “всички диади в $A[1 \dots n]$ се намират в $A[i \dots j]$ ”. Виждаме, че инвариантът се запазва. ✓

Терминация. Да разгледаме последното достигане на ред 2. Точно едно от следните две е вярно.

- $i < j$ и $A[i] + A[j] = m$. Изпълнението отива на ред 3, условието е истина и на ред 4 алгоритъмът връща 1. Доказахме верността на теоремата в този случай: масивът съдържа поне една диада, а алгоритъмът връща единица. ✓.
- $\neg(i < j)$, тоест, $i \geq j$. Но инвариантът казва, че всички диади се намират в $A[i \dots j]$. Ако $i \geq j$, подмасивът $A[i \dots j]$ има по-малко от два елемента. Тогава той не съдържа нито една диада; по дефиниция, всяка диада се състои от точно два различни елемента от масива, така че в масив с по-малко от два елемента “няма място” за диада. Изпълнението отива на ред 9, където алгоритъмът връща 0. Доказахме верността на теоремата в този случай: масивът не съдържа нито една диада, а алгоритъмът връща нула. ✓. □

2.1.3.5 ALG 3SUMCOUNT

Задачата е частен случай на Задача 22 при $k = 3$, като S е обикновено множество, а не мултимножество. Защо не позволяваме S да е мултимножество става ясно на стр. 290.

Изч. Задача 10: 3SUMCOUNT, ВЕРСИЯ ЗА ТЪРСЕНЕ

екземпляр: Множество от реални числа S .

решение: Броят на подмножествата $\{a_1, a_2, a_3\} \subseteq S$, такива че $a_1 + a_2 + a_3 = 0$.

В Задача 9 имаше число m като част от екземпляра и се искаше двете числа да се сумират до m . Тук се иска трите числа да се сумират до 0. Тази разлика е незначителна. Както се казва в Подсекция 4.2.7, вариантът, в който трите числа се сумират до m , е практически същият, понеже, ако извадим $m/3$ от всяко число на S , да има три числа със сума m в оригиналния S е същото като да има три числа със сума 0 в модифицирания S .

За практическото значение на задачата 3SUM вижте Подсекция 4.2.7. Тук само ще разгледаме алгоритъм за нея, наречен ALG 3SUMCOUNT, и ще докажем коректността му.

ALG 3SUMCOUNT($A[1..n]$: сортиран масив от две по две различни цели числа; $n \geq 3$)

```

1  c ← 0
2  for i ← 1 to n - 2
3      p ← i + 1, q ← n
4      d ← 0
5      while p < q do
6          if A[i] + A[p] + A[q] = 0
7              d++
8              p++, q--
9          else if A[i] + A[p] + A[q] > 0
10             q--
11         else
12             p++
13     c ← c + d
14 return c

```

За удобство да дефинираме *триада* като всяка триелементна поредица $\langle A[p], A[q], A[r] \rangle$ на $A[1..n]$, такава че $1 \leq p < q < r \leq n$ и $A[p] + A[q] + A[r] = 0$.

Теорема 11: Коректността на ALG 3SUMCOUNT

Нека $n \geq 3$. За всеки числен масив $A[1..n]$, който е сортиран и няма повтарящи се елементи, ALG 3SUMCOUNT($A[1..n]$) връща броя на триадите в $A[1..n]$.

Доказателство: ALG 3SUMCOUNT е първият итеративен алгоритъм с вложеност на цикли измежду алгоритмите, които верифицираме. Наблюдателният читател(ка) сигурно е забелязал(а), че може да се каже, че ALG 3SUMCOUNT за всяко i пуска вариант на ALG 2SUM, който записва в променливата d броя на диадите в $A[i+1..n]$, сумиращи се до $-A[i]$, а променливата c акумулира тези бройки на ред 13. Но верификацията на ALG 3SUMCOUNT ще бъде направена формално и от първи принципи – все едно, че не знаем за ALG 2SUM.

Доказателството ще бъде съобразено с факта, че алгоритъмът има **for**-цикъл (редове 2–13), който на свой ред съдържа **while**-цикъл (редове 5–12). Инвариантът, от който следва директно Теорема 11, е за външния цикъл. Това е смислено: разгледан на най-високо ниво, алгоритъмът се състои от инициализация $c \leftarrow 0$, следвана от **for**-цикъла, следван от връщане на c . Ако успеем да докажем смислен и полезен предикат с променлива i за всяко достигане на ред 2, от този предикат ще получим верността на теоремата, замествайки i с $n+1$. Този предикат няма нужда да казва каквото и да е за **while**-цикъла! Достатъчно е предикатът да казва нещо за броя на триадите и елемента i . Когато доказваме този предикат, ние трябва да отчетем ефекта от изпълнението на вътрешния цикъл, но това касае само **доказателството** на предиката, а не неговата **формулировка**. Ето една възможност за въпросния инвариант.

Инвариант 1: Верификацията на ALG 3SUMCOUNT

Всеки път, когато изпълнението е на ред 2, променливата c съдържа броя на триадите, чийто минимален елемент се намира в подмасива $A[1..i-1]$.

Това е напълно смислен предикат: тъй като всяка триада се състои от различни числа, тя има точно един минимален елемент и множеството от триадите може да бъде разбито по позицията на минималния елемент. Предикатът не казва нищо за вътрешния цикъл – това не е необходимо засега. Ще се замислим за вътрешния цикъл, когато се наложи.

База: При първото достигане на ред 2, c съдържа 0 заради ред 1, а i съдържа 1 заради неясното присвояване на ред 2. От една страна, броят на триадите, чийто минимален елемент е в подмасива $A[1 \dots i - 1]$, е нула, понеже този подмасив е $A[1 \dots 0]$, тоест, празен. От друга страна, $c = 0$. ✓

Поддръжка: Да допуснем, че твърдението е вярно за някое достигане на ред 2, което не е последното. Ще докажем, че инвариантът се запазва след изпълнението на тялото на **for**-цикъла. На ред 3, p се инициализира с $i + 1$, а q се инициализира с n . На ред 4, d се инициализира с 0. Следва изпълнението на **while**-цикъла.

Предишните итеративни алгоритми, които верифицирахме, са с единствен цикъл. При някои, например FIND SUM на стр. 46, преминаването през тялото на цикъла е линейно в смисъл, че първо се изпълнява първата инструкция, после втората инструкция, и така нататък, докато тялото бъде изпълнено докрай. При други като MAX UNSORTED, ITER на стр. 49 преминаването е по-сложно, защото тялото съдържа **if**, така че изпълнението може да мине по два различни пътя в зависимост от верността на булевото условие на **if**-а. Това прави доказателството по-дълго, но то остава с **фиксирана дължина**.

Сегашния алгоритъм ALG3 SUMCOUNT е радикално различен заради вложения цикъл. Ние не можем да направим доказателство с фиксирана дължина за поддръжката на инварианта на **for**-а чрез “разгъване” на изпълнението на **while**-а, защото **не знаем точно колко пъти ще се изпълни while-ът**. Налага да осмислим работата на **while**-а и да разберем какъв ефект има изпълнението му. Става дума не за едно “завъртане” на **while**-а, а за цялото му изпълнение в рамките на едно изпълнение на **for**-а. Налага се да формулираме нов инвариант, който касае само **while**-а, и да докажем този нов инвариант. Този нов инвариант е напълно различен от инварианта за **for**-цикъла! Когато докажем инварианта за **while**-а, във фазата **Терминация** ще получим твърдение, което описва ефекта от работата на **while**-а. Разполагайки с това твърдение, можем да се върнем към фазата **Поддръжка** от доказателството на инварианта за **for**-цикъла.

Ето едно помощно твърдение за ефекта от работата на вътрешния цикъл. То е по отношение на **едно единствено** изпълнение на външния цикъл. Това е много важно! В крайна сметка, ние искаме да докажем фазата **Поддръжка** за работата на външния цикъл, а за тази цел има смисъл да разгледаме всички изпълнения на вътрешния цикъл от **едно единствено** изпълнение на външния цикъл.

Лема 4: За ефекта от изпълнението на вътр. цикъл на ALG 3SUMCOUNT

Когато изпълнението достигне ред 13, променливата d съдържа броя на триадите, чийто минимален елемент е $A[i]$.

Доказателство: Ще докажем Лема 4 чрез следния инвариант.

Инвариант

В рамките на едно изпълнение на **for**-цикъла (редове 2–13), всеки път, когато изпълнението е на ред 5:

- ① за всяка триада $\langle A[i], A[j], A[k] \rangle$ в $A[1 \dots n]$ е вярно, че $j \in \{i + 1, \dots, p - 1\}$ тстк $k \in \{q + 1, \dots, n\}$.
- ② d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$ в $A[1 \dots n]$, за които е вярно, че $j \in \{i + 1, \dots, p - 1\}$ и $k \in \{q + 1, \dots, n\}$.

Доказателство: Доказателството ползва следните очевидни твърдения. Нека $A[1 \dots n]$ е сортиран, без повтарящи се елементи, и нека $1 \leq i < j < k \leq n$.

Факт1 Ако $A[i] + A[j] + A[k] \leq 0$, то $A[i] + A[j] + A[k'] < 0$ за всяко $k' < k$.

Факт2 Ако $A[i] + A[j] + A[k] \geq 0$, то $A[i] + A[j'] + A[k] > 0$ за всяко $j' > j$.

Факт3 Ако $\langle A[i], A[j], A[k] \rangle$ е триада, то $\langle A[i], A[j], A[k'] \rangle$ не е триада за никое $k' < k$ и $\langle A[i], A[j'], A[k] \rangle$ не е триада за никое $j' > j$.

Щом числата във входа са уникални, **Факт3** е очевидно следствие на **Факт1** и **Факт2**.

База: $p = i + 1$ и $q = n$ заради присвояванията на ред 3. Тогава множествата $\{i + 1, \dots, p - 1\}$ и $\{q + 1, \dots, n\}$ са съответно $\{i + 1, \dots, i\}$ и $\{n + 1, \dots, n\}$, така че и двете са празни. Тогава ① е вярно в празния смисъл.

Да видим ②. От една страна, $d = 0$ заради присвояването на ред 4. От друга страна, множествата $\{i + 1, \dots, i\}$ и $\{n + 1, \dots, n\}$ са празни, така че твърдението става “ d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \emptyset$ и $k \in \emptyset$.” Но има точно нула триади, в които средният елемент е от празното множество (алтернативно, в които максималният елемент е от празното множество). Виждаме, че и ② е изпълнено. ✓

Поддръжка: Нека ① и ② са в сила за някое достигане на ред 5, което не е последното. Да кажем, че в това е моментът t от дискретното време на алгоритъма. Следните три възможности са взаимно изключващи се и са изчерпателни:

- $\langle A[i], A[p], A[q] \rangle$ е триада. Първо да се убедим, че ① се запазва. От **Факт3** знаем, че $\langle A[i], A[p], A[q] \rangle$ е единствената триада, съдържаща $A[i]$ и $A[p]$, и е единствената триада, съдържаща $A[i]$ и $A[q]$. От това и от индуктивното предположение следва, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, p\}$ тстк $k \in \{q, \dots, n\}$. След инкрементирането на p и декрементирането на q на ред 8, по отношение на новите стойности на p и q , е вярно, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, p - 1\}$ тстк $k \in \{q - 1, \dots, n\}$. Наистина, ① се запазва. ✓

Сега да се убедим, че ② се запазва. Пак разглеждаме момента t . Щом $\langle A[i], A[p], A[q] \rangle$ е единствената триада, съдържаща $A[i]$ и $A[p]$, и е единствената триада, съдържаща $A[i]$ и $A[q]$, то броят на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i + 1, \dots, p\}$ и $k \in \{q, \dots, n\}$, е с единица по-голям от броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i + 1, \dots, p - 1\}$ и $k \in \{q + 1, \dots, n\}$. От ② знаем, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i + 1, \dots, p - 1\}$ и $k \in \{q + 1, \dots, n\}$. Тогава броят на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i + 1, \dots, p\}$ и $k \in \{q, \dots, n\}$, е $d + 1$. След инкрементирането на d на ред 7 е вярно, че d е броят на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i + 1, \dots, p\}$ и $k \in \{q, \dots, n\}$, е $d + 1$. След инкрементирането на p и декрементирането на q на ред 8, по отношение

на новите стойности на p и q , е вярно, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i+1, \dots, p-1\}$ и $k \in \{q+1, \dots, n\}$. Виждаме, че ② се запазва. ✓

- $A[i] + A[p] + A[q] > 0$. Да се убедим, че ① се запазва. От **Факт2** знаем, че $A[i]$ и $A[q]$ не образуват триада нито с $A[p+1]$, нито с $A[p+2]$, и така нататък, нито с $A[q-1]$. Заедно с индуктивното предположение ①, това влече, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i+1, \dots, p-1\}$ тстк $k \in \{q, \dots, n\}$. От друга страна, условието на ред 9 е истина и на ред 10, q бива декрементирано. Спрямо новата стойност на q е вярно, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i+1, \dots, p-1\}$ тстк $k \in \{q+1, \dots, n\}$. Виждаме, че ① се запазва. ✓

Да се убедим, че ② се запазва. Пак разглеждаме момента t . Припомняме си, че $A[i]$ и $A[q]$ не образуват триада нито с $A[p+1]$, нито с $A[p+2]$, и така нататък, нито с $A[q-1]$. Заедно с индуктивното предположение ②, това влече, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i+1, \dots, p-1\}$ и $k \in \{q, \dots, n\}$. На ред 10, q бива декрементирано. Спрямо новата стойност на q е вярно, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i+1, \dots, p-1\}$ и $k \in \{q+1, \dots, n\}$. Виждаме, че ② се запазва. ✓

- $A[i] + A[p] + A[q] < 0$. Да се убедим, че ① се запазва. От **Факт1** знаем, че $A[i]$ и $A[p]$ не образуват триада нито с $A[q-1]$, нито с $A[q-2]$, и така нататък, нито с $A[p+1]$. Заедно с индуктивното предположение ①, това влече, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i+1, \dots, p\}$ тстк $k \in \{q+1, \dots, n\}$. От друга страна, условието на ред 9 е лъжа и на ред 12, p бива инкрементирано. Спрямо новата стойност на p е вярно, че за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i+1, \dots, p-1\}$ тстк $k \in \{q+1, \dots, n\}$. Виждаме, че ① се запазва. ✓

Да се убедим, че ② се запазва. Пак разглеждаме момента t . Припомняме си, че $A[i]$ и $A[p]$ не образуват триада нито с $A[q-1]$, нито с $A[q-2]$, и така нататък, нито с $A[p+1]$. Заедно с индуктивното предположение ②, това влече, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i+1, \dots, p\}$ и $k \in \{q+1, \dots, n\}$. На ред 12, p бива инкрементирано. Спрямо новата стойност на p е вярно, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i+1, \dots, p-1\}$ и $k \in \{q+1, \dots, n\}$. Виждаме, че ② се запазва. ✓

Терминация: Изпълнението е на ред 5 за последен път. Нека $\tilde{p}$ и $\tilde{q}$ са стойностите съответно на p и q в този момент. Шом е на ред 5 за последен път, вярно е, че $\neg(\tilde{p} < \tilde{q})$, което е същото като $\tilde{p} \geq \tilde{q}$. Сега ще се убедим, че има точно две възможности за големините на $\tilde{p}$ и $\tilde{q}$:

Възможност1: $\tilde{q} = \tilde{p}$.

Възможност2: $\tilde{q} = \tilde{p} - 1$.

Нека $\hat{p}$ и $\hat{q}$ са съответно стойностите на променливите p и q при предното достигане на ред 5. Сегашните $\tilde{p}$ и $\tilde{q}$ са получени от $\hat{p}$ и $\hat{q}$ по точно един от следните три начина.

- Последното изпълнение на тялото на **while**-цикъла е достигнало ред 8. Тогава $\tilde{p} = \hat{p} + 1$ и $\tilde{q} = \hat{q} - 1$. Това означава, че при последното минаване през тялото на **while**-цикъла е била открита триада $\langle A[i], A[\hat{p}], A[\hat{q}] \rangle$ и двата индекса p и q са “мръднали”: p нагоре и q надолу. Следните два случая са изчерпателни.

- ♦ $\hat{p} = \hat{q} - 1$. Тогава $\tilde{q} = \tilde{p} - 1$. Това е **Възможност2**.

♦ $\hat{p} = \hat{q} - 2$. Тогава $\tilde{q} = \tilde{p}$. Това е **Възможност1**.

- Последното изпълнение на тялото на **while**-цикъла е достигнало ред 10. Тогава $\tilde{p} = \hat{p}$ и $\tilde{q} = \hat{q} - 1$. Това означава, че при последното минаване през тялото на **while**-цикъла не е била открита триада и само q е “мръднал”, а p е “останал на място”. Тогава $\tilde{q} = \tilde{p}$. Това е **Възможност1**.
- Последното изпълнение на тялото на **while**-цикъла е достигнало ред 12. Тогава $\tilde{p} = \hat{p} + 1$ и $\tilde{q} = \hat{q}$. Това означава, че при последното минаване през тялото на **while**-цикъла не е била открита триада и само p е “мръднал”, а q е “останал на място”. Тогава $\tilde{q} = \tilde{p}$. Това е **Възможност1**.

Да разгледаме **Възможност1**. За по-голяма яснота, нека стойността $\tilde{p} = \tilde{q}$ бъде наречена r . Да заместим p , и q в инварианта с r . Получаваме

- ① за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, r - 1\}$ тстк $k \in \{r + 1, \dots, n\}$.
- ② d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i + 1, \dots, r - 1\}$ и $k \in \{r + 1, \dots, n\}$.

Потенциален проблем за докателството е това, че множествата $\{i + 1, \dots, r - 1\}$ и $\{r + 1, \dots, n\}$ не са разбиване на $\{i + 1, \dots, n\}$, защото $A[r]$ остава извън тях. А в крайна сметка ние искаме да покажем, че променливата d съдържа броя на **всички** триади с минимален елемент $A[i]$ (Лема 4). Сега ще се убедим, че проблем няма, понеже не може да има триади с минимален елемент $A[i]$, съдържащи $A[r]$. Да допуснем противното: поне една триада с минимален елемент $A[i]$ съдържа $A[r]$. Следните две възможности са изчерпателни.

- $A[r]$ е максималният елемент на тази триада. Тогава тя е от вида $\langle A[i], A[y], A[r] \rangle$, като $y \in \{i + 1, \dots, r - 1\}$. Но това директно противоречи на ①, което казва “за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, r - 1\}$ тстк $k \in \{r + 1, \dots, n\}$ ”: щом средният елемент $A[y]$ е от $A[i + 1 \dots r - 1]$, няма как максималният елемент да не е от $A[r + 1 \dots n]$.
- $A[r]$ е средният елемент на тази триада. Тогава тя е от вида $\langle A[i], A[r], A[z] \rangle$, като $z \in \{r + 1, \dots, n\}$. Но това директно противоречи на ①, което казва “за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, r - 1\}$ тстк $k \in \{r + 1, \dots, n\}$ ”: щом максималният елемент $A[z]$ е от $A[r + 1 \dots n]$, няма как средният елемент да не е от $A[i + 1 \dots r]$.

И така, $A[r]$ не може да участва в нито една триада с минимален елемент $A[i]$. Тогава ① и ② казват, че d съдържа точно броя на триадите с минимален елемент $A[i]$.

Да разгледаме **Възможност2**. За по-голяма яснота, нека стойността $\tilde{p}$ бъде наречена r . Тогава $\tilde{q} = r - 1$. Да заместим p с r и q с $r - 1$ в инварианта. Получаваме

- ① за всяка триада $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i + 1, \dots, r - 1\}$ тстк $k \in \{r, \dots, n\}$.
- ② d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, за които е вярно, че $j \in \{i + 1, \dots, r - 1\}$ и $k \in \{r, \dots, n\}$.

Тъй като множествата $\{i + 1, \dots, r - 1\}$ и $\{r, \dots, n\}$ са разбиване на $\{i + 1, \dots, n\}$, инвариантът казва, че d съдържа точно броя на триадите с минимален елемент $A[i]$. ✓

С това доказателството на Лема 4 приключва. □

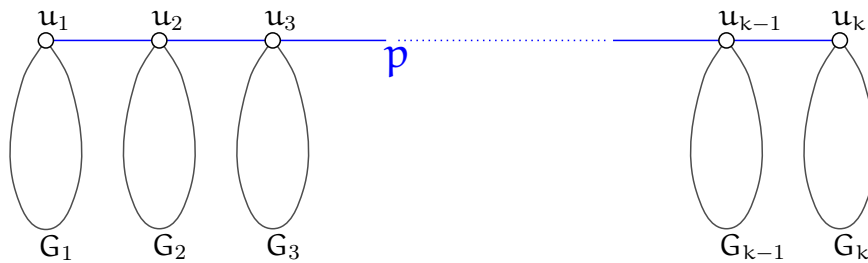
Връщаме се към фазата **Поддръжка** от доказателството на Инвариант 1. Съгласно Лема 4, на ред 13, d съдържа броя на триадите, чиито минимален елемент е $A[i]$.

Тогава след присвояването на ред 13, s вече съдържа броя на триадите, чийто минимален елемент се намира в подмасива $A[1 \dots i]$. При следващото достигане на ред 2, променливата i се инкрементира. По отношение на новото i е вярно, че s съдържа броя на триадите, чийто минимален елемент се намира в подмасива $A[1 \dots i - 1]$. Виждаме, че Инвариант 1 се запазва. ✓

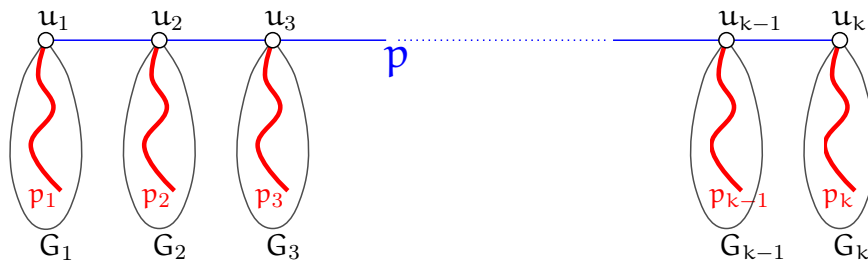
Терминация: Да разгледаме последното достигане на ред 2. Тогава $i = n - 1$. Получаваме, че s съдържа броя на триадите, чийто минимален елемент се намира в $A[1 \dots n - 2]$. Но това са всички триади! Други триади не може да има, защото всяка триада е от три различни елемента, ерго, най-малкият от тях не може да е на позиция, по-голяма от $n - 2$. Тогава на ред 14 алгоритъмът ALG 3SUMCOUNT връща броя на всички триади в $A[1 \dots n]$. □

2.1.3.6 ALG MAX SUM PLUS DIST

Задачата, която ще разгледаме, има графова интерпретация. Да си представим свързан граф G , в който има път $p = u_1, u_2, \dots, u_k$, такъв че всяко ребро от p е мост (Определение 81). Това означава, че ако изтрием от G ребрата на p , то G ще се разпадне на k свързани компоненти $G_1, \dots, G_k$, като $u_i \in G_i$ за $i \in \{1, \dots, k\}$. Ето рисунка на G , като ребрата на p са в синьо.



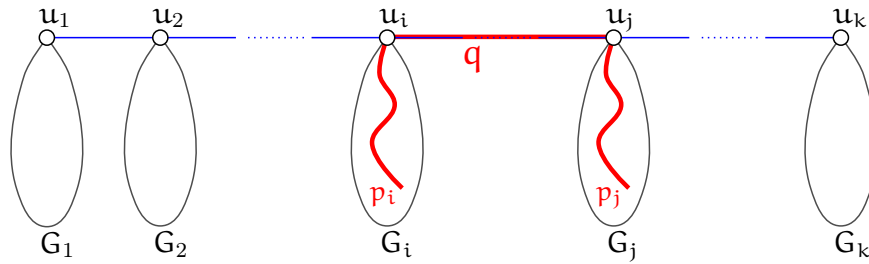
Да допуснем, че за $i \in \{1, \dots, k\}$, p_i е най-дълъг път в G_i с един край u_i . Очевидно $|p_i| \geq 0$, като границата е точна, понеже G_i може да е тривиален.



Дадени са дължините $|p_i|$, за $i \in \{1, \dots, k\}$. Трябва да се намери дължината на най-дълъг път q в G , който съдържа поне едно ребро от p . Очевидно q се състои от “сцепването” на тези три пътя:

- p_i ,
- подпътя на p между u_i и u_j ,
- и p_j ,

за някакви i и j , такива че $1 \leq i < j \leq k$.



Очевидно $|q| = |p_i| + j - i + |p_j|$.

Ако игнорираме графовата интерпретация, задачата е тази.

Изн. Задача 11: MAX SUM PLUS DIST

екземпляр: Масив $A[1..n]$ от естествени числа, като $n \geq 2$.

решение: $\max \{A[s] + A[t] + t - s \mid 1 \leq s < t \leq n\}$.

Наивният подход е да се опитат всички ненаредени двойки различни елементи. Тук ще направим нещо по-умно.

Алгоритъм 3: Alg Max Sum Plus Dist

ALG MAX SUM PLUS DIST($A[1..n]$: масив от естествени числа, като $n \geq 2$)

```

1  създай масив  $B[1..n]$  от естествени числа
2   $B[1] \leftarrow A[1]$ 
3   $opt \leftarrow -\infty$ 
4   $i \leftarrow 2$ 
5  do
6     $B[i] \leftarrow \max(B[i-1], A[i] - (i-1))$ 
7     $opt \leftarrow \max(opt, B[i-1] + A[i] + (i-1))$ 
8     $i++$ 
9  while  $i \leq n$ 
10 return  $opt$ 
```

Да направим на ръка три итерации на цикъла на ALG MAX SUM PLUS DIST, за да придобием интуиция за това какво става. $B[1] = A[1]$ поначало, opt се инициализира с $-\infty$, i се инициализира с 2 и влизаме в цикъла.

първа итерация, $i = 2$:

$$\begin{aligned}
 B[2] &= \max \{B[1], A[2] - (2 - 1)\} \\
 &= \max \{A[1], A[2] - 1\} \\
 opt &= \max \{-\infty, A[1] + A[2] + (2 - 1)\} \\
 &= A[1] + A[2] + 1
 \end{aligned}$$

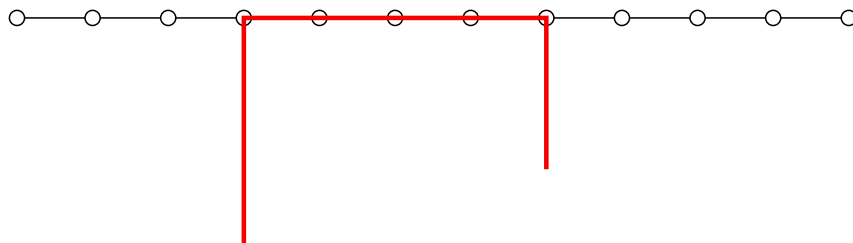
втора итерация, $i = 3$: $B[3] = \max \{B[2], A[3] - (3 - 1)\}$, така че

$$\begin{aligned} B[3] &= \max \{ \max \{A[1], A[2] - 1\}, A[3] - 2 \} \\ &= \max \{A[1], A[2] - 1, A[3] - 2\} \\ \text{opt} &= \max \{A[1] + A[2] + 1, \max \{A[1], A[2] - 1\} + A[3] + (3 - 1)\} \\ &= \max \{A[1] + A[2] + 1, \max \{A[1] + A[3] + 2, A[2] - 1 + A[3] + 2\}\} \\ &= \max \{A[1] + A[2] + 1, A[2] + A[3] + 1, A[1] + A[3] + 2\} \end{aligned}$$

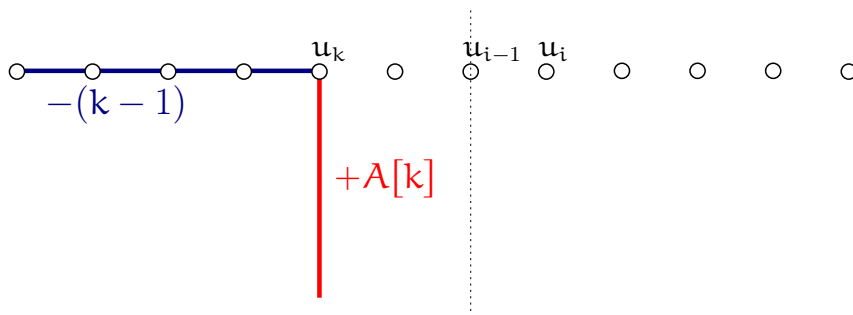
трета итерация, $i = 4$: $B[4] = \max \{B[3], A[4] - (4 - 1)\}$, така че

$$\begin{aligned} B[4] &= \max \{ \max \{A[1], A[2] - 1, A[3] - 2\}, A[4] - (4 - 1) \} \\ &= \max \{A[1], A[2] - 1, A[3] - 2, A[4] - 3\} \\ \text{opt} &= \max \{ \max \{A[1] + A[2] + 1, A[2] + A[3] + 1, A[1] + A[3] + 2\}, \\ &\quad \max \{A[1], A[2] - 1, A[3] - 2\} + A[4] + (4 - 1) \} \\ &= \max \{ \max \{A[1] + A[2] + 1, A[2] + A[3] + 1, A[1] + A[3] + 2\}, \\ &\quad \max \{A[1] + A[4] + 3, A[2] + A[4] + 2, A[3] + A[4] + 1\} \} \\ &= \max \{ A[1] + A[2] + 1, A[2] + A[3] + 1, A[3] + A[4] + 1, \\ &\quad A[1] + A[3] + 2, A[2] + A[4] + 2, A[1] + A[4] + 3 \} \end{aligned}$$

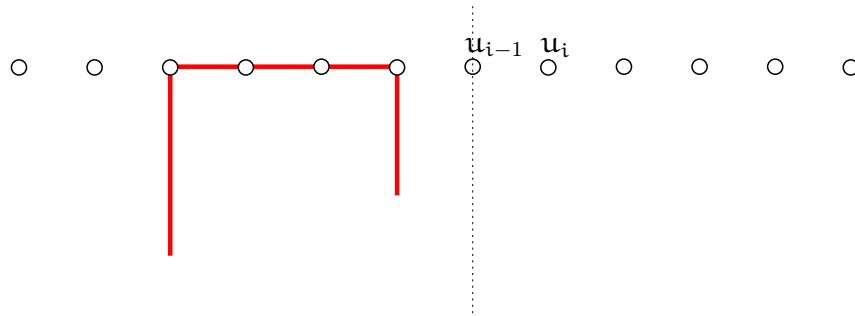
Може да направим следната неформална, картинна аргументация за коректността на алгоритъма, представяйки си графа и масива едновременно. Търсим “скоба” с максимална дължина в целия $A[1 \dots n]$:



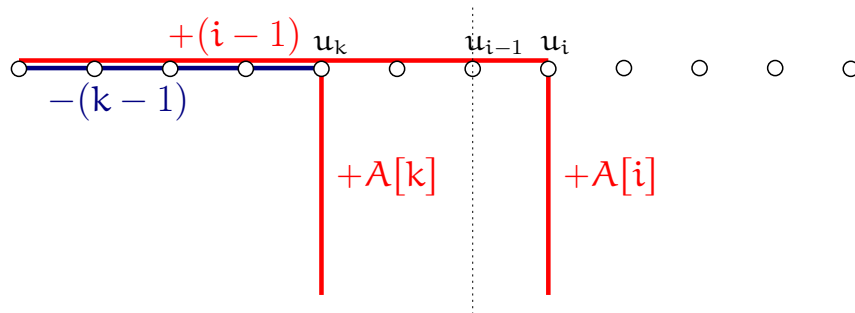
За дадено i , B_{i-1} е максималната стойност на $A[k] - (k - 1)$, по всички $k \in \{1, \dots, i - 1\}$. Можем да си представим B_{i-1} като състоящ се от път с положителна дължина $|p_k| = A[k]$ и още един път с отрицателна дължина $-(k - 1)$:



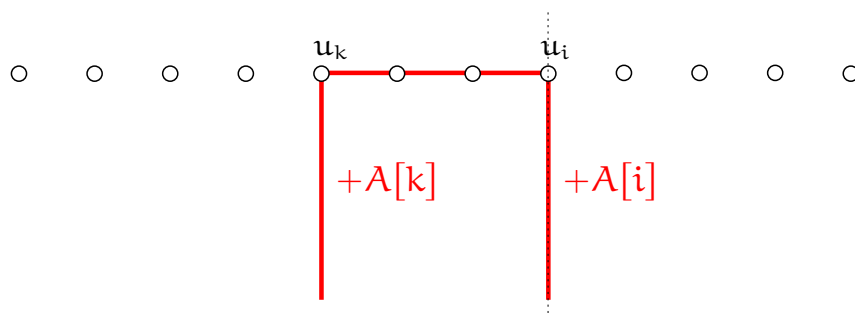
Пак за това i , стойността на opt преди присвояването на ред 7 е максималната дължина на “скоба” в $A[1 \dots i - 1]$:



Пак за това i , сумата $B[i-1] + A[i] + (i-1)$ в присвояването на ред 7 може да си представим така:



Събирайки $+(i-1)$ с $-(k-1)$, можем да опростим до:



Но това е “скоба” с максимална дължина в $A[1..i]$, едното “рамо” на която е $A[i]$. Заклучаваме, че след присвояването на ред 7, стойността на opt е максимум на тези две: максимална дължина на “скоба” в $A[1..i-1]$ и максимална дължина на “скоба” в $A[1..i]$, едното “рамо” на която е $A[i]$. Този максимум е максимална дължина на “скоба” в $A[1..i]$.

И така, за всяко i , присвояването на ред 7 дава на opt стойност, равна на максимална дължина на “скоба” в $A[1..i]$. Ерго, в края на алгоритъма наистина opt съдържа максимална дължина на “скоба” в $A[1..n]$.

Сега ще докажем формално коректността на алгоритъма.

Теорема 12: Коректността на алгоритъма ALG MAX SUM PLUS DIST

ALG MAX SUM PLUS DIST($A[1..n]$) връща $\max \{A[s] + A[t] + t - s \mid 1 \leq s < t \leq n\}$.

Доказателство: Следното твърдение е инвариант за **do-while** цикъла (редове 5–9). Тъй като цикълът е с постусловие, има смисъл да “закачим” предиката за ред 9, тъй като там е управлението на цикъла.

Инвариант

При всяко достигане на ред 9:

- ① $B[i-1]$ съдържа $\max \{A[k] - (k-1) \mid 1 \leq k \leq i-1\}$,
- ② opt съдържа $\max \{A[s] + A[t] + (t-s) \mid 1 \leq s < t \leq i-1\}$

База. При първото достигане на ред 9, i съдържа 3 заради присвояването на ред 4 и инкрементацията на ред 8, така че $B[i-1]$ е всъщност $B[2]$.

От една страна, ако четем псевдокода, $B[2]$ съдържа $\max \{A[1], A[2] - 1\}$ заради присвояванията на ред 2 и ред 6; забележете, че при първото достигане на ред 6, i съдържа 2. От друга страна, по отношение на текущото i , което е 3, множеството $\{A[k] - (k-1) \mid 1 \leq k \leq i-1\}$, за което говори инварианта, е $\{A[1] - 0, A[2] - 1\}$. Заклучаваме, че ① е истина. ✓

От една страна, ако четем псевдокода, opt съдържа $A[1] + A[2] + 1$ заради присвояването на ред 7; в този момент $i = 2$. От друга страна, по отношение на текущото (при достигането на ред 9) $i = 3$, множеството $\{A[s] + A[t] + (t-s) \mid 1 \leq s < t \leq i-1\}$, за което говори инварианта, всъщност е $\{A[1] + A[2] + (2-1)\}$. Тогава ② също е истина. ✓

Поддръжка: Нека инвариантът е истина за някое достигане на ред 9, което не е последното. Ще се убедим, че той се запазва при изпълнението на тялото на цикъла.

На ред 6, $B[i]$ получава стойност $\max \{B[i-1], A[i] - (i-1)\}$. По допускане,

$$B[i-1] = \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i-1] - (i-2)\}$$

Тогава

$$B[i] = \max \{ \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i-1] - (i-2)\}, A[i] - (i-1) \}$$

което може да запишем накратко като

$$B[i] = \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i] - (i-1)\}$$

След инкрементирането на i на ред 8, изразено чрез новото i , това става

$$B[i-1] = \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i-1] - (i-2)\}$$

Виждаме, че ① отново е истина.

На ред 7, opt получава стойност $\max(opt, B[i-1] + A[i] + (i-1))$. По допускане, старата стойност на opt е

$$opt_{old} = \max \{A[s] + A[t] + (t-s) \mid 1 \leq s < t \leq i-1\}$$

а $B[i-1]$ е

$$B[i-1] = \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i-1] - (i-2)\}$$

Тогава $B[i-1] + A[i] + (i-1)$ е

$$\begin{aligned} & \max \{A[1], A[2] - 1, A[3] - 2, \dots, A[i-1] - (i-2)\} + A[i] + (i-1) = \\ & \max \{A[1] + A[i] + (i-1), A[2] - 1 + A[i] + (i-1), \dots, A[i-1] - (i-2) + A[i] + (i-1)\} = \\ & \max \{A[k] + A[i] + (i-k) \mid 1 \leq k \leq i-1\} \end{aligned}$$

И така:

- opt_{old} съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1 \dots i-1]$,
- а $B[i-1] + A[i] + (i-1)$ съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1 \dots i]$, единият от които обаче е $A[i]$.

Тогава $\max\{\text{opt}_{\text{old}}, B[i-1] + A[i] + (i-1)\}$, което новото t , съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1 \dots i]$. След инкрементирането на ред 8, изразено в новото i , това става

$$\text{opt} = \max\{A[s] + A[t] + (t - s) \mid 1 \leq s < t \leq i - 1\}$$

Виждаме, че ② отново е истина.

Терминация: Разглеждаме момента, в който изпълнението е на ред 9 за последен път. Очевидно i съдържа $n + 1$ в този момент. От ② получаваме, замествайки i с $n + 1$, че opt съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1 \dots n]$. И на ред 10 алгоритъмът връща точно това. $\square$

2.1.3.7 Намиране на най-малкото положително число, което липсва

Даден е масив от цели числа $A[1 \dots n]$. Задачата е да намери най-малкото положително число, което не е в A .

Наивният алгоритъм. Очевидният подход е да се сортират числата и след това с едно сканиране отляво надясно да се разбере кое е най-малкото положително липсващо число. Примерно, ето така.

SMALLEST POSITIVE MISSING, NAIVE($A[1 \dots n]$)

```

1  SORT(A)
2  for i ← 1 to n
3      if A[i] > 0
4          break
5  if i = n + 1
6      return 1
7  else
8      k ← 1
9      while i ≤ n and A[i] = k do
10         k++, i++
11  return k
```

Ето аргумент за коректността. Ефектът от първия цикъл (редове 2–4) е, че при достигането на ред 5, $A[1 \dots i-1]$ се състои точно от неположителните елементи на входния масив, а променливата i е или $n + 1$, или от $\{1, \dots, n\}$.

- Ако $i = n + 1$, положителни числа във входния масив няма, така че най-малкото липсващо число е най-малкото положително число, което е 1. На ред 6 алгоритъмът връща точно 1. ✓

- В противен случай, $i \in \{1, \dots, n\}$. Тогава i е валиден индекс в A и освен това $A[i]$ е най-малкото положително число в A , понеже A е сортиран на ред 1. Изпълнението на алгоритъма отива на ред 8, където на k се присвоява 1. Изпълнението на алгоритъма отива на ред 9.

Нека i' е стойността на i при първото достигането на ред 9. Следното твърдение е инвариант за втория цикъл (редове 9–10).

Инвариант

Всеки път, когато изпълнението на SMALLEST POSITIVE MISSING, NAIVE е на ред 9,

❶ $i = i' + k - 1$ и

❷ $A[i' .. i - 1] = [1, 2, \dots, k - 1]$.

База: При първото достигане на ред 9 имаме $i' = i$ и $k = 1$, така че равенството $i = i' + k - 1$ всъщност е $i' = i' + 1 - 1$, което със сигурност е вярно. Ерго, ❶ е в сила. ✓

Да видим ❷. $A[i' .. i - 1]$ е $A[i' .. i' - 1]$, който е празният подмасив. От друга страна, $[1, 2, \dots, k - 1]$ е $[1, \dots, 0]$, който също е празен. И ❷ е в сила. ✓

Подпр.: Нека ❶ и ❷ са в сила за някое достигане на ред 9, което не е последното. Щом не е последното, трябва да е вярно, че $A[i] = k$ и $i \leq n$. От $A[i] = k$ и предположението $A[i' .. i - 1] = [1, 2, \dots, k - 1]$ следва, че $A[i' .. i] = [1, 2, \dots, k]$.

На ред 10 променливите i и k се инкрементират. В термините на новите i и k отново е вярно, че подмасивът $A[i' .. i - 1] = [1, 2, \dots, k - 1]$. Заключаваме, че при следващото достигане на ред 9, ❷ е изпълнено. ✓

❶ също остава вярно, понеже, щом поначало $i = i' + k - 1$ и после променливите i и k се инкрементират, равенството се запазва. ✓

Доказахме инварианта. Сега сме във фазата **Терминация** на доказателството. Разглеждаме последното достигане на ред 9. При последното достигане на ред 9 имаме $\neg(i \leq n \wedge A[i] = k)$, което е еквивалентно на $i > n \vee A[i] \neq k$.

Първо да допуснем, че $i > n$. Тъй като i се инкрементира с единица на ред 9, всъщност $i = n + 1$. Заместваме в ❷ i с $n + 1$ и получаваме " $A[i' .. n] = [1, 2, \dots, k - 1]$ ". Имайки предвид, че масивът е сортиран и елементите на $A[1 .. i' - 1]$ са неположителните числа на входа, заключаваме, че най-малкото положително число, което не се съдържа във входния масив, е k . Но на ред 11 алгоритъмът връща k . ✓

Сега да допуснем, че $i \leq n$, но $A[i] \neq k$. От ❷ знаем, че $A[i' .. i - 1] = [1, 2, \dots, k - 1]$. Имайки предвид, че масивът е сортиран и $A[1 .. i' - 1]$ са неположителните числа на входа, очевидно е, че $A[k] > k$. Ерго, най-малкото положително число, което не се съдържа във входния масив, е k . Но на ред 11 алгоритъмът връща k . ✓ □

Алгоритъмът SMALLEST POSITIVE MISSING, NAIVE, чиято коректност току-що доказахме, е неефикасен. Той има сложност по време (Определение 18) $\Omega(n \lg n)$ (Определение 22) заради сортирането на ред 1 (Следствие 32).

Ефикасният алгоритъм. Най-малкото положително число, което липсва, може да се намери ефикасно ето така.

```

SMALLEST POSITIVE MISSING(A[1 .. n])
1  found ← FALSE
2  for i ← 1 to n
3      if A[i] = 1
4          found ← TRUE
5          break
6  if not found
7      return 1
8  for i ← 1 to n
9      if A[i] ≤ 0 or A[i] > n
10         A[i] ← 1
11  for i ← 1 to n
12     A[(A[i] - 1) mod n + 1] += n
13  for i ← 1 to n
14     if A[i] ≤ n
15         return i
16  return n + 1

```

Да видим малък пример за работата на SMALLEST POSITIVE MISSING. Нека $n = 8$ и $A = [5, 2, 5, 3, 4, 1, -1, 9]$. Очевидно най-малкото липсващо положително число е 6.

Първият **for**-цикъл терминира с $\text{found} = \text{TRUE}$. Условието на ред 6 е лъжа и изпълнението отива на ред 8. След втория **for**-цикъл, A става $[5, 2, 5, 3, 4, 1, 1, 1]$. Двете новопоявили се единици (в зелено) са на мястото на -1 и 9 .

Изпълнението отива на ред 11. Индексната променлива взема последователно стойностите $1, 2, \dots, 8$. Да видим какво се случва.

i = 1: $A[i]$ е 5, $A[i] - 1$ е 4, $(4 \bmod 8) + 1$ е 5, $A[5]$ е 4, новата стойност на $A[5]$ е $4 + 8 = 12$.
Вече $A = [5, 2, 5, 3, 12, 1, 1, 1]$.

i = 2: $A[i]$ е 2, $A[i] - 1$ е 1, $(1 \bmod 8) + 1$ е 2, $A[2]$ е 2, новата стойност на $A[2]$ е $2 + 8 = 10$.
Вече $A = [5, 10, 5, 3, 12, 1, 1, 1]$.

i = 3: $A[i]$ е 5, $A[i] - 1$ е 4, $(4 \bmod 8) + 1$ е 5, $A[5]$ е 12, новата стойност на $A[5]$ е $12 + 8 = 20$.
Вече $A = [5, 10, 5, 3, 20, 1, 1, 1]$.

i = 4: $A[i]$ е 3, $A[i] - 1$ е 2, $(2 \bmod 8) + 1$ е 3, $A[3]$ е 5, новата стойност на $A[3]$ е $5 + 8 = 13$.
Вече $A = [5, 10, 13, 3, 20, 1, 1, 1]$.

i = 5: $A[i]$ е 20, $A[i] - 1$ е 19, $(19 \bmod 8) + 1$ е 4, $A[4]$ е 3, новата стойност на $A[4]$ е $3 + 8 = 11$.
Вече $A = [5, 10, 13, 11, 20, 1, 1, 1]$.

i = 6: $A[i]$ е 1, $A[i] - 1$ е 0, $(0 \bmod 8) + 1$ е 1, $A[1]$ е 5, новата стойност на $A[1]$ е $5 + 8 = 13$.
Вече $A = [13, 10, 13, 11, 20, 1, 1, 1]$.

i = 7: $A[i]$ е 1, $A[i] - 1$ е 0, $(0 \bmod 8) + 1$ е 1, $A[1]$ е 13, новата стойност на $A[1]$ е $13 + 8 = 21$.
Вече $A = [21, 10, 13, 11, 20, 1, 1, 1]$.

$i = 8$: $A[i]$ е 1, $A[i] - 1$ е 0, $(0 \bmod 8) + 1$ е 1, $A[1]$ е 21, новата стойност на $A[1]$ е $21 + 8 = 29$.
Вече $A = [29, 10, 13, 11, 20, 1, 1, 1]$.

И накрая, четвъртият **for**-цикъл връща $i = 6$, понеже $A[6] = 1$ е първият елемент на A , по-малък или равен на 8.

Този пример би трябвало да поясни какво прави третият **for**-цикъл, който е същността на алгоритъма. За всяко число m , което се среща в A преди началото на третия **for**-цикъл, $A[m]$ получава нова стойност чрез добавяне на 8, така че става по-голямо от 8. Ключовото наблюдение е, че при това $A[m]$ не се променя по модул 8. $A[i]$ последователно взема стойностите 5, 2, 5, 3, 20, 1, 1, 1 в този ред. Но 20 по модул 8 е 4, така че, в някакъв смисъл, $A[i]$ последователно взема стойностите 5, 2, 5, 3, 4, 1, 1, 1. Ерго, $A[i]$ не взема стойност 6 и $A[6]$ си остава със стойността 1 отпреди началото на третия **for**-цикъл.

Сега ще докажем формално коректността на алгоритъма. В следните лемии и теореми, с $\mathring{A}$ означаваме входния масив, а с A означаваме текущия масив.

Лема 5: Първият цикъл на SMALLEST POSITIVE MISSING

Ефектът от изпълнението на първия цикъл (редове 2–4) на алгоритъма SMALLEST POSITIVE MISSING е, че при терминирането, ако $1 \in \mathring{A}$, то $\text{found} = \text{TRUE}$, а в противен случай $\text{found} = \text{FALSE}$.

Доказателство: Следният инвариант е в сила за **for**-цикъл на редове 2–4.

Инвариант

Всеки път, когато изпълнението на SMALLEST POSITIVE MISSING е на ред 2, променливата found съдържа FALSE и $1 \notin A[1 \dots i - 1]$.

База. Да разгледаме първото достигане на ред 2. Променливата found има стойност FALSE заради присвояването на ред 1. В този момент $i = 1$, така че $A[1 \dots i - 1]$ е празен, а празният масив не съдържа нито едно 1. Инвариантът е в сила. ✓

Поддръжка. Нека инвариантът е в сила за някое достигане на ред 2, което не е последното. Щом не е последното, $i \leq n$ и $A[i] \neq 1$. По допускане, $1 \notin A[1 \dots i - 1]$. Щом $1 \notin A[1 \dots i - 1]$ и $A[i] \neq 1$, заключаваме, че $1 \notin A[1 \dots i]$.

Щом $i \leq n$, изпълнението отива на ред 3. Булевото условие на ред 3 е лъжа, понеже $A[i] \neq 1$, така че изпълнението отива на ред 2. Там i се инкрементира. По отношение на новото i е вярно, че $1 \notin A[1 \dots i - 1]$. Освен това, променливата found не беше променена, така че тя отново съдържа FALSE.

Терминация. Разглеждаме последното достигане на ред 2. Или $i = n + 1$, или са верни тези двете: $i \leq n$ и $A[i] = 1$. Използваме очевидния факт, че първият цикъл не променя масива, така че A е същият като $\mathring{A}$ при последното достигане на ред 2.

- Ако $i = n + 1$, заместваем i с $n + 1$ в инварианта и получаваме “ found съдържа FALSE и $1 \notin A[1 \dots n]$ ”. Заключаваме, че ако $1 \notin \mathring{A}$, то $\text{found} = \text{FALSE}$ при терминирането на цикъла.
- Ако $i \leq n$ и $A[i] = 1$, изпълнението отива на ред 3, където булевото условие е истина, така че изпълнението отива на ред 4, където found получава стойност TRUE, след което цикълът терминира на ред 5. Заключаваме, че ако $1 \in \mathring{A}$, то $\text{found} = \text{TRUE}$ при терминирането на цикъла. □

Следствие 4

Ако входният масив не съдържа 1, SMALLEST POSITIVE MISSING връща 1 на ред 7.
Ако входният масив съдържа 1, изпълнението достига ред 8.

Разглеждаме възможността изпълнението да достигне началото на втория **for**-цикъл на ред 8. Това означава, че в следните разсъждения приемаме, че входът съдържа единица. С $\ddot{A}$ означаваме масива A при термилирането на втория **for**-цикъл (редове 8–10).

Определение 10: $\text{El}(B)$

В текущия контекст, ако B е масив, с “ $\text{El}(B)$ ” означаваме $\{x \in \{1, \dots, n\} \mid x \in B\}$. Примерно, ако $n = 8$ и $A = [5, 2, 5, 3, 4, 1, -1, 9]$, то $\text{El}(A) = \{1, 2, 3, 4, 5\}$.

Лема 6: Вторият цикъл на SMALLEST POSITIVE MISSING

Ефектът от изпълнението на втория цикъл (редове 8–10) на алгоритъма SMALLEST POSITIVE MISSING е следният. При термилирането,

$$\begin{aligned} \forall j \in \{1, \dots, n\} : \ddot{A}[j] &\in \{1, \dots, n\} \\ \text{El}(\ddot{A}) &= \text{El}(\mathring{A}) \end{aligned}$$

Доказателство: Следният инвариант е в сила за **for**-цикълъ на редове 8–10.

Инвариант

Всеки път, когато изпълнението на SMALLEST POSITIVE MISSING е на ред 8:

- ① $\forall j \in \{1, \dots, i-1\} : A[j] \in \{1, \dots, n\}$
- ② $\forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$
- ③ $\forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$

База. Да разгледаме първото достигане на ред 8. Тогава $i = 1$, така че $\{1, \dots, i-1\}$ е празното множество, така че ①, ② и ③ са верни в празния смисъл. ✓

Поддръжка. Разглеждаме някой момент t , в който изпълнението е на ред 8 и тялото на цикъла ще се изпълни поне още веднъж. Допускаме, че ①, ② и ③ са в сила в момента t . Приемаме за очевидно, че в момента t е вярно, че $A[i, \dots, n] = \mathring{A}[i, \dots, n]$.

Следните три възможности в момента t са взаимно изключващи се и изчерпателни.

- $A[i] = 1$. Тогава $\mathring{A}[i] = 1$. По отношение на доказателството на ①, заключаваме, че

$$\forall j \in \{1, \dots, i\} : A[j] \in \{1, \dots, n\} \tag{2.14}$$

По отношение на доказателството на ②, в момента t от индуктивното допускане имаме

$$\forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

Но би-импликацията

$$\underbrace{A[i] = 1}_{\text{TRUE}} \leftrightarrow \underbrace{\mathring{A}[i] = 1}_{\text{TRUE}} \vee \underbrace{\mathring{A}[i] \notin \{1, \dots, n\}}_{\text{FALSE}}$$

е истина. Тогава в момента t :

$$\forall j \in \{1, \dots, i\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\} \quad (2.15)$$

По отношение на доказателството на ③, в момента t от индуктивното допускане имаме

$$\forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

Би-импликацията

$$\underbrace{A[i] \in \{2, \dots, n\}}_{\text{FALSE}} \leftrightarrow \underbrace{\mathring{A}[i] \in \{2, \dots, n\}}_{\text{FALSE}}$$

е истина. Тогава в момента t :

$$\forall j \in \{1, \dots, i\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\} \quad (2.16)$$

Булевото условие на ред 9 е лъжа, така че ред 10 не се изпълнява, а изпълнението отива на ред 8, където i се инкрементира. Предвид (2.14), (2.15) и (2.16), по отношение на новото i е вярно, че

$$\begin{aligned} \forall j \in \{1, \dots, i-1\} : A[j] \in \{1, \dots, n\} \\ \forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\} \\ \forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\} \end{aligned}$$

Но това са точно ①, ② и ③. ✓

- **$A[i] \in \{2, \dots, n\}$.** Тогава $\mathring{A}[i] \in \{2, \dots, n\}$. По отношение на доказателството на ①, заключаваме, че

$$\forall j \in \{1, \dots, i\} : A[j] \in \{1, \dots, n\} \quad (2.17)$$

По отношение на доказателството на ②, в момента t от индуктивното допускане имаме

$$\forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[i] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

Но би-импликацията

$$\underbrace{A[i] = 1}_{\text{FALSE}} \leftrightarrow \underbrace{\mathring{A}[i] = 1}_{\text{FALSE}} \vee \underbrace{\mathring{A}[i] \notin \{1, \dots, n\}}_{\text{FALSE}}$$

е истина. Тогава в момента t :

$$\forall j \in \{1, \dots, i\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\} \quad (2.18)$$

По отношение на доказателството на ③, в момента t от индуктивното допускане имаме

$$\forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

Но би-импликацията

$$\underbrace{A[i] \in \{2, \dots, n\}}_{\text{TRUE}} \leftrightarrow \underbrace{\mathring{A}[j] \in \{2, \dots, n\}}_{\text{TRUE}}$$

е истина. Тогава в момента t :

$$\forall j \in \{1, \dots, i\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\} \quad (2.19)$$

Булевото условие на ред 9 е лъжа, така че ред 10 не се изпълнява, а изпълнението отива на ред 8, където i се инкрементира. Предвид (2.17), (2.18) и (2.19), по отношение на новото i е вярно, че

$$\begin{aligned} \forall j \in \{1, \dots, i-1\} : A[j] &\in \{1, \dots, n\} \\ \forall j \in \{1, \dots, i-1\} : A[j] = 1 &\rightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\} \\ \forall j \in \{1, \dots, i-1\} : A[j] &\in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\} \end{aligned}$$

Но това са точно ①, ② и ③. ✓

- **$A[i] \notin \{1, \dots, n\}$.** Тогава $\mathring{A}[i] \notin \{1, \dots, n\}$. Първо да се убедим, че ① остава в сила. Булевото условие на ред 9 е истина, така че изпълнението отива на ред 10, където на $A[i]$ се присвоява 1. Заключаваме, че

$$\forall j \in \{1, \dots, i\} : A[j] \in \{1, \dots, n\}$$

Изпълнението отива отново на ред 8, където i се инкрементира. Спрямо новото i е вярно, че

$$\forall j \in \{1, \dots, i-1\} : A[j] \in \{1, \dots, n\}$$

Виждаме, че ① остава в сила. ✓

Сега разглеждаме ②. От индуктивното допускане знаем, че в момента t :

$$\forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

Забелязваме, че би-импликацията

$$\underbrace{A[i] = 1}_{\text{FALSE}} \leftrightarrow \underbrace{\mathring{A}[i] = 1}_{\text{FALSE}} \vee \underbrace{\mathring{A}[i] \notin \{1, \dots, n\}}_{\text{TRUE}}$$

е лъжа. Но булевото условие на ред 9 е истина, така че изпълнението отива на ред 10, където на $A[i]$ се присвоява 1. Сега вече би-импликацията

$$\underbrace{A[i] = 1}_{\text{TRUE}} \leftrightarrow \underbrace{\mathring{A}[i] = 1}_{\text{FALSE}} \vee \underbrace{\mathring{A}[i] \notin \{1, \dots, n\}}_{\text{TRUE}}$$

е истина. Имаме право да кажем, че

$$\forall j \in \{1, \dots, i\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

Изпълнението отива отново на ред 8, където i се инкрементира. Спрямо новото i е вярно, че

$$\forall j \in \{1, \dots, i-1\} : A[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

Виждаме, че и ② остава в сила. ✓

Сега разглеждаме ③. От индуктивното допускане знаем, че в момента t :

$$\forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

Но би-импликацията

$$\underbrace{A[i] \in \{2, \dots, n\}}_{\text{FALSE}} \leftrightarrow \underbrace{\mathring{A}[j] \in \{2, \dots, n\}}_{\text{FALSE}}$$

е истина. Булевото условие на ред 9 е истина, така че изпълнението отива на ред 10, където на $A[i]$ се присвоява 1. Би-импликацията

$$\underbrace{A[i] \in \{2, \dots, n\}}_{\text{FALSE}} \leftrightarrow \underbrace{\mathring{A}[j] \in \{2, \dots, n\}}_{\text{FALSE}}$$

продължава да е истина. Имаме право да кажем, че

$$\forall j \in \{1, \dots, i\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

Изпълнението отива отново на ред 8, където i се инкрементира. Спрямо новото i е вярно, че

$$\forall j \in \{1, \dots, i-1\} : A[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

Виждаме, че и ③ остава в сила. ✓

Терминация. Разглеждаме последното достигане на ред 8. В сила е $i = n + 1$. Заместваме i с $n + 1$ в инварианта и получаваме

$$\mathbf{A}: \forall j \in \{1, \dots, n\} : \ddot{A}[j] \in \{1, \dots, n\}$$

$$\mathbf{B}: \forall j \in \{1, \dots, n\} : \ddot{A}[j] = 1 \leftrightarrow \mathring{A}[j] = 1 \vee \mathring{A}[j] \notin \{1, \dots, n\}$$

$$\mathbf{C}: \forall j \in \{1, \dots, n\} : \ddot{A}[j] \in \{2, \dots, n\} \leftrightarrow \mathring{A}[j] \in \{2, \dots, n\}$$

В лемата се твърди, че

$$\begin{aligned} \forall j \in \{1, \dots, n\} : \ddot{A}[j] \in \{1, \dots, n\} \\ \text{El}(\ddot{A}) = \text{El}(\mathring{A}) \end{aligned}$$

Първото твърдение вече доказахме – то е **A**. Остава да докажем, че $\text{El}(\ddot{A}) = \text{El}(\mathring{A})$. На прост български, това казва, че числата от $\{1, \dots, n\}$, които се срещат във входа, са точно тези числа от $\{1, \dots, n\}$, които се срещат в масива след третия цикъл.

- Разглеждаме произволно число x от $\{1, \dots, n\}$, което се среща във входа. Ако $x = 1$, то x се среща в $\ddot{A}$ заради **B**. Ако $x > 1$, то x се среща в $\ddot{A}$ заради **C**.
- Сега разглеждаме произволно число x от $\{1, \dots, n\}$, което не се среща във входа. Тъй като **B** и **C** са би-импликации, заключаваме, че x не се среща в $\ddot{A}$. $\square$

С $\ddot{A}$ означаваме масива A при терминирането на третия **for**-цикъл (редове 8–10).

Лема 7: Третият цикъл на SMALLEST POSITIVE MISSING

Ефектът от изпълнението на третия цикъл (редове 11–12) на алгоритъма SMALLEST POSITIVE MISSING е следният. При терминирането е изпълнено

$$\forall i \in \{1, \dots, n\} : \ddot{A}[i] > n \leftrightarrow i \in \ddot{A}$$

Доказателство: Следният инвариант е в сила за **for**-цикъла на редове 11–12.

Инвариант

Всеки път, когато изпълнението на SMALLEST POSITIVE MISSING е на ред 11:

- ❶ $\forall j \in \{1, \dots, n\} : A[j] \geq 1$.
- ❷ $\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots i - 1]$.
- ❸ $\forall j \in \{1, \dots, n\} : A[j] \equiv \ddot{A}[j] \pmod{n}$.

База. Разглеждаме първото достигане на ред 11. В този момент $i = 1$ и, съгласно Лема 6, $A[j] \in \{1, \dots, n\}$ за всяко $j \in \{1, \dots, n\}$. Заключаваме, че ❶ е вярно. Нещо повече, хем $A[j] > n$ е лъжа за всяко $j \in \{1, \dots, n\}$, хем $\ddot{A}[1 \dots i - 1] = \ddot{A}[1 \dots 0]$ е празният масив, така че и ❷ е вярно. А ❸ е вярно, понеже в този момент $A = \ddot{A}$ по конструкцията.

Поддръжка. Нека t е момент от работата на алгоритъма, в който изпълнението е на ред 11 и тялото на цикъла ще се изпълни поне още веднъж. Допускаме, че инвариантът е в сила в момента t .

Ред 12 се изпълнява безусловно. Очевидно $(A[i] - 1) \bmod n \in \{0, \dots, n - 1\}$, така че $((A[i] - 1) \bmod n) + 1 \in \{1, \dots, n\}$, така че на ред 12, индексът $((A[i] - 1) \bmod n) + 1$ е легитимен индекс в A . Тоест, няма опит за достъп извън рамките на масива.

Нека t' е момента на следващото достигане на ред 11. Ще покажем, че инвариантът остава в сила в момента t' .

- Ще покажем, че ❶ е изпълнено в момента t' . Единствената промяна на елемент от масива между моментите t и t' е увеличаването с n на един единствен елемент. По допускане, в момента t елементите на масива са положителни. Тогава в момента t' те също са положителни. ✓
- Ще покажем, че ❷ е изпълнено в момента t' . По допускане, ❷ е изпълнено в момента t . На прост български: в момента t , елементите на текущия масив, по-големи от n , са точно тези, чиито индекси са елементите на $\ddot{A}[1 \dots i - 1]$. За краткост, дефинираме, че $\ell = ((A[i] - 1) \bmod n) + 1$.

Лема 8: $\ell = \ddot{A}[i]$

В текущия контекст, $\ell = \ddot{A}[i]$.

Доказателство: От теорията на числата знаем [61, (3.21), стр. 82], че за произволни реални числа x и y , където $y \neq 0$:

$$x \bmod y \stackrel{\text{def}}{=} x - y \left\lfloor \frac{x}{y} \right\rfloor$$

Освен това, от част ③ на инварианта знаем, че $A[i] = \ddot{A}[i] + kn$ за някакво естествено k . Тогава

$$\begin{aligned} \ell &= ((A[i] - 1) \bmod n) + 1 = \\ &= ((\ddot{A}[i] + kn - 1) \bmod n) + 1 = \\ &= \ddot{A}[i] + kn - 1 - n \left\lfloor \frac{\ddot{A}[i] + kn - 1}{n} \right\rfloor + 1 = \\ &= \ddot{A}[i] + kn - n \left\lfloor \frac{\ddot{A}[i] - 1}{n} + \frac{kn}{n} \right\rfloor = \\ &= \ddot{A}[i] + kn - n \left\lfloor \frac{\ddot{A}[i] - 1}{n} + k \right\rfloor = \\ &= \ddot{A}[i] + kn - n \left\lfloor \frac{\ddot{A}[i] - 1}{n} \right\rfloor - kn = \\ &= \ddot{A}[i] - n \left\lfloor \frac{\ddot{A}[i] - 1}{n} \right\rfloor \end{aligned}$$

Но $\left\lfloor \frac{\ddot{A}[i] - 1}{n} \right\rfloor = 0$, понеже $\ddot{A}[i] \in \{1, \dots, n\}$. Тогава

$$\ddot{A}[i] - n \left\lfloor \frac{\ddot{A}[i] - 1}{n} \right\rfloor = \ddot{A}[i]$$

Заклучаваме, че $\ell = \ddot{A}[i]$. □

- ◆ Нека $A[\ell] > n$ преди изпълнението на ред 12. След изпълнението на ред 12, $A[\ell]$ се е увеличил с n . При това увеличение, множеството от индексите на елементите на A , по-големи от n , не се променя.

Твърдим, че множеството от елементите на $\ddot{A}[1 \dots i - 1]$ е равно на множеството от елементите на $\ddot{A}[1 \dots i]$. Твърдението е същото като $\ddot{A}[i] \in \ddot{A}[1 \dots i - 1]$. Допуснали сме, че $A[\ell] > n$. Част ② на инварианта казва, че индексът ℓ е елемент на $\ddot{A}[1 \dots i - 1]$. Но от Лема 8 знаем, че $\ell = \ddot{A}[i]$. Тогава наистина $\ddot{A}[i] \in \ddot{A}[1 \dots i - 1]$.

И така, в този подслучай хем множеството от индексите на елементите на A , по-големи от n , остава същото след изпълнението на тялото на цикъла, хем множеството от елементите на $\ddot{A}[1 \dots i - 1]$ е равно на множеството от елементите на $\ddot{A}[1 \dots i]$. Тогава $\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots i]$.

Изпълнението отива на ред 11, където i се инкрементира. По отношение на новото i отново е вярно, че $\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots i - 1]$. Заклучаваме, че

❷ е в сила в момента t' . ✓

- Нека $A[\ell] \leq n$ преди изпълнението на ред 12. От част ❶ на инварианта знаем, че $A[\ell] \geq 1$ в този момент. След изпълнението на ред 12, $A[\ell]$ се е увеличил с n , ставайки поне $n + 1$. Тогава множеството от индексите на елементите на масива A , по-големи от n , нараства. Негов елемент вече е и ℓ .

От Лема 8 знаем, че $\ell = \ddot{A}[i]$. И така, в този подслучай хем към множеството от индексите на елементите на A , по-големи от n , се добавя ℓ , хем $\ell \in \ddot{A}[1 \dots i]$. Тогава $\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots i]$.

Изпълнението отива на ред 11, където i се инкрементира. По отношение на новото i отново е вярно, че $\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots i - 1]$. Заклучаваме, че ❷ е в сила в момента t' . ✓

- Ще покажем, че ❸ е изпълнено в t' . От част ❸ на инварианта знаем, че $\forall j \in \{1, \dots, n\} : A[j] \equiv \ddot{A}[j] \pmod n$ преди присвояването на ред 12. Наистина, добавянето на n към един елемент на масива не променя това. ✓

Терминация. При последното достигане на ред 11 имаме $i = n + 1$. Заместваме i с $n + 1$ в ❷ и получаваме

$$\forall j \in \{1, \dots, n\} : A[j] > n \leftrightarrow j \in \ddot{A}[1 \dots n]$$

Тъй като става дума за масива в края на третия цикъл, можем да препишем това така

$$\forall i \in \{1, \dots, n\} : \ddot{A}[i] > n \leftrightarrow i \in \ddot{A}[1 \dots n]$$

Но кои i от $\{1, \dots, n\}$ се срещат в $\ddot{A}$? По дефиниция (Определение 10), това са точно елементите на $\text{El}(\ddot{A})$. Но от Лема 6 знаем, че $\text{El}(\ddot{A}) = \text{El}(\ddot{A})$. Заклучаваме, че

$$\forall i \in \{1, \dots, n\} : \ddot{A}[i] > n \leftrightarrow i \in \ddot{A}$$

Кое и трябваше да покажем. □

Лема 9: Четвъртият цикъл на SMALLEST POSITIVE MISSING

Ефектът от изпълнението на четвъртия цикъл (редове 13–15) на алгоритъма SMALLEST POSITIVE MISSING е следният. Ако $\ddot{A}$ има елементи, по-малки или равни на n , то алгоритъмът връща индекса на най-левия от тях. В противен случай, изпълнението достига ред 16.

Доказателство: Ако $\ddot{A}[1] \leq n$, очевидно алгоритъмът връща 1 на ред 15. В този случай твърдението очевидно е в сила.

Да допуснем, че $\ddot{A}[1] > n$. Следният инвариант е в сила за **for**-цикъл на редове 13–15.

Инвариант

Всеки път, когато изпълнението е на ред 13, $\forall j \in \{1, \dots, i - 1\} : A[j] > n$.

База. Разглеждаме второто достигане на ред 13. В този момент, $i = 2$. Ерго, $\{1, \dots, i - 1\} = \{1\}$. Но ние работим под допускането, че $\ddot{A}[1] > n$. Тогава $A[1] > n$, така че твърдението е вярно. ✓

Поддръжка. Разглеждаме някое достигане на ред 13, което не е последното. Такова достигане трябва да има, защото работим под допускането, че $\ddot{A}[1] > n$. Допускаме, че инвариантът е в сила в този момент, тоест, $\forall j \in \{1, \dots, i-1\} : A[j] > n$. Щом достигането не е последното, $A[i] > n$. Имаме право да кажем, че $\forall j \in \{1, \dots, i\} : A[j] > n$. Условието на ред 14 е лъжа и изпълнението отива отново на ред 13, където i се инкрементира. Спрямо новото i отново е вярно, че $\forall j \in \{1, \dots, i-1\} : A[j] > n$. ✓

Терминация. Цикълът терминира по точно един от тези два начина: или изпълнението достига ред 15, където алгоритъмът връща i , или изпълнението достига ред 13 с $i = n + 1$.

- Да допуснем, че цикълът терминира през ред 15. За да се достигне ред 15, трябва условието на ред 14 да е истина. Това означава, че $A[i] \leq n$ за текущото i , като $i \geq 2$. От инварианта знаем, че $\forall j \in \{1, \dots, i-1\} : A[j] > n$. Заклучаваме, че хем $\ddot{A}$ има елементи, по-малки или равни на n , хем алгоритъмът връща индекса на най-левия от тях, точно както се твърди в лемата. ✓
- Да допуснем, че цикълът терминира през ред 13. Заместваме i с $n + 1$ в инварианта и получаваме $\forall j \in \{1, \dots, i-1\} : A[j] > n$. И така, хем всички елементи са по-големи от n , хем изпълнението отива на ред 16, точно както се твърди в лемата. ✓ □

Теорема 13: Коректността на SMALLEST POSITIVE MISSING

Ако $\text{El}(\ddot{A}) \neq \{1, \dots, n\}$, то $\text{SMALLEST POSITIVE MISSING}(\ddot{A})$ връща $\min \{1, \dots, n\} \setminus \text{El}(\ddot{A})$.
В противен случай, $\text{SMALLEST POSITIVE MISSING}(\ddot{A})$ връща $n + 1$.

Доказателство: Първо да допуснем, че $\text{El}(\ddot{A}) \neq \{1, \dots, n\}$.

- Нека $1 \notin \text{El}(\ddot{A})$. Тогава, съгласно Следствие 4, $\text{SMALLEST POSITIVE MISSING}(\ddot{A})$ връща 1. Но в този случай $\min \{1, \dots, n\} \setminus \text{El}(\ddot{A}) = 1$. Заклучаваме, че $\text{SMALLEST POSITIVE MISSING}(\ddot{A})$ наистина връща $\min \{1, \dots, n\} \setminus \text{El}(\ddot{A})$. ✓
- Нека $1 \in \text{El}(\ddot{A})$. Очевидно е, че $\min \{1, \dots, n\} \setminus \text{El}(\ddot{A}) > 1$ в този случай. Съгласно Следствие 4, изпълнението достига ред 8. Изпълнява се вторият **for**-цикъл и после третият **for**-цикъл. Съгласно Лема 7, при терминирането на третия цикъл, за всеки елемент на масива, е вярно, че той е по-голям от n тстк неговият индекс—а той е между 1 и n —е число, което се среща във входа. При това положение, числата, които не се срещат във входа, са индексите на елементите, които не се срещат във входа. Нещо повече, най-малкият такъв индекс е най-малкото число от $\{1, \dots, n\}$, което не се среща във входа. Сега прилагаме Лема 9 и заключаваме, че алгоритъмът връща най-малкото число от $\{1, \dots, n\}$, което не се среща във входа. ✓

Сега да допуснем, че $\text{El}(\ddot{A}) = \{1, \dots, n\}$. Съгласно Следствие 4, изпълнението достига ред 8. Изпълнява се вторият **for**-цикъл и после третият **for**-цикъл. Съгласно Лема 7, при терминирането на третия цикъл, за всеки елемент на масива, е вярно, че той е по-голям от n тстк неговият индекс—а той е между 1 и n —е число, което се среща във входа. Но в този случай, всички числа между 1 и n се срещат във входа. Заклучаваме, че всички числа в масива са по-големи от n . Сега прилагаме Лема 9 и заключаваме, че изпълнението достига ред 16, където алгоритъмът връща $n + 1$. □

2.1.3.8 BOYER–MOORE за намиране на мажорант

Даден е масив от цели числа $A[1 \dots n]$, където $n \geq 2$. *Мажорант* на масива е число, такова че повече от половината елементи на масива са равни на него. Задачата е да се конструира алгоритъм с линейна сложност по време, чийто вход е масивът A , а изходът е или мажорант на A , ако такъв съществува, или съобщение, че A няма мажорант, в противен случай, и да се докаже прецизно коректността му.

Мажорант е частен случай на мода и всеки алгоритъм за намиране на мода (вижте Подсекция 4.2.5) може да се модифицира тривиално, така че да или да извежда мажорант, или да извежда съобщение, че няма мажорант. Но този начин за намиране на мажорант не е достатъчно ефикасен: всеки алгоритъм за намиране на мода, базиран на сравнения, работи във време $\Omega(n \lg n)$ (вижте Подсекция 13.3.2), а тук се иска линеен алгоритъм.

Следният алгоритъм за намиране на мажорант в линейно време е известен от 90-те години на XX век [27].

BOYER-MOORE($A[1 \dots n]$: int, $n \geq 2$)

```

1  c ← 0, i ← 1
2  do
3      if c = 0
4          m ← A[i]
5          c ← 1
6      else
7          if m = A[i]
8              c++
9          else
10             c--
11         i++
12 while i ≤ n
13 c ← 0
14 for k ← 1 to n
15     if A[k] = m
16         c++
17 if c ≥ ⌊ $\frac{n}{2}$ ⌋ + 1
18     return m
19 else
20     return няма мажорант
```

Преди доказателство за коректност ще направим няколко дефиниции. Нека $S[1 \dots n]$ е целочислен масив и $S[p \dots q]$ е негов непразен подмасив. Казваме, че $S[p]$ е *лидърът* на $S[p \dots q]$. *Дисбалансът* в $S[p \dots q]$ е следната разлика: броят на срещанията на лидера минус броя на останалите елементи на $S[p \dots q]$. Казваме, че $S[p \dots q]$ е *регулярен*, ако неговият дисбаланс е неотрицателен, и че $S[p \dots q]$ е *балансиран*, ако неговият дисбаланс е нула. Примерно, ако $S = [2, 5, 1, 3, 6, 1, 2, 2, 3]$, то $S[1 \dots 3] = [2, 5, 1]$ не е регулярен, $S[7 \dots 9] = [2, 2, 3]$ е регулярен, но не е балансиран, $S[1 \dots 1] = [2]$ също е регулярен, но не балансиран, а $S[3 \dots 6] = [1, 3, 6, 1]$ е балансиран.

Мажорантното разбиване на S ще наричаме редицата $\text{maj}(S)$ от подмасиви на S :

$$\text{maj}(S) \stackrel{\text{def}}{=} (S[1 \dots i_1], S[i_1 + 1 \dots i_2], S[i_2 + 1 \dots i_3], \dots, S[i_{t-1} + 1 \dots i_t], S[i_t + 1 \dots n])$$

където $i_1, \dots, i_t$ са уникалните индекси, такива че $1 < i_1 < i_2 < \dots < i_{t-1} < i_t < n$, като $t \geq 0$ и освен това

- $S[1 .. i_1]$ е най-късият балансиран подмасив с лидер $S[1]$,
- $S[i_1 + 1 .. i_2]$ е най-късият балансиран подмасив с лидер $S[i_1 + 1]$,
- $S[i_2 + 1 .. i_3]$ е най-късият балансиран подмасив с лидер $S[i_2 + 1]$,
- ...
- $S[i_{t-1} + 1 .. i_t]$ е най-късият балансиран подмасив с лидер $S[i_{t-1} + 1]$,
- $S[i_t + 1 .. n]$ е регулярен. Освен това, или $S[i_t + 1 .. n]$ е най-късият балансиран подмасив с лидер $S[i_t + 1]$, или $S[i_t + 1]$ не е лидер балансиран подмасив.

Примерно, ако $S = [2, 5, 1, 3, 6, 1, 2, 2, 3]$, то $\text{maj}(S) = ([2, 5], [1, 3], [6, 1], [2, 2, 3])$, като $[2, 5]$, $[1, 3]$ и $[6, 1]$ са балансираны, а $[2, 2, 3]$ не е балансиран. Сега да разгледаме друг пример. Ако $S' = [2, 2, 5, 1, 3, 3, 6, 3, 1, 1, 2, 2, 3, 4]$, то $\text{maj}(S') = ([2, 2, 5, 1], [3, 3, 6, 3, 1, 1], [2, 2, 3, 4])$, като и трите подмасиви са балансираны. Като трети пример, ако $S'' = [2, 2, 2, 3, 4]$, то $\text{maj}(S'') = ([2, 2, 2, 3, 4])$, като масивът не е балансиран; в този случай $t = 0$ и $i_t = 0$.

Наблюдение 9

В контекста на горните дефиниции, нека $A[p .. q]$ е елемент на $\text{maj}(S)$. Тогава за всяко $\ell \in \{p, \dots, q\}$, подмасивът $A[p .. \ell]$ е регулярен.

Лема 10

При последното достигане на ред 12, променливата m съдържа лидера на последния масив от $\text{maj}(S)$.

Доказателство: Следното твърдение е инвариант на **do-while** цикъла на редове 2–12. По отношение на произволно достигане на ред 12 дефинираме, че *текущият подмасив* е този елемент на $\text{maj}(A)$, който съдържа $A[i - 1]$.

Инвариант

Всеки път, когато изпълнението е на ред 12, ако текущият подмасив е $A[p .. q]$, то m съдържа лидера му, а c съдържа дисбаланса на $A[p .. i - 1]$.

Доказателство.

База. В началото на алгоритъма, c получава стойност 0 и i получава стойност 1 на ред 1. Условието на ред 3 е истина, така че m получава стойност $A[1]$ на ред 4, а c получава стойност 1 на ред 5. После i се инкрементира на ред 11 и става 2. Разглеждаме първото достигане на ред 12. Тъй като $i = 2$, текущият подмасив $A[p .. q]$ е този елемент на $\text{maj}(S)$, който съдържа $A[1]$. Тогава $A[p .. q]$ е $A[1 .. q]$, за някакво q . Подмасивът $A[p .. i - 1]$, за който говори инварианта, е $A[1 .. 1]$.

От една страна, лидерът на текущия подмасив е $A[1]$. От друга страна, $m = A[1]$. От една страна, дисбалансът на $A[p .. i - 1]$ е 1. От друга страна, $c = 1$. Заключаваме, че инвариантът е в сила при първото достигане на ред 12. ✓

Поддръжка. Разглеждаме някое достигане на ред 12, което не е последното. Нека t е моментът от дискретното време на алгоритъма, в който става това достигане. Нека t' е моментът на следващото достигане на ред 12; такъв момент очевидно съществува. Допускаме, че инвариантът е в сила в момента t . Съгласно Наблюдение 9 и допускането, с се явява дисбаланс на регулярен масив, така че $s \geq 0$. Тогава следните случаи са взаимно изключващи се и изчерпателни.

- $s = 0$. От допускането следва, че $A[p \dots i-1]$ е балансиран, от което следва, че $A[p \dots i-1]$ е масив-елемент на $\text{maj}(S)$, от което пък следва, че $A[i]$ е първият елемент от следващия масив от $\text{maj}(S)$. Нека този следващ масив е $A[p' \dots q']$, където $p' = i$.

Разглеждаме изпълнението на тялото на **do-while** цикъла от момента t до момента t' . Условието на ред 3 е истина, така че на ред 4, m получава стойност $A[i]$, който е лидерът на $A[p' \dots q']$, а на ред 5, s получава стойност 1, която е дисбалансът на масива $A[p' \dots i]$. След това i се инкрементира на ред 11. Спрямо новото i е вярно, че s съдържа дисбаланса на $A[p' \dots i-1]$.

Изпълнението отива на ред 12. Това е моментът t' . Ясно е, че текущият подмасив в този момент е $A[p' \dots q']$. Заклучаваме, че в момента t' , m съдържа лидера на текущия подмасив $A[p' \dots q']$, а s съдържа дисбаланса на $A[p' \dots i-1]$. Инвариантът е в сила. ✓

- $s > 0$ и $A[i]$ е равен на лидера на текущия подмасив. Щом $s > 0$, от допускането следва, че $A[p \dots i-1]$ не е масив-елемент на $\text{maj}(S)$, от което пък следва, че $A[i]$ също е елемент от $A[p \dots q]$.

Разглеждаме изпълнението на тялото на **do-while** цикъла от момента t до момента t' . Условието на ред 3 е лъжа, така че изпълнението отива на ред 6. Условието на ред 7 е истина, защото по допускане, m съдържа лидера на $A[p \dots q]$. На ред 8, s се инкрементира. Предвид допускането, s вече съдържа дисбаланса на $A[p \dots i]$. На ред 11, i се инкрементира. Спрямо новото i е вярно, че s съдържа дисбаланса на $A[p \dots i-1]$.

Изпълнението отива на ред 12. Това е моментът t' . Ясно е, че текущият подмасив в този момент е пак $A[p \dots q]$ и m , която не беше променяна от момента t до момента t' , съдържа лидера на текущия подмасив, а s съдържа дисбаланса на $A[p' \dots i-1]$. ✓

- $s > 0$ и $A[i]$ не е равен на лидера на текущия подмасив. Щом $s > 0$, от допускането следва, че $A[p \dots i-1]$ не е масив-елемент на $\text{maj}(S)$, от което пък следва, че $A[i]$ също е елемент от $A[p \dots q]$.

Разглеждаме изпълнението на тялото на **do-while** цикъла от момента t до момента t' . Условието на ред 3 е лъжа, така че изпълнението отива на ред 6. Условието на ред 7 е лъжа, защото по допускане, m съдържа лидера на $A[p \dots q]$. На ред 10, s се декрементира. Предвид допускането, s вече съдържа дисбаланса на $A[p \dots i]$. На ред 11, i се инкрементира. Спрямо новото i е вярно, че s съдържа дисбаланса на $A[p \dots i-1]$.

Изпълнението отива на ред 12. Това е моментът t' . Ясно е, че текущият подмасив в този момент е пак $A[p \dots q]$ и m , която не беше променяна от момента t до момента t' , съдържа лидера на текущия подмасив, а s съдържа дисбаланса на $A[p' \dots i-1]$. ✓

Терминация. Разглеждаме последното достигане на ред 12, което става в някакъв момент $\tilde{t}$. В този момент очевидно $i = n + 1$. Тогава текущият подмасив е този елемент на $\text{maj}(S)$, който съдържа $A[n]$. Но това очевидно е последният масив от $\text{maj}(S)$. От инварианта знаем, че m съдържа лидера му. Следователно, в момента $\tilde{t}$, променливата m съдържа лидера на последния масив от $\text{maj}(S)$. □

Лема 11

Разглеждаме входния масив A на BOYER-MOORE. Нека x е лидерът на последния масив от $\text{maj}(A)$. За всеки y от A , такъв че $y \neq x$, е вярно, че y не е мажорант на A .

Доказателство: Ключовото наблюдение е, че нито един масив от $\text{maj}(A)$, който не е последният, сам по себе си няма мажорант, защото е балансиран; балансиран масив очевидно не може да има мажорант.

Нека “конкатенацията” на масивите от $\text{maj}(A)$ без последния се нарича B . Нека последният масив от $\text{maj}(A)$ се нарича D . Нека $b = |B|$ и $d = |D|$. Забелязваме, че b е четно, понеже B е конкатенация от балансиран масиви, а всеки балансиран масив има четна големина.

Разглеждаме произволен елемент y от A , такъв че $y \neq x$. Забелязваме, че y се среща най-много $\frac{b}{2}$ пъти в B , защото B е конкатенация от балансиран масиви, и y се среща най-много $\lfloor \frac{d}{2} \rfloor$ пъти в D , защото D е регулярен и лидерът му е x , а $x \neq y$ по дефиниция. Заклучаваме, че y се среща не повече от $\lfloor \frac{n}{2} \rfloor$ в A . За да бъде мажорант на A , елемент трябва да се среща поне $\lfloor \frac{n}{2} \rfloor + 1$ в A . $\square$

Лема 12

Променливата s на ред 17 съдържа броя на появите на m в A .

Доказателство: Следното твърдение е инвариант за **for**-цикъла на редове 14–17.

Инвариант

За всяко достигане на ред 14, s съдържа броя на срещанията на m в $A[1 \dots k-1]$.

База. В сила е $k = 1$ и $s = 0$ при първото достигане на ред 14. Наистина, m се среща 0 пъти в празния подмасив $A[1 \dots 0]$.

Поддръжка. Разглеждаме някое достигане на ред 14, което не е последното. Допускаме, че твърдението е вярно. Разглеждаме два случая.

- $A[k] = m$. Тогава, от една страна, m се среща $s + 1$ пъти в $A[1 \dots k]$. От друга страна, условието на ред 15 е истина, s се инкрементира и спрямо новото s е вярно, че m се среща s пъти в $A[1 \dots k]$. След инкрементирането на k на ред 14, спрямо новото k , пак е вярно, че m се среща s пъти в $A[1 \dots k-1]$.
- $A[k] \neq m$. Тогава, от една страна, m се среща s пъти в $A[1 \dots k]$. От друга страна, условието на ред 15 е лъжа, така че s не се инкрементира и остава вярно, че m се среща s пъти в $A[1 \dots k]$. След инкрементирането на k на ред 14, спрямо новото k , пак е вярно, че m се среща s пъти в $A[1 \dots k-1]$.

Терминация. При последното достигане на ред 14 е вярно, че $k = n + 1$. Заместваме в инварианта и получаваме “ s съдържа броя на срещанията на m в $A[1 \dots n]$ ”. $\square$

Теорема 14: Коректността на BOYER-MOORE

BOYER-MOORE($A[1 \dots n]$) връща или стойността на мажорант на A , ако такъв съществува, или съобщение, че няма мажорант, в противен случай.

Доказателство: От Лема 10 знаем, че при последното достигане на ред 12, променливата m съдържа лидера на последния масив от $\text{maj}(S)$. От Лема 11 знаем, че ако A има мажорант, това е лидерът на последния масив от $\text{maj}(S)$. Това означава, че когато изпълнението е на ред 14 за първи път, единственият възможен мажорант на A е m . От Лема 12 знаем, че на ред 17, променливата s съдържа броя на появите на m в A . Заклучаваме, че ако s на ред 17 е поне $\lfloor \frac{n}{2} \rfloor + 1$, то A има мажорант и това е m , а в противен случай A няма мажорант. От друга страна, ако на ред 17, $s \geq \lfloor \frac{n}{2} \rfloor + 1$, алгоритъмът връща m , в противен случай връща съобщение, че няма мажорант. $\square$

2.1.3.9 Една транспортна задача

Нека $C_1, C_2, \dots, C_n$ са градове, наредени в линейна наредба в този ред. Да кажем, по протежението на едно шосе. Нека d_i е разстоянието в метри между C_i и C_{i+1} , за $1 \leq i < n$.

В началния момент, всеки C_i притежава някаква наличност $x_i \in \mathbb{Z}$ от някакъв ресурс. Да кажем, литри вода. Ако $x_i < 0$, това, което C_i притежава не е реална вода, а недостиг на вода. Примерно, ако $x_1 = -5\,500$, то C_1 има недостиг от 5 500 литра вода в началото; ако по някое време транспортираме 6 000 литра вода в C_1 , то C_1 ще има 500 литра реална вода.

Водата може да се транспортира между съседни градове, но транспортът е със загуби. Загубата на вода в литри при транспортиране е равна на метрите, на които се транспортира, а ако при транспортирането количеството вода стане нула, то остава нула. Може да си мислите, че цистерните, с които се транспортира, са надупчени и по време на транспортирането от тях изтича вода с константна скорост като литри на изминат метър, докато има вода изобщо. Формално казано: ако започнем да транспортираме ℓ литра вода от C_i до C_{i+1} или от C_{i+1} до C_i , количеството, което ще пристигне, е $\max\{\ell - d_i, 0\}$.

За всеки C_i е дадена константа $m_i \in \mathbb{Z}^+$, която има смисъл на литри вода, от които се нуждае C_i . Казваме, че C_i е *удовлетворен*, ако $m_i \leq x_i$; в противен случай е *неудовлетворен*. Пита се, ако има неудовлетворени градове в началния момент, дали има начин да се транспортира вода между градове по такъв начин, че всички градове да се окажат удовлетворени? Имайки предвид, че транспортирането на вода е със загуби.

Изч. Задача 12: ТРАНСПОРТИРАНЕ СЪС ЗАГУБИ

екземляр: $d_1 > 0, \dots, d_{n-1} > 0$: разстояния, $x_1, \dots, x_n$: начални наличности вода, $m_1 > 0, \dots, m_n > 0$: минимални количества вода.

въпрос: Има ли начин да бъде транспортирана вода между градовете (при положение, че транспортирането е със загуби) по такъв начин, че всеки град да бъде удовлетворен?

Определение 11: Добра редица и дефицитна редица

За всяко $i \in \{1, \dots, n\}$, дефинираме, че редицата от градове $\langle C_1, C_2, \dots, C_i \rangle$ е $\mathcal{A}_i$. Казваме, че $\mathcal{A}_i$ е *добра*, ако нейните градове са удовлетворени или има начин да бъдат удовлетворени с транспортиране на вода само помежду им. В противен случай, $\mathcal{A}_i$ е *дефицитна*.

Определение 12: q-излишна, q-дефицитна, критична и изолирана редица

Нека $q \in \mathbb{Z}^+$. За всяко $i \in \{1, \dots, n\}$, казваме, че $\mathcal{A}_i$ е:

- *q-излишна*, ако $\mathcal{A}_i$ е добра и

$$q = \max \{y \in \mathbb{Z}^+ \mid \text{ако количеството в } C_i \text{ бъде намалено с } y, \mathcal{A}_i \text{ остава добра}\}$$

- *q-дефицитна*, ако $\mathcal{A}_i$ е дефицитна и

$$q = \max \{y \in \mathbb{Z}^+ \mid \text{ако количеството в } C_i \text{ бъде увеличено с } y - 1, \\ \mathcal{A}_i \text{ остава дефицитна}\}$$

- *критична*, ако $\mathcal{A}_i$ е добра, но не е 1-излишна.
- *изолирана*, ако $\mathcal{A}_i$ е добра, но ако количеството вода в C_i бъде намалено с $\geq d_i$ литра, то $\mathcal{A}_i$ става дефицитна.

Наблюдение 10

Забележете следните неща.

1. Всяка $\mathcal{A}_i$ е точно едно от тези три:
 - или е q-излишна за точно едно $q \in \mathbb{Z}^+$,
 - или е критична,
 - или е q-дефицитна за точно едно $q \in \mathbb{Z}^+$.
2. Ако $i < n$ и $\mathcal{A}_i$ е изолирана, от $\mathcal{A}_i$ не може да изнесем никакво количество, част от което да достигне C_{i+1} , като при това $\mathcal{A}_i$ остане добра.
3. Всяка критична редица е изолирана, но не всяка изолирана е критична. Поради това, критична редица е строго частен случай на изолирана редица.

Ето няколко малки примери.

Следният алчен алгоритъм решава ТРАНСПОРТИРАНЕ СЪС ЗАГУБИ.

TRANSPORT($[d_1 \dots d_{n-1}]$, $[x_1 \dots x_n]$, $[m_1 \dots m_n]$)

```

1   $d_n \leftarrow 0$ 
2   $s \leftarrow 0$ 
3  for  $i \leftarrow 1$  to  $n$ 
4       $\Delta \leftarrow x_i - m_i$ 
5      if  $s + \Delta \geq 0$ 
6           $s \leftarrow \max \{s + \Delta - d_i, 0\}$ 
7      else
8           $s \leftarrow s + \Delta - d_i$ 
9  if  $s \geq 0$ 
10     return TRUE
11 else
12     return FALSE
```

Теорема 15: Коректността на TRANSPORT

Алгоритъмът TRANSPORT връща TRUE тстк $\mathcal{A}_n$ е добра.

Доказателство: Следното твърдение е инвариант за **for**-цикъла (редове 3–8).

Инвариант 2: Цикълът на TRANSPORT

Всеки път, когато изпълнението на TRANSPORT е на ред 3:

- ① ако $s > 0$, то $\mathcal{A}_{i-1}$ е $(s + d_{i-1})$ -излишна,
- ② ако $s = 0$, то $\mathcal{A}_{i-1}$ е изолирана,
- ③ ако $s < 0$, то $\mathcal{A}_{i-1}$ е $|s + d_{i-1}|$ -дефицитна.

База. При първото изпълнение на ред 3, $i = 1$, така че $\mathcal{A}_{i-1}$ е празна. От една страна, празната редица от градове е изолирана в празния смисъл, защото е добра (всички нейни градове—няма такива—са удовлетворени), но количеството вода в нея не може да бъде намалено с никакво положително число, като тя остане добра. От друга страна, $s = 0$ заради присвояването на ред 2. ✓

Поддръжка. Да допуснем, че инвариантът е в сила за някое достигане на ред 3, което не е последното. Ще наричаме стойността на s в този момент “ s_{old} ”. И ще наричаме стойността на s след престоящото присвояване (било на ред 6, било на ред 8) “ s_{new} ”.

Започва изпълнението на тялото на цикъла. Δ получава стойност $x_i - m_i$ на ред 4.

Случай I: $s_{old} > 0$. По допускане, $\mathcal{A}_{i-1}$ е $(s_{old} + d_{i-1})$ -излишна. Това означава, че можем да доставим s_{old} литра вода в C_i като транспортираме $s_{old} + d_{i-1}$ литра от C_{i-1} към C_i , губейки d_{i-1} литра по пътя, като при това $\mathcal{A}_{i-1}$ остава добра. Но ако транспортираме по-голямо количество от C_{i-1} към C_i , $\mathcal{A}_{i-1}$ би станала дефицитна.

Случай I.1: $s_{old} + \Delta \geq 0$. В този подслучай се прави присвояването на ред 6. Тези два факта:

1. $\mathcal{A}_{i-1}$ е добра и C_i може да получи още s_{old} литра, като $\mathcal{A}_{i-1}$ остава добра.
2. $s_{old} + \Delta \geq 0$.

влекат, че $\mathcal{A}_i$ е добра. Нещо повече, $s_{old} + \Delta$ е максималното количество, което може да бъде транспортирано от C_i към C_{i+1} , като при това $\mathcal{A}_i$ остане добра. Ерго, $\mathcal{A}_i$ е $(s_{old} + \Delta)$ -излишна.

Случай I.1.a: Да допуснем, че $s_{old} + \Delta - d_i > 0$. Тогава $s_{new} = s_{old} + \Delta - d_i$. Ще докажем, че $\mathcal{A}_i$ е $(s_{new} + d_i)$ -излишна. Знаем, че $\mathcal{A}_i$ е добра и $(s_{old} + \Delta)$ -излишна. Забелязваме, че $s_{old} + \Delta = s_{old} + \Delta - d_i + d_i = s_{new} + d_i$ и заключаваме, че $\mathcal{A}_i$ е $(s_{new} + d_i)$ -излишна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $(s + d_{i-1})$ -излишна.

Случай I.1.b: Нека допуснем, че $s_{old} + \Delta - d_i \leq 0$. Тогава очевидно $s_{new} = 0$. Ние знаем, че $\mathcal{A}_i$ е добра в **Случай I**. Но очевидно $s_{old} + \Delta - d_i - \epsilon < 0$ за всяко $\epsilon > 0$. Заключаваме, че $\mathcal{A}_i$ е изолирана.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, че $\mathcal{A}_{i-1}$ е изолирана.

Случай I.2: Да допуснем, че $s_{old} + \Delta < 0$. Това означава, че $\Delta < 0$, понеже $s_{old} > 0$.

Но тогава $\mathcal{A}_i$ е дефицитна, защото, ако доставим в C_i повече от s_{old} литра от C_{i-1} , трябва да транспортираме повече от $s_{old} + d_{i-1}$ литра от C_{i-1} към C_i и тогава $\mathcal{A}_{i-1}$ ще стане дефицитна. Да си припомним, че по допускане, $\mathcal{A}_{i-1}$ е $(s_{old} + d_{i-1})$ -излишна.

За да направим $\mathcal{A}_i$ добра, трябва да транспортираме $|s_{old} + \Delta|$ литра—но не по-малко—в C_i от C_{i+1} . Тогава $\mathcal{A}_i$ е $|s_{old} + \Delta|$ -излишна по определение. Имайки предвид, че $s_{old} + \Delta = s_{old} + \Delta - d_i + d_i = s_{new} + d_i$, заключаваме, че $\mathcal{A}_i$ е $|s_{new} + d_i|$ -дефицитна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $|s_{new} + d_{i-1}|$ -дефицитна.

Случай II: $s_{old} = 0$. По допускане, $\mathcal{A}_{i-1}$ е изолирана. Това означава, че е добра, но не може да транспортираме дори един литър от нея към C_i , като при това тя остане добра. Следователно, състоянието на $\mathcal{A}_i$ зависи напълно от Δ и d_i .

Случай II.1: Да допуснем, че $s_{old} + \Delta \geq 0$. Тоест, $\Delta \geq 0$. Тогава се изпълнява присвояването на ред 6.

Случай II.1.a: Да допуснем, че $\Delta - d_i > 0$. Тогава $s_{new} = \Delta - d_i$. $\mathcal{A}_i$ е добра и можем да транспортираме Δ литра—но не повече—от C_i към C_{i+1} , като при това $\mathcal{A}_i$ остане добра. Тогава $\mathcal{A}_i$ е Δ -излишна, тоест, $(s_{new} + d_i)$ -излишна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $(s + d_{i-1})$ -излишна.

Случай II.1.b: Да допуснем, че $\Delta - d_i \leq 0$. Тогава $s_{new} = 0$. Ясно е, че $\mathcal{A}_i$ е добра, щом $\mathcal{A}_{i-1}$ е добра и $\Delta \geq 0$, но нито един литър не може да се транспортира от C_i към C_{i+1} , без $\mathcal{A}_i$ да стане дефицитна. Тогава $\mathcal{A}_i$ е изолирана.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е изолирана.

Случай II.2: Да допуснем, че $s_{old} + \Delta < 0$. Тоест, $\Delta < 0$. Тогава се изпълнява присвояването на ред 8, така че $s_{new} = \Delta - d_i$. Това е отрицателно количество, понеже $d_i > 0$. Ясно е, че $\mathcal{A}_i$ е дефицитна. Тя става добра, ако поне $|\Delta|$ литра—но не по-малко—бъдат доставени в C_i от C_{i+1} . Тогава $\mathcal{A}_i$ е $|\Delta|$ -дефицитна, тоест, $|s_{new} + d_i|$ -дефицитна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $|s_{new} + d_{i-1}|$ -дефицитна.

Случай III: $s_{old} < 0$. По допускане, $\mathcal{A}_{i-1}$ е $|s_{old} + d_{i-1}|$ -дефицитна. Това означава, че $\mathcal{A}_{i-1}$ е дефицитна и остава дефицитна, освен ако в C_{i-1} не бъдат доставени $|s_{old} + d_{i-1}|$ литра—но не по-малко—от C_i . За да доставим поне $|s_{old} + d_{i-1}|$ литра от C_i назад в C_{i-1} , трябва да транспортираме поне $|s_{old} + d_{i-1}| + d_{i-1}$ литра от C_i , за да компенсираме за загубата на d_{i-1} литра по пътя между C_i и C_{i-1} .

Твърдим, че $|s_{old}| > d_{i-1}$. За да се убедим в това, да забележим, че отрицателното количество s_{old} е било присвоено на s на ред 8 по време на предишното изпълнение на **for**-цикъла. Забележете, че предишно изпълнение е имало, инак s_{old} би било 0. Сега разглеждаме предишната итерация. За да може изпълнението да е било на ред 8, трябва $s + \Delta$ да е било отрицателно тогава. Но $-d_{i-1}$ на ред 8 е отрицателно, понеже разстоянията са положителни. Следователно, абсолютната стойност на това, което е било присвоено на s на ред 8 в предната итерация, е била строго по-голяма от d_{i-1} . С други думи, $|s_{old}| > d_{i-1}$.

Докажем, че $|s_{old}| > d_{i-1}$. Имайки предвид, че $s_{old} < 0$ и $d_{i-1} > 0$, очевидно $|s_{old} + d_{i-1}| = |s_{old}| - d_{i-1}$, откъдето $|s_{old} + d_{i-1}| + d_{i-1} = |s_{old}| - d_{i-1} + d_{i-1} = |s_{old}|$. И така, количеството вода, което трябва да се транспортира от C_i към C_{i-1} , за да стане $\mathcal{A}_{i-1}$ добра, е поне $|s_{old}|$. Ако е по-малко, $\mathcal{A}_{i-1}$ ще си остане дефицитна.

Случай III.1: Да допуснем, че $s_{old} + \Delta \geq 0$. Тогава се изпълнява присвояването на ред 6.

Случай III.1.a: Да допуснем, че $s_{old} + \Delta - d_i > 0$. Тогава $s_{new} = s_{old} + \Delta - d_i$. Щом s_{old} е отрицателно и $-d_i$ е отрицателно, трябва да е вярно, че Δ положително. Нещо повече,

трябва да е вярно, че $\Delta > |s_{old}| + d_i$.

Сега ще докажем, че $\mathcal{A}_i$ е $(s_{old} + \Delta)$ -излишна. Транспортираме $|s_{old}|$ литра от C_i към C_{i-1} , и по този начин $\mathcal{A}_{i-1}$ става добра. Но при това количеството вода в C_i спада на $s_{old} + \Delta$ литра; да си припомним, че s_{old} е отрицателно, така че $s_{old} + \Delta$ е по-малко от Δ , но е положително. Тъй като $|s_{old}|$ е граничното количество, няма начин да останем с повече от $s_{old} + \Delta$ литра в C_i и $\mathcal{A}_{i-1}$ да стане добра.

Тогава в C_i остава положителен “излишък” $s_{old} + \Delta$, а редицата $\mathcal{A}_{i-1}$ е добра, така че $\mathcal{A}_i$ е $(s_{old} + \Delta)$ -излишна. Но $s_{old} + \Delta = s_{new} + d_i$. Тогава $\mathcal{A}_i$ is $(s_{new} + d_i)$ -излишна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $(s + d_{i-1})$ -излишна.

Случай III.1.b: Да допуснем, че $s_{old} + \Delta - d_i \leq 0$. Тогава $s_{new} = 0$.

Ще покажем, че $\mathcal{A}_i$ е изолирана. Количеството вода в C_i е Δ . Но $\Delta \leq -s_{old} + d_i$, като $-s_{old}$ е положително число. В **Случай III**, редицата $\mathcal{A}_{i-1}$ е дефицитна. Транспортираме $|s_{old}|$ литра от C_i назад към C_{i-1} , правейки $\mathcal{A}_{i-1}$ добра. Знаем, че транспортирането на по-малко вода би оставило $\mathcal{A}_{i-1}$ дефицитна.

След транспортирането, C_i остава удовлетворен, имайки d_i литра в себе си, което е повече от $\Delta = x_i - m_i$. Заклучаваме, че $\mathcal{A}_i$ е добра след транспортирането. Обаче не е възможно да бъде направена $\mathcal{A}_{i-1}$ добра и C_i да бъде удовлетворен и да бъде доставено каквото и да е положително количество вода в C_{i+1} от C_i . Ерго, $\mathcal{A}_i$ е изолирана.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е изолирана.

Случай III.2: Да допуснем, че $s_{old} + \Delta < 0$. Изпълнява се присвояването на ред 8 и $s_{new} = s_{old} + \Delta - d_i$. Това е отрицателно количество, понеже $d_i > 0$.

Ще докажем, че $\mathcal{A}_i$ е $|s_{old} + \Delta|$ -дефицитна. Да си припомним, че $\mathcal{A}_{i-1}$ е дефицитна и тя остава дефицитна, освен ако поне $|s_{old}|$ литра не бъдат транспортирани в C_{i-1} . $\Delta = x_i - m_i$ може да е положително, нула, или отрицателно – това не знаем. От значение е фактът, че Δ , намалена с $|s_{old}|$, тоест $s_{old} + \Delta$, е отрицателно количество. Поради това, $\mathcal{A}_i$ е дефицитна. Нещо повече. За да направим $\mathcal{A}_i$ добра, количеството вода в C_i трябва да бъде увеличено с поне $|s_{old} + \Delta|$; всяко по-малко увеличение оставя $\mathcal{A}_i$ дефицитна.

За да се убедим, че последното е истина, забелязваме, че в текущите допускания, $\Delta - |s_{old}| + |s_{old} + \Delta| = 0$. На свой ред, това е истина, понеже

$$\forall x, y \in \mathbb{R}, \text{ такава че } x < 0 \text{ и } x + y < 0 : y - |x| + |x + y| = 0$$

Щом $|s_{old} + \Delta|$ е минималното количество вода, което трябва да се достави в C_i от C_{i+1} , за да стане $\mathcal{A}_i$ добра, то $\mathcal{A}_i$ е $|s_{old} + \Delta|$ -дефицитна. Но това е същото като $\mathcal{A}_i$ да е $|s_{new} - d_i|$ -дефицитна.

При следващото достигане на ред 3, i се инкрементира с 1. По отношение на новото i , вярно е, $\mathcal{A}_{i-1}$ е $|s_{new} + d_{i-1}|$ -дефицитна.

Терминация. Да разгледаме момента, в който изпълнението е на ред 3 за последен път. Ясно е, че $i = n + 1$.

Ако s в този момент е неотрицателно, то $\mathcal{A}_{i-1} = \mathcal{A}_n$ е или s -излишна, (припомняме си, че $d_n = 0$), или изолирана. И в двата случая е добра. Съответно, алгоритъмът връща TRUE на ред 9. Ако s е отрицателно, то $\mathcal{A}_{i-1} = \mathcal{A}_n$ е дефицитна. Съответно, алгоритъмът връща FALSE на ред 12. $\square$

2.2 Анализ на сложността

“Сложност на алгоритъм” е мярка за това, колко ресурси ползва този алгоритъм. “Сложен” в този смисъл означава “ползва много ресурси”. За какви ресурси става дума?

- Ресурсът, който най-често имаме предвид, е времето. Естествено, искаме алгоритмите ни да работят бързо. В този смисъл, “качествен алгоритъм” е алгоритъм, който работи бързо върху всички входове. Но “бързо” е разговорен термин. Прецизните разсъждения се базират на понятието “сложност по време” (на английски е *time complexity*), което ще разгледаме след малко. Засега само казваме, че алгоритъм с висока сложност по време е алгоритъм, който е бавен.
- Следващият по важност ресурс е паметта, която ползва алгоритъмът. Естествено, искаме алгоритмите ни да ползват малко памет. В този смисъл, “качествен алгоритъм” е алгоритъм, който ползва малко памет върху всички входове. Прецизните разсъждения се базират на понятието “сложност по памет” (на английски е *space complexity*), което ще разгледаме след малко. Засега само казваме, че алгоритъм с висока сложност по памет е алгоритъм, който ползва много памет.

Допълнение 11: Други ресурси освен времето и паметта

По правило учебниците по алгоритми не разглеждат други ресурси освен време и памет. Но, както е посочено в [125, стр. 156], има много други възможни ресурси, наречени там *complexity measures*, които можем да разгледаме. От особен интерес е така нареченият *ink*, което е броят на презаписвания на клетки от паметта. Този ресурс очевидно е свързан с енергията, която реален софтуер, базиран на алгоритъма, би ползвал върху реален компютър. Свързан с “мастилото”, но различен ресурс, е броят на достъпите до клетките на паметта. Разликата има практическо значение дотолкова, доколкото записването в паметта е по-енергоемко от четенето от паметта. Повече информация за тези ресурси и сложността, базирана на тях, има в [82] и [79].

Има елегантна, много обща дефиниция на Blum на “complexity measure”, която покрива времето, паметта и “мастилото”. Интересуваният се читател може да я намери в [125, стр. 156], като оригиналната статия е [22].

При разпределено изчисление (*distributed computation*) се говори за комуникационна сложност (*communication complexity*), измерваща количеството комуникация между независими изчисляващи агенти, но това е съвсем друг изчислителен модел от този, който сме приели в тези лекции, а именно със само един изчисляващ агент.

2.2.1 Големина на входа

Големина на входа на алгоритъм, или още размер на входа (на английски *input size*), е ключово понятие за дефинирането на сложността на алгоритмите, защото сложността е функция на размера на входа. Прецизното определение е следното.

Определение 13: Големина на входа – определение, което няма да използваме

За всеки алгоритъм, големината на входа за конкретен вход x е дължината на стринга, който кодира x в избраната система за кодиране.

Определение 13 е полезно за теория на сложността, която отчита и кодирането на данните. Тъй като вече приехме, че няма да разглеждаме кодирания, Определение 13 не върши работа за целите на този курс.

Терминът “големина на входа” ще дефинираме непрецизно, с голяма доза условност. Да си припомним Конвенция 1: всяка атомарна променлива се побира в единица памет, независимо от своето съдържание; с други думи, препълване на отредената памет не може да се получи, колкото и голямо да е съдържанието. Ако мислим за регистрите или думите в паметта на нашия абстрактен компютър, всеки регистър може да побере всяко число и всяка дума от паметта с произволен достъп може да побере всяко число. Да, това е математически абсурд и се опровергава тривиално с принципа на Dirichlet, но е удобно опростяване.

Определение 14: Големина на входа – непрецизно определение, което ще ползваме

Ако алгоритъмът е върху числен масив, то големината на входа е броят на числата в масива. Ако алгоритъмът е върху графи, големината на входа е сумата от броя на върховете и броя на ребрата. Това важи както за графи без тегла, така и за тегловни графи. Ако алгоритъмът е върху стрингове, големината на входа е броят на символите.

Това разбиране за големина на входа отговаря добре на модела, който вече приехме, в който числата са базови елементи, а елементарните операции върху тях стават за единица време. Практиката е показала, че това разбиране за големина на входа обикновено води до смислени резултати. Както ще видим нататък, има и изключения.

Конвенция 2: Буквата n в контекста на анализа на алгоритми

При анализа на алгоритми, по правило буквата n означава големината на входа. Изключенията са малко и ще се споменават изрично.

Сравнете Конвенция 2 с Конвенция 10 за графите.

Има едно изключение. Ако разглеждаме алгоритъм, чийто вход е квадратна матрица $n \times n$, или поне основната част от входа е такава матрица, сложността се изразява като функция на n . Но броят на елементите на матрицата е n^2 , а не n . Следователно, изразявайки сложността като функция на n , ние я изразяваме не чрез големината на входа, а чрез корен-квадратен от големината на входа[†].

Конвенция 3: Големина на входа при квадратни масиви

Прието е, когато входът или основната част на входа е квадратен масив $n \times n$, сложността на алгоритъма да се изразява като функция на n . Това нарушава правилото, според което сложността се изразява чрез големината на входа.

2.2.2 Въведение в сложността по време

Да разгледаме някакъв прост алгоритъм, например сортиращия алгоритъм INSERTION SORT. Псевдокодът, който ще разгледаме, е по учебника CLR (Cormen, Leiserson, Rivest) [34].

[†]Типичен пример е традиционният алгоритъм за умножение на матрици. Ако умножаваме квадратни, $n \times n$ матрици, казваме, че сложността е $\Theta(n^3)$, но алгоритъмът не е кубичен в размера на входа! Алгоритъмът всъщност има сложност $\Theta(N\sqrt{N})$, където N е големината на входа. За да разберете тази забележка под черта, трябва да знаете смисъла на Θ -нотацията, а тя се дефинира в Подсекция 2.3.3.

INSERTION SORT($A[1 \dots n]$: масив от естествени числа)

```

1  for i ← 2 to n
2      key ← A[i]
3      j ← i - 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j--
7      A[j + 1] ← key

```

Колко бързо работи INSERTION SORT? От практическа гледна точка, този въпрос е за физическото време, което отнема работата на някаква негова програмна реализация върху истински компютър. За да отговорим, трябва да можем да отговорим на тези въпроси:

1. За какъв вход става дума? Тук има два подвъпроса:
 - (а) Колко е n ? Ясно е, че колкото по-голямо е n , толкова повече време отнема изпълнението.
 - (б) За дадено n , какви са конкретните стойности на числата във входния масив[†]?

Този въпрос е на алгоритмично ниво. Следващите въпроси касаят по-ниското ниво на софтуерна имплементация.

2. Каква е конкретната програмна реализация? Според Skiena (вж. цитата на стр. 3), алгоритъмът е идеята зад някаква програма. Но всяка истинска програма е повече от идеята зад нея – тя е конкретна реализация на идеята на конкретен език за програмиране. Очевидно някои реализации са по-бързи от други, защото са по-грамотно написани. Компиляторът не може да компенсира неграмотно писане на софтуер.
3. Какъв е компилаторът?
4. Каква е виртуалната машина, ако има такава?
5. Каква е операционната система?
6. Каква е компютърната архитектура?
7. Какво е натоварването на компютъра в момента? По правило, компютрите изпълняват много процеси едновременно в *multitasking*, превключвайки бързо текущия активен процес и създавайки илюзия за истински паралелно изпълнение. Ерго, програмата, реализираща INSERTION SORT, най-вероятно няма да разполага с лукса да е единствената активна програма върху компютъра и времето за работата ѝ ще зависи и от другите активни програми.

Би трябвало да е очевидно, че е практически **невъзможно** е да пресметнем с точност до наносекунда времето, което конкретна програмна реализация на INSERTION SORT върху конкретна компютърна архитектура и операционна система в контекста на конкретно натоварване би отнела при изпълнение върху конкретен вход[‡].

[†]Лесно се забелязва, че INSERTION SORT работи най-бързо, ако входът вече е сортиран, например $[1, 2, 3, 4]$, и най-бавно, когато входът е сортиран обратно, например $[4, 3, 2, 1]$.

[‡]Странична забележка: съществуват специализирани компютри, изпълняващи критични дейности в *реално време*, при които наистина са необходими твърди гаранции за бързодействието на програмите, но това е далече отвъд материала в този курс.

Толкова прецизен отговор не е и необходим. Изчислителната техника се променя бързо, технологиите еволюират непрекъснато, така че, дори да имаме такъв суперпрецизен отговор с точност до наносекунда, много скоро той ще бъде нерелевантен.

В нашия курс, разсъждавайки за бързодействието, ние се фокусираме върху алгоритмите, а не върху програмните им реализации. Разликата между бързодействието на две програми за една и съща задача, които реализират два различни алгоритъма, по правило са много по-драстични от разликите в бързодействието на две програми, реализиращи един и същи алгоритъм.

Наблюдение

По отношение на сложните, нетривиални задачи, печелившата стратегия за бързодействието е подобрене на алгоритъма. Печалбата от по-бързия алгоритъм е толкова по-видима, колкото по-голям е входът.

Както вече споменахме в Конвенция 1, допускаме, че всяка елементарна инструкция на алгоритъма, който разглеждаме, се изпълнява за единица време. Инструкциите на истински процесор се изпълняват за различен брой тактове, но в нашата идеализирана изчислителна среда опростяваме нещата. По този начин, времето за изпълнение на алгоритъма върху някакъв вход е точно **броят на стъпките** по време на работата му. Ако при това допускане, за една и съща задача има два алгоритъма, ALG1 и ALG2, и ALG1 е по-бърз от ALG2, по правило, в реалния свят програмната реализация на ALG1 ще е по-бърза от тази на ALG2, като разликата ще е толкова по-очевидна, колкото по-голям е входът.

Като пример за прилагането на тези опростяващи допускания да разгледаме задачата СОРТИРАНЕ. Сортиращият алгоритъм INSERTION SORT, както ще видим от (2.23) на стр. 105, в най-лошия случай работи във време, пропорционално на n^2 . Сортиращият алгоритъм HEAPSORT, както ще видим на стр. 345, в най-лошия случай работи във време, пропорционално на $n \cdot \lg n$. Да разгледаме стойностите на n^2 и $n \cdot \lg n$ за няколко различни n . За целите на този пример логаритъмът е с основа 10.

n	10	100	1 000	100 000	1 000 000	100 000 000
n^2	100	10 000	1 000 000	10 000 000 000	1 000 000 000 000	10 000 000 000 000 000
$n \cdot \lg n$	10	200	3 000	500 000	6 000 000	800 000 000

Разбира се, числата в тази таблица не са точният брой елементарни инструкции; точният брой, както казахме, е пропорционален на тези числа. Но на практика тези коефициенти на пропорционалност не са големи. И така, при вход с размер сто милиона, огромната разлика между 10 000 000 000 000 000 и 800 000 000—**осем десетични порядъка**—води до това, че програмата, реализираща HEAPSORT, е **смазващо** по-бърза от програмата, реализираща INSERTION SORT, независимо от неща като използван компилатор, ниво на оптимизация на компилирането, операционна система, процесор, програмистки трикове за подобряване на скоростта и така нататък.

Току-що видяхме пример за това, че предимството, което дава по-бързият алгоритъм, доминира предимството на по-бързата компютърна технология. Съществуват много по-екстремни примери за предимствата, които дават бързите алгоритми. Тук е мястото на една неформална дефиниция: *наивен алгоритъм* е алгоритъм, който може да бъде предложен само от най-общи съображения и здрав разум, без използване на изтънчена теория; по правило терминът се използва в контекста на задачи, за които съществуват значително по-добри, но и по-неочевидни алгоритми. В Лекция 12 ще разгледаме множество интересни и важни

задачи, за които наивният алгоритъм се изпълнява в най-лошия случай за време, което е пропорционално на, да кажем, 2^n , докато за същата задача има по-съвършени алгоритми, чието време за изпълнение е пропорционално на, да кажем, n^2 . Разликата между нарастването на n^2 и 2^n е много по-драстична от разликата между $n \lg n$ и n^2 , която вече разгледахме. За илюстрация на разликата между n^2 и 2^n , да разгледаме, примерно, $n = 200$. От една страна, $200^2 = 40\,000$, което е нищо за съвременен компютър. От друга страна, $2^{200} = 1\,606\,938\,044\,258\,990\,275\,541\,962\,092\,341\,162\,602\,522\,202\,993\,782\,792\,835\,301\,376 \approx 1.6 \times 10^{60}$ е практически безкрайност; от изложението в Допълнение 12 следва, че 2^{200} като брой инструкции, които трябва да бъдат изпълнени, е **далече отвъд възможностите на всеки истински компютър**, както в момента, така и в бъдеще. Това заслужава да бъде повторено.

Забележка

В реалния свят, програма, чието цялостно изпълнение изисква 2^{200} стъпки, няма да завърши работата си **НИКОГА**, независимо от използваната хардуерна и софтуерна технология за реализация на платформата, върху която тази програма работи. Ограничението идва от фундаментални принципи на физиката и **НЕ МОЖЕ** да бъде заобиколено чрез никакви технологични иновации и подобрения.

Допълнение 12: Принцип на Landauer

Принципът на Landauer [103] казва, че има долна граница за енергията, необходима за обръщането на един бит от нула в единица или обратно, като функция на температурата, при условие, че изчислението е необратимо^a във времето. Този резултат свързва две много различни области на познанието: термодинамиката, която е дял на физиката, и теорията на цифровите изчислителните машини. Става дума за реални изчислителни машини, а не за изчислителни модели като машини на Turing и така нататък. Технологично, изчислителните машини може да са конструирани по много начини: с вакуумни лампи, с дискретни транзистори, с интегрални схеми, може да работят със светлина и да са базирани на фотонни превключватели, а в миналото е имало идеи да бъдат реализирани чрез релета или дори чисто механично. Принципът на Landauer е напълно независим от технологията на изчислителната машина, което го прави универсален, а долната граница, която той налага, е незаобиколима, както са незаобиколими принципите на термодинамиката. По думите на Bennett [13]:

In his classic paper, Rolf Landauer (1961) attempted to apply thermodynamic reasoning to digital computers. Paralleling the fruitful distinction in statistical physics between macroscopic and microscopic degrees of freedom, he noted that some of a computer's degrees of freedom are used to encode the logical state of the computation, and these information bearing degrees of freedom (IBDF) are by design sufficiently robust that, within limits, the computer's logical (i.e., digital) state evolves deterministically as a function of its initial value, regardless of small fluctuations or variations in the environment or in the computer's other non-information bearing degrees of freedom (NIBDF). While a computer as a whole (including its power supply and other parts of its environment), may be viewed as a closed system obeying reversible laws of motion (Hamiltonian or, more properly for a quantum system, unitary dynamics), Landauer noted that the logical state often evolves irreversibly, with two or more distinct logical states having a single logical successor. Therefore, because Hamiltonian/unitary

dynamics conserves (fine-grained) entropy, the entropy decrease of the IBDF during a logically irreversible operation must be compensated by an equal or greater entropy increase in the NIBDF and environment. This is Landauer's principle.

Съгласно принципа на Landauer, всяко битово обръщане при необратимо във времето изчисление “иска” като енергия в джаули поне

$$E = k_B \cdot T \cdot \ln 2$$

където k_B е константата на Boltzmann, приблизително равна на $1.38 \times 10^{-23} \text{ J K}^{-1}$, T е температурата на устройството в Келвинови градуси, а $\ln 2$ е, разбира се, натуралният логаритъм на двойката. По понятни причини, количеството $k_B \cdot T \cdot \ln 2$ е известно като *граница на Landauer*.

Авторът на лекционните записки признава, че познанията му в термодинамиката са нищожни, но въпреки това на най-общо ниво има нещо като разбиране за същността на принципа на Landauer. Принципът е базиран на понятието *ентропия*, която в някакъв смисъл е *мярка за неподредеността в дадена система*. Изчисление, което е необратимо във времето, както всеки процес, който е необратим във времето, води до нарастване на ентропията, според термодинамиката. Количеството $k_B \cdot \ln 2$ съответства на нарастването на ентропията при обръщане на един бит, а произведението от нарастването на ентропията и температурата има смисъл на енергия.

На пръв поглед, ако температурата е нула, границата на Landauer също става нула. Но на практика това няма как да стане. Температурата не може да бъде нула. В нашата Вселена има *реликтовото лъчение*, остатък от Големия взрив, изотропно електромагнитно лъчение, пронизващо пространството. Дори само заради реликтовото лъчение, температурата на тяло⁶ никога няма да бъде нула, ако ще тялото да е далече от Земята, извън Слънчевата система или дори извън Млечния път.

Ако вземем нереалистично ниската стойност от един келвинов градус за температура на нашата изчисляваща машина, имаме долна граница от около 10^{-23} джаула енергия за едно битово обръщане. За 10^{60} битови обръщания тази долна граница става около 10^{37} джаула. Това е приблизително равно на енергията, която Слънцето излъчва в продължение на *хиляда години*. Не енергията от Слънцето, която попада върху Земята, а **цялата енергия**, която нашата звезда произвежда. И тъй като $10^{60} \approx 2^{200}$, то за 2^{200} битови обръщания се иска енергия **поне** колкото една звезда генерира за хиляда години. Ако увеличим малко степенния показател, да кажем 2^{250} , магнитуда на енергията, която се иска за толкова битови обръщания нараства до около 10^{52} джаула. Това е много повече от 10^{45} джаула, енергията, *която се отделя при избухване на супернова*, и е от порядъка на енергията, *отделяна при експлозии на галактични ядра*. На читателя остава да сметне за коя стойност на n е вярно, че 2^n битови обръщания при един келвинов градус на компютъра искат енергия поне 10^{69} джаула съгласно принципа на Landauer, където 10^{69} джаула е *тоталната маса-енергия в наблюдаемата Вселена*.

Има доста автори, които са критични към принципа на Landauer, както посочва Bennett [13]. Някои автори отричат връзката между физически величини като топлина абстрактно-математически свойства като обратимост на логически операции. Други автори посочват, че принципът на Landauer е в сила само при изчисления, които са необратими във времето, а принципно е възможно да се реализират изчисления, които са обратими. Въпреки че дебатите продължават, преобладаващото мнение е, че принци-

път на Landauer е валиден, а освен това има и негови експериментални потвърждения [85].

По правило учебниците по алгоритми още в уводната глава показват нагледно зашеметяващото нарастване на функции като 2^n и $n!$ и показват убедително, че дори ако даден реален компютър изпълнява много повече елементарни операции в секунда от най-добрите днешни образци, той е безнадеждно бавен за програми, искащи от порядъка на 2^n или $n!$ стъпки, дори за скромни стойности на n . Като пример за това, вижте Problem 1-1 в [34, стр. 14]. По същество, този тип аргумент казва “Няма да ви стигне времето!”. Аргументацията с принципа на Landauer казва друго: “Няма да ви стигне енергията!”.

Добре известният *закон на Moore* казва, че гъстота на гейтовете в интегралните схеми се удвоява на горе-долу 18 месеца, което се интерпретира и като “производителността на компютрите се удвоява на 18 месеца”. Дори хора, които не са виждали това твърдение формулирано в явен вид, мислят, че технологичният прогрес увеличава производителността с някаква мултипликативна константа на единица време⁶ и това винаги ще е така. Ако това е неизменно вярно, то в недалечно бъдеще наистина ще има компютри, способни да извършат и 2^{200} стъпки, и 2^{2000} стъпки. Но принципът на Landauer казва, че последното е невъзможно, от което следва, че експоненциалното нарастване на производителността ще спре с достигането на ограничения, произтичащи от фундаменталните закони на физиката.

Читателят, интересуваш от физическите ограничения върху компютърната производителност, може да намери статията на Lloyd [109] за особено информативна, защото тя разглежда най-различни ограничения върху производителността, а не само това, произтичащо от принципа на Landauer. Примерно, там се твърди, че 10^{50} елементарни операции в секунда е горна граница, произтичаща от закона на Einstein $E = mc^2$.

⁴На английски терминът за “необратимо във времето изчисление” е *time-irreversible computation*. Това означава изчисление, при което не може да определим входните стойности от резултата. Примерно, елемент тип конюнкция, на английски *AND-gate*: ако резултатът е 0, не може да кажем дали входът е бил (0, 0) или (0, 1) или (1, 0).

⁶Става дума за тяло, “оставено на мира”. В някои лабораторни експерименти тела придобиват температура, много близка до нула келвинови градуса, но за това “изпомпване на топлина” се иска енергия, и то много.

⁷Това е същото като експоненциален растеж, защото сумата на геометрична прогресия е експоненциална функция.

2.2.3 Определение на сложността по време

Сложността по време е функция на големината на входа. Но има много входове с една и съща големина. По-лошо, в нашия опростен модел, в който всяко число има един и същи размер (единица), има **безброй много** входове за всяка големина на входа[†]. Да разгледаме отново сортиращ алгоритъм. Да кажем, че сортираме цели числа. Множеството от входовете с големина 1 е $\mathbb{Z}$. Множеството от входовете с големина 2 е $\mathbb{Z}^2$. Множеството от входовете с големина 3 е $\mathbb{Z}^3$. И така нататък. В общия случай, множеството от входовете с големина n

[†]При машините на Turing (Определение 120), които са друг изчислителен модел, няма проблеми с дефинирането на “големина на входа”. Големината на входа на машина на Turing е дължината на входния стринг и за всяка големина на входа n , броят на различните входове е $|\Sigma|^n$, където Σ е входната азбука съгласно Определение 120.

е $\mathbb{Z}^n$. Виждаме, че множеството от входовете е (изброимо) безкрайно за всяко n . Следните разсъждения ни позволяват да разглеждаме само краен брой входове за този алгоритъм.

Конвенция 4: Редиците са само крайни

За целите на тези лекционни записки, “редица” означава по подразбиране “крайна редица”. Причината е, че алгоритъм не може да има вход безкрайна редица. Тъй като редиците са крайни, ще се въздържаме от използване на “вектор” като синоним на “крайна редица”, освен в изключителни случаи, например в контекста на булевите функции.

Определение 15: Подобни редици

За всяко $n \in \mathbb{N}^+$, две редици от $\mathbb{Z}^n$ наричаме *подобни*, ако съдържат един и същи брой от минималния елемент, един и същи брой от следващия по големина елемент, и така нататък, един и същи брой от максималния елемент.

Като пример, редиците $(55, 6, 55, 7075, 6)$ и $(36, 36, 37, 37, 38)$ са подобни, защото

- и в двата има два минимални елемента, а именно 6 и 6 в X и 36 и 36 в Y ,
- и в двата има два средни по големина елемента, а именно 55 и 55 в X и 37 и 37 в Y ,
- и в двата има един максимален елемент, а именно 7075 в X и 38 в Y .

А редиците $(55, 6, 55, 7075, 6)$ и $(36, 36, 37, 37, 37)$ не са подобни. Очевидно е, че всеки две редици, в които няма повтаряне на елементи, са подобни.

Определение 16: Съществено подобни и съществено различни редици

За всяко $n \in \mathbb{N}^+$, две редици от $\mathbb{Z}^n$ наричаме *съществено подобни*, ако са подобни и освен това всички появи на минималния елемент и в двете са на едни и същи позиции, всички появи на следващия по големина елемент и в двете са на едни и същи позиции, и така нататък, всички появи на максималния елемент и в двете са на едни и същи позиции. Две редици с еднаква дължина, които не са съществено подобни, са *съществено различни* независимо от това дали са подобни или не.

Като пример, редиците $(55, 6, 55, 7075, 6)$ и $(36, 36, 37, 37, 38)$ не са съществено подобни, въпреки че са подобни, понеже минималният елемент в първата—той е 6—е на позиции 2 и 5, а минималният елемент във втората—той е 36—е на позиции 1 и 2. Ерго, $(55, 6, 55, 7075, 6)$ и $(36, 36, 37, 37, 38)$ са съществено различни. $(55, 6, 55, 7075, 6)$ и $(36, 36, 36, 36, 36)$ също са съществено различни. Редиците $(6, 6, 55, 55, 7075)$ и $(36, 36, 37, 37, 38)$ са съществено подобни. Понятията “подобни” и “съществено подобни” не са приложими за редици с различни дължини.

Определение 17: Релация $\hat{R}_n$ на неразличимост върху целочислени редици

За всяко $n \in \mathbb{N}^+$, за всеки $X, Y \in \mathbb{Z}^n$, $X \hat{R}_n Y$ тогава и само тогава, когато X и Y са съществено подобни.

Тривиално се показва, че за всяко n , $\hat{R}_n$ е релация на еквивалентност. Нейните класове на еквивалентност са само краен брой за всяко n . Интересен въпрос е колко точно са тези класове

на еквивалентност. Ако не допускаме повтаряне на елементи, то класовете на еквивалентност са $n!$, защото всеки клас се определя от това на коя позиция е минималният елемент, на коя е следващият по големина, и така нататък, на коя позиция е максималният елемент, така че просто броим пермутациите.

Ако обаче допускаме повтаряне на елементи, класовете на еквивалентност се броят от така наречените *ordered Bell numbers*; на български можем да ги наречем “числа на Bell с наредба”. n -тото число на Bell с наредба е значително по-голямо от $n!$, но извеждането на точна формула е сравнително сложно, затова го отделяме от основния текст в Допълнение 13.

Допълнение 13: P_n : числата на Bell с наредба

Искаме да преброим по колко начина могат да бъдат ранкирани n елемента, които са два по два различни, но ранговете им не са непременно различни. Тези комбинаторни обекти се наричат *weak orderings*; на български нека кажем “слаби наредби”. Knuth ги нарича “weak orderings” в главата за сортиране в ТАОСР [94, стр. 194, задача 3]. Като брой те са точно колкото пълните преднаредби (Определение 35 в Допълнение 18); неслучайно формалното определение на задачата СОРТИРАНЕ ползва “пълна преднаредба” (Задача 13).

Нека броят на слабите наредби на n елемента е P_n . Ето две (еквивалентни) формули за P_n , които се извеждат с различни съображения.

P_n , изразено чрез числата на Stirling от втори род. Да си припомним, че $\{n \atop k\}$ за $1 \leq k \leq n$ е броят на разбиванията на n -елементно множество на точно k подмножества; четете се “числото на Stirling от втори род n -подмножество- k ”. Повече за тези числа има в Подсекция 12.3.2, където са показани и ефикасни алгоритми за пресмятането им.

И така, дадени са n елемента, но някои може да имат еднакви големина (при което си остават различни елементи). Нека броят на различните големина е k ; в екстремните случаи, $k = 1$ означава всички елементи да имат една и съща големина, а $k = n$ означава да няма еднакви големина. За фиксирано k , по $\{n \atop k\}$ начина може да бъдат групирани числата в точно k класа, като числата във всеки клас са с еднаква големина. Например, $\{4 \atop 2\} = 7$, и наистина има точно 7 начина a_1, a_2, a_3 и a_4 да бъдат групирани в два класа; тоест, да имат точно две различни големина:

a_1, a_2 и a_3 с една големина, a_4 с друга големина
 a_1, a_2 и a_4 с една големина, a_3 с друга големина
 a_1, a_3 и a_4 с една големина, a_2 с друга големина
 a_2, a_3 и a_4 с една големина, a_1 с друга големина
 a_1 и a_2 с една големина, a_3 и a_4 с друга големина
 a_1 и a_3 с една големина, a_2 и a_4 с друга големина
 a_1 и a_4 с една големина, a_2 и a_3 с друга големина

За всяко различно групиране на елементите в k класа, тези класове може да бъдат подредени линейно по $k!$ начина. Например, при $n = 4$ и $k = 2$, има точно $14 = \{4 \atop 2\}2!$

слаби наредби на a_1, a_2, a_3 и a_4 , които можем да опишем така:

$$a_1 = a_2 = a_3 < a_4$$

$$a_4 < a_1 = a_2 = a_3$$

$$a_1 = a_2 = a_4 < a_3$$

$$a_3 < a_1 = a_2 = a_4$$

$$a_1 = a_3 = a_4 < a_2$$

$$a_2 < a_1 = a_3 = a_4$$

$$a_2 = a_3 = a_4 < a_1$$

$$a_1 < a_2 = a_3 = a_4$$

$$a_1 = a_2 < a_3 = a_4$$

$$a_3 = a_4 < a_1 = a_2$$

$$a_1 = a_3 < a_2 = a_4$$

$$a_2 = a_4 < a_1 = a_3$$

$$a_1 = a_4 < a_2 = a_3$$

$$a_2 = a_3 < a_1 = a_4$$

Щом k може да взема стойности от 1 до n , броят на класовете на еквивалентност на $\hat{R}_n$ е

$$P_n = \sum_{k=1}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\} k! \quad (2.20)$$

Примерно, $P_4 = 75$.

Само $\sum_{k=1}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ е броят на разбиванията на n -елементно множество. Това количество се нарича *число на Bell* и се бележи с " B_n ". Ясно е защо казваме "число на Bell с наредба" за $P_n = \sum_{k=1}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\} k!$: защото разглеждаме и наредбите на дяловете на разбиване.

P_n , изразено чрез рекурентно уравнение. В сила е

$$P_n = \begin{cases} 1, & \text{ако } n = 0 \\ \sum_{k=1}^n \binom{n}{k} P_{n-k}, & \text{ако } n \geq 1 \end{cases}$$

Ще докажем това с комбинаторни разсъждения. Ако $n = 0$, няма елементи за подреждане в слаба наредба и има една такава наредба, а именно празната. Нека $n \geq 1$. Слабите наредби на n елемента се разбиват по броя k на елементите с най-малка големина, където $k \in \{1, \dots, n\}$. Да кажем, че стойността на най-малката големина е m . За всяко фиксирано k , по $\binom{n}{k}$ начина можем да изберем кои елементи да са с големина m , а за всеки такъв избор има P_{n-k} слаби наредби на останалите $n - k$ елемента, като обаче всички големина на останалите надвишават m .

Независимо от начина на извеждане, $P_0 = 1, P_1 = 1, P_2 = 3, P_3 = 13, P_4 = 75, P_5 = 541, P_6 = 4683, P_7 = 47293$ и така нататък. Това е редица [A000670](#) в онлайн енциклопедията на целочислените редици.

Да продължим с разсъжденията за броя на входовете на сортирац алгоритъм. Разглеждаме

само такива сортирания, които се базират на сравнения на числата. Съществуват и други възможности за сортиране, ако са дадени някакви ограничения за входа; например, ако сортираме само нули и единици, може просто да преброим колко са нулите. Но ние тук разгледаме само сортирания, базирани на сравнения. Лесно се вижда, че ако сортирането е базирано на сравнения, то сортирането на следните входове

$(2, 1, 3)$

$(2000, 1000, 3000)$

$(2 \times 10^9, 1, 10^{10^{10}})$

ще протече по един и същи начин, и то в много силен смисъл. Да си представим произволен сортиращ алгоритъм, базиран на сравнения, и три негови копия, всяко от които е пуснато върху един от тези три входа, като и трите копия “живеят” в едно и също дискретно време. Очевидно е, че и трите копия ще завършат работата си за еднакъв брой стъпки, като на всяка стъпка ще се изпълнява една и съща инструкция и от трите копия[†]. Защо е така? Защото тези три входа са съществено подобни.

Това е в сила за всякакви множества от входове, които са съществено подобни.

Наблюдение 11: Неразличимост на съществено подобни входове

Всеки сортиращ алгоритъм, базиран на сравнения, работи по един и същи начин върху съществено подобни входове в смисъл, че изпълнението отнема един и същи брой стъпки и във всеки момент от изпълнението, една и съща инструкция е текущата.

Естествено, не твърдим, че алгоритъмът връща едно и също! Ако входът е $(2, 1, 3)$, алгоритъмът връща $(1, 2, 3)$, а ако входът е $(2 \times 10^9, 1, 10^{10^{10}})$, алгоритъмът връща $(1, 2 \times 10^9, 10^{10^{10}})$.

Наблюдение 12: Краен брой съществено различни начини на работа на сорт. алг

По отношение на кой да е сортиращ алгоритъм, базиран на сравнения, за всеки два входа с големина n , които са от един и същи клас на еквивалентност на $\hat{R}_n$, алгоритъмът работи по един и същи начин. Тогава броят на различните начини, по които може да работи алгоритъмът, е не повече от P_n , дефинирано в Допълнение 13.

Наблюдение 12 може да бъде пренесено и при другите изчислителни задачи и техните алгоритми, които ще разгледаме в тези лекции, но само концептуално. Примерно, трудно е[‡] да намерим точна формула за броя на съществено различните входове за алгоритъма на Kruskal или алгоритъма на Dijkstra. Но трябва да е ясно, че за всеки алгоритъм, който ще разгледаме, безкрайното множество от входове с големина n може да бъде разбито на краен брой класове, върху всеки от които алгоритъмът работи по един и същи начин.

[†]Разбира се, това е при нереалистичното допускане, че $10^{10^{10}}$ има размер единица и действията върху него стават за единица време.

[‡]Това е меко казано. Авторът на записките няма представа как да се подходи.

Определение 18: Сложност по време

Нека P е изчислителна задача и A е алгоритъм за нея. За всяка големина на входа $n \in \mathbb{N}^+$, нека $\mathcal{I}(n)$ е крайното множество от съществено различните входове с големина n . За всеки вход κ , нека $f(\kappa)$ е броят стъпки, които се изпълняват от $A(\kappa)$. Тогава, за всяко n , *сложността по време на A в най-лошия случай* е

$$T_A(n) = \max \{f(\kappa) \mid \kappa \in \mathcal{I}(n)\}$$

сложността по време на A в най-добрия случай е

$$P_A(n) = \min \{f(\kappa) \mid \kappa \in \mathcal{I}(n)\}$$

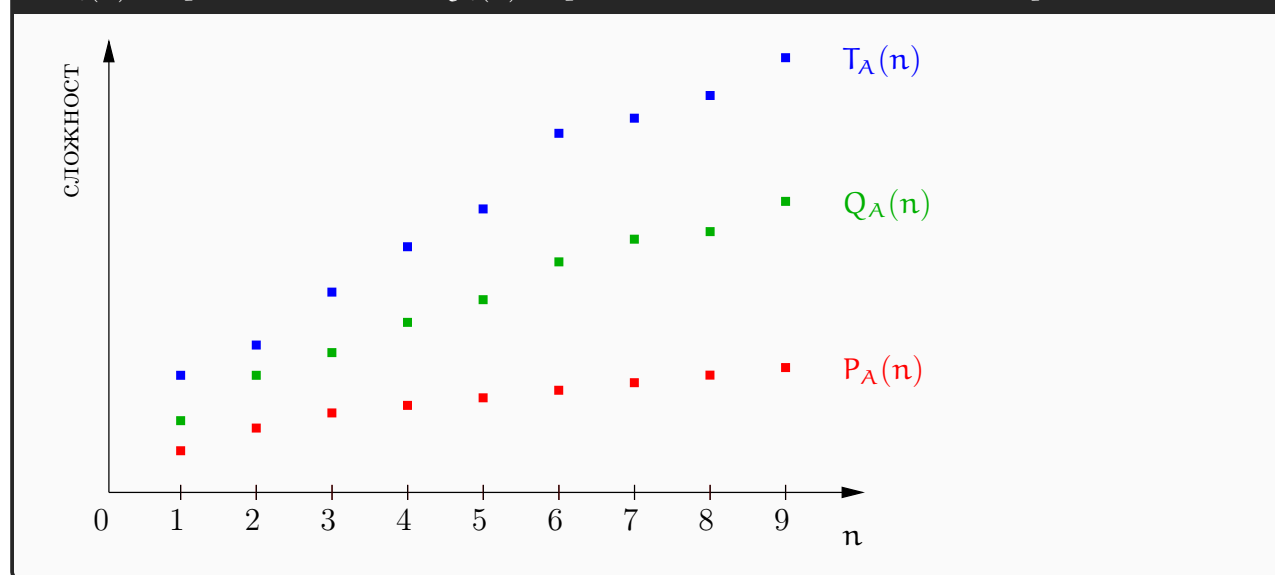
и *средната сложност по време на A* е

$$Q_A(n) = \frac{1}{|\mathcal{I}(n)|} \sum_{\kappa \in \mathcal{I}(n)} f(\kappa)$$

Забележка: използваната нотацията, например " $T_A(n)$ " и т. н., не е общоприета.

Фигура 2.1 показва нагледно трите вида сложност.

Фигура 2.1 : Сложност по време в най-лошия случай $T_A(n)$, в най-добрия случай $P_A(n)$ и средна сложност $Q_A(n)$. Ординатата е сложността като брой стъпки.



Средната сложност винаги е между сложността в най-лошия и сложността в най-добрия случай. По правило функциите на сложността са строго нарастващи, защото (по правило) един и същи алгоритъм работи по-бавно върху по-голям вход, но има и изключения, както ще видим надолу в примера с Евклидовия алгоритъм.

На практика най-често използвана е сложността по време в най-лошия случай (*worst-case time complexity* на английски), по две основни причини. Първо, изследването на сложността в най-лошия случай е много по-лесно от изследването на средната сложност, в което ще се убедим в Лекция 6 при анализа на QUICKSORT. Анализирането на средната сложност изисква значително по-задълбочени математически познания и значително по-сложни техники, дори

при неявното допускане в определението на $Q_A(n)$, че всички входове с дадена големина са еднакво вероятни. Втората причина е, че резултатът за най-лошия случай е твърда гаранция (че по-лошо не може да бъде), за разлика от резултата за средния случай.

Сложността по време в най-добрия случай почти не се ползва. Причината е, че сложността по време в най-добрия случай на всеки[†] алгоритъм може да бъде подобрена с един прост трик, който е формално валиден, но е напълно безполезен на практика. Да разгледаме като пример задачата СОРТИРАНЕ (Задача 13). Трикът, по отношение на сортирането, е този: в самото начало, алгоритъмът проверява дали някоя конкретна пермутация на входните елементи дава наредба в ненамаляващ ред. Точно коя пермутация бива проверена, няма значение. Да кажем, че проверката да е дали

$$a_1 \leq a_2 \leq \dots \leq a_n$$

Ако това се окаже истина, то входът вече е сортиран и няма какво повече да се прави, така че алгоритъмът приключва работата си. По този начин сложността по време в най-добрия случай става $\Theta(n)$ (Подсекция 2.3.3), защото е възможно входът да е сортиран поначало. Ако не е сортиран поначало? Тогава тази проверка се оказва излишна и алгоритъмът продължава със същинското сортиране. В този курс ще разгледаме бързи сортиращи алгоритми, работещи във време $\Theta(n \lg n)$, и не толкова бързи сортиращи алгоритми, работещи във време $\Theta(n^2)$. Ключовото наблюдение е, че гореспоменатият трик може да бъде приложен към **всеки** от тях, така че всеки сортиращ алгоритъм да има сложност по време в най-добрия случай $\Theta(n)$. С което става напълно безсмислено да изследваме тази сложност по време: щом сортиращите алгоритми лесно могат да станат еднакво бързи в най-добрия случай, защо изобщо да разглеждаме тази мярка?

Нарекохме този трик “безполезен на практика” по причина, която може да бъде обяснена на езика на вероятностите. Ако входът е случаен, вероятността да се окаже сортиран е пренебрежимо малка и клони много бързо към нула с нарастването на n . Ерго, вредата от новодобавените $n - 1$ проверки за софтуерната реализация надхвърля ползата от тях, във вероятностния смисъл[‡].

2.2.4 Сложност по памет

Сложността по памет на даден алгоритъм A върху даден вход е броят на елементите памет, които A ползва, без да броим паметта, в която се разполага входа и паметта, в която се разполага изходът. С други думи, гледаме само **работната памет** на алгоритъма; това е съвкупността от променливите му, които се ползват, за да се получи изходът от входа. Сложност по памет в най-лошия, средния и най-добрия случай се дефинира по начин, аналогичен на сложността по време.

Една значителна принципна разлика между сложността по време и сложността по памет е, че сложността по време—било в най-лошия случай, било средната—по правило е строго растяща функция на големината на входа, докато е напълно възможно даден алгоритъм да ползва само константна работна памет за **всяка** големина на входа. Алгоритми, които ползват константна работна памет, се наричат *in-place*.

[†]“Всеки” може да е прекалено силно, но със сигурност всеки алгоритъм, който ще разгледаме в този курс.

[‡]Аналогично, във вероятностния смисъл, финансовата вреда от хазарта надхвърля финансовата полза от хазарта. Става дума за играещия на хазарт; за организатора на хазарта, ползата е очевидно голяма.

Наблюдение 13: Сложността по време ограничава отгоре сложността по памет

Сложността по време не може да е по-малка от сложността по памет. Нека A е произволен алгоритъм, $T_A(n)$ е сложността му по време в най-лошия случай, а $S_A(n)$ е сложността му по памет в най-лошия случай. Тогава, $\forall n : S_A(n) \leq T_A(n)$. Причината е, че за да извърши достъп до дадена променлива от работната си памет, A трябва да използва поне единица от дискретното си време; достъпът не може да стане за нула време.

Допълнение 14: Сл. по време ограничава сл. по памет отгоре и отдолу

В нашия не особено реалистичен изчислителен модел числата имат размер на представянето единица и същевременно могат да са произволно големи. Това позволява сложността по памет на някои алгоритми да е произволно по-малка от сложността по време, като в екстремния случай, сложността по памет е константна.

В контраст с това, при машините на Turing (Определение 120) сложността по памет не може да е произволно по-малка от сложността по време по простата причина, че ако машината повтори конфигурация, то тя ще я и потрети и така нататък и няма да спре никога. Конфигурацията на машина на Turing (Определение 121) се определя от състоянието и съдържанието на лентата (и позицията на главата, но да оставим това настрана за момента, то не е съществено). Състоянията са константен брой. Съдържанието на лентата е крайно, въпреки че лентата е безкрайна; съдържанието е само тази крайна (и непрекъсната) част от лентата, която е била достъпвана от машината по време на работата. Ако лентовата азбука е Γ и съдържанието на лентата има дължина n , от елементарни комбинаторни съображения следва, че има най-много $|\Gamma|^n$ различни възможности за съдържанието на лентата. Ерго, ако машината е работила поне $|\Gamma|^n + 1$ стъпки, тя е имала едно и също съдържание на лентата в различни моменти съгласно принципа на Dirichlet.

Ако машината спира върху всеки вход, тя не може да повтаря конфигурации. Оттук следва, че има константи c_1 и c_2 , такива че броят стъпки на машината за всеки вход с размер n е ограничен отгоре от $c_1^{c_2 \cdot L(n)}$, където $L(n)$ е максимумът на размера на съдържанието на лентата при работа върху входове с размер n .

Наблюдение 14

В изчислителни модели, отчитащи адекватно сложността по памет, сложността по време е ограничена отгоре от някаква експоненциална функция на сложността по памет. В такива модели, in-place алгоритми са някаква екзотика: за да е in-place алгоритъм в такъв модел, трябва да не чете входа изобщо или да чете само константна част от входа или, ако чете целия вход, да “запомня” само константна по размер информация, примерно четността на булев стринг.

Практически полезни алгоритми, които са in-place, като INSERTION SORT например, може да има само в изчислителен модел, който прави нереалистични допускания за паметта.

В нотацията на Наблюдение 13, $T_A(n) \leq c_1^{c_2 \cdot S_A(n)}$. Иначе казано, използвайки Θ -нотацията (Подсекция 2.3.3),

$$\Theta(\lg T_A(n)) \leq S_A(n) \leq T_A(n)$$

2.2.5 Проблеми със сложността по време, до които води нашият модел

Да разгледаме отново Евклидовия алгоритъм в модерната му формулировка от Подсекция 1.2.2, независимо дали в рекурсивния или итеративния вариант. Да се опитаме да направим груб анализ на сложността му по време в най-лошия случай. Очевидно, за някои двойки входни числа, а именно тези, за които a е кратно на b , алгоритъмът ще завършва много бързо. За други двойки входни числа алгоритъмът ще работи много повече. От [93] става ясно, че за всяко цяло положително число m , колкото и голямо да е, има двойка (a, b) , такава че Евклидовият алгоритъм с вход (a, b) извършва повече от m стъпки. Но в нашия опростен модел, всички двойки (a, b) имат един и същи размер (на представянето), а именно 2, защото се състоят от две числа, а ние приехме, че всяко число има размер единица. Тогава всеки вход има размер точно 2. Следва, че функциите на сложността, които имат аргумент-размера на входа, не могат да бъдат дефинирани смислено, защото имаме само една големина на вход, а именно 2. За тази големина имаме входове, за които алгоритъмът работи в повече стъпки от всяко предварително избрано число. Излиза, че сложността по време е безкрайност ...

Този парадоксален извод се дължи само на допускането, че всяко число има размер единица. Ако приемем обаче по-сложния модел, в който всяко $k \in \mathbb{N}^+$ има размер, пропорционален на $\log_2 k^\dagger$, парадоксът изчезва. Тогава Евклидовият алгоритъм работи във време, пропорционално на квадрата на размера на входа (вж. [93]).

Както вече казахме в Подсекция 1.1.5, моделите налагат някакви опростявания, иначе нямаше да са модели. Доколкото тези опростявания са смислени и полезни на практика, съответните модели също са смислени и полезни. Но в случаите, в които опростяванията и допусканията водят до безсмислени и безполезни резултати, съответните модели стават неизползваеми и трябва да се търсят други модели, по-сложни и детайлни. Така че моделът, в който всяко число има големина единица, не е иманентно лош или сбъркан, просто в някои случаи е полезен (например, сортиращите алгоритми), в други, не е (например, Евклидовият алгоритъм).

Аналогично, пътната карта на България е модел на истинската пътна мрежа, защото съдържа само най-важната информация, а не цялата информация. Тази карта, в която градовете са отбелязани с точки, е много полезна в някои случаи и напълно безполезна в други случаи. Ако искаме да стигнем с кола от София до Варна, например, и не знаем пътя, пътната карта, в която Варна е точка, е полезна до момента, в който влезем във Варна. За отиването до конкретен адрес във Варна, тази картата вече не върши работа. И така, в някои случаи въпросната карта е полезен модел, в други случаи – не. По същия начин, моделът, в който числата имат големина единица, е полезен в някои случаи и безполезен в други.

2.3 Асимптотични нотации

2.3.1 Опит за намиране на точната сложност по време на прости алгоритми

Да разгледаме отново алгоритъма INSERTION SORT, въведен на стр. 91. Въпреки че в Подсекция 2.2.2 установихме, че е безсмислено да търсим точен израз за сложността му по време като функция от големината на входа, с учебна цел сега ще се опитаме да направим точно

[†] $\log_2 k$ е горе-долу броят на битовете, необходими за записването на n в двоична позиционна бройна система.

това. Работим под допускането, че числата имат големина единица и че всяка елементарна инструкция отнема единица време.

INSERTION SORT($A[1..n]$: масив от цели числа)

```

1  for i ← 2 to n
2      key ← A[i]
3      j ← i - 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j--
7      A[j + 1] ← key

```

Колко стъпки отнема изпълнението на този алгоритъм върху n числа? Използваме следните съображения.

- В нашия опростен модел това, което е на ред 1—а именно инициализацията на управляващата променлива i , проверката за продължаване и инкрементирането—е една инструкция. Тя се изпълнява n пъти. Да поясним защо ред 1 се изпълнява n пъти. По принцип, всяка инструкция от вида `for i ← a to b` се изпълнява точно $b - a + 2$ пъти (ако $a \leq b$), защото:
 - ♦ тялото на цикъла се изпълнява точно $b - a + 1$ пъти,
 - ♦ а `for i ← a to b` се изпълнява веднъж за всяко изпълнение на тялото цикъла и после **още веднъж**, когато условието за продължаване вече не е вярно.

В този случай, $n - 2 + 2$ е точно n .

Допълнение 15: Цикъл for на ниво машинни инструкции

Върху реален компютър, **for**-цикъл се реализира на ниско ниво горе-долу по следния начин. Разгледайте този фрагмент от програма на C:

```
for (i=2; i <= n; i++);
```

Той се компилира върху Intel iCore3 процесор, Ubuntu Linux, 64 битов gcc компилатор без оптимизация, в следния код на асемблер (адресите и отместванията, естествено, варира):

```

0x4005bb <main+30> movl $0x2,-0x4(%rbp)
0x4005c2 <main+37> jmp 0x4005c8 <main+43>
0x4005c4 <main+39> addl $0x1,-0x4(%rbp)
0x4005c8 <main+43> mov -0x8(%rbp),%eax
0x4005cb <main+46> cmp %eax,-0x4(%rbp)
0x4005ce <main+49> jle 0x4005c4 <main+39>

```

Както виждаме, в реалния свят инкрементирането с единица `addl`, проверката за край на изпълнението на цикъла `cmp`, и условният скок `jle` в изпълнението, са отделни инструкции.

- Тялото на цикъла **for** на редове 1–7 се изпълнява, както казахме, $n - 2 + 1 = n - 1$ пъти. Следователно, всеки от редовете 2, 3 и 7 се изпълнява $n - 1$ пъти, и тъй като всеки от тях има по една инструкция, която се изпълнява за една стъпка, това са общо $3(n - 1)$ стъпки.

- При всяко изпълнение на **for** цикъла, а такива има $n - 1$ на брой, изпълнението достига ред 4 (началото на **while** цикъла) поне веднъж. Но при дадено изпълнение на **for** цикъла, ред 4 може да бъде изпълняван повече от веднъж – това зависи от конкретния вход. С други думи, точно колко пъти ще се изпълни вложеният цикъл при някакво i зависи от сравненията между $A[j]$ и key . Освен това, вложеният цикъл не се изпълнява за една инструкция. Следователно, за да оценим точно колко пъти ще бъдат изпълнени всеки от редовете 4–6, трябва да извършим по-подробен анализ.
 - ♦ Нека k_i е броят на изпълненията на ред 4 за всяко $i \in \{2, 3, \dots, n\}$. Очевидно $k_i \in \{1, 2, \dots, i\}$, защото ред 4 се изпълнява поне веднъж и най-много i пъти. Защо най-много i пъти? – защото j може да стане най-малко 0, бивайки инициализирано със стойност $i - 1$ на ред 3.
 - ♦ Всеки от редове 5 и 6 се изпълнява $k_i - 1$ пъти. Тогава ред 4 се изпълнява $\sum_{i=2}^n k_i$ пъти **общо за цялото изпълнение на алгоритъма**, а редове 5 и 6 се изпълняват по $\sum_{i=2}^n (k_i - 1)$ пъти. Общо изпълнението на **while** цикъла “струва” $\sum_{i=2}^n k_i + 2 \sum_{i=2}^n (k_i - 1) = (3 \sum_{i=2}^n k_i) - 2(n - 1)$ стъпки по време на цялото изпълнение на алгоритъма.

И така, в нашия опростен модел, изпълнението на INSERTION SORT става в

$$n + 3(n - 1) + 3 \left(\sum_{i=2}^n k_i \right) - 2(n - 1) = 2n - 1 + 3 \sum_{i=2}^n k_i \quad (2.21)$$

стъпки.

За да довършим анализа, трябва да определим минималната и максималната възможна стойност на сумата. k_i може да е най-малко 1, когато тялото на **while** цикъла не се изпълнява изобщо, и най-много i , когато тялото на **while** цикъла се изпълнява $i - 1$ пъти. От това следва, че сложността по време в най-добрия случай е

$$P(n) = 2n - 1 + 3 \sum_{i=2}^n 1 = 2n - 1 + 3n - 3 = 5n - 4 \quad (2.22)$$

а сложността по време в най-лошия случай е

$$T(n) = 2n - 1 + 3 \sum_{i=2}^n (i - 1) = 2n - 1 + \frac{3n(n - 1)}{2} = \frac{3n^2 + n}{2} - 1 \quad (2.23)$$

Да анализираме сложността на още един алгоритъм по аналогичен начин.

SELECTION SORT($A[1 \dots n]$: масив от цели числа)

```

1  for i ← 1 to n - 1
2      for j ← i + 1 to n
3          if A[j] < A[i]
4              swap(A[i], A[j])
```

Ред 1 се изпълнява общо $n - 1 - 1 + 2 = n$ пъти. Ред 2 се изпълнява $n - i + 1$ за всяко i , общо $\sum_{i=1}^{n-1} n - i + 1 = \frac{(n+2)(n-1)}{2}$. Ред 3 се изпълнява $n - i$ за всяко i , общо $\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2}$. Ред 4 се изпълнява общо k пъти за някакво k , такова че $0 \leq k \leq \frac{n(n-1)}{2}$, в зависимост от

конкретния вход. Общо сложността е

$$n + \frac{(n+2)(n-1)}{2} + \frac{n(n-1)}{2} + k = n^2 + n - 1 + k \quad (2.24)$$

Тогава сложността в най-добрия случай е

$$P'(n) = n^2 + n - 1 \quad (2.25)$$

и сложността в най-лошия случай е

$$T'(n) = n^2 + n - 1 + \frac{n(n-1)}{2} = \frac{3n^2 + n}{2} - 1 \quad (2.26)$$

2.3.2 Отказ от точна оценка и търсене на приблизителна оценка на сложността

От изложението в Подсекция 2.2.2 трябва да е станало ясно, че точни изрази като (2.22), (2.23), (2.25) и (2.26) не са необходими. Нашият модел на сложността е достатъчно далеч от реалността, така че събираемостта -1 , което имаме в (2.26), няма практически никакво значение по отношение на каквато и да е софтуерна реализация на SELECTION SORT. Не можем да предвидим времето за изпълнение с точност до наносекунда, така че по отношение на реално изпълнение, **събираемостта -1 не ни казва нищо**. Нещо повече, цялото събираемо $\frac{1}{2}n - 1$ е безсмислено. Ако разполагаме с конкретна реализация на алгоритъма и я тестваме върху различни тестови входове, за които имаме най-лоша сложност (обратно сортирани редици) и после анализираме данните, ще стане ясно, че бързодействието се интерполира от някаква квадратична функция, и толкова. Фактори като конкретна операционна система, компилатор, архитектура и виртуална машина, както и уменията на програмиста да пише бързи програми върху конкретната машина, със сигурност ще се отразят повече на реалното бързодействие от $\frac{1}{2}n - 1$.

Това, което има значение в (2.26) е фактът, че от трите събираеми най-бързо растящото е квадратичната функция $\frac{3}{2}n^2$. Игнорираме множителя $\frac{3}{2}$ и казваме, че алгоритъмът има **квадратична сложност по време**. Никакъв избор на конкретна операционна система, компилатор, програмистки трикове и така нататък не могат да подобрят квадратичната сложност до, да речем, линейна (след малко ще дефинираме прецизно тези понятия).

В заключение, същественото за сложностите на INSERTION SORT и SELECTION SORT е, че в най-лошия случай[†] те са квадратични функции.

Да обобщим. В анализа на сложността се интересуваме само от тенденцията на изменение на бързодействието при **неограничено нарастване** на големината на входа. Никаква конкретна големина на входа не ни интересува. Никаква нейна конкретна стойност, ако ще да е $10\,000\,000^{10\,000\,000^{10\,000\,000}}^{10\,000\,000}$, не е достатъчно голяма, за да спрем до нея. Поради това правим следните опроставяния.

Опроставяне 1. Ако изразът за сложността на даден алгоритъм е сума и едно от събираемите е функция, която расте асимптотично по-бързо от сумата на останалите събираеми в смисъла на Определение 28, разглеждаме само това събираемо, игнорирайки останалите.

[†]Казахме, че сложността в най-добрия случай не е особено информативна и не се ползва, но си заслужава да се отбележи, че в най-добрия случай INSERTION SORT има сложност, която е линейна функция, без алгоритъмът да е модифициран чрез изкуствено пришити инструкции за само един най-добър случай.

Опроставяне 2. За всеки два алгоритъма, ако техните изрази за сложността са асимптотично еквивалентни функции в смисъла на Определение 25, смятаме, че тези алгоритми работят еднакво бързо.

Опростяване 1 не винаги е реалистично. Смисълът да се разглежда само тенденцията на нарастването, когато n клони към безкрайност, е ясен: по-големите входове са по-интересни. Но на практика това е вярно само донякъде. На практика, наистина поведението на алгоритъм за сортиране върху вход с големина 10 000 000 е по-интересно от поведението му върху вход с големина 10, но едва ли можем да твърдим, че поведението му върху вход с големина 10 000 000^{10 000 000} е още по-интересно, а това върху вход с големина 10 000 000^{10 000 000^{10 000 000}}, дори още по-интересно. Последните две числа са напълно отвъд възможностите на всеки истински компютър, сегашен и бъдещ, така че програмата, реализираща въпросния алгоритъм, никога няма да има възможност да демонстрира бързината си върху входове с такива размери. Когато се фокусираме само върху водещото събираемо, ние твърдим, че измежду два алгоритъма, единият от които има сложност $n^2 + n$, а другият, $n + 10\,000\,000^{10\,000\,000^{10\,000\,000}}$ n , вторият работи по-бързо, защото неговата водеща функция n расте по-бавно от n^2 , водещата функция на първия. Да, очевидно съществува стойност на големината на входа n_0 , такава че за всяко $n \geq n_0$, вторият алгоритъм е по-бърз, но това n_0 е твърде голямо. Както вече стана ясно, то е число, което никога не е големина на входа в реалния свят.

Опроставяне 2, тоест игнорирането на мултипликативната константа във водещата функция, също не винаги е реалистично. Според него, два алгоритъма, единият от които има сложност $n^2 + n$, а другият има сложност $10\,000\,000^{10\,000\,000^{10\,000\,000}}$ n^2 , са еднакво бързи. “Оправданието” ни да правим това опростяване е, че алгоритмите, които се използват на практика и почиват върху прагматични идеи, по правило имат сложности, в които тези мултипликативни константи не са фрапиращо големи и никога не са от порядъка на $10\,000\,000^{10\,000\,000^{10\,000\,000}}$. В теорията обаче са известни алгоритми, при които тази мултипликативна константа е кула от степени на двойката

$$2^{2^{2^{\dots^2}}}$$

чиято височина е кула от степени на двойката, чиято височина е кула от степени на двойката, и така нататък няколко пъти [84]. Очевидно такъв алгоритъм е непрактичен дори за вход с големина единица заради въпросната мултипликативна константа.

В заключение, когато разглеждаме тенденцията на нарастването на броя на стъпките на алгоритъма при неограничено нарастване на големината на входа, ние опростяваме нещата, понеже **така е по-лесно**. Практиката показва, че често—но **не винаги**—результатите, които ни дава това опростяване, са добре корелирани с качествата на реалните програми, реализиращи съответните алгоритми.

По думите на Jack Edmonds [42, стр. 451]:

... we can define $F_A(N)$ to be the least upper bound of the cost of applying algorithm A to problems of size N .

When the measure of problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of $F_A(N)$ (let us call it the order of difficulty of algorithm A) is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes. It is one criterion showing how good the algorithm is—not merely in comparison with other algorithms for the same class of problems, but also on the whole how good in comparison with itself.

Edmonds посочва още една причина да разглеждаме асимптотиката. “Rigged” означава “опорочен”, и наистина всеки алгоритъм може да бъде “опорочен”, като бъде направен изкуствено бавен върху малките входове. Ако разглеждаме не асимптотиката на функцията на сложността, а я разглеждаме само до някаква крайна големина на входа n' (смятайки, че по-големи входове няма да възникнат на практика), то, правейки алгоритъма бавен за малките входове, можем да направим така, че той да работи (почти) еднакво бързо върху входове с всяка големина (ненадхвърляща това n'), от което да изглежда, че сложността е константна. Примерно, можем да “опорочим” квадратичен алгоритъм като INSERTION SORT, правейки го привидно линеен (Определение 21. В контраст с това, ако разглеждаме асимптотиката, няма как да направим INSERTION SORT да изглежда линеен посредством изкуствени трикове; можем да го направим кубичен или експоненциален, но не и линеен.

Ще завършим подсекцията с един цитат от книгата “Computational Complexity” на Papadimitriou [125, стр. 7].

Any attempt, in any field of mathematics, to capture an intuitive, real-life notion (for example, that of a “smooth function” in real analysis) by a mathematical concept (such as C_∞) is bound to include certain undesirable specimens, while excluding others that arguably should be embraced.

Следователно, както и да опитваме да формализираме житейското понятие “бърз алгоритъм” или дори “бърза програма”, неизбежно е според формалните ни дефиниции някои алгоритми, които не бихме нарекли бързи, да бъдат класифицирани като такива, а от друга страна, алгоритми, които смятаме за прилично бързи, да бъдат класифицирани като бавни.

2.3.3 Асимптотична нотация Θ

Използвайки асимптотичните нотации, които сега ще дефинираме, можем да изразяваме сложността на изследваните от нас алгоритми **приблизително**. Практиката е показала, че тези приближения са смислени: ако сложността на алгоритъм А, изразена в Θ -нотация, е по-малка от сложността на алгоритъм В, пак в Θ -нотация, твърде възможно е практическата реализация на алгоритъм А да работи по-бързо от практическата реализация на алгоритъм В.

Определение 19: Асимптотично положителна функция

Нека $f : \mathbb{R}^+ \rightarrow \mathbb{R}$. Казваме, че f е *асимптотично положителна*, ако

$$\exists n_0 \in \mathbb{R}^+ \forall n \geq n_0 : f(n) > 0$$

На прост български, иска се функцията да е строго положителна от някоя стойност на аргумента нататък. Това допуска да съществуват стойности на аргумента, за които функцията е отрицателна, стига тези стойности на аргумента да са само донякъде.

Конвенция 5

Отсега нататък по подразбиране всички функции са асимптотично положителни. Ако има изключения, това ще се споменава изрично.

По правило функциите, които разглеждаме, имат домейн и кодомейн $\mathbb{R}^+$, което означава, че са строго положителни и това е по-рестриктивно от това да са асимптотично положителни. В редки случаи обаче, когато изваждаме функция от функция, може за някои (положителни) стойности на аргумента разликата на функциите да не е положителна. Но важното е, че от някаква стойност на аргумента нататък, разликата на функциите е положителна.

Нотация 1: $\forall n \nearrow$

В тези лекционни записки, “ $\forall n \nearrow$ ” е кратък запис за “ $\exists n_0 \forall n \geq n_0$ ”. На прост български се казва “за всички достатъчно големи n ”.

Определение 20: $\Theta(g(n))$

За всяка функция $g(n)$:

$$\Theta(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c_1, c_2 > 0 \forall n \nearrow : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Формално, $\Theta(g(n))$ е безкрайно множество от функции. Ако искаме да кажем, че $h(n)$ е една от тях, пишем “ $h(n) = \Theta(g(n))$ ” наместо формално коректното “ $h(n) \in \Theta(g(n))$ ”. Причината е историческа – прието е да се използва знакът за равенство, а не формално коректният знак за принадлежност към множество. Изразът “ $h(n) = \Theta(g(n))$ ” се чете така: “ $h(n)$ е Тета-голямо от $g(n)$ ”.

Ето пример за педантично доказателство, че една функция е Тета-голямо от друга функция.

Задача 1

Докажете, че $n^2 + 10n + 12 = \Theta(n^2)$, използвайки Определение 20.

Решение: Съгласно определението, трябва да покажем константи $c_1, c_2 > 0$ и стойност n_0 на аргумента, такива че за всяко $n \geq n_0$ да е вярно

$$0 \leq c_1 n^2 \leq n^2 + 10n + 12 \leq c_2 n^2$$

Това са всъщност три неравенства. Първото неравенство $0 \leq c_1 n^2$ е очевидно: щом $c_1 > 0$, то можем да вземем $n_0 = 1$ и тогава $\forall n \geq n_0 : 0 \leq c_1 n^2$. Да разгледаме второто неравенство $c_1 n^2 \leq n^2 + 10n + 12$. Имаме право да разделим двете страни на n^2 , защото разглеждаме само положителни n :

$$c_1 n^2 \leq n^2 + 10n + 12 \leftrightarrow c_1 \leq 1 + \frac{10}{n} + \frac{12}{n^2}$$

Можем да вземем просто $c_1 = 1$, понеже $\forall n \geq n_0 : 1 \leq 1 + \frac{10}{n} + \frac{12}{n^2}$. Тук отново n_0 е 1. Да разгледаме третото неравенство $n^2 + 10n + 12 \leq c_2 n^2$:

$$n^2 + 10n + 12 \leq c_2 n^2 \leftrightarrow 1 + \frac{10}{n} + \frac{12}{n^2} \leq c_2$$

Ако искаме да използваме същата стойност на n_0 , която използвахме в предните две неравенства, а именно $n_0 = 1$, можем да вземем някакво голямо c_2 , да кажем $c_2 = 100$, и тогава лесно се вижда, че $\forall n \geq n_0 : 1 + \frac{10}{n} + \frac{12}{n^2} \leq 100$. Други стойности за c_2 и n_0 също биха свършили работа за доказателството, примерно $c_2 = 3$ и $n_0 = 12^\dagger$.

Ако в първото, второто и третото неравенство сме открили различни стойности на n_0 , всяка от които върши работа за съответното неравенство, то за цялото доказателство е достатъчно да вземем най-голямата от тях. В конкретния случай, ако за първото и второто неравенство $n_0 = 1$ върши работа, а за третото работа върши $n_0 = 12$, то за цялото доказателство ще вземем $n_0 = 12$. $\square$

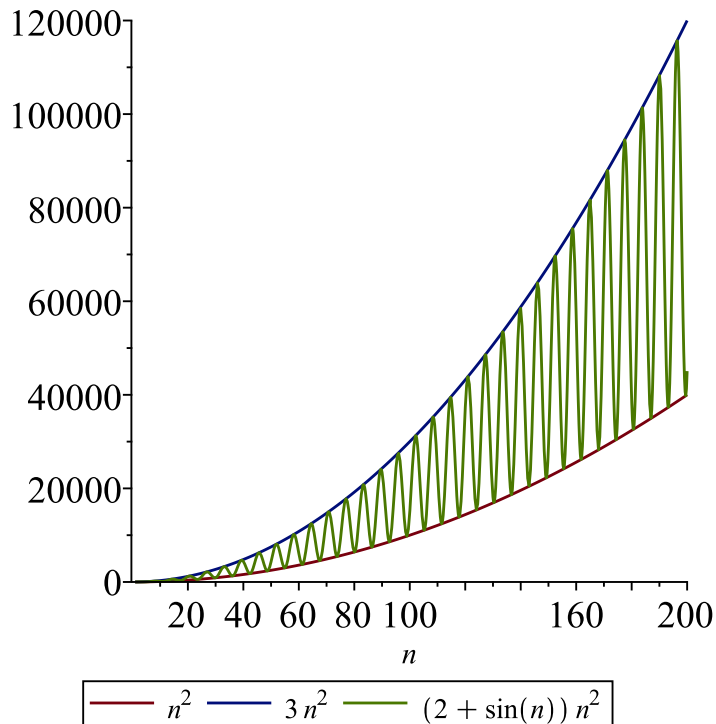
Наблюдение 15

Забележете, че “ $\lim_{n \rightarrow \infty} \frac{h(n)}{g(n)}$ съществува и е равно на някакво L , такова че $0 < L < \infty$ ” е по-силно твърдение от “ $h(n) = \Theta(g(n))$ ”. С други думи, Тета-нотацията е по-обща от изразяването чрез граница, защото границата $\lim_{n \rightarrow \infty} \frac{h(n)}{g(n)}$ може да не съществува и въпреки това $h(n)$ да е Тета-голямо от $g(n)$.

Като пример за това да разгледаме $(2 + \sin n)n^2$ и n^2 . Вярно е, че $(2 + \sin n)n^2 = \Theta(n^2)$. За доказателството можем да вземем константи $c_1 = 1$ и $c_2 = 3$. Обаче границата $\lim_{n \rightarrow \infty} \frac{(2 + \sin n)n^2}{n^2} = \lim_{n \rightarrow \infty} 2 + \sin n$ не съществува. Разгледайте Фигура 2.2[†]. Графиката на $(2 + \sin n)n^2$ осцилира между графиките на n^2 и $3n^2$, като амплитудата на осцилирането нараства неограничено с нарастването на аргумента. Следователно, във формализма на границите не бихме могли да изразим директно, че $(2 + \sin n)n^2$ и n^2 са асимптотично “близки”.

[†]Очевидно съществуват много наредени тройки (c_1, c_2, n_0) , които вършат работа за доказателството.

[†]Фигура 2.2 е направена с Maple(TM).

Фигура 2.2 : Графиките на n^2 , $3n^2$ и $(2 + \sin n)n^2$.**Теорема 16: Свойства на нотацията Θ**

За всички функции $f(n)$, $g(n)$ и $h(n)$:

$$f(n) = \Theta(f(n)) \quad (2.27)$$

$$f(n) = \Theta(g(n)) \rightarrow g(n) = \Theta(f(n)) \quad (2.28)$$

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n)) \quad (2.29)$$

Доказателство: Равенствата 2.27 и 2.29 следват очевидно от рефлексивността и транзитивността на релацията “ $\leq$ ” в Определение 20. Да разгледаме равенство 2.28. В термините на Определение 20, то казва, че:

$$\begin{aligned} & (\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)) \rightarrow \\ & (\exists c'_1, c'_2 > 0 \exists n'_0 > 0 \forall n \geq n'_0 : 0 \leq c'_1 \cdot f(n) \leq g(n) \leq c'_2 \cdot f(n)) \end{aligned}$$

Но това е очевидно вярно, ако вземем $c'_2 = \frac{1}{c_1}$, $c'_1 = \frac{1}{c_2}$, и $n'_0 = n_0$. □

Следствие 5

От равенство 2.28 на Теорема 16 веднага следва, че $f(n) = \Theta(g(n))$ тогава и само тогава, когато $g(n) = \Theta(f(n))$.

Определение 21

Линеен е всеки алгоритъм, чиято сложност по време в най-лошия случай е $\Theta(n)$. Аналогично, ако сложността по време в най-лошия случай е $\Theta(n^2)$, $\Theta(n^3)$ или $\Theta(n^4)$, казваме съответно, че алгоритъмът е *квадратичен*, *кубичен* или *квартичен*.

Използвайки Тета-нотацията, можем да запишем сложността в най-лошия случай $T(n)$ на INSERTION SORT и $T'(n)$ на SELECTION SORT така:

$$T(n) = \Theta(n^2)$$

$$T'(n) = \Theta(n^2)$$

От гледна точка на Тета-асимптотиката, по-прецизен анализ на сложността на тези алгоритми няма. Те имат квадратична сложност по време и толкова.

Конвенция 6: Еднопосочност на знака за равенство при Θ

Записът $\Theta(n^2) = T(n)$ е **некоректен** и не се използва. Знакът за равенство в Тета-изразите е еднопосочен: Тета-нотацията може да се появява или само вдясно, което видяхме досега, или вдясно и вляво, което ще видим след малко, но никога само вляво. Примери за появата на Тета вляво и вдясно са изрази като

$$n^2 + \Theta(n) = \Theta(n^2)$$

В този случай Тета-нотациите вляво и вдясно имат съвсем различен смисъл. Тета-нотацията вдясно представлява безкрайно множество от функции, а тази вляво е *анонимна функция*. С други думи, Тета-нотацията вляво представлява не множество функции, а само една, като всичко, което знаем за нея е, че принадлежи на множеството $\Theta(n)$. Целият израз се чете така: сумата на n^2 и коя да е функция от множеството $\Theta(n)$ е функция от множеството $\Theta(n^2)$.

Допълнение 16: Друга интерпретация на появата на Θ вляво

Според Knuth [91], изрази като " $n^2 + \Theta(n) = \Theta(n^2)$ " се четат по следния начин. Нека A и B са множества от функции. Тогава $A + B$ е множеството от функции $\{f + g \mid f \in A, g \in B\}$. В примера " $n^2 + \Theta(n) = \Theta(n^2)$ " изразът вляво се интерпретира точно по този начин: нека $A = \{n^2\}$, а B е безкрайното множество от функции $\Theta(n)$ според Определение 20; тогава " $n^2 + \Theta(n)$ " е множеството от функции, всяка от които е сума на n^2 и някоя функция от $\Theta(n)$.

Тогава знакът за равенство се интерпретира не като знак за принадлежност към множество " $\in$ ", а като знак за **подмножество** " $\subseteq$ ". При тази интерпретация, " $n^2 + \Theta(n) = \Theta(n^2)$ " се чете така "множеството от функции, всяка от които е сумата на n^2 и някоя функция от $\Theta(n)$, е подмножество на множеството $\Theta(n^2)$."

Конвенция 7: В Θ -нотацията се слага най-простият израз

Формално погледнато, дали ще пишем

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n = \Theta(n^2)$$

или

$$n^2 = \Theta(10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n)$$

е без значение. И двете твърдения са верни. На практика вторият запис **не се ползва**. В скобите на Тета-нотацията слагаме възможно най-простия израз.

Конвенция 8: $\Theta(1)$ означава анонимна константа

В теорията на алгоритмите е прието да се ползва “ $\Theta(1)$ ” като синоним на “анонимна константа” и “сложност $\Theta(1)$ ” като синоним на “константна сложност”. Оттам и “ n^c , където c е константа” в Определение 30 може да се запише кратко “ $n^{\Theta(1)}$ ”.

Единицата не е специална и на теория всяко друго число би могло да се използва за същата цел, но това не е прието. “ $\Theta(2)$ ”, “ $\Theta(1000)$ ” или “ $\Theta(\pi)$ ” изглеждат странно и претенциозно. Да се ползват би било лош стил.

Вместо $\Theta(1)$ понякога се пише $O(1)$ —вижте Определение 22, (2.30) надолу—като смисълът е абсолютно същият; става дума за положителна анонимна константа.

Константна сложност по време при алгоритмите практически не се среща. Само най-прости алгоритми като

- алгоритъм, чийто вход има константна големина, примерно сумиране на две числа,
- или алгоритъм, който винаги връща едно и също нещо, игнорирайки входа,

имат сложност по време $\Theta(1)$. Ние такива алгоритми няма да разглеждаме, освен някои процедури, ползвани от основния разглеждан алгоритъм, примерно RELAX на стр. 529, чийто вход има константа големина. Всеки интересен самостоятелен алгоритъм има сложност по време, която е някаква растяща функция на големината на входа. Заслужава да се отбележи, че ако сложността по време на алгоритъм е $\Theta(f(n))$, където $f(n)$ е функция, която расте асимптотично по-бавно от линейна в смисъла на Определение 22, то този алгоритъм не “чете” целия вход, защото само достъпът до всеки елемент на входа иска линейно време. Пример на такъв алгоритъм е двоичното търсене (Алгоритъм 4), при който $f(n) = \lg n$; това е доказано в Подсекция 2.4.3.

От друга страна, алгоритми със сложност по памет $\Theta(1)$ се срещат често. Както вече бе казано на стр. 101, такива алгоритми се наричат in-place.

2.3.4 Други асимптотични нотации: O , Ω , o и ω

Както казахме, що се отнася до асимптотиката на нарастването, Тета-нотацията дава възможно най-подробната информация. Въпреки че Тета-нотацията е доста “хлабава”—примерно, $\frac{n^3}{1000}$ и $1000^{1000}n^3$ са Тета една от друга—поякога не можем да намерим дори Тета-оценката на някаква функция. За такива случаи има други нотации, по-слаби от Тета-та, които са все пак информативни за асимптотиката на нарастването.

Определение 22: $O(g(n)), \Omega(g(n)), o(g(n)), \omega(g(n))$

За всяка функция $g(n)$:

$$O(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0 \forall n \nearrow : 0 \leq f(n) \leq c \cdot g(n)\} \quad (2.30)$$

$$\Omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0 \forall n \nearrow : 0 \leq c \cdot g(n) \leq f(n)\} \quad (2.31)$$

$$o(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0 \forall n \nearrow : 0 \leq f(n) < c \cdot g(n)\} \quad (2.32)$$

$$\omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0 \forall n \nearrow : 0 \leq c \cdot g(n) < f(n)\} \quad (2.33)$$

Както и при Тета-нотацията, принадлежността към тези множества означаваме не с “ ϵ ”, а с “ $=$ ”, примерно пишем $f(n) = O(g(n))$, $h(n) = \omega(\phi(n))$ и така нататък.

Определение 23: Асимптотични граници

Ако $f(n) = O(g(n))$, казваме, че $g(n)$ е *асимптотична горна граница* за $f(n)$. Ако $f(n) = o(g(n))$, казваме, че $g(n)$ е *строга асимптотична горна граница* за $f(n)$. Ако $f(n) = \Omega(g(n))$, казваме, че $g(n)$ е *асимптотична долна граница* за $f(n)$. Ако $f(n) = \omega(g(n))$, казваме, че $g(n)$ е *строга асимптотична долна граница* за $f(n)$.

Ето примери. В сила са следните съждения:

$$\begin{array}{llllll} n = O(n^2) & n^2 = O(n^2) & n^2 + 1000n + 10000 = O(n^2) & n^3 \neq O(n^2) & 1 = O(n^2) \\ n = o(n^2) & n^2 \neq o(n^2) & n^2 + 1000n + 10000 \neq o(n^2) & 10n^2 \neq o(n^2) & 1 = o(n^2) \\ n \neq \Omega(n^2) & n^2 = \Omega(n^2) & n^2 + 1000n + 10000 \neq \omega(n^2) & n^3 = \Omega(n^2) & n^3 = \omega(n^2) \end{array}$$

Съответно, функцията n^2 е асимптотична горна граница за всяка от функциите n , n^2 , $n^2 + 1000n + 10000$ и 1 , но не е асимптотична горна граница за функцията n^3 ; функцията n^2 е строга асимптотична горна граница за функциите n и 1 , но не е строга асимптотична горна граница за функциите n^2 , $n^2 + 1000n + 10000$ и $10n^2$, и така нататък.

Всички тези твърдения може да бъдат доказани лесно чрез дефинициите. Примерно, то формално доказателство, че $n^3 \neq O(n^2)$. Да допуснем противното, тоест, че $n^3 = O(n^2)$. Тогава от (2.30) знаем, че съществува положително c , такова че за всички достатъчно големи n е изпълнено: $n^3 \leq c \cdot n^2$. Разделяме на n^2 и получаваме еквивалентното неравенство $n \leq c$. Веднага се вижда, че това не може да е вярно, понеже n е променлива, а c е константа. Колкото и голямо c да изберем, съществува n , което е по-голямо от c . Ерго, функцията n^2 не е асимптотична горна граница за функцията n^3 . ✓

Забележете съществената разлика между $O(g(n))$ и $o(g(n))$ (а също така и, дуално, между $\Omega(g(n))$ и $\omega(g(n))$). Ако $f(n) = O(g(n))$, то за **някаква** положителна константа c , $f(n)$ не надвишава $c \cdot g(n)$, $\forall n \nearrow$. Ако $f(n) = o(g(n))$, то за **всяка** положителна константа c , $f(n)$ “изостава” от $c \cdot g(n)$, $\forall n \nearrow$. Неформално говорейки, нотацията o -малко казва, че можем да “отдалечим” $f(n)$ “надолу” от $g(n)$ на каквато си искаме мултипликативна константа, стига да разглеждаме достатъчно големи стойности на аргумента.

В Теорема 17 и Теорема 18, знаците за равенство означават именно равенства между множества, а не принадлежност към множества.

Теорема 17: Θ е сечението на O и Ω

За всяка функция $g(n)$, $O(g(n)) \cap \Omega(g(n)) = \Theta(g(n))$.

Доказателство: Да разгледаме произволна $f(n) \in O(g(n)) \cap \Omega(g(n))$. Съгласно Определение 22(2.30) и Определение 22(2.31), това е същото като

$$(\exists c' > 0 \exists n'_0 \forall n \geq n'_0 : f(n) \leq c' \cdot g(n)) \wedge (\exists c'' > 0 \exists n''_0 \forall n \geq n''_0 : c'' \cdot g(n) \leq f(n))$$

Но тогава за същите c' и c'' и за всяко $n \geq \max\{n'_0, n''_0\}$ е изпълнено

$$c'' \cdot g(n) \leq f(n) \leq c' \cdot g(n)$$

Тогава, съгласно Определение 20, $f(n) \in \Theta(g(n))$. ✓

В обратната посока, да допуснем, че $f(n) \in \Theta(g(n))$. Тогава, съгласно Определение 20:

$$\exists c_1, c_2 > 0 \exists n_0 \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Тривиално следва, че

$$(\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)) \wedge (\exists c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) \leq f(n))$$

Съгласно Определение 22, това е същото като

$$f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)).$$

□

Теорема 18: Несъвместимости между асимптотичните нотации

За всяка функция $g(n)$:

$$o(g(n)) \cap \omega(g(n)) = \emptyset \quad (2.34)$$

$$O(g(n)) \cap \omega(g(n)) = \emptyset \quad (2.35)$$

$$o(g(n)) \cap \Omega(g(n)) = \emptyset \quad (2.36)$$

Доказателство: Ще докажем (2.34). Да допуснем, че съществува $f(n) \in o(g(n)) \cap \omega(g(n))$. Съгласно Определение 22(2.32) и Определение 22(2.33), това е същото като

$$(\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)) \wedge (\forall c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) < f(n))$$

Щом и двата израза започват със “за всяко c ”, то имаме право да вземем едно и също c' и в двата. Заключаваме, че има стойност на аргумента $\tilde{n}$, такава че за всяко $n > \tilde{n}$:

$$f(n) < c' \cdot g(n) \wedge f(n) > c' \cdot g(n)$$

Това очевидно е невъзможно. ✓

Ще докажем (2.35). Да допуснем, че съществува $f(n) \in O(g(n)) \cap \omega(g(n))$. Съгласно Определение 22(2.30) и Определение 22(2.33), това е същото като

$$(\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)) \wedge (\forall c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) < f(n))$$

Щом изразът вдясно започва със “за всяко c ”, в частност предикатът е истина за стойността на c , за която изразът вляво е истина. Заключаваме, че има $c > 0$ и има стойност на аргумента

$\tilde{n}$, такава че за всяко $n > \tilde{n}$:

$$f(n) \leq c \cdot g(n) \wedge f(n) > c \cdot g(n)$$

Това очевидно е невъзможно. ✓

Ще докажем (2.36). Да допуснем, че съществува $f(n) \in o(g(n)) \cap \Omega(g(n))$. Съгласно Определение 22(2.32) и Определение 22(2.31), това е същото като

$$(\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)) \wedge (\exists c > 0 \exists n_0 \forall n \geq n_0 : c \cdot g(n) \leq f(n))$$

Щом изразът вляво започва със “за всяко c ”, в частност предикатът е истина за стойността на c , за която изразът вдясно е истина. Заключаваме, че има $c > 0$ и има стойност на аргумента $\tilde{n}$, такава че за всяко $n > \tilde{n}$:

$$f(n) < c \cdot g(n) \wedge f(n) \geq c \cdot g(n)$$

Това очевидно е невъзможно. ✓

□

В [34, стр. 52], следното свойство се нарича *transpose symmetry*. На български се казва *транспонирана симетрия*. O и Ω са транспонирано симетрични, а също и o и ω са транспонирано симетрични.

Теорема 19: Транспонирана симетрия между асимптотични нотации

За всички функции $f(n)$, $g(n)$:

$$f(n) = O(g(n)) \leftrightarrow g(n) = \Omega(f(n)) \quad (2.37)$$

$$f(n) = o(g(n)) \leftrightarrow g(n) = \omega(f(n)) \quad (2.38)$$

Доказателство: Ще докажем (2.37). Наистина, съгласно Определение 22(2.30), $f(n) = O(g(n))$ е същото като

$$\exists c > 0 \exists n_0 \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Но $f(n) \leq c \cdot g(n)$ е същото като $\frac{1}{c}f(n) \leq g(n)$; тъй като $c > 0$, $\frac{1}{c}$ е дефинирано. Заключаваме, че

$$\exists c' > 0 \exists n_0 \forall n \geq n_0 : c' \cdot f(n) \leq g(n)$$

а именно за $c' = \frac{1}{c}$. Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 22(2.31).

Ще докажем (2.38). Наистина, съгласно Определение 22(2.32), $f(n) = o(g(n))$ е същото като

$$\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)$$

Но $f(n) < c \cdot g(n)$ е същото като $\frac{1}{c}f(n) < g(n)$; тъй като $c > 0$, $\frac{1}{c}$ е дефинирано. Нещо повече, всяко положително число може да се представи като $\frac{1}{c}$, за някое положително c . Заключаваме, че

$$\forall c' > 0 \forall n \nrightarrow : c' \cdot f(n) < g(n)$$

Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 22(2.31). □

Нотациите o и ω може да се дефинират чрез граници.

Наблюдение 16

За всички функции $f(n)$, $g(n)$:

$$f(n) = o(g(n)) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Задача 2

Нека f е асимптотично положителна функция. Докажете или опровергайте, че

$$o(f) = O(f) \setminus \Theta(f)$$

Решение: Твърдението не е вярно. Нека

$$f(x) = 2^{\lceil x \rceil} \text{ за всяко положително реално } x$$

$$g(x) = 2^{2\lceil x \rceil} \text{ за всяко положително реално } x$$

В сила са следните твърдения.

- $g(x) = O(f(x))$, понеже съществува положителна константа c , такава че $2^{2\lceil x \rceil} \leq c \cdot 2^{\lceil x \rceil}$ за всички достатъчно големи стойности на x . Примерно, $c = 1$.
- $g(x) \notin \Theta(f(x))$, понеже $(g(x))^2 = f(x)$ за не-цели стойности на x .
- $g(x) \notin o(f(x))$, защото за всяко положително x_1 съществува $x_2 > x_1$, такава че $g(x_2) = f(x_2)$; а именно, x_2 е цяло число, по-голямо от x_1 .

Тогава $g \in O(f) \setminus \Theta(f)$, но $g \notin o(f)$. Тогава множествата $o(f)$ и $O(f) \setminus \Theta(f)$ не са равни. $\square$

Допълнение 17: Нотациите O и o в анализа

В математическия анализ нотациите O и o се ползват *от 19 век*. Това е много преди модерната теория на алгоритмите.

Да видим пример. Развиваме функцията $\sin(x)$ в ред на Taylor в околността на точка 0:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots$$

Често срещан запис на това развиване е:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + O(x^9) \quad (2.39)$$

(2.39) е по-прецизен запис от

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

защото “ $\approx$ ” може да означава какво ли не. (2.39) казва, че $\sin(x)$ е **точно** равен на

сумата на $x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$ и някаква функция, която в околността на точка 0 намалява, в асимптотичния смисъл, като x^9 ^a. Очевидно нотацията “ $O(x^9)$ ” в (2.39) има смисъл, различен от този на Определение 22(2.30). В (2.39) се има предвид не тенденция при неограничено нарастване на аргумента—както е в Определение 22—а тенденция, когато аргументът клони към определена стойност, в случая 0.

Записът “ $f(x) = O(g(x))$ при $x \rightarrow a$ ” е кратък запис за следното:

$$\exists c > 0 \exists \delta > 0 : |f(x)| \leq cg(x), \text{ ако } 0 < |x - a| < \delta$$

В (2.39) “ $x \rightarrow 0$ ” е изпуснато, но то се подразбира.

За нотацията o нещата са аналогични. Записът “ $f(x) = o(g(x))$ при $x \rightarrow a$ ” е кратък запис за следното:

$$\forall c > 0 \exists \delta > 0 : |f(x)| \leq cg(x), \text{ ако } 0 < |x - a| < \delta$$

В заключение, нотациите O и o винаги изразяват тенденции за нарастване или намаляване на функции. Ако x е аргументът, тези тенденции са или при $x \rightarrow a$ за някакво фиксирано a , или при $x \rightarrow \infty$. Когато изследваме сложността на алгоритми, ние се интересуваме само от тенденцията при $x \rightarrow \infty$, поради което асимптотичните нотации в тези лекционни записки винаги са спрямо тенденцията на неограничено нарастване на аргумента.

^a“намалява”, защото аргументът е по-малък от единица.

2.3.5 Релации $\asymp$, $\leq$, $<$, $\geq$ и $>$ и някои техни свойства

Определение 24: Релация $\asymp$ над множеството на асимпт. положителните функции

За всички функции $f(n)$ и $g(n)$, $f(n) \asymp g(n)$ тогава и само тогава, когато $f(n) = \Theta(g(n))$.

Нотацията, използваща символа “ $\asymp$ ”, е алтернатива на нотацията, използваща Θ : вместо “ $f(n) = \Theta(g(n))$ ” можем да запишем “ $f(n) \asymp g(n)$ ” и обратно.

Ако ползваме записа с “ $\asymp$ ” вместо “ Θ ”, необходимостта от Конвенция 6 изчезва по очевидни причини: в израза няма “ Θ ”. Също така изчезва и необходимостта от Конвенция 7, понеже двете страни на знака $\asymp$ са равнопоставени. Ерго, следните два записа са еднакво приемливи:

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n \asymp n^2$$

$$n^2 \asymp 10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n$$

Следствие 6 се доказва тривиално с Теорема 16 и Следствие 5.

Следствие 6

$\asymp$ е релация на еквивалентност.

Следствие 6 ни дава основание да направим следните дефиниции.

Определение 25: Асимптотична еквивалентност

Релацията $\asymp$ се нарича *релация на асимптотична еквивалентност*. За всеки две функции $f(n)$ и $g(n)$, ако $f(n) \asymp g(n)$, казваме, че $f(n)$ и $g(n)$ са *асимптотично еквивалентни*.

Определение 26: Релации $\leq$, $<$, $\geq$ и $>$

За всички функции $f(n)$ и $g(n)$:

- $f(n) < g(n)$ тогава и само тогава, когато $f(n) = o(g(n))$,
- $f(n) \leq g(n)$ тогава и само тогава, когато $f(n) = O(g(n))$,
- $f(n) > g(n)$ тогава и само тогава, когато $f(n) = \omega(g(n))$,
- $f(n) \geq g(n)$ тогава и само тогава, когато $f(n) = \Omega(g(n))$.

Релацията $<$ се нарича *релация на строго асимптотично предшество*. Релацията $\leq$ се нарича *релация на асимптотично предшество*. Релацията $>$ се нарича *релация на строго асимптотично следване*. Релацията $\geq$ се нарича *релация на асимптотично следване*.

Определение 27: Релации на асимптотични сравнения

Петте релации $\asymp$, $\leq$, $<$, $\geq$ и $>$ са известни като *релациите на асимптотични сравнения*. За всеки две функции $f(n)$ и $g(n)$, ако поне една от тези пет релации е в сила между тях, казваме, че $f(n)$ и $g(n)$ са *асимптотично сравними*. В противен случай казваме, че $f(n)$ и $g(n)$ са *асимптотично несравними*.

От симетрията на $\asymp$ и транспонираната симетрия (вж. Следствие 9) на $\{\leq, \geq\}$ и на $\{<, >\}$ следва, че по отношение на асимптотичната сравнимост няма значение коя от $f(n)$ и $g(n)$ е записана вляво и коя, вдясно. С други думи, ако за някоя от петте релации, да я запишем като $\triangleright$, е вярно, че $f(n) \triangleright g(n)$, то съществува релация измежду петте, да я запишем като $\triangleleft$, такава че $g(n) \triangleleft f(n)$.

Съгласно Теорема 24(2.52), асимптотично несравними функции съществуват.

Определение 28: Асимптотично по-бавно и асимптотично по-бързо нарастване

Ако $f(n) < g(n)$, казваме, че $f(n)$ *расте асимптотично по-бавно от $g(n)$* . Ако $f(n) > g(n)$, казваме, че $f(n)$ *расте асимптотично по-бързо от $g(n)$* .

Заради транспонираната симетрия (вж. Следствие 9) на $\{<, >\}$ имаме право освен това да кажем, че $g(n)$ *расте асимптотично по-бързо от $f(n)$* и че $g(n)$ *расте асимптотично по-бавно от $f(n)$* .

Теорема 17, Определение 24 и Определение 26 влекат непосредствено Следствие 7.

Следствие 7

За всички функции $f(n)$ и $g(n)$, $f(n) \asymp g(n) \leftrightarrow f(n) \leq g(n) \wedge f(n) \geq g(n)$.

Теорема 18 и Определение 26 влекат непосредствено Следствие 8.

Следствие 8: Несъвместими релации на асимптотични сравнения

Не съществуват функции $f(n)$ и $g(n)$, такива че:

$$f(n) < g(n) \wedge f(n) > g(n)$$

$$f(n) \leq g(n) \wedge f(n) > g(n)$$

$$f(n) < g(n) \wedge f(n) \geq g(n)$$

Теорема 19 и Определение 26 влекат непосредствено Следствие 9.

Следствие 9: Транспонирана симетрия между релации

За всички функции $f(n)$, $g(n)$:

$$f(n) \leq g(n) \leftrightarrow g(n) \geq f(n)$$

$$f(n) < g(n) \leftrightarrow g(n) > f(n)$$

Наблюдение 17 е точно съответствие на Наблюдение 16:

Наблюдение 17

За всички функции $f(n)$, $g(n)$:

$$f(n) < g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) > g(n) \leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Теорема 20: $<$ влече $\leq$ и $>$ влече $\geq$

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) < g(n) \rightarrow f(n) \leq g(n) \tag{2.40}$$

$$f(n) > g(n) \rightarrow f(n) \geq g(n) \tag{2.41}$$

Доказателство: Доказателството е тривиално и в двата случая. За (2.40), ако за всяко положително c и всички достатъчно големи стойности на аргумента n е вярно, че $f(n) < c \cdot g(n)$, то очевидно съществува положително c , такова че за всички достатъчно големи стойности на аргумента n : $f(n) \leq c \cdot g(n)$. За (2.41) доказателството е аналогично. $\square$

Теорема 21: $\leq$ не влече $<$ и $\geq$ не влече $>$

Нека $f(n)$ и $g(n)$ са произволни асимптотично положителни функции. Тогава

$$\neg(f(n) \leq g(n) \rightarrow f(n) < g(n)) \tag{2.42}$$

$$\neg(f(n) \geq g(n) \rightarrow f(n) > g(n)) \tag{2.43}$$

Доказателство: Като контрапример и за (2.42), и за (2.43) да разгледаме $f(n) = n$ и $g(n) = n$. Очевидно $f(n) \leq g(n)$ и $f(n) \geq g(n)$, но $f(n) \not< g(n)$ и $f(n) \not> g(n)$. $\square$

Теорема 22: $<$ е несъвместима с $\geq$ и $>$ е несъвместима с $\leq$

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \geq g(n) \rightarrow f(n) \nless g(n) \quad (2.44)$$

$$f(n) < g(n) \rightarrow f(n) \nless g(n) \quad (2.45)$$

$$f(n) \leq g(n) \rightarrow f(n) \nless g(n) \quad (2.46)$$

$$f(n) > g(n) \rightarrow f(n) \nless g(n) \quad (2.47)$$

Доказателство: Всяка от тези импликации се доказва тривиално с допускане на противното. Да разгледаме само (2.44). Противното е

$$\begin{aligned} \neg(f(n) \geq g(n) \rightarrow f(n) \nless g(n)) &\equiv (* \text{ свойства на импликацията } *) \\ \neg(\neg(f(n) \geq g(n)) \vee \neg(f(n) < g(n))) &\equiv (* \text{ з-н на De Morgan, з-н за дв. отриц. } *) \\ f(n) \geq g(n) \wedge f(n) < g(n) \end{aligned}$$

Но от Следствие 8 знаем, че такива $f(n)$ и $g(n)$ няма. □

Теорема 23: $<$ и $>$ са несъвместими с $\asymp$

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \asymp g(n) \rightarrow f(n) \nless g(n) \quad (2.48)$$

$$f(n) < g(n) \rightarrow f(n) \neq g(n) \quad (2.49)$$

$$f(n) \asymp g(n) \rightarrow f(n) \nless g(n) \quad (2.50)$$

$$f(n) > g(n) \rightarrow f(n) \neq g(n) \quad (2.51)$$

Доказателство: Следват от Теорема 22 и Следствие 7. Да разгледаме само (2.48). От Следствие 7 имаме $f(n) \asymp g(n) \rightarrow f(n) \geq g(n)$. От Теорема 22 имаме $f(n) \geq g(n) \rightarrow f(n) \nless g(n)$. Прилагаме правилото хипотетичен силгизъм от съждителната логика, а именно $p \rightarrow q \wedge q \rightarrow r \vdash p \rightarrow r$, и получаваме желанния резултат. □

Въведохме пет релации на асимптотични сравнения на функции. От най-общи комбинаторни съображения има не повече от $2^5 = 32$ възможности тези релации да са или да не са в сила за произволни функции. Теорема 24 казва, че тези възможности са точно шест.

Теорема 24: Различните възможности при асимптотично сравняване на функции

За всеки две асимптотично положителни функции $f(n)$ и $g(n)$ е в сила е точно едно от следните:

$$f(n) \nless g(n) \wedge f(n) \nless g(n) \wedge f(n) \neq g(n) \wedge f(n) \nless g(n) \wedge f(n) \nless g(n) \quad (2.52)$$

$$f(n) \nless g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \nless g(n) \wedge f(n) \nless g(n) \quad (2.53)$$

$$f(n) < g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \nless g(n) \wedge f(n) \nless g(n) \quad (2.54)$$

$$f(n) \nless g(n) \wedge f(n) \leq g(n) \wedge f(n) \asymp g(n) \wedge f(n) \nless g(n) \wedge f(n) \geq g(n) \quad (2.55)$$

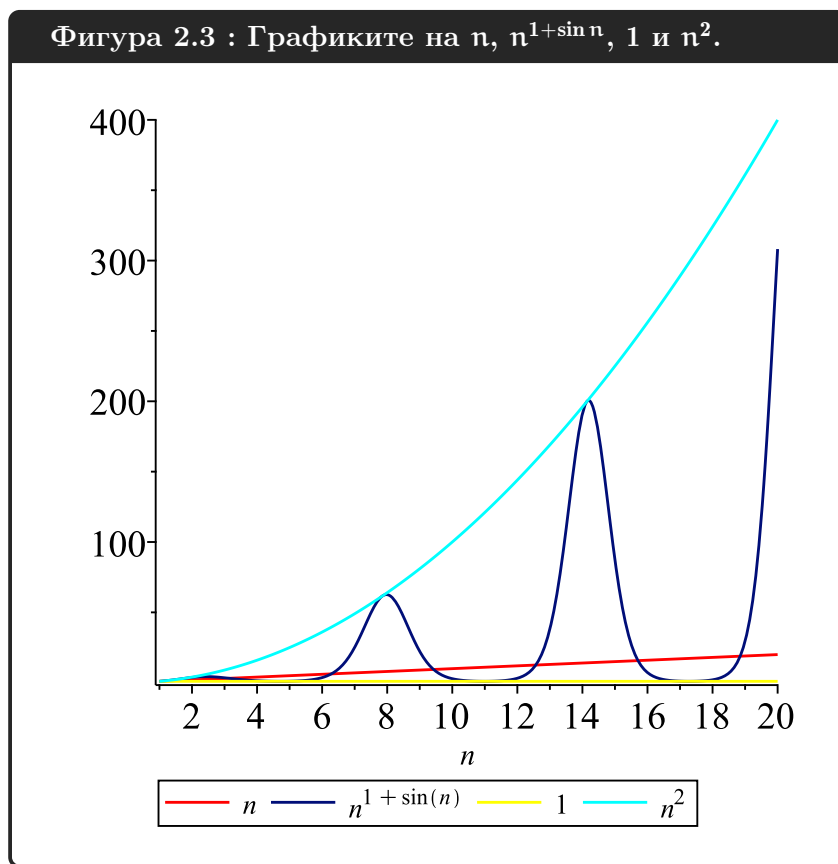
$$f(n) \nless g(n) \wedge f(n) \nless g(n) \wedge f(n) \neq g(n) \wedge f(n) > g(n) \wedge f(n) \geq g(n) \quad (2.56)$$

$$f(n) \nless g(n) \wedge f(n) \nless g(n) \wedge f(n) \neq g(n) \wedge f(n) \nless g(n) \wedge f(n) \geq g(n) \quad (2.57)$$

Доказателство: Първо ще покажем по една двойка функции като пример за всяка от тези шест възможности. После ще покажем, че останалите комбинации са невъзможни.

$f(n) \nlessdot g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \nlessdot g(n) \wedge f(n) \nlessdot g(n)$ (2.52): Дефиниращата характеристика на този случай е $f(n) \leq g(n) \wedge f(n) \nlessdot g(n)$, понеже е достатъчно и $\leq$, и $\nlessdot$ да не са в сила, за да не е в сила нито една от останалите три релации. Това се доказва лесно с негацията на Следствие 7 и контрапозитивните на твърденията от Теорема 20.

Ето пример за такива функции, взет от [34, стр. 52]. Нека $f(n) = n$ и $g(n) = n^{1+\sin n}$. Тъй като $1 + \sin n$ осцилира между 0 и 2, функцията $n^{1+\sin n}$ осцилира между 1 и n^2 [†]. Ясно е, че за всяка константа $c > 0$ и за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $\neg(f(n') \leq c \cdot g(n'))$. Но това е точно отрицанието на $f(n) \leq g(n)$. Ясно е, че за всяка константа $c > 0$ и за всяка стойност на аргумента n_0 съществува $n'' > n_0$, такова че $\neg(f(n'') \geq c \cdot g(n''))$. Но това е точно отрицанието на $f(n) \geq g(n)$. Накратко, $f(n) \leq g(n)$ и $f(n) \nlessdot g(n)$. Фигура 2.3[‡] илюстрира графиките на $f(n)$ и $g(n)$, а също така и на 1 и n^2 .



$f(n) \nlessdot g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \nlessdot g(n) \wedge f(n) \nlessdot g(n)$ (2.53): Един пример за такива функции е $f(n) = n^{1+\sin n}$ и $g(n) = n^2$ (Фигура 2.3 илюстрира и тези разсъждения). Вярно е, че $f(n) \leq g(n)$: вземаме $c = 1$ и забелязваме, че за всяко $n \geq 1$ е вярно, че $f(n) \leq cg(n)$.

От друга страна, $f(n) \nlessdot g(n)$, понеже съществува константа $c > 0$, такова че за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') \geq c \cdot g(n')$; например, $c = 1$.

[†]В този случай осцилирането е в степенния показател, а не в множител, както беше в примера на Фигура 2.2, така че тези функции не са Тета една от друга, за разлика от функциите, показани на Фигура 2.2.

[‡]Фигура 2.3 е направена с Maple(TM).

Също така е вярно, че $f(n) \not\leq g(n)$, понеже за всяка константа $c > 0$ и за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') < c \cdot g(n')$.

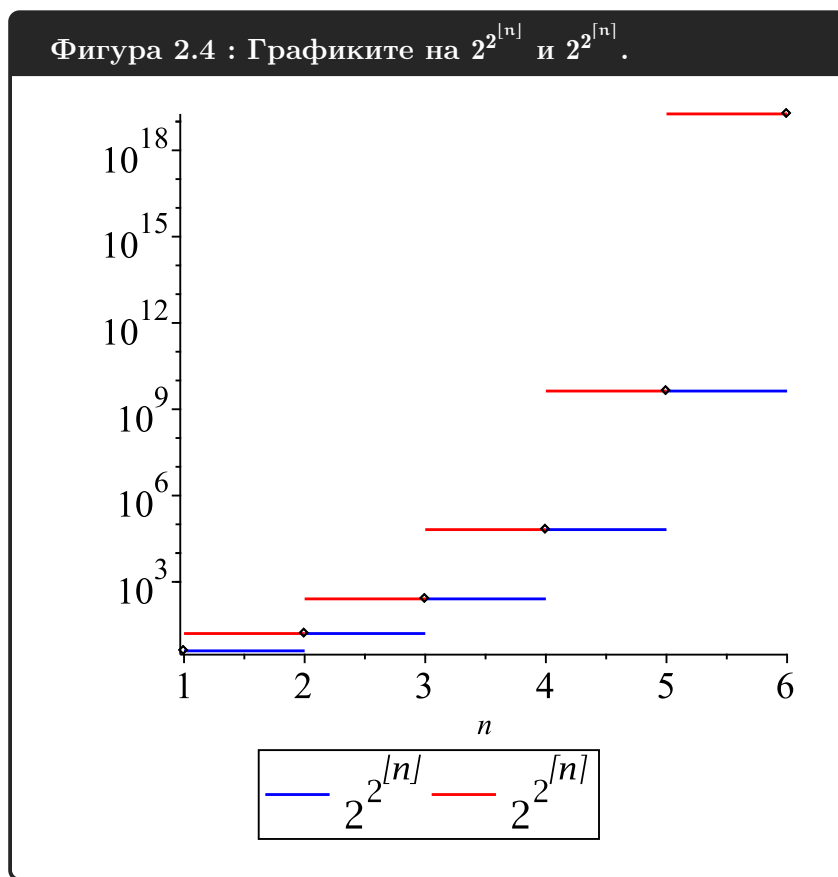
От това, че $f(n) \leq g(n)$ и $f(n) \not\leq g(n)$ следва, че $f(n) \neq g(n)$ съгласно Следствие 7.

И накрая, $f(n) \not\leq g(n)$, което следва от това, че $f(n) \leq g(n)$ и Теорема 22.

Друга двойка функции, която е подходящ пример, е $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$. Ако $n \in \mathbb{N}$, то $f(n) = g(n)$. Ако обаче $n \in \mathbb{R}^+ \setminus \mathbb{N}$, $f(n) = \sqrt{g(n)}$, понеже

$$n \in \mathbb{R}^+ \setminus \mathbb{N} \rightarrow \lfloor n \rfloor = \lceil n \rceil - 1 \rightarrow 2^{\lfloor n \rfloor} = 2^{\lceil n \rceil - 1} \rightarrow 2^{2^{\lfloor n \rfloor}} = 2^{2^{\lceil n \rceil - 1}} = 2^{\frac{1}{2} 2^{\lceil n \rceil}} = \sqrt{2^{2^{\lceil n \rceil}}}$$

Фигура 2.4[†] показва графиките на тези две функции. Функциите се изравняват върху всяко цяло число. След това, в отворения интервал до следващото цяло число, $f(n) = 2^{2^{\lfloor n \rfloor}}$ остава същата, а $g(n) = 2^{2^{\lceil n \rceil}}$ има стойност, която е предната ѝ стойност, повдигната на квадрат. Фигурата е с логаритмичен мащаб по ординатата поради изключително бързото нарастване на двойната експонента.



$f(n) < g(n) \wedge f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \not\leq g(n)$ (2.54): Пример за такива функции е $f(n) = n$ и $g(n) = n^2$. Това, че $f(n) < g(n)$, $f(n) \leq g(n)$, $f(n) \neq g(n)$, $f(n) \not\leq g(n)$ и $f(n) \not\leq g(n)$, е очевидно.

$f(n) \not\leq g(n) \wedge f(n) \leq g(n) \wedge f(n) \asymp g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \geq g(n)$ (2.55): Пример за такива функции е $f(n) = n$ и $g(n) = n + 1$. Това, че $f(n) \not\leq g(n)$, $f(n) \leq g(n)$, $f(n) \asymp g(n)$, $f(n) \not\leq g(n)$ и $f(n) \geq g(n)$, е очевидно.

[†]Фигура 2.4 е направена с Maple(TM).

$f(n) \not\prec g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \neq g(n) \wedge f(n) > g(n) \wedge f(n) \geq g(n)$ (2.56): Примерът за (2.54), но с разменени $f(n)$ и $g(n)$, е валиден за този случай.

$f(n) \not\prec g(n) \wedge f(n) \not\leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \not\prec g(n) \wedge f(n) \geq g(n)$ (2.57): Двата примера за (2.53), но с разменени $f(n)$ и $g(n)$, са валидни за този случай. $\square$

Сега да разгледаме невъзможните комбинации. Всички комбинации с поне едно от следните:

$$f(n) > g(n) \wedge f(n) \not\leq g(n)$$

$$f(n) < g(n) \wedge f(n) \not\leq g(n)$$

са невъзможни, защото $>$ влече $\geq$ и $<$ влече $\leq$ (Теорема 20). Всички комбинации с поне едно от следните:

$$f(n) = g(n) \wedge f(n) \not\leq g(n)$$

$$f(n) \not\leq g(n) \wedge f(n) = g(n)$$

$$f(n) \leq g(n) \wedge f(n) \neq g(n) \wedge f(n) \geq g(n)$$

са невъзможни, защото $=$ е в сила тогава и само тогава, когато и $\geq$, и $\leq$ са в сила (Следствие 7). И накрая, всички комбинации с поне едно от следните:

$$f(n) < g(n) \wedge f(n) \geq g(n)$$

$$f(n) \leq g(n) \wedge f(n) > g(n)$$

са невъзможни, защото $<$ и $\geq$ са несъвместими, а също така $\leq$ и $>$ са несъвместими (Следствие 8). $\square$

Следствие 10: $\leq$ и $\geq$ са фундаменталните асимптотични сравнения

Нека $f(n)$ и $g(n)$ са произволни асимптотично положителни функции. Нека $f(n)$ и $g(n)$ са асимптотично сравними. Тогава

$$f(n) \leq g(n) \text{ или } g(n) \leq f(n)$$

Заради транспонираната симетрия имаме право да кажем това по друг начин:

$$f(n) \geq g(n) \text{ или } g(n) \geq f(n)$$

Теорема 25: Повдигането на константна степен запазва $\asymp$

За всеки две функции $f(n)$ и $g(n)$ и за всяка константа $k \in \mathbb{R}^+$:

$$f(n) \asymp g(n) \leftrightarrow (f(n))^k \asymp (g(n))^k$$

Доказателство: В едната посока, нека

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

за някакви положителни константи c_1 и c_2 и за всички $n \geq n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на k -та степен (k е положително) и получаваме

$$c_1^k (g(n))^k \leq (f(n))^k \leq c_2^k (g(n))^k, \text{ за всяко } n \geq n_0$$

Но тогава $(f(n))^k \asymp (g(n))^k$, понеже c_1^k и c_2^k са положителни константи.

В другата посока, нека

$$c_1(g(n))^k \leq (f(n))^k \leq c_2(g(n))^k, \text{ за всяко } n \geq n_0$$

за някакви положителни константи c_1 и c_2 и за всички $n \geq n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на $\frac{1}{k}$ -та степен ($\frac{1}{k}$ е положително) и получаваме

$$\sqrt[k]{c_1}g(n) \leq f(n) \leq \sqrt[k]{c_2}g(n), \text{ за всяко } n \geq n_0$$

Но тогава $f(n) \asymp g(n)$, понеже $\sqrt[k]{c_1}$ и $\sqrt[k]{c_2}$ са положителни константи. □

Теорема 26: Водещото събираемо определя асимптотиката

Нека $f(n)$ е асимптотично положителна функция. Нека $f(n) = g(n) + h(n)$, където $h(n)$ е или не е асимптотично положителна, но във всеки случай $|h(n)| < g(n)$. Тогава $f(n) \asymp g(n)$.

Доказателство: Първо ще покажем, че $g(n)$ е асимптотично положителна. Да допуснем противното. Негацията на съждението в Определение 19, по отношение на $g(n)$, е:

$$\forall n_0 \in \mathbb{R}^+ \exists n \geq n_0 : g(n) \leq 0$$

Но тогава е невъзможно да е имаме $|h(n)| = o(g(n))$, понеже това е същото (Определение 22, (2.32)) като за всяка положителна константа c да има стойност на аргумента n_0 , такава че за всяко $n \geq n_0$ да е вярно, че $0 < |h(n)| < c \cdot g(n)$.

И така, $g(n)$ е асимптотично положителна. Съгласно Определение 19, съществува n_0 , такава че $\forall n \geq n_0 : g(n) > 0$. Забележете, че е възможно $g(n) + h(n) < g(n) \forall n \nrightarrow$, понеже е възможно $h(n) < 0 \forall n \nrightarrow$. Но

$$\frac{1}{2}g(n) \leq g(n) + h(n) \leq 2g(n) \forall n \nrightarrow$$

понеже има стойност на аргумента $n' \geq n_0$, такава че за $n \geq n'$ е изпълнено $|h(n)| < \frac{1}{2}g(n)$. Но $g(n) + h(n) = f(n)$ по условие. Тогава е изпълнено

$$\forall n \geq n' : \frac{1}{2}g(n) \leq f(n) \leq 2g(n)$$

Съгласно Определение 20, в сила е $f(n) \asymp g(n)$. □

Следствие 11

Нека $p(n)$ е произволен асимптотично положителен полином, такъв че $\deg(p(n)) = k$. Тогава $p(n) \asymp n^k$.

Да настояваме за положителни коефициенти е прекалено рестриктивно. Допускаме някои коефициенти на полинома да са отрицателни. Коефициентът пред събираемото от най-висока степен обаче е положителен, иначе полиномът не би бил асимптотично положителен.

Доказателство: Нека $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$. Очевидно a_k не е нула, иначе полиномът нямаше да е от степен k , и не е отрицателно число, иначе полиномът нямаше да е асимптотично положителна функция.

Ясно е, че $|a_{k-1}n^{k-1} + \dots + a_1n + a_0| < a_k n^k$, понеже $|a_{k-1}n^{k-1} + \dots + a_1n + a_0| \leq |a_{k-1}|n^{k-1} + \dots + |a_1|n + |a_0| \forall n \geq 1$ и $\lim_{n \rightarrow \infty} \frac{|a_{k-1}|n^{k-1} + \dots + |a_1|n + |a_0|}{a_k n^k} = 0$. Сега прилагаме Теорема 26 с $p(n)$ като $f(n)$, $a_k n^k$ като $g(n)$ и $a_{k-1}n^{k-1} + \dots + a_1n + a_0$ като $h(n)$. $\square$

Теорема 27 казва, че експоненциалната трансформация върху две функции “изостря” асимптотичните разлики в нарастването им, ако функциите са неограничено нарастващи и имат различно асимптотично нарастване.

Теорема 27: Експонирането запазва <

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции и $a > 1$ е константа. Тогава

$$f(n) < g(n) \rightarrow a^{f(n)} < a^{g(n)}$$

Доказателство: Трябва да докажем, че:

$$\forall c > 0 \exists n' \forall n \geq n' : a^{f(n)} < c \cdot a^{g(n)} \quad (2.58)$$

Но (2.58) може да се препише по следния еквивалентен начин, като k съответства на $\log_a c$ в (2.58):

$$\forall k \exists n' \forall n \geq n' : a^{f(n)} < a^k a^{g(n)} \quad (2.59)$$

Трябва да се има предвид, че k не е непременно положително, за разлика от c в (2.58). Вземаме логаритъм при основа a от двете страни на (2.59) и получаваме:

$$\forall k \exists n' \forall n \geq n' : f(n) < k + g(n) \quad (2.60)$$

Трябва да докажем (2.60).

По условие имаме:

$$\forall c > 0 \exists n_0 \forall n \geq n_0 : f(n) < c \cdot g(n)$$

Щом това е в сила за всяко $c > 0$, в частност то е в сила за $c = \frac{1}{2}$. И така:

$$\exists n_0 \forall n \geq n_0 : f(n) < \frac{g(n)}{2} \quad (2.61)$$

Но $g(n)$ е растяща и неограничена. Тогава:

$$\forall k \exists n_1 \forall n \geq n_1 : 0 < k + \frac{g(n)}{2} \quad (2.62)$$

(2.62) е еквивалентно на:

$$\forall k \exists n_1 \forall n \geq n_1 : \frac{g(n)}{2} < k + g(n) \quad (2.63)$$

От (2.61) и (2.63) заключаваме, че:

$$\forall k \exists n'' \forall n \geq n'' : f(n) < k + g(n) \quad (2.64)$$

Но (2.64) е същото като (2.60). С това доказателството е готово. $\square$

Следствие 12

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции. Тогава

$$\lg f(n) < \lg g(n) \rightarrow f(n) < g(n)$$

Теорема 28 в някакъв смисъл е конверсна на Теорема 27. Според Теорема 28, ако образите на две нарастващи и неограничени функции след експоненциална трансформация са асимптотично еквивалентни, то и самите функции са асимптотично еквивалентни.

Теорема 28: Логаритмуването запазва $\asymp$

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции и нека $a > 1$ е константа. Тогава

$$a^{f(n)} \asymp a^{g(n)} \rightarrow f(n) \asymp g(n)$$

Доказателство: Нека

$$\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 \cdot a^{g(n)} \leq a^{f(n)} \leq c_2 \cdot a^{g(n)}$$

Щом $a > 1$ и $f(n)$ и $g(n)$ са положителни, то за всички достатъчно големи n е вярно, че експонентите са строго по-големи от единица. Тогава вземаме логаритъм при основа a от двете страни и получаваме:

$$\log_a c_1 + g(n) \leq f(n) \leq \log_a c_2 + g(n)$$

Забелязваме, че щом $g(n)$ е нарастваща и неограничена, то за всяка константа k_1 , такава че $0 < k_1 < 1$, в сила е $k_1 \cdot g(n) \leq \log_a c_1 + g(n)$, за всички достатъчно големи n . И това е така независимо от това, дали логаритъмът е положителен или отрицателен или е нула. После забелязваме, че щом $g(n)$ е нарастваща и неограничена, то за всяка константа k_2 , такава че $k_2 > 1$, в сила е $\log_a c_2 + g(n) \leq k_2 \cdot g(n)$ за всички достатъчно големи n . И това е така независимо от това, дали логаритъмът е положителен или отрицателен или е нула. Заклучаваме, че съществува n_1 , такова че:

$$\forall n \geq n_1 : k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$$

Но тогава $f(n) \asymp g(n)$. □

Следствие 13

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции. Тогава

$$f(n) \asymp g(n) \rightarrow \lg f(n) \asymp \lg g(n)$$

Дотук разглеждахме свойства на релациите на асимптотични сравнения върху две функции, които по правило наричахме “ $f(n)$ ” и “ $g(n)$ ”. Сега ще разгледаме свойства на тези релации върху функции, едната от които се получава от другата чрез полилогаритмуване или експониране или итерирано логаритмуване.

Определение 29: Полилогаритмично растяща функция

За всяка $f(n)$, такава че $f(n) \asymp (\log_a n)^k$, където $a, k \in \mathbb{R}^+$ са константи и $a > 1$, казваме, че $f(n)$ *расте полилогаритмично*.

Определение 30: Полиномиално растяща функция

За всяка $f(n)$, такава че $f(n) \asymp n^c$, където $c \in \mathbb{R}^+$ е константа, казваме, че $f(n)$ *расте полиномиално*.

Забележете, че в математиката е прието полиномите да имат цели степенни показатели, така че $n^{\frac{1}{2}}$ не е полином. Но съгласно Определение 30, функцията $n^{\frac{1}{2}}$ расте полиномиално.

Определение 31: Експоненциално растяща функция

За всяка $f(n)$, такава че $f(n) \asymp a^{n^k}$, където $a, k \in \mathbb{R}^+$ са константи и $a > 1$, казваме, че $f(n)$ *расте експоненциално*.

Когато чуем “експоненциална функция”, обикновено си представяме функции като 2^n или e^n или 10^n ; изобщо, a^n за някое $a > 1$. Но съгласно Определение 31, функции като 2^{n^2} и e^{n^5} и $(\sqrt{2})^{n^{\sqrt{2}}}$ също са експоненциално растящи функции.

Теорема 29: Полилогаритмичното нарастване е по-бавно от полиномиалното

Нека $a, k, c \in \mathbb{R}^+$ са произволни константи, като $a > 1$. Нека $f(n)$ е неограничено и строго растяща диференцируема функция. В сила е:

$$(\log_a f(n))^k < (f(n))^c$$

Доказателство:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{(f(n))^c}{(\log_a f(n))^k} &= // \text{полагаме } b \leftarrow \frac{c}{k} \\ \lim_{n \rightarrow \infty} \frac{((f(n))^b)^k}{(\log_a f(n))^k} &= \\ \lim_{n \rightarrow \infty} \left(\frac{(f(n))^b}{\log_a f(n)} \right)^k &= // \frac{(f(n))^b}{\log_a f(n)} \text{ е непрекъснатата} \\ \left(\lim_{n \rightarrow \infty} \frac{(f(n))^b}{\log_a f(n)} \right)^k &= // \text{прилагаме правилото на l'Hôpital} \\ \left(\lim_{n \rightarrow \infty} \frac{b(f(n))^{b-1} f'(n)}{\left(\frac{1}{\ln a}\right) \left(\frac{1}{f(n)}\right) f'(n)} \right)^k &= // f'(n) \neq 0 \\ \left(\lim_{n \rightarrow \infty} (\ln a)^b (f(n))^b \right)^k &= // k > 0 \end{aligned}$$

∞

□

Теорема 30: Полиномиалното нарастване е по-бавно от експоненциалното

Нека $a, k, c \in \mathbb{R}^+$ са произволни константи, като $a > 1$. Нека $f(n)$ е неограничено растяща диференцируема функция. В сила е:

$$(f(n))^k < a^{(f(n))^c}$$

Доказателство: Да вземем $\log_a$ от двете страни. Лявата страна става $k \log_a f(n)$ и дясната страна става $(f(n))^c$. Но $k \log_a f(n) < (f(n))^c$ съгласно Теорема 29. Прилагаме Следствие 12 и получаваме желанния резултат. $\square$

Следствие 14 е частен случай на Теорема 29 и Теорема 30, ако $f(n) = n$.

Следствие 14: Полилогаритмично преди полиномиално преди експоненциално

Нека $a, k, \epsilon \in \mathbb{R}^+$ са произволни константи, като $a > 1$. В сила е:

$$(\log_a n)^k < n^\epsilon < a^{n^k}$$

Казано на прост български, всяка полилогаритмично растяща функция расте асимптотично по-бавно от всяка полиномиално растяща функция, а всяка полиномиално растяща функция расте асимптотично по-бавно от всяка експоненциално растяща функция.

И в края на подсекцията ще разгледаме асимптотични сравнения, включващи итерирани логаритми. Става дума за функции като $\lg \lg n$ (двоен логаритъм), $\lg \lg \lg n$ (троен логаритъм) и така нататък. Тъй като самият логаритъм е изключително бавно растяща функция, такива неколkokратни композиции на логаритъма със себе си водят до функции, които са практически константи. В смисъл, че за всяка стойност на аргумента, която може да възникне на практика, функционалната стойност може да се ограничи отгоре от малка константа. Примерно, нека $n = 1\,000\,000\,000\,000$. Тогава

$$\lg n \approx 39.86313714$$

$$\lg \lg n \approx 5.316983347$$

$$\lg \lg \lg n \approx 2.410607947$$

Ако увеличим аргумента до $n = 10^{15}$, имаме

$$\lg n \approx 49.82892142$$

$$\lg \lg n \approx 5.638911441$$

$$\lg \lg \lg n \approx 2.495416686$$

Дори за двойния логаритъм имаме право да кажем, че е константа **на практика**. Да не говорим за тройния, четворния и така нататък. Ако изследваме програма, която реализира алгоритъм със сложност по време $\Theta(f(n) \cdot \lg \lg n)$ за някаква “разумна” $f(n)$ като n или n^2 , можем спокойно да игнорираме множителя $\lg \lg n$ и да мислим, че сложността на програмата е $\Theta(f(n))$.

Но от **теоретична гледна точка**, всяка от тези функции е неограничено растяща, бивайки композиция от неограничено растящи функции. Поради това, за нас тези функции са именно такива: неограничено растящи.

Определение 32: Итериран логаритъм

За всяко цяло $i \geq 0$ и всяко реално $a > 1$, дефинираме *итерирания логаритъм* $\log_a^{(i)} n$ така:

$$\log_a^{(i)} n = \begin{cases} n, & \text{ако } i = 0 \\ \log_a \left(\log_a^{(i-1)} n \right), & \text{ако } i > 0 \text{ и } \log_a^{(i-1)} n > 0 \\ \text{недефинирано,} & \text{ако } i > 0 \text{ и } \log_a^{(i-1)} n < 0, \text{ или } \log_a^{(i-1)} n \text{ е недефинирано} \end{cases}$$

Забележете, че $\lg^{(i)} n$ **не е** същото като $\lg^i n$. $\lg^{(i)} n$ е итерираният логаритъм

$$\underbrace{\lg \lg \cdots \lg n}_{i \text{ пъти}}$$

докато $\lg^i n$ е $(\lg n)^i$, тоест, $\lg n$, повдигнат на i -та степен.

Забележете още, че записът $\lg^{(i)} n$ е по-прецизен от $\underbrace{\lg \lg \cdots \lg n}_{i \text{ пъти}}$, ако искаме да разглеж-

даме конкретни стойности на аргумента. Примерно, ако започнем с аргумент 3 и ползваме запис $\lg \cdots \lg 3$, ще получаваме

$$\begin{aligned} \lg 3 &\approx 1.584962501 \\ \lg \lg 3 &\approx .6644487077 \\ \lg \lg \lg 3 &\approx -.5897702603 \\ \lg \lg \lg \lg 3 &\approx -.7617750199 + 4.532360143 \cdot i \\ \lg \lg \lg \lg \lg 3 &\approx 2.200357409 + 2.506415242 \cdot i \\ &\dots \end{aligned}$$

А ако ползваме Определение 32, ще получим

$$\begin{aligned} \lg^{(1)} 3 &\approx 1.584962501 \\ \lg^{(2)} 3 &\approx .6644487077 \\ \lg^{(3)} 3 &\approx -.5897702603 \\ \lg^{(4)} 3 &\text{ е недефинирано} \end{aligned}$$

Разбира се, ако ни интересува само тенденцията при неограничено нарастване на аргумента, тези две интерпретации на $\lg^{(i)} n$ са еквивалентни.

Теорема 31: Итерирането на логаритъма дава по-бавно асимпт. нарастване

Нека $a > 1$ е реална константа, $i \in \mathbb{N}^+$, а $g(n)$ е неограничено растяща диференцируема функция. Тогава

$$\log_a^{(i+1)} g(n) < \log_a^{(i)} g(n)$$

Доказателство: Твърдението следва веднага от Теорема 29 с $k = 1$ и $c = 1$, ако $f(n)$ от Теорема 29 е $\log_a^{(i)} g(n)$. $\square$

2.3.6 Асимптотични еквивалентности на често срещани функции

“Истинската” апроксимация на Stirling казва, че

$$n! \sim \sqrt{2\pi n} \frac{n^n}{e^n}$$

За целите на тези лекции следният по-слаб резултат е напълно достатъчен. Това, че мултипликативната константа е $\sqrt{2\pi}$, е без значение за Тета-нотацията.

Теорема 32: Апроксимация на Stirling, слаба версия

В сила е

$$n! \asymp \sqrt{n} \cdot \frac{n^n}{e^n} \quad (2.65)$$

Доказателство Това доказателство е взето от статия на Dan Romik [134]. Ще докажем, че съществуват положителни константи c_1 и c_2 , такива че

$$\forall n \nexists : c_1 \cdot \sqrt{n} \frac{n^n}{e^n} \leq n! \leq c_2 \cdot \sqrt{n} \frac{n^n}{e^n}$$

Нотация 2: “{x}” вместо “x − [x]”

Ползваме следната нотация само в рамките на това доказателство. Нека x е реално число. “{x}” е кратък запис за $x - [x]$. Ако x е положително, то “{x}” означава дробната му част. Примерно, $\{2.4\} = 0.4$ и $\{2\} = 0$.

Нотацията не е особено удачна, понеже фигурните скоби типично се използват за множества, а в тази нотация, {x} **не е** множеството с единствен елемент x . Но този запис за остатъка е приет на практика, примерно в Concrete Mathematics на Knuth [61, (3.8) на стр. 70].

Нека $n \in \mathbb{N}^+$. Очевидно, в сила е

$$\ln n! = \sum_{k=1}^n \ln k = \sum_{k=1}^n (\ln k - 0) = \sum_{k=1}^n (\ln k - \ln 1) = \sum_{k=1}^n \left(\ln x \Big|_1^k \right) = \sum_{k=1}^n \int_1^k \frac{1}{x} dx \quad (2.66)$$

Твърдим, че

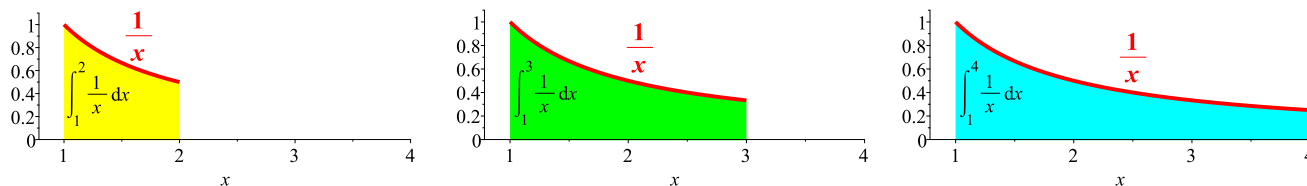
$$\sum_{k=1}^n \int_1^k \frac{1}{x} dx = \int_1^n \frac{n - [x]}{x} dx \quad (2.67)$$

Ще се убедим в истинността на (2.67) с малък пример. Нека $n = 4$. Тогава лявата страна на (2.67) е $\int_1^1 \frac{1}{x} dx + \int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx$. Тъй като $\int_1^1 \frac{1}{x} dx = 0$, лявата страна на (2.67) може да се запише като сума от три интеграла:

$$\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx \quad (2.68)$$

Ето илюстрация на трите интеграла от (2.68)[†].

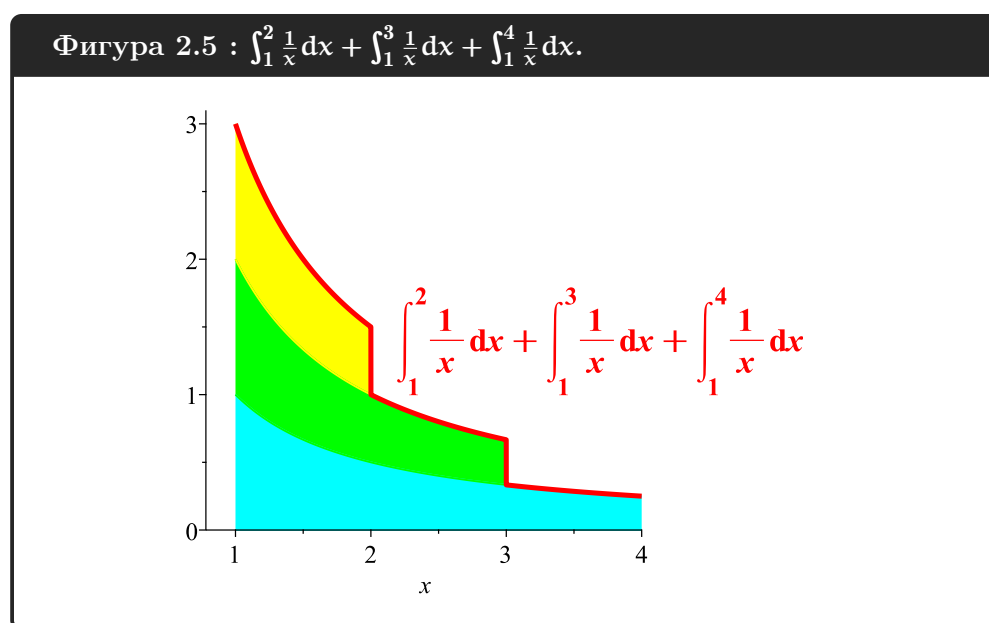
[†]Фигурите са направени с Maple(TM).



Твърдим, че

$$\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx = \int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx \quad (2.69)$$

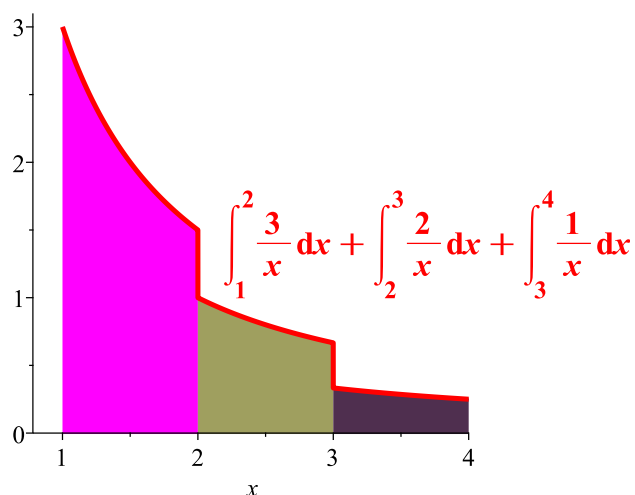
Нагледно неформално доказателство има на Фигура 2.5[‡] и Фигура 2.6[§]. Фигура 2.5 показва $\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx$ като трите площи са наложени една върху друга. Фигура 2.6 показва същия район от равнината, но сега разбит на площи съгласно $\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx$: върху $[1, 2]$ се сумират три от районите под хиперболата $\frac{1}{x}$, върху $[2, 3]$ се сумират два от районите под хиперболата $\frac{1}{x}$, а върху $[3, 4]$ имаме само района под $\frac{1}{x}$.



[‡]Фигура 2.5 е направена с Maple(ТМ).

[§]Фигура 2.6 е направена с Maple(ТМ).

Фигура 2.6 : $\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx$.



Обаче

$$\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx = \int_1^4 \frac{4 - [x]}{x} dx \quad (2.70)$$

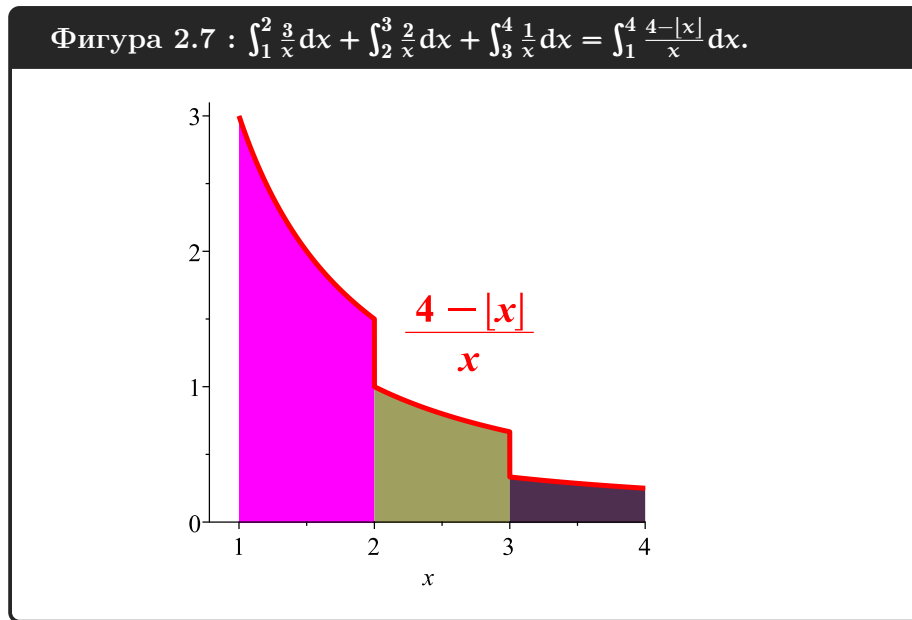
Причината е, че следните две функции $f_1(x)$ и $f_2(x)$ върху интервала $[1, 4]$ съвпадат:

$$f_1(x) = \begin{cases} \frac{3}{x}, & \text{ако } 1 \leq x \leq 2 \\ \frac{2}{x}, & \text{ако } 2 < x \leq 3 \\ \frac{1}{x}, & \text{ако } 3 < x \leq 4 \end{cases}$$

$$f_2(x) = \frac{4 - [x]}{x}$$

Уравнение (2.70) е илюстрирано на Фигура 2.7[†].

[†]Фигура 2.7 е направена с Maple(TM).



След като сме убедени в истинността на (2.67) и предвид очевидното $-\lfloor x \rfloor = \{x\} - x$, продължаваме така:

$$\ln n! = \int_1^n \frac{n - \lfloor x \rfloor}{x} dx = \int_1^n \frac{n + \{x\} - x}{x} dx \quad (2.71)$$

Ако сега продължим по този начин:

$$\begin{aligned} \int_1^n \frac{n + \{x\} - x}{x} dx &= \int_1^n \frac{n}{x} dx - \int_1^n \frac{x}{x} dx + \int_1^n \frac{\{x\}}{x} dx = \\ n \int_1^n \frac{1}{x} dx - \int_1^n 1 \cdot dx + \int_1^n \frac{\{x\}}{x} dx &= n \left(\ln x \Big|_1^n \right) - x \Big|_1^n + \int_1^n \frac{\{x\}}{x} dx = \\ n (\ln n - 0) - (n - 1) + \int_1^n \frac{\{x\}}{x} dx &= n \ln n - (n - 1) + \int_1^n \frac{\{x\}}{x} dx \end{aligned} \quad (2.72)$$

няма да стигнем до успех. Ясно е, че $\int_1^n \frac{\{x\}}{x} dx = O(\ln n)$ предвид факта, че $\frac{\{x\}}{x} < \frac{1}{x}$ навсякъде в $[1, n]$, тъй като $0 \leq \{x\} < 1$; ерго, $\int_1^n \frac{\{x\}}{x} dx < \int_1^n \frac{1}{x} dx$, а $\int_1^n \frac{1}{x} dx = \ln n$. Ерго, асимптотиката на сумата $n \ln n - (n - 1) + \int_1^n \frac{\{x\}}{x} dx$ (2.72) се определя от събираемото $n \ln n$. Тогава $\ln n! \asymp n \lg n$. Но този резултат е прекалено слаб. Ние го получаваме в Теорема 33 като леко следствие от Теорема 32. А за да докажем самата Теорема 32, трябва да направим нещо по-умно от извеждането на (2.72).

Продължаваме така:

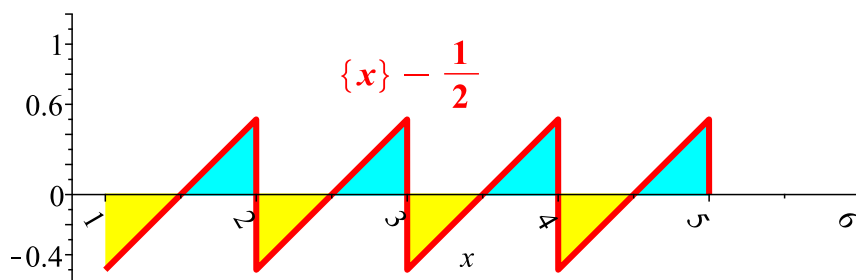
$$\int_1^n \frac{n + \{x\} - x}{x} dx = \int_1^n \frac{(n + \frac{1}{2}) - x + (\{x\} - \frac{1}{2})}{x} dx = \quad (2.73)$$

$$\left(n + \frac{1}{2} \right) \ln n - (n - 1) + \int_1^n \frac{\{x\} - \frac{1}{2}}{x} dx \quad (2.74)$$

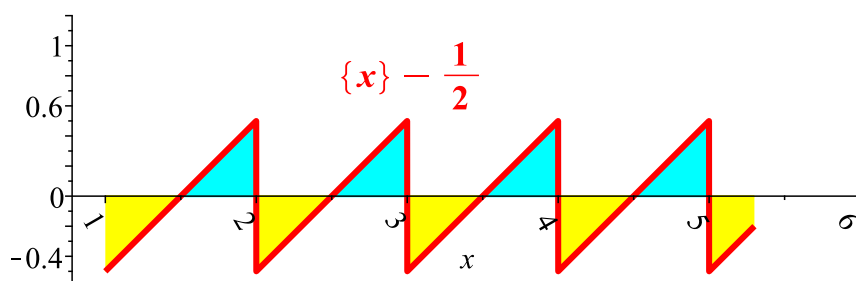
Да разгледаме $\int_1^n \frac{\{x\} - \frac{1}{2}}{x} dx$. Ще покажем, че това събираемо е ограничено отгоре и отдолу от константи. За целта разглеждаме $\int_1^{+\infty} \frac{\{x\} - \frac{1}{2}}{x} dx$. Но $\int_1^{+\infty} \frac{\{x\} - \frac{1}{2}}{x} dx$ е сходящ по критерия на Dirichlet (вижте например [111, Theorem 9.20, стр. 291–292]). Критерият е приложим по следните причини.

- $\int_1^y (\{x\} - \frac{1}{2}) dx$ за всяко $y > 1$ е ограничен от константа, макар и да не е сходящ. Той е лицето на отрязък от “трионообразна” фигура, като това лице е 0 за цели y , а за нецели y е число между $-\frac{1}{8}$ и 0.

Ето илюстрация за $y = 5$. Сумата от лицата на жълтите райони е равна на сумата от лицата на светлосините райони, но с обратен знак, така че интегралът е нула[†].



Ето илюстрация за $y = 5.3$. Интегралът е лицето на най-десния жълт район (с отрицателен знак), понеже останалите лица се канцелират взаимно[‡].



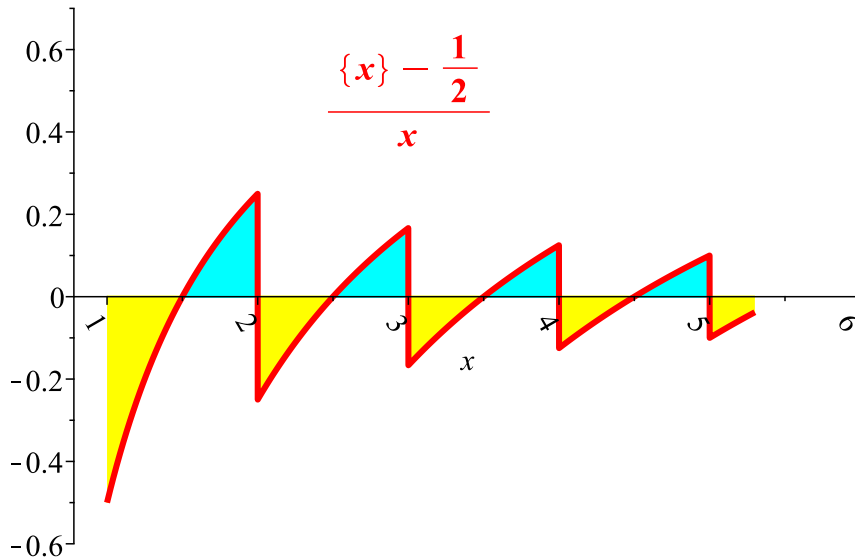
- Функцията $\{x\} - \frac{1}{2}$ е очевидно интегрируема по Riemann във всеки интервал $[1, y] \subset [1, +\infty)$.
- Функцията $\frac{1}{x}$ е непрекъснато диференцируема върху $(1, +\infty)$ и клони монотонно към 0, когато x клони към безкрайност.

Образно казано, множителят $\frac{1}{x}$ свива прогресивно “зъбите на триона” ето така[§]:

[†]Фигурата е направена с Maple(TM).

[‡]Фигурата е направена с Maple(TM).

[§]Фигурата е направена с Maple(TM).



Според Maple (TM), $\int_1^{+\infty} \frac{\{x\} - \frac{1}{2}}{x} dx \approx -0.810614668$.

Щом $\int_1^n \frac{\{x\} - \frac{1}{2}}{x} dx$ в (2.74) е ограничен отгоре и отдолу от константи, то съществуват константи c_3 и c_4 , такива че

$$\left(n + \frac{1}{2}\right) \ln n - n + c_3 \leq \ln n! \leq \left(n + \frac{1}{2}\right) \ln n - n + c_4$$

за всички достатъчно големи n . Но това е същото като

$$\left(\ln \frac{n^{n+\frac{1}{2}}}{e^n}\right) + c_3 \leq \ln n! \leq \left(\ln \frac{n^{n+\frac{1}{2}}}{e^n}\right) + c_4$$

Тогава, за някакви положителни константи c_1 и c_2 е в сила

$$c_1 \cdot \sqrt{n} \frac{n^n}{e^n} \leq n! \leq c_2 \cdot \sqrt{n} \frac{n^n}{e^n}$$

за всички достатъчно големи n . □

Една забележка върху доказателството на Теорема 32. В (2.73) извадихме и добавихме $\frac{1}{2}$. За да “работи” доказателството, трябва тази стойност да е именно $\frac{1}{2}$. Интегралът $\int_1^y \left(\{x\} - \frac{1}{2}\right) dx$ не дивергира при $y \rightarrow \infty$ само защото вадим точно $\frac{1}{2}$, поради което “зъбите на триона” — може би без един — над абсцисата се канцелират като площ с тези под абсцисата. Всяка друга стойност би накарала интеграла да дивергира. Забележете също така, че множителят $\sqrt{n}$ в асимптотиката на $n!$ се появи именно защото добавихме $\frac{1}{2}$ в (2.74).

Теорема 33: Асимптотиката на $\lg n!$

В сила е

$$\lg n! \asymp n \lg n \tag{2.75}$$

Доказателство: От Теорема 32 знаем, че $n! \asymp \sqrt{n} \frac{n^n}{e^n}$. Прилагаме Следствие 13 и получаваме:

$$\lg n! \asymp \lg \left(\sqrt{n} \frac{n^n}{e^n} \right)$$

Но

$$\lg \left(\sqrt{n} \frac{n^n}{e^n} \right) = \frac{1}{2} \lg n + n \lg n - n \lg e$$

Забелязваме, че $|-n \lg e + \frac{1}{2} \lg n| < n \lg n$ и прилагаме Теорема 26. Заклучаваме, че $\lg n! \asymp n \lg n$. $\square$

Теорема 34: Асимптотиката на $\sum_{k=1}^n k \lg k$

$$\sum_{k=1}^n k \lg k \asymp n^2 \lg n.$$

Доказателство: Нека $m = \lfloor \frac{n}{2} \rfloor$, $A = \sum_{k=1}^{m-1} k \lg k$ и $B = \sum_{k=m}^n k \lg k$. Първо ще покажем, че $B \asymp n^2 \lg n$. В сила е

$$\begin{aligned} \sum_{k=m}^n m \lg m &\leq \sum_{k=m}^n k \lg k \leq \sum_{k=m}^n n \lg n \leftrightarrow \\ (m \lg m)(n - m + 1) &\leq B \leq (n \lg n)(n - m + 1) \leftrightarrow \\ \underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor \right) \left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}_C &\leq B \leq \underbrace{(n \lg n) \left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}_D \end{aligned}$$

Очевидно $C \asymp n^2 \lg n$ и $D \asymp n^2 \lg n$. От това следва, че $B \asymp n^2 \lg n$.

Сега ще покажем, че $A \leq B$. A има $\frac{n-2}{2}$ събираеми, ако n е четно, и $\frac{n-3}{2}$ събираеми, ако n е нечетно. Във всеки случай, A има по-малко събираеми от B . Нещо повече, всяко събираемо на A е по-малко от всяко събираемо на B . Ерго, $A \leq B$. Щом $\sum_{k=1}^n k \lg k = A + B$, заключаваме, че $\sum_{k=1}^n k \lg k \asymp n^2 \lg n$. $\square$

Важно свойство на биномния коефициент е, че средният биномен коефициент $\binom{n}{n/2}$ расте, в асимптотичния смисъл, почти като 2^n . За удобство, да допуснем, че n е четно.

Теорема 35: Асимптотиката на $\binom{n}{n/2}$

В сила е

$$\binom{n}{\frac{n}{2}} \asymp \frac{2^n}{\sqrt{n}}$$

Доказателство: Знаем, че:

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\left(\frac{n}{2}!\right)^2}$$

Прилагаме Теорема 32 върху факториелите и получаваме

$$\frac{n!}{\left(\frac{n}{2}!\right)^2} \asymp \frac{\sqrt{n} \frac{n^n}{e^n}}{\left(\sqrt{\frac{n}{2}} \frac{\frac{n}{2}^{\frac{n}{2}}}{e^{\frac{n}{2}}}\right)^2} \asymp \sqrt{n} \frac{n^n}{e^n} \times \frac{e^n}{n^{\frac{n}{2}}} = \frac{2^n}{\sqrt{n}}$$

$\square$

Нотация 3: “ H_n ” означава n -тата парциална сума на хармоничния ред

С “ H_n ” означаваме n -тата парциална сума на хармоничния ред. Иначе казано:

$$H_n = \sum_{k=1}^n \frac{1}{k}$$

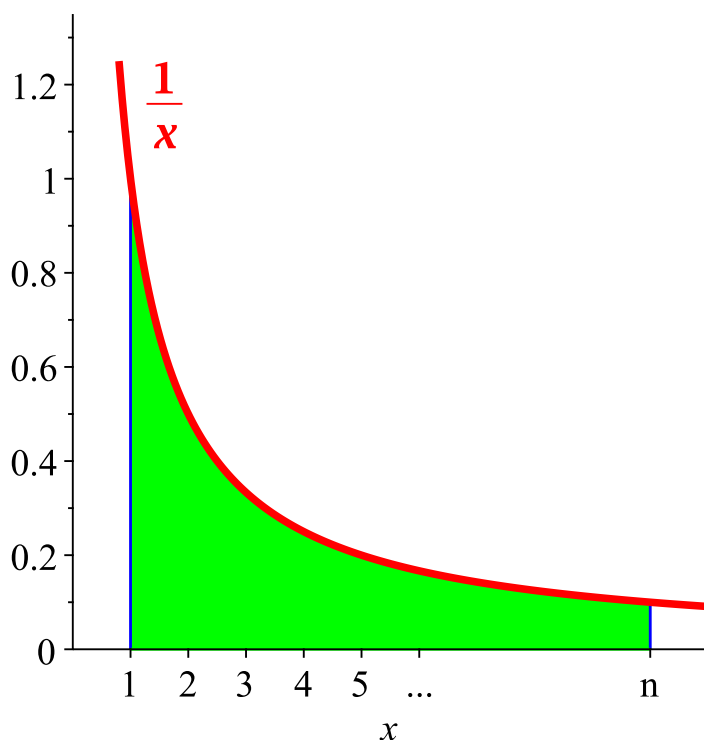
Теорема 36: Асимптотиката на H_n

В сила е

$$H_n \asymp \lg n$$

Доказателство: Ще ограничим $\int_1^n \frac{1}{x} dx$ отгоре и отдолу със стъпаловидни функции, отговарящи на парциални суми на хармоничния ред. Знаем, че за всяко $n \in \mathbb{N}^+$, $\int_1^n \frac{1}{x} dx$ е площта на района, ограден от графиката на $\frac{1}{x}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата. Това е илюстрирано на Фигура 2.8[†].

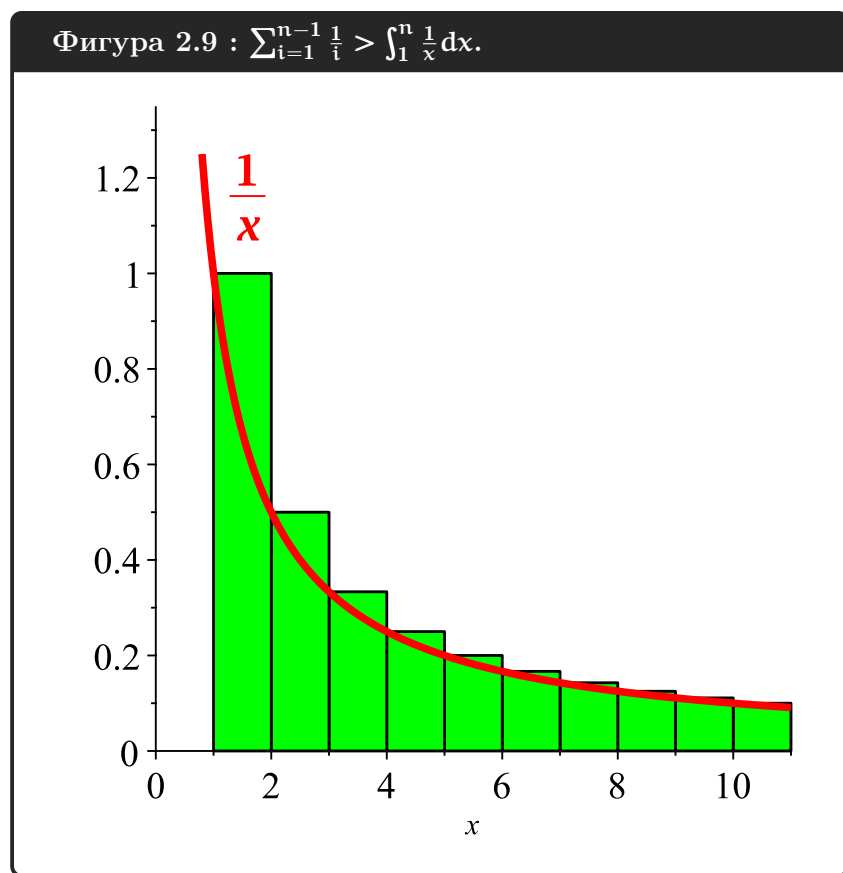
Фигура 2.8 : $\int_1^n \frac{1}{x} dx$ е площта на зеления район.



Да разгледаме стъпаловидната функция $\frac{1}{[x]}$ върху интервала $[1, n]$. Вярно е, че $\forall x \in [1, n] : \frac{1}{[x]} \geq \frac{1}{x}$, понеже $[x] \leq x$ за всяко x . Забелязваме, че площта на района, ограден от $\frac{1}{[x]}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата е точно $\sum_{i=1}^{n-1} \frac{1}{i}$, защото този район се разбива на $n - 1$ правоъгълника, едната страна на всеки от които е с дължина единица, а

[†]Фигура 2.8 е направена с Maple(TM).

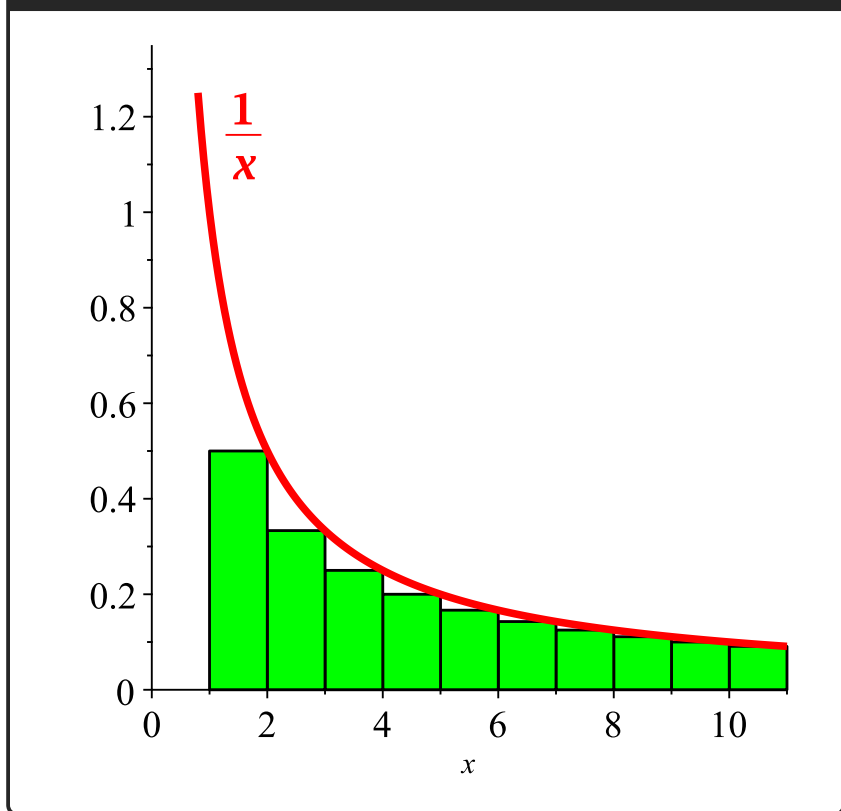
другата е $\frac{1}{i}$ за някое $i \in \{1, 2, \dots, n-1\}$. Очевидно този район има площ, по-голяма от площта под $\frac{1}{x}$. С други думи, $\sum_{i=1}^{n-1} \frac{1}{i} > \int_1^n \frac{1}{x} dx$. Това е илюстрирано на Фигура 2.9[†].



Сега да разгледаме стъпаловидната функция $\frac{1}{[x]}$ върху интервала $[1, n]$. $\forall x \in [1, n] : \frac{1}{[x]} \leq \frac{1}{x}$, понеже $[x] \geq x$ за всяко x . Забелязваме, че площта на района, ограден от $\frac{1}{[x]}$, абсцисата и двете вертикални линии през точките 1 и n от абсцисата е точно $\sum_{i=2}^n \frac{1}{i}$, защото този район се разбива на $n - 1$ правоъгълника, едната страна на всеки от които е с дължина единица, а другата е $\frac{1}{i}$ за някое $i \in \{2, 3, \dots, n\}$. Очевидно този район има площ, по-малка от площта под $\frac{1}{x}$. С други думи, $\int_1^n \frac{1}{x} dx > \sum_{i=2}^n \frac{1}{i}$. Това е илюстрирано на Фигура 2.10[‡].

[†]Фигура 2.9 е направена с Maple(TM).

[‡]Фигура 2.10 е направена с Maple(TM).

Фигура 2.10 : $\int_1^n \frac{1}{x} dx > \sum_{i=2}^n \frac{1}{i}$.

Покажем, че

$$\sum_{i=2}^n \frac{1}{i} < \int_1^n \frac{1}{x} dx < \sum_{i=1}^{n-1} \frac{1}{i}$$

Но $\sum_{i=2}^n \frac{1}{i} = \left(\sum_{i=1}^n \frac{1}{i}\right) - 1 = H_n - 1$, а $\sum_{i=1}^{n-1} \frac{1}{i} = H_{n-1}$. Тогава

$$H_n - 1 < \int_1^n \frac{1}{x} dx < H_{n-1}$$

Но $\int_1^n \frac{1}{x} dx = \ln x \Big|_1^n = \ln n - \ln 1 = \ln n - 0 = \ln n$. Тогава:

$$H_n - 1 < \ln n < H_{n-1}$$

Очевидно съществуват положителни константи c_1 и c_2 и стойност на аргумента n_0 , такива че за всички $n > n_0$ е вярно

$$c_1 \cdot H_n \leq \ln n \leq c_2 \cdot H_n$$

Например, $c_1 = \frac{1}{2}$, $c_2 = 1$ и $n_0 = 1000$.

Заклучаваме, че $\ln n \asymp H_n$.

□

2.3.7 Задачи с решения върху асимптотични сравнения

2.3.7.1 Свойства на релациите на асимптотични сравнения

Задача 3

Нека $f(n) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ и $g(n) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$. Известно е, че $f(n)$ и $g(n)$ са монотонно растящи. Известно е, че съществува естествено $d \geq 2$, такова че за всяко цяло положително k , ако $n = d^k$, то $f(n) \leq g(n)$. Известно е, че $g(n) \leq g\left(\frac{n}{d}\right)$. Докажете, че $f(n) \leq g(n)$.

Решение: Дадени са три ограничения за f и g . Първото е, че са монотонно растящи. Функция $\phi(n)$ е монотонно растяща тстк

$$n_1 \leq n_2 \rightarrow \phi(n_1) \leq \phi(n_2)$$

Второто ограничение казва следното. Разглеждаме числата, които са точни степени на d . Ограничението казва, че

$$\begin{aligned} f(d) &\leq g(d) \\ f(d^2) &\leq g(d^2) \\ f(d^3) &\leq g(d^3) \\ f(d^4) &\leq g(d^4) \\ &\dots \end{aligned}$$

Третото ограничение казва, че съществува положителна константа c , такава че за всички достатъчно големи n е изпълнено

$$g(n) \leq c \cdot g\left(\frac{n}{d}\right)$$

В частност, ако $n = d^{k+1}$, то

$$g(d^{k+1}) \leq c \cdot g(d^k)$$

Лесно се вижда, че всъщност $c \geq 1$ заради монотонното нарастване на g ; ако допуснем, примерно, $c = \frac{1}{2}$, би следвало

$$g(d^2) \leq \frac{1}{2}g(d) \leftrightarrow g(d) \geq 2g(d^2)$$

което е несъвместимо с монотонното нарастване на g и факта, че $d < d^2$.

Иска се да докажем, че съществува положителна константа c_1 и положителна стойност n_0 на аргумента, такива че за $n \geq n_0$ е изпълнено

$$f(n) \leq c_1 \cdot g(n) \tag{2.76}$$

Разглеждаме два случая.

- n е точна степен на d . По условие имаме $f(n) \leq 1 \cdot g(n)$.
- n не е точна степен на d . Очевидно съществува цяло положително k , такова че

$$d^k < n < d^{k+1}$$

Тъй като f е монотонно растяща, в сила е

$$f(n) \leq f(d^{k+1})$$

Но $f(d^{k+1}) \leq g(d^{k+1})$ по условие. Тогава

$$f(n) \leq g(d^{k+1})$$

Както вече знаем, $g(d^{k+1}) \leq c \cdot g(d^k)$. Тогава

$$f(n) \leq c \cdot g(d^k)$$

Тъй като g е монотонно растяща, в сила е

$$g(d^k) \leq g(n)$$

Заклучаваме, че

$$f(n) \leq c \cdot g(n)$$

С което доказахме (2.76). Избираме $n_0 = d$ и $c_1 = c$. Тъй като $c \geq 1$, изборът $c_1 = c$ “покрива” и двата случая, които разгледахме поотделно. $\square$

Задача 4

Нека $a \in \mathbb{R}$ и $b \in \mathbb{R}^+$. Докажете, че $(n + a)^b \asymp n^b$.

Решение: Ще докажем твърдението педантично, използвайки Определение 20. Ще намерим положителни константи c_1 , c_2 и стойност на аргумента n_0 , такива че

$$0 \leq c_1 \cdot n^b \leq (n + a)^b \leq c_2 \cdot n^b$$

за всяко $n \geq n_0$.

Очевидно

$$n + a \leq n + |a| \leq 2n, \text{ ако } n \geq |a|$$

$$n + a \geq n - |a| \geq \frac{n}{2}, \text{ ако } \frac{n}{2} \geq |a|; \text{ тоест, } n \geq 2|a|$$

Тогава

$$\frac{1}{2}n \leq n + a \leq 2n, \text{ ако } n \geq 2|a|$$

Повдигаме на степен b и получаваме

$$\left(\frac{1}{2}\right)^b n^b \leq (n + a)^b \leq 2^b n^b$$

Вече имаме доказателство със $c_1 = \left(\frac{1}{2}\right)^b$, $c_2 = 2^b$ и $n_0 = \lceil 2|a| \rceil$. $\square$

Задача 5

Докажете или опровергайте, че за всеки две асимптотично положителни функции $f(n)$ и $g(n)$ е в сила

$$\max(f(n), g(n)) \asymp f(n) + g(n)$$

Решение: Твърдението е вярно. Ще докажем, че съществуват положителни константи c_1 и c_2 и стойност на аргумента n_0 , такива че $\forall n \geq n_0$:

$$0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$$

Щом $f(n)$ и $g(n)$ са асимптотично положителни,

$$\exists n'_0 \forall n \geq n'_0 : f(n) > 0$$

$$\exists n''_0 \forall n \geq n''_0 : g(n) > 0$$

Нека $n'''_0 = \max\{n'_0, n''_0\}$. Очевидно за всяко $n \geq n'''_0$ е вярно

$$0 \leq c_1(f(n) + g(n)),$$

ако $c_1 > 0$. Също така е очевидно, че за всяко $n \geq n'''_0$:

$$0 \leq \frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \text{ и} \\ f(n) + g(n) \geq \max(f(n), g(n)),$$

което можем да запишем като

$$0 \leq \frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

Имаме доказателство с $n_0 = n'''_0$, $c_1 = \frac{1}{2}$ и $c_2 = 1$. □

Задача 6

Вярно ли е, че за всеки две асимптотично положителни функции $f(n)$ и $g(n)$ е в сила

$$\max(f(n), g(n)) \asymp f(n) - g(n)$$

Допуснете, че $f(n) - g(n)$ е асимптотично положителна функция.

Решение: Това не е вярно. Контрапример е $f(n) = n^3 + n^2$ и $g(n) = n^3 + n$. Очевидно $\max(f(n), g(n)) = n^3 + n^2 = f(n)$, за всички достатъчно големи n . Очевидно $f(n) - g(n) = n^2 - n$, като $f(n) - g(n)$ е асимптотично положителна функция. Обаче $n^3 + n^2 \not\asymp n^2 - n$. □

Задача 7

Докажете или опровергайте, че за всеки две асимптотично положителни функции $f(n)$ и $g(n)$ е в сила

$$\min(f(n), g(n)) \asymp f(n) + g(n)$$

Решение: Твърдението не е вярно. Разгледайте $f(n) = n$ и $g(n) = 1$ като контрапример. Тогава $\min(f(n), g(n)) = 1$ и $f(n) + g(n) = n + 1$, но $n + 1 \neq 1$. $\square$

Задача 8

Вярно ли е, че за всяка асимптотично положителна функция $f(n)$ е в сила

$$f(n) \leq (f(n))^2?$$

Решение: В общия случай не е вярно. Да разгледаме като контрапример $f(n) = \frac{1}{n}$, която е асимптотично положителна функция. Да, това е намаляваща функция, която едва ли би се появила в анализа на естествени алгоритми, но, формално, щом е асимптотично положителна, имаме право да я разглеждаме. Имаме

$$\lim_{n \rightarrow \infty} \frac{f(n)}{(f(n))^2} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{n^2}} = \lim_{n \rightarrow \infty} \frac{n}{1} = \infty$$

Съгласно Наблюдение 17, в сила е $f(n) > (f(n))^2$. Тогава, съгласно Теорема 22 (2.47), в сила е $f(n) \not\leq (f(n))^2$.

Ако обаче $f(n)$ е неограничено растяща, твърдението е вярно:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{(f(n))^2} = \lim_{n \rightarrow \infty} \frac{1}{f(n)} = 0$$

Тогава от Наблюдение 17 имаме $f(n) < (f(n))^2$, което, съгласно Теорема 20 (2.40), влече $f(n) \leq (f(n))^2$. $\square$

Задача 9

Нека $f(n)$ е асимптотично положителна функция. Кое от следните твърдения е вярно и кое не е вярно?

$$f\left(\frac{n}{2}\right) \asymp f(n) \tag{2.77}$$

$$f\left(\frac{n}{2}\right) < f(n) \tag{2.78}$$

$$f\left(\frac{n}{2}\right) \leq f(n) \tag{2.79}$$

Решение: Нито едно не е вярно. Като контрапример за (2.77) разгледайте $f(n) = 2^n$. Тогава

$$\lim_{n \rightarrow \infty} \frac{f\left(\frac{n}{2}\right)}{f(n)} = \lim_{n \rightarrow \infty} \frac{2^{\frac{n}{2}}}{2^n} = \lim_{n \rightarrow \infty} \frac{1}{2^{\frac{n}{2}}} = 0$$

така че $f\left(\frac{n}{2}\right) < f(n)$. Съгласно Теорема 23 (2.49), в сила е $f\left(\frac{n}{2}\right) \neq f(n)$.

Като контрапример за (2.78) разгледайте $f(n) = 2n$. Тогава $f\left(\frac{n}{2}\right) = n$ и очевидно

$$f\left(\frac{n}{2}\right) \asymp f(n)$$

Съгласно Теорема 23 (2.48), в сила е $f\left(\frac{n}{2}\right) \not\leq f(n)$.

Като контрапример за (2.79) разгледайте $f(n) = \frac{1}{2^n}$. Тогава $f\left(\frac{n}{2}\right) = \frac{1}{2^{\frac{n}{2}}}$, така че

$$\lim_{n \rightarrow \infty} \frac{f\left(\frac{n}{2}\right)}{f(n)} = \lim_{n \rightarrow \infty} \frac{\frac{1}{2^{\frac{n}{2}}}}{\frac{1}{2^n}} = \lim_{n \rightarrow \infty} \frac{2^n}{2^{\frac{n}{2}}} = \lim_{n \rightarrow \infty} 2^{\frac{n}{2}} = \infty$$

така че $f\left(\frac{n}{2}\right) > f(n)$. Съгласно Теорема 22 (2.47), в сила е $f\left(\frac{n}{2}\right) \not\asymp f(n)$. $\square$

Задача 10

Нека k е положителна константа и n е цяло число. Докажете, че

$$1 + k + k^2 + k^3 + \dots + k^n \asymp \begin{cases} 1, & \text{ако } k < 1 \\ n, & \text{ако } k = 1 \\ k^n, & \text{ако } k > 1 \end{cases}$$

Решение: Добре известно е, че

$$1 + k + k^2 + k^3 + \dots + k^n = \begin{cases} \frac{k^{n+1}-1}{k-1}, & \text{ако } k \neq 1 \\ n+1, & \text{ако } k = 1 \end{cases} \quad (2.80)$$

Ако $0 < k < 1$, геометричният ред $\sum_{t=0}^{\infty} k^t$ е сходящ със сума $\frac{1}{1-k}$, така че

$$1 + k + k^2 + k^3 + \dots + k^n < \sum_{t=0}^{\infty} k^t = \frac{1}{1-k} \asymp 1$$

Ако $k = 1$, имаме

$$1 + k + k^2 + k^3 + \dots + k^n = n + 1 \asymp n$$

Ако $k > 1$, от (2.80) имаме

$$1 + k + k^2 + k^3 + \dots + k^n = \frac{k^{n+1}-1}{k-1} = \frac{k \cdot k^n}{k-1} - \frac{1}{k-1} = \frac{k}{k-1} k^n + \frac{1}{1-k}$$

Но $\frac{k}{k-1}$ е положителна константа, а $\left|\frac{1}{1-k}\right| < \frac{k}{k-1} k^n$. Прилагаме Теорема 26 и заключаваме, че $1 + k + k^2 + k^3 + \dots + k^n \asymp k^n$. $\square$

Задача 11

Нека $f(n)$, $g(n)$ и $h(n)$ са асимптотично положителни функции, такива че $f(n) \asymp g(n)$ и $h(n) \asymp g(n)$. Докажете или опровергайте всяко от следните твърдения:

$$\begin{aligned} \sqrt{f(n)h(n)} &\asymp g(n) \\ \sqrt{(f(n))^2 + (h(n))^2} &\asymp g(n) \end{aligned}$$

Решение: И двете твърдения са верни. По условие,

$$\exists c_1 > 0 \exists c_2 > 0 \exists n_1 > 0 \forall n \geq n_1 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\exists c_3 > 0 \exists c_4 > 0 \exists n_2 > 0 \forall n \geq n_2 : c_3 g(n) \leq h(n) \leq c_4 g(n)$$

Нека $n' = \max\{n_1, n_2\}$. За всяко n , такова че $n \geq n'$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad (2.81)$$

$$c_3 g(n) \leq h(n) \leq c_4 g(n) \quad (2.82)$$

Ако умножим (2.81) и (2.82), получаваме

$$c_1 c_3 (g(n))^2 \leq f(n) h(n) \leq c_2 c_4 (g(n))^2$$

Тъй като функциите са асимптотично положителни, БОО можем да допуснем, че и трите са положителни за стойности на аргумента $\geq n'$. Тогава имаме право да коренуваме. Коренувайки, получаваме

$$\sqrt{c_1 c_3} g(n) \leq \sqrt{f(n) h(n)} \leq \sqrt{c_2 c_4} g(n)$$

Следователно, съществуват константи c' и c'' , а именно $c' = \sqrt{c_1 c_3}$ и $c'' = \sqrt{c_2 c_4}$ и стойност на аргумента n' , такива че за всяко $n \geq n'$:

$$c' g(n) \leq \sqrt{f(n) h(n)} \leq c'' g(n)$$

Заклучаваме, че $\sqrt{f(n) h(n)} \asymp g(n)$.

Сега да повдигнем (2.81) и (2.82) на квадрат. Получаваме

$$c_1^2 (g(n))^2 \leq (f(n))^2 \leq c_2^2 (g(n))^2 \quad (2.83)$$

$$c_3^2 (g(n))^2 \leq (h(n))^2 \leq c_4^2 (g(n))^2 \quad (2.84)$$

Тогава

$$\begin{aligned} c_1^2 (g(n))^2 + c_3^2 (g(n))^2 &\leq (f(n))^2 + (h(n))^2 \leq c_2^2 (g(n))^2 + c_4^2 (g(n))^2 \leftrightarrow \\ (c_1^2 + c_3^2) (g(n))^2 &\leq (f(n))^2 + (h(n))^2 \leq (c_2^2 + c_4^2) (g(n))^2 \leftrightarrow \\ \sqrt{c_1^2 + c_3^2} g(n) &\leq \sqrt{(f(n))^2 + (h(n))^2} \leq \sqrt{c_2^2 + c_4^2} g(n) \end{aligned}$$

Следователно, съществуват константи c' и c'' , а именно $c' = \sqrt{c_1^2 + c_3^2}$ и $c'' = \sqrt{c_2^2 + c_4^2}$ и стойност на аргумента n' , такива че за всяко $n \geq n'$:

$$c' g(n) \leq \sqrt{(f(n))^2 + (h(n))^2} \leq c'' g(n)$$

Заклучаваме, че $\sqrt{(f(n))^2 + (h(n))^2} \asymp g(n)$. □

Задача 12

Нека $f(n)$ и $g(n)$ са асимптотично положителни функции. Докажете или опровергайте, че ако $f(n) \leq n$, то $g(f(n)) \leq g(n)$.

Решение: Твърдението не е вярно. Контрапример е $f(n) = 2n$ и $g(n) = 2^n$. Тогава $g(f(n)) = 2^{2n} = 4^n$. Но $4^n \not\leq 2^n$. □

Задача 13

Нека $n \in \mathbb{N}$ и $f(n) \asymp g(n)$. Докажете или опровергайте, че

- а) $2f(n) \geq g(n) + 2$;
- б) $2f(n) \geq g(n + 2)$;
- в) $2f(n) \geq g(n + 2)$ или $2f(n) \leq g(n + 2)$.

Решение:

- а) Твърдението не е вярно. Щеше да е вярно, ако функциите бяха растящи, но такова нещо не е казано. Щом използваме асимптотичните нотации, то налице е имплицитно допускане, че функциите са асимптотично положителни. Друго имплицитно допускане няма.

Като контрапример да разгледаме $f(n) = g(n) = \frac{1}{n}$. Тези функции са положителни за положителни стойности на n , следователно имаме право да ги разглеждаме. Със сигурност $f(n) \asymp g(n)$ е вярно щом функциите съвпадат. Да допуснем, че $2f(n) \geq g(n) + 2$. По дефиниция това е кратък запис на

$$\exists c' \exists n'_0 \forall n \geq n'_0 : c' \frac{1}{n} + 2 \leq 2 \frac{1}{n}$$

Но тогава

$$\exists c' \exists n'_0 \forall n \geq n'_0 : c' + 2n \leq 2$$

Това е абсурд, имайки предвид, че c' е константа, а n е променлива, която расте неограничено.

- б) Твърдението не е вярно. Дори ако разгледаме растящи функции, пак не е вярно. Нека $f(n) = g(n) = 2^{2^n}$. Със сигурност $f(n) \asymp g(n)$ е вярно щом функциите съвпадат.

Но забележете, че $g(n + 1) = 2^{2^{n+1}} = (g(n))^2 = (f(n))^2$. Тогава $g(n + 2) = (f(n))^4$. Очевидно не е вярно, че $2f(n) \geq (f(n))^4$ за растяща функция $f(n)$.

- в) Твърдението не е вярно. Нека отново $f(n) = g(n)$, дефинирани върху $\mathbb{N}^+$ така

$$f(n) = g(n) = \begin{cases} 2^{2^n}, & \text{ако } n \text{ е четно} \\ 1, & \text{ако } n \bmod 4 = 1 \\ 2^{2^n}, & \text{ако } n \bmod 4 = 3 \end{cases}$$

Както вече видяхме, $2^{2^{n+2}} = (2^{2^n})^4$.

От една страна, не е вярно, че $2f(n) \geq g(n + 2)$, защото стойността на функцията върху $n + 2$ е четвъртата степен на стойността на функцията върху n . По-подробно казано, за всяка предварително избрана константа c и за всяко n_0 съществува (четно) $n \geq n_0$, такива че $2f(n) < cg(n + 2)$. Но това е точно логическото отрицание на условието за $2f(n) \geq g(n + 2)$.

Да разгледаме нечетните числа. Ще разглеждаме двойки последователни нечетни числа, по-малкото от които дава остатък 3 по модул 4, което означава, че по-голямото

(тоест, следващото нечетно число) дава остатък 1 по модул 4. Съществуват безброй много такива двойки. Тогава съществуват безброй много двойки $n, n + 2$, такива че $f(n) = g(n) = 2^{2^n}$ и $f(n + 2) = g(n + 2) = 1$. И така, за всяка предварително избрана константа c и за всяко n_0 съществува $n \geq n_0$, такива че $2f(n) > cg(n + 2)$. Но това е точно логическото отрицание на условието за $2f(n) \leq g(n + 2)$.

Докажем, че нито $2f(n) \geq g(n + 2)$, нито $2f(n) \leq g(n + 2)$. □

2.3.7.2 Асимптотични сравнения на двойки функции

Определение 33: Сравняване на двойка функции по асимптотично нарастване

Нека са дадени асимптотично положителни функции $f(n)$ и $g(n)$. Да бъдат сравнени по асимптотично нарастване означава да се определи коя от шестте възможности съгласно Теорема 24—или (2.52), или (2.53), или (2.54), или (2.55), или (2.56), или (2.57)—е в сила.

Наблюдение 18

Ако $f(n) < g(n)$, Теорема 24 казва, че задължително $f(n) \leq g(n)$. Дуално, ако $f(n) > g(n)$, то задължително $f(n) \geq g(n)$. И накрая, ако $f(n) \asymp g(n)$, то задължително $f(n) \leq g(n)$ и $f(n) \geq g(n)$. Поради това, за краткост в описанието на решението на задача от вида “Сравнете по асимптотично нарастване $f(n)$ и $g(n)$ ”:

- ако $f(n) < g(n)$, не пишем и $f(n) \leq g(n)$; то се подразбира,
- ако $f(n) > g(n)$, не пишем и $f(n) \geq g(n)$; то се подразбира,
- ако $f(n) \asymp g(n)$, не пишем нито $f(n) \leq g(n)$, нито $f(n) \geq g(n)$; те се подразбират.

Задача 14

Сравнете $f(n) = n^{\lg n}$ и $g(n) = (\lg n)^n$ по асимптотично нарастване.

Решение: Да логаритмуваме двете функции. Получаваме

$$\lg f(n) = (\lg n)(\lg n) = \lg^2 n$$

$$\lg g(n) = n \cdot \lg \lg n$$

Но $n \cdot \lg \lg n > \lg^2 n$, защото $\lg \lg n$ е неограничено растяща. От друга страна, $n > \lg^2 n$ от Следствие 14. Тогава $n \cdot \lg \lg n > \lg \lg n$ от транзитивността на релацията $>$. Тоест, $\lg g(n) > \lg f(n)$. Съгласно Следствие 12, в сила е $g(n) > f(n)$. □

Задача 15

Сравнете $\lfloor \lg n \rfloor!$ и $\lceil \lg n \rceil!$ по асимптотично нарастване.

Решение: Приели сме, че $n \in \mathbb{R}^+$. Ако $\lg n \in \mathbb{R}^+ \setminus \mathbb{N}^+$, в сила е $\lceil \lg n \rceil = \lfloor \lg n \rfloor + 1$, откъдето

$$\lceil \lg n \rceil! = (\lfloor \lg n \rfloor + 1) \lfloor \lg n \rfloor! \tag{2.85}$$

Тогава $\lfloor \lg n \rfloor! \neq \lceil \lg n \rceil!$, защото

$$\neg(\exists c > 0 \exists n_0 \forall n \geq n_0 : \lfloor \lg n \rfloor! \leq c \lceil \lg n \rceil!) \leftrightarrow \\ \forall c > 0 \forall n_0 \exists n \geq n_0 : \lfloor \lg n \rfloor! > c \lceil \lg n \rceil!$$

Последното е очевидно предвидно предвид (2.85).

От друга страна, $\lfloor \lg n \rfloor! \prec \lceil \lg n \rceil!$, защото за всяко n , което е точна степен на двойката, имаме $\lfloor \lg n \rfloor = \lceil \lg n \rceil$.

Със сигурност е вярно, че $\lfloor \lg n \rfloor! \leq \lceil \lg n \rceil!$, тъй като $\lfloor \lg n \rfloor! \leq \lceil \lg n \rceil!$ за всяко n .

Щом $\lfloor \lg n \rfloor! \neq \lceil \lg n \rceil!$ и $\lfloor \lg n \rfloor! \prec \lceil \lg n \rceil!$ и $\lfloor \lg n \rfloor! \leq \lceil \lg n \rceil!$, единствената релация измежду петте релации на асимптотични сравнения, която е в сила за въпросните две функции в този ред, е $\leq$; това отговаря на (2.53) в Теорема 24. Ситуацията е аналогична на сравнението на $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$ на стр. 122. $\square$

Задача 16

Сравнете $f(n) = n^{\lg \lg \lg n}$ и $g(n) = (\lg n)!$ по асимптотично нарастване.

Решение: Възниква въпросът как да интерпретираме факториела на логаритъма при положение, че логаритъмът може да не е цяло число. Не е добра идея да запишем $g(n)$ като $\lfloor \lg n \rfloor!$ или $\lceil \lg n \rceil!$, защото тези две функции не са асимптотично еквивалентни (Задача 15). Ако факториелът в $(\lg n)!$ се апроксимира с апроксимацията на Stirling (Теорема 32), получаваме

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{e^{\lg n}} = \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}} \quad (2.86)$$

Тук ще разгледаме не директно $(\lg n)!$, а логаритъмът на тази функция.

Да логаритмуваме двете функции. Получаваме $\lg f(n) = (\lg n) \cdot (\lg \lg \lg n)$ и $\lg g(n) = \lg((\lg n)!)$. Да заместим $\lg n$ с m в $g(n)$. Получаваме $g(n) = m!$. Тогава $\lg g(n) = \lg(m!)$. Но ние знаем от Теорема 33, че $\lg m! \asymp m \lg m$. Тогава $\lg g(n) \asymp m \lg m$. Това е същото като $\lg g(n) \asymp (\lg n) \cdot (\lg \lg n)$. Същият резултат се получава и при логаритмуване на двете страни на (2.86).

Но $(\lg n) \cdot (\lg \lg n) > (\lg n) \cdot (\lg \lg \lg n)$, защото $\lg \lg n > \lg \lg \lg n$, което следва от Теорема 31. Тоест, $\lg g(n) > \lg f(n)$. Съгласно Следствие 12, в сила е $g(n) > f(n)$. $\square$

Задача 17

Нека $n!! = (n!)!$. Сравнете $f(n) = n!!$ и $g(n) = (n-1)!! \cdot ((n-1)!)^{n!}$ по асимптотично нарастване.

Решение: Първо да опитаме с логаритмуване. Имаме

$$\lg f(n) = \lg(n!!) \\ \lg g(n) = \lg((n-1)!! \cdot ((n-1)!)^{n!}) = \lg((n-1)!!) + n! \cdot \lg((n-1)!)$$

Прилагаме Теорема 33 и получаваме

$$\lg f(n) \asymp n! \cdot \lg n! \asymp n! \cdot n \cdot \lg n \\ \lg g(n) \asymp (n-1)! \cdot \lg(n-1)! + n! \cdot (n-1) \lg(n-1) \asymp \\ (n-1)! \cdot (n-1) \cdot \lg(n-1) + n! \cdot (n-1) \cdot \lg(n-1)$$

Да разгледаме $\lg g(n)$. Твърдим, че $n! \cdot (n-1) \cdot \lg(n-1) > (n-1)! \cdot (n-1) \cdot \lg(n-1)$. Наистина,

$$\lim_{n \rightarrow \infty} \frac{n! \cdot (n-1) \cdot \lg(n-1)}{(n-1)! \cdot (n-1) \cdot \lg(n-1)} = \lim_{n \rightarrow \infty} \frac{n \cdot (n-1)! \cdot (n-1) \cdot \lg(n-1)}{(n-1)! \cdot (n-1) \cdot \lg(n-1)} = \lim_{n \rightarrow \infty} n = \infty$$

така че, съгласно Наблюдение 17, наистина $n! \cdot (n-1) \cdot \lg(n-1) > (n-1)! \cdot (n-1) \cdot \lg(n-1)$. Прилагаме Теорема 26 и заключаваме, че $\lg g(n) \asymp n! \cdot (n-1) \cdot \lg(n-1)$. И така,

$$\lg f(n) \asymp n! \cdot n \cdot \lg n$$

$$\lg g(n) \asymp n! \cdot (n-1) \cdot \lg(n-1)$$

Но тогава $\lg f(n) \asymp \lg g(n)$. От този резултат не можем да заключим нищо за асимптотичното сравнение на $f(n)$ и $g(n)$.

С Maple(TM) табулираме стойности, точни и приблизителни, на $f(n)$ и $g(n)$ за малки n .

n	1	2	3	4	5	6	7	10
$f(n)$	1	2	720	$\approx 6 \times 10^{23}$	$\approx 6 \times 10^{198}$	$\approx 2 \times 10^{1746}$	$\approx 4 \times 10^{16473}$	$\approx 9 \times 10^{22228103}$
$g(n)$	1	1	128	$\approx 3 \times 10^{21}$	$\approx 2 \times 10^{189}$	$\approx 6 \times 10^{1695}$	$\approx 2 \times 10^{16147}$	$\approx 2 \times 10^{22035201}$

От тези числени експерименти става ясно, че $f(n)$ расте **много** по-бързо от $g(n)$. Интуитивно е ясно, че частното $\frac{f(n)}{g(n)}$ не може да бъде ограничено отгоре от константа при неограничено нарастване на n . Как да покажем обаче, че $f(n) > g(n)$?

Да положим $(n-1)! = v$. С оглед на тази субституция имаме $n! = nv$, така че сравняваме по асимптотично нарастване $\phi(n, v) = (nv)!$ и $\psi(n, v) = v! \times v^{nv}$. Да приложим апроксимацията на Stirling към ϕ и ψ .

$$\phi(n, v) \asymp \sqrt{nv} \frac{(nv)^{nv}}{e^{nv}} = \sqrt{2\pi nv} \frac{n^{nv} \cdot v^{nv}}{e^{nv}}$$

$$\psi(n, v) \asymp \sqrt{v} \frac{v^v}{e^v} \times v^{nv}$$

Да разгледаме частното на двете функции в десните страни.

$$\frac{\sqrt{nv} \frac{n^{nv} \cdot v^{nv}}{e^{nv}}}{\sqrt{v} \frac{v^v}{e^v} \times v^{nv}} = \sqrt{n} \cdot \frac{\frac{n^{nv}}{e^{nv}}}{\frac{v^v}{e^v}} = \sqrt{n} \cdot \frac{n^{nv}}{e^{(n-1)v} \cdot v^v}$$

Да игнорираме множителя $\sqrt{n}$. Ако и без него получим, че частното клони към безкрайност, неговото връщане няма да промени този факт, понеже $\sqrt{n}$ е неограничено растяща функция. И така, разглеждаме

$$\frac{n^{nv}}{e^{(n-1)v} \cdot v^v} = \left(\frac{n^n}{e^{(n-1)} \cdot v} \right)^v$$

Да разгледаме основата $\frac{n^n}{e^{(n-1)} \cdot v}$. Ще покажем, че това частно клони към безкрайност при неограничено нарастване на n , и така ще получим желанния резултат. Забележете, че това е същото като да вземем $\sqrt[n]{\frac{n^n}{e^{(n-1)} \cdot v}}$, тоест, $\sqrt[n]{\frac{n^n}{e^{(n-1)} \cdot v}}$. За да се освободим от v в частното, спомняме

си, че $v = (n-1)!$, и прилагаме апроксимацията на Stirling за $(n-1)!$.

$$\frac{n^n}{e^{(n-1)} \cdot v} = \frac{n^n}{e^{(n-1)} \cdot (n-1)!} \asymp \frac{n^n}{e^{(n-1)} \cdot \sqrt{n-1} \frac{(n-1)^{n-1}}{e^{n-1}}} = \frac{1}{\sqrt{n-1}} \cdot \frac{n^{n-1} \cdot n}{(n-1)^{n-1}} =$$

$$\frac{n}{\sqrt{n-1}} \cdot \left(\frac{n}{n-1}\right)^{n-1}$$

Но очевидно

$$\lim_{n \rightarrow \infty} \frac{n}{\sqrt{n-1}} \cdot \left(\frac{n}{n-1}\right)^{n-1} = \infty$$

откъдето следва, че $f(n) > g(n)$. □

2.3.7.3 Подреждане на функции по асимптотично нарастване

Често срещана задача е тази: дадени са функции $f_1(n), f_2(n), \dots, f_k(n)$ и се иска те да бъдат подредени по асимптотично нарастване. **Винаги** в такава задача допускаме неявно, че всеки две от функциите са асимптотично сравними; с други думи, (2.52) от Теорема 24 не се допуска. Нещо повече, (2.53) и (2.57) от Теорема 24 не се допускат, така че единствено възможни са (2.54), (2.55) и (2.56). Ерго, за всеки две функции f_i, f_j е в сила точно едно от следните:

$$\begin{aligned} f_i(n) < f_j(n) \wedge f_i(n) \leq f_j(n) \\ f_i(n) > f_j(n) \wedge f_i(n) \geq f_j(n) \\ f_i(n) \leq f_j(n) \wedge f_i(n) \asymp f_j(n) \wedge f_i(n) \geq f_j(n) \end{aligned}$$

Тъй като $f_i(n) < f_j(n)$ влече $f_i(n) \leq f_j(n)$ и $f_i(n) > f_j(n)$ влече $f_i(n) \geq f_j(n)$ и $f_i(n) \asymp f_j(n)$ влече $f_i(n) \leq f_j(n) \wedge f_i(n) \geq f_j(n)$, можем да опростим описанието на трите възможности до:

$$\begin{aligned} f_i(n) < f_j(n) \\ f_i(n) > f_j(n) \\ f_i(n) \asymp f_j(n) \end{aligned}$$

И така, да бъдат сравнени по асимптотично нарастване $f_i(n)$ и $f_j(n)$ означава да се определи дали $f_i(n) < f_j(n)$, или $f_i(n) > f_j(n)$, или $f_i(n) \asymp f_j(n)$. Заради транспонираната симетрия това са всъщност само две възможности: или едната функция расте асимптотично по-бързо от другата, или двете са асимптотично еквивалентни. Ограниченията, които наложихме, не позволяват друго.

Частен случай: няма асимптотично еквивалентни функции. Първо да разгледаме частния случай, в който нито две функции не са асимптотично еквивалентни. При това положение, да бъдат наредени функциите по асимптотично нарастване е същото като да бъде намерена уникалната пермутация $\pi: \{1, \dots, k\} \rightarrow \{1, \dots, k\}$, такава че:

$$f_{\pi(1)} < f_{\pi(2)} < \dots < f_{\pi(k-1)} < f_{\pi(k)} \quad (2.87)$$

Естествено, можем да ги подредим и в обратния ред:

$$f_{\pi(k)} > f_{\pi(k-1)} > \dots > f_{\pi(2)} > f_{\pi(1)} \quad (2.88)$$

като (2.87) и (2.88) са еквивалентни заради транспонираната симетрия, но предпочитаме (2.87), защото при сортирането по подразбиране сортираме в нарастващ, или поне ненамаляващ, ред (Задача 13).

Дотук стана ясно какво се иска в задачата: да се намери строгата линейна наредба π на дадените функции. Сега разсъждаваме как това да стане ефикасно. Ако бъдат сравнени по асимптотично нарастване всеки две $f_i(n)$ и $f_j(n)$, където $1 \leq i < j \leq n$, то π ще бъде намерена тривиално. Но тези двойки функции са $\binom{k}{2}$ на брой, така че дори за неголеми стойности на k става непрактично да се правят всички тези сравнения на ръка. Примерно, за $k = 20$ ненаредените двойки функции са $\frac{20 \cdot 19}{2} = 190$. Ключовото наблюдение е, че е достатъчно да бъдат сравнени всеки две функции, които са **непосредствени съседи** в (2.87). По този начин се правят само $k - 1$ сравнения, което е същността на Наблюдение 19. Забележете, че Наблюдение 19 е аналогично на Теорема 108 от Лекция 13.

Наблюдение 19

Да допуснем, че сме сравнили асимптотично $f_{\pi(1)}$ с $f_{\pi(2)}$, $f_{\pi(2)}$ с $f_{\pi(3)}$, и така нататък, $f_{\pi(k-1)}$ с $f_{\pi(k)}$, по този начин намирайки

$$f_{\pi(1)} < f_{\pi(2)} < \dots < f_{\pi(k-1)} < f_{\pi(k)}$$

За всеки две $f_i(n)$ и $f_j(n)$, които не са непосредствени съседи в (2.87), фактът, че $f_i(n) < f_j(n)$, следва от $f_{\pi(1)} < f_{\pi(2)} < \dots < f_{\pi(k-1)} < f_{\pi(k)}$ и транзитивността на релацията $<$. Поради това е излишно да се сравняват функции, които не са непосредствени съседи. Не е грешно, но е излишен труд.

Всяка двойка непосредствени съседи **трябва** да бъде сравнена, инак няма да знаем коя от тях расте асимптотично по-бързо. За непосредствените съседи транзитивността не помага. Тези сравнения **трябва** да се направят. Останалите сравнения може да не се правят.

В крайна сметка, решението се състои в извършване на точно $k - 1$ сравнения по асимптотично нарастване, а именно на двойките непосредствени съседи.

В светлината на Наблюдение 19, за да подредим по асимптотично нарастване $f_1(n), \dots, f_k(n)$, напълно достатъчно е да идентифицираме двойките непосредствени съседи, които са $k - 1$ на брой и да извършим експлицитно сравнение на всяка от тези двойки. Резултатите от тези сравнения влекат (2.87).

А как да идентифицираме двойките непосредствени съседи, без да сравняваме всяка двойка функции? Рецепта за общия случай няма, но може да ползваме добри практики и опита си в решаването на подобни задачи. По правило, в тези задачи $\{f_1(n), \dots, f_k(n)\}$ се разбива на малък брой подмножества $F_1, \dots, F_t$, такива че очевидно всяка функция от F_1 расте асимптотично по-бавно от всяка функция от F_2 , всяка функция от F_2 расте асимптотично по-бавно от всяка функция от F_3 , и така нататък. Ако намерим такива множества $F_1, \dots, F_t$, решаването на задачата се свежда до намирането на линейната поднаредба с $<$ на функциите във всяко от множествата. След това окончателното решение лесно се “сглобява” от тези вериги. Примерно, ако функциите за сравняване са тези:

$$\lg n, \quad \frac{\lg n}{\lg \lg n}, \quad 2n^2, \quad n^3 + \binom{n}{2}, \quad 2^n, \quad 3^{\frac{n}{\lg n}}, \quad n!$$

удачни множества са

$$F_1 = \left\{ \lg n, \frac{\lg n}{\lg \lg n} \right\}$$

$$F_2 = \left\{ 2n^2, n^3 + \binom{n}{2} \right\}$$

$$F_3 = \left\{ 2^n, 3^{\frac{n}{\lg n}}, n! \right\}$$

Линейните наредби $<$ в рамките на всяко от множествата са съответно

$$\frac{\lg n}{\lg \lg n} < \lg n$$

$$2n^2 < n^3 + \binom{n}{2}$$

$$3^{\frac{n}{\lg n}} < 2^n < n!$$

откъдето решението е

$$\frac{\lg n}{\lg \lg n} < \lg n < 2n^2 < n^3 + \binom{n}{2} < 3^{\frac{n}{\lg n}} < 2^n < n!$$

Разбира се, формалното решение на задачата не говори за F_1 , F_2 и F_3 – тези множества са за черновата! Те служат само за наше улеснение в намирането на двойките непосредствени съседи. Формалното решение казва само неща от рода на

Да разгледаме функциите $\lg n$ и $\frac{\lg n}{\lg \lg n}$. Ще покажем, че $\frac{\lg n}{\lg \lg n} < \lg n$.

.....някак доказваме, че $\frac{\lg n}{\lg \lg n} < \lg n$

Да разгледаме функциите $\lg n$ и $2n^2$. Ще покажем, че $\lg n < 2n^2$.

.....някак доказваме, че $\lg n < 2n^2$

Да разгледаме функциите $2n^2$ и $n^3 + \binom{n}{2}$. Ще покажем, че $2n^2 < n^3 + \binom{n}{2}$.

.....някак доказваме, че $2n^2 < n^3 + \binom{n}{2}$

и така нататък

Общият случай: може да има асимптотично еквивалентни функции. Сега да разгледаме общия случай, в който е възможно за $i \neq j$ да е изпълнено $f_i(n) \asymp f_j(n)$. Ако има асимптотично еквивалентни функции, решението не е едно единствено, ако под “решение” имаме предвид линейното разполагане на функциите отляво надясно. Примерно, ако трябва да се сравнят $\lg n$, $n^2 + n$, $\binom{n}{2}$ и 2^n , всяко от следните е решение:

$$\lg n < n^2 + n \asymp \binom{n}{2} < 2^n$$

$$\lg n < \binom{n}{2} \asymp n^2 + n < 2^n$$

Да разгледаме релацията $\leq$ върху функциите $f_1(n), \dots, f_k(n)$. Тя е пълна преднаредба (Определение 35), понеже в текущите допускания всеки две функции са асимптотично сравними. Тъй като $\asymp$ е релация на еквивалентност (Следствие 6), имаме право да разгледаме фактор-релацията $\leq / \asymp$. Знаем от Лема 13, че $\leq / \asymp$ е частична наредба. Следователно, в текущите допускания $\leq / \asymp$ е линейна наредба.

Класовете на еквивалентност на $\leq$ са максималните по включване подмножества на $\{f_1(n), \dots, f_k(n)\}$, всяко от което се състои от асимптотично еквивалентни функции[†]. Нека класовете на еквивалентност на $\leq$ са m на брой. Да ги наречем $F_1, F_2, \dots, F_m$. Да кажем, че $|F_1| = k_1, |F_2| = k_2, \dots, |F_m| = k_m$. Очевидно $k_1 + k_2 + \dots + k_m = k$. БОО, нека

- всяка функция от F_1 расте асимптотично по-бавно от всяка функция от F_2 ,
- всяка функция от F_2 расте асимптотично по-бавно от всяка функция от F_3 ,
- и така нататък,
- всяка функция от F_{m-1} расте асимптотично по-бавно от всяка функция от F_m .

С известна злоупотреба с нотацията “ $<$ ” можем да напишем това така

$$F_1 < F_2 < F_3 < \dots < F_{m-1} < F_m$$

Решението на задачата да се подредят по асимптотично нарастване $f_1(n), \dots, f_k(n)$ се състои от следните две дейности.

1. Да се намерят класовете на еквивалентност. За $1 \leq i \leq m$, да се намери F_i означава да се извършат такива асимптотични сравнения на двойки функции от F_i , от които сравнения да следва, че $\forall g, h \in F_i : g \asymp h$.

Ако се направят всички възможни $\binom{k_i}{2}$ сравнения и резултатът от всяко сравнение е, че функциите са Θ една от друга, очевидно ще следва, че всички тези функции са в един и същи клас на еквивалентност. Но не е необходимо да се правят всички тези сравнения! Ако преименуваме елементите на F_i като $g_1(n), g_2(n), \dots, g_{k_i}(n)$, достатъчно е да покажем, че

$$\begin{aligned} g_1(n) &\asymp g_2(n) \\ g_2(n) &\asymp g_3(n) \\ &\dots \\ g_{k_i-1}(n) &\asymp g_{k_i}(n) \end{aligned}$$

Това са $k_i - 1$ сравнения. Разбира се, сравненията може да са други. Примерно тези

$$\begin{aligned} g_1(n) &\asymp g_2(n) \\ g_1(n) &\asymp g_3(n) \\ &\dots \\ g_1(n) &\asymp g_{k_i-1}(n) \\ g_1(n) &\asymp g_{k_i}(n) \end{aligned}$$

[†]Примерно, ако функциите са $\lg n, n^2 + n, \binom{n}{2}, 2^n, 3 \lg n + \lg \lg n$, класовете на еквивалентност са $\{\lg n, 3 \lg n + \lg \lg n\}, \{n^2 + n, \binom{n}{2}\}$ и $\{2^n\}$.

Виждаме, че пак са $k_i - 1$ на брой, но имат друга структура.

Минималното множество от сравнения, от които следва, че всички тези k_i на брой функции са Θ една от друга, представляват покриващо дърво на пълния граф с върхове $g_1(n), g_2(n), \dots, g_{k_i}(n)^\dagger$. От това следва, че броят им е $k_i - 1$: колкото са ребрата на дърво с k_i върха[‡].

Всичко това има значение за икономичността на решението, а не за коректността му! Искане в максимална степен да се възползваме от транзитивността на $\asymp$. Да повторим: не е грешка да се направят всички $\binom{k_i}{2}$ сравнения на функции от F_i , но това включва излишен труд, ако $k_i > 2$.

Ето пример. Нека

$$F_1 = \left\{ n^2, 100n^2, \binom{n}{2}, 2n^2 + n + 4, \frac{1}{3}n^2 + n\sqrt{n}, \sum_{i=1}^n i \right\}$$

Имаме $|F_1| = 6$. Може да извършим следните пет сравнения

- (а) сравняваме n^2 със $100n^2$,
- (б) сравняваме n^2 с $\binom{n}{2}$,
- (в) сравняваме n^2 с $2n^2 + n + 4$,
- (г) сравняваме n^2 с $\frac{1}{3}n^2 + n\sqrt{n}$,
- (д) сравняваме n^2 със $\sum_{i=1}^n i$

и да заключим, че всички тези функции са Θ една от друга. Не е задължително да са точно **тези** сравнения, но трябва да са пет на брой и да представляват покриващо дърво на пълния граф с върхове шестте функции.

2. Да се подредят класовете на еквивалентност чрез $<$. Това е лесно. За всеки два класа на еквивалентност F_i и F_j , които са непосредствени съседи в окончателната наредба, сравняваме коя да е функция $f'(n) \in F_i$ с коя да е функция $f''(n) \in F_j$. Това е напълно достатъчно. Няма смисъл да сравняваме други функции от F_i с други функции от F_j ; ако $f'(n) < f''(n)$, очевидно е вярно, че $h'(n) < h''(n)$ за всяка $h'(n) \in F_i$ и всяка $h''(n) \in F_j$.

Също така е безсмислено да се сравняват функции от класове на еквивалентност, които не са непосредствени съседи. Съображенията са същите като съображенията, поради които няма смисъл да се сравняват функции, които не са непосредствени съседи в окончателната наредба, при допускането, че няма асимптотично еквивалентни функции.

Общият брой на сравненията е

$$\begin{aligned} (k_1 - 1) + & \quad // \text{ за установяване на взаимната асимпт. еквивалентност на } \phi\text{-ите от } F_1 \\ (k_2 - 1) + & \quad // \text{ за установяване на взаимната асимпт. еквивалентност на } \phi\text{-ите от } F_2 \\ \dots & \\ (k_m - 1) + & \quad // \text{ за установяване на взаимната асимпт. еквивалентност на } \phi\text{-ите от } F_m \\ m - 1 & \quad // \text{ за подреждане на класовете в линейна наредба} \end{aligned}$$

[†]Оттук и броя на различните начини да се извършат тези сравнения е $k_i^{k_i-2}$.

[‡]Забележете, че това е в сила дори когато $k_i = 1$. Тогава единствената функция във F_i не е асимптотично еквивалентна на никоя друга функция измежду $f_1(n), \dots, f_k(n)$, така че извършваме $1 - 1 = 0$ сравнения за “намирането”—което е празното действие, понеже не правим нищо—на F_i .

Но тази сума е

$$(k_1 + k_2 + \dots + k_m - m) + (m - 1) = k_1 + k_2 + \dots + k_m - 1 = k - 1$$

И така, необходимият и достатъчен брой сравнения и в общия случай е: броят на всички функции, минус единица.

Задача 18

Нека $a > 1$ е реална константа. Подредете по асимптотично нарастване функциите a^n , $n!$, n^n и a^{a^n} . Интерпретирайте $n!$ чрез апроксимацията на Stirling (2.65), понеже n може да не е цяло число.

Решение: Да сравним a^n с $n!$. Да вземем логаритъм от двете функции. Очевидно $\lg a^n \asymp n$, докато $\lg n! \asymp n \lg n$ съгласно (2.75). Тогава $\lg a^n < \lg n!$. Тогава $a^n < n!$ съгласно Следствие 12.

Да сравним $n!$ с n^n . Тук трикът с логаритмуването не работи, понеже се получават функции, които са Тета една от друга, което не ни казва в какво отношение са първообразите. Да разгледаме границата $\lim_{n \rightarrow \infty} \frac{n!}{n^n}$. Ще покажем, че границата съществува и е 0. Наистина, след като заместим $n!$ с $\sqrt{n} \cdot \frac{n^n}{e^n}$, получаваме

$$\lim_{n \rightarrow \infty} \frac{\sqrt{n} \cdot \frac{n^n}{e^n}}{n^n} = \lim_{n \rightarrow \infty} \sqrt{n} \cdot \frac{1}{e^n} = 0$$

Тогава $n! < n^n$ съгласно Наблюдение 17.

И накрая, да сравним n^n с a^{a^n} . Да вземем логаритъм от двете функции. Очевидно $\lg n^n \asymp n \lg n$, докато $\lg a^{a^n} \asymp a^n$. Тъй като $n \lg n < n^2$, а $n^2 < a^n$ съгласно Следствие 14, в сила е $n \lg n < a^n$. Съгласно Следствие 12, в сила е $n^n < a^{a^n}$.

Окончателната наредба е

$$a^n < n! < n^n < a^{a^n}$$

□

Задача 19

Подредете по асимптотично нарастване n^n , n^{n+1} , $(n+1)^n$ и $(n+1)^{n+1}$.

Решение: Да сравним n^n с $(n+1)^n$. Да разгледаме границата

$$\lim_{n \rightarrow \infty} \frac{(n+1)^n}{n^n} = \lim_{n \rightarrow \infty} \left(\frac{n+1}{n} \right)^n = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^n = e$$

Заклучаваме, че $n^n \asymp (n+1)^n$.

Да сравним n^n с $n^{(n+1)}$. Но $n^{(n+1)} = n \cdot n^n$, така че явно $n^n < n^{(n+1)}$.

Да сравним $n^{(n+1)}$ с $(n+1)^{(n+1)}$. Да разгледаме границата

$$\lim_{n \rightarrow \infty} \frac{(n+1)^{n+1}}{n^{n+1}} = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^{n+1} = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right)^n \cdot \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n} \right) = e \cdot 1 = e$$

Тогава $n^{(n+1)} \asymp (n+1)^{(n+1)}$.

Окончателната наредба е

$$n^n \asymp (n+1)^n < n^{(n+1)} \asymp (n+1)^{(n+1)}$$

□

Задача 20

Наредете по асимптотично нарастване следните функции. Обосновете отговорите си. Напишете в явен вид наредбата. *Забележка: всички логаритми са с основа 2.*

$$\begin{aligned} f_1(n) &= 2n, & f_2(n) &= n^{n!}, & f_3(n) &= 2^{(\lg n)^2}, & f_4(n) &= 2^{\lg(n^2)}, & f_5(n) &= \frac{\sqrt{n}}{\lg \lg n}, \\ f_6(n) &= \binom{2n}{2}^2, & f_7(n) &= \lg \binom{2n}{n}, & f_8(n) &= \sqrt[3]{n} \lg \lg n, & f_9(n) &= n^{\lg \lg n}, & f_{10}(n) &= n!^n \end{aligned}$$

Решение: Ще извършим девет сравнения на функции.

(i) Ще покажем, че $f_8 < f_5$. Наистина,

$$f_8(n) = \sqrt[3]{n} \lg \lg n = n^{\frac{1}{3}} \lg \lg n < \quad // \text{ Теорема 31}$$

$$n^{\frac{1}{3}} \lg n < \quad // \text{ Следствие 14}$$

$$n^{\frac{1}{3}} n^{\frac{1}{12}} = n^{\frac{5}{12}} < \quad // \text{ Следствие 14}$$

$$n^{\frac{5}{12}} \cdot \frac{n^{\frac{1}{12}}}{\lg \lg n} = \frac{n^{\frac{6}{12}}}{\lg \lg n} = \frac{\sqrt{n}}{\lg \lg n} = f_5(n)$$

(ii) Ще покажем, че $f_5 < f_1$. Но това е напълно очевидно, понеже $\sqrt{n} < n$, така че $\frac{\sqrt{n}}{\lg \lg n} < 2n$.

(iii) Ще покажем, че $f_1 \asymp f_7$. Използваме Теорема 35, която казва, че $\binom{n}{\frac{n}{2}} \asymp \frac{2^n}{\sqrt{n}}$. Тогава $\binom{2n}{n} \asymp \frac{4^n}{\sqrt{n}}$. Тогава $\lg \binom{2n}{n} \asymp \lg \left(\frac{4^n}{\sqrt{n}} \right) = n \lg 4 - \frac{1}{2} \lg n \asymp n$, така че $2n \asymp \lg \binom{2n}{n}$.

(iv) Ще покажем, че $f_1 < f_4$. Веднага се вижда, че

$$f_4(n) = 2^{\lg(n^2)} = n^2$$

Това, че $n < n^2$, е очевидно.

(v) Ще покажем, че $f_4 < f_6$. Помним, че $f_4(n) = n^2$. От друга страна,

$$f_6(n) = \binom{2n}{2}^2 = \left(\frac{2n(2n-1)}{2} \right)^2 \asymp n^4$$

Твърдението става очевидно.

(vi) Ще покажем, че $f_6 < f_9$. Помним, че $f_6(n) \asymp n^4$. От друга страна, двойният логаритъм е растяща функция на n , така че $n^4 < n^{\lg \lg n}$.

(vii) Ще покажем, че $f_9 < f_3$. Първо преобразуваме f_3 така:

$$f_3(n) = 2^{(\lg n)^2} = 2^{(\lg n) \cdot (\lg n)} = 2^{\lg(n^{\lg n})} = n^{\lg n}$$

Това, че $n^{\lg \lg n} < n^{\lg n}$, се вижда веднага, понеже логаритъмът е растяща функция.

(viii) Ще покажем, че $f_3 < f_{10}$. Логаритмуваме двете функции:

$$\begin{aligned} \lg(n^{\lg n}) &= (\lg n)(\lg n) = (\lg n)^2 \\ \lg(n!^n) &= n \lg n! \asymp n^2 \lg n \quad // \text{ съгласно Теорема 33} \end{aligned}$$

Но $(\lg n)^2 < n^2 \lg n$ съгласно Следствие 14. Заклучаваме, че $n^{\lg n} < n!^n$.

(ix) Ще покажем, че $f_{10} < f_2$. Вече видяхме, че $\lg f_{10}(n) \asymp n^2 \lg n$. От друга страна, $\lg(n^{n!}) \asymp n! \lg n$. Тъй като факториелът очевидно расте асимптотично по-бързо от всяка полиномиална функция, заключаваме, че $\lg f_{10}(n) < \lg f_2(n)$. Оттук веднага следва $f_{10} < f_2$.

Окончателната наредба е:

$$f_8 < f_5 < f_1 \asymp f_7 < f_4 < f_6 < f_9 < f_3 < f_{10} < f_2$$

□

Задача 21

Наредете по асимптотично нарастване следните функции. Обосновете отговорите си. Напишете в явен вид наредбата. *Забележка: всички логаритми са с основа 2.*

$$\begin{aligned} f_1 &= (\lg n)^{\lg n}, & f_2 &= (\lg(\sqrt{n}))^{\lg(\sqrt{n})}, & f_3 &= (\lg n)^{(\lg n)^{\lg n}} \\ f_4 &= 2^{2^{n-1}}, & f_5 &= \sum_{k=0}^n \binom{n}{k}, & f_6 &= \binom{2n}{n} \end{aligned}$$

Решение: Ще извършим пет сравнения на функции.

(i) Ще докажем, че $f_2 < f_1$. Не е добра идея да се логаритмуват двете функции, защото образите след логаритмичната трансформация са асимптотично еквивалентни. Действаме така:

$$f_2 = (\lg(\sqrt{n}))^{\lg(\sqrt{n})} = \left(\frac{1}{2} \lg n\right)^{\frac{1}{2} \lg n} \leq (\lg n)^{\frac{1}{2} \lg n} = \left((\lg n)^{\frac{1}{2}}\right)^{\lg n} = \left(\sqrt{\lg n}\right)^{\lg n}$$

Но $(\sqrt{\lg n})^{\lg n} < (\lg n)^{\lg n}$, понеже

$$\lim_{n \rightarrow \infty} \frac{(\sqrt{\lg n})^{\lg n}}{(\lg n)^{\lg n}} = \lim_{n \rightarrow \infty} \left(\frac{\sqrt{\lg n}}{\lg n}\right)^{\lg n} = \lim_{n \rightarrow \infty} \left(\frac{1}{\sqrt{\lg n}}\right)^{\lg n} = \lim_{n \rightarrow \infty} \frac{1}{(\sqrt{\lg n})^{\lg n}} = 0$$

Тогава наистина $f_2 < f_1$.

(ii) Ще докажем, че $f_1 < f_5$. От Теоремата на Newton следва, че $f_5 = 2^n$. Сравняваме $(\lg n)^{\lg n}$ с 2^n . Логаритмуваме двете функции. Сега сравняваме $\lg f_1(n) = (\lg n)(\lg \lg n)$ с $\lg f_5(n) = n$.

Но всяка полилогаритмично нарастваща функция расте асимптотично по-бавно от всяка полиномиално растяща функция (Следствие 14), така че $(\lg n)^2 < n$. Имайки предвид, че $(\lg n)(\lg \lg n) \leq (\lg n)^2$ (Теорема 31), заключаваме, че $(\lg n)(\lg \lg n) < n$. Съгласно Следствие 12, в сила е $f_1 < f_5$.

(iii) Ще докажем, че $f_5 < f_6$. От Теоремата на Newton следва, че $f_5 = 2^n$. Теорема 35 казва, че $f_6 \asymp \frac{4^n}{\sqrt{n}}$. Тривиално се доказва, че $\lim_{n \rightarrow \infty} \frac{4^n}{2^n \sqrt{n}} = \infty$. Докажем, че $f_5 < f_6$.

(iv) Ще докажем, че $f_6 < f_3$. Както вече видяхме, $f_6 \asymp \frac{4^n}{\sqrt{n}}$. Сравняваме $\frac{4^n}{\sqrt{n}}$ и f_3 , логаритмувайки и двете. Очевидно

$$\lg f_6 = \lg \frac{4^n}{\sqrt{n}} \asymp n \quad (2.89)$$

$$\lg f_3 = \left((\lg n)^{\lg n} \right) (\lg n) \quad (2.90)$$

Но $n < (\lg n)^{\lg n}$. За да се убедим в това, достатъчно е да логаритмуваме двете функции. Образът на функцията вляво е $\lg n$, а на функцията вдясно е $(\lg n)(\lg \lg n)$. Очевидно е, че $\lg n < (\lg n)(\lg \lg n)$, понеже $\lg \lg n$ е неограничено растяща. Заключаваме, че наистина $n < (\lg n)^{\lg n}$.

От това и (2.89) следва, че $\lg f_6 < (\lg n)^{\lg n}$. Но $(\lg n)^{\lg n} < \left((\lg n)^{\lg n} \right) (\lg n)$, понеже $\lg n$ е неограничено растяща. Тогава $\lg f_6 < \left((\lg n)^{\lg n} \right) (\lg n)$. От това и (2.90) следва, че $\lg f_6 < \lg f_3$. Съгласно Следствие 12, $f_6 < f_3$.

(v) Ще докажем, че $f_3 < f_4$. Логаритмуваме двете функции и вече сравняваме $\left((\lg n)^{\lg n} \right) (\lg n)$ с 2^{n-1} . Дясната от тези расте по-бързо в асимптотичния смисъл – факт, който може да установим с още едно логаритмуване. Желаният резултат следва веднага.

От (i)–(v) следва наредбата:

$$f_2 < f_1 < f_5 < f_6 < f_3 < f_4$$

□

Задача 22

Наредете по асимптотично нарастване следните дванадесет функции. Обосновете отговорите си. Напишете в явен вид наредбата.

$$\begin{array}{llll} f_1(n) = n^2 & f_2(n) = n \binom{n}{2} & f_3(n) = \sum_{i=1}^n \binom{n}{i}^2 & f_4(n) = \sum_{i=1}^{\infty} \frac{n^2}{2^i} \\ f_5(n) = \sqrt[n]{n!} & f_6(n) = \lg \lg \lg n & f_7(n) = 2^{2^n} & f_8(n) = 2^{2^{n+2}} \\ f_9(n) = 2^{n^2} & f_{10}(n) = \lg \lg \lg \binom{n}{2} & f_{11}(n) = 2^{n!} & f_{12}(n) = \sum_{k=1}^{n^2} \frac{n^2}{k} \end{array}$$

Решение: Ще извършим единадесет сравнения на функции.

(i) Да сравним $f_6(n)$ с $f_{10}(n)$. Тъй като $\binom{n}{2} \asymp n^2$, в сила е

$$f_{10}(n) \asymp \lg \lg \lg n^2 = \lg \lg (2 \lg n) \asymp \lg \lg \lg n$$

Тогава

$$f_{10}(n) \asymp f_6(n)$$

(ii) Да сравним $f_6(n)$ с $f_5(n)$. Ще опростим $f_5(n)$, използвайки апроксимацията на Stirling:

$$\sqrt[n]{n!} \asymp \sqrt[n]{\sqrt{n} \frac{n^n}{e^n}} = \sqrt[n]{n} \frac{n}{e} \asymp n^{\frac{1}{2n}} \sqrt[n]{n}$$

Очевидно $\lg \lg \lg n < n^{\frac{1}{2n}} \sqrt[n]{n}$, така че $f_6(n) < f_5(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n)$$

(iii) Да сравним $f_5(n)$ с $f_1(n)$. Очевидно е, че $n^{\frac{1}{2n}} \sqrt[n]{n} < n^2$, така че $f_5(n) < f_1(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n)$$

(iv) Да сравним $f_1(n)$ с $f_4(n)$. Очевидно $f_4(n) = n^2 \sum_{i=1}^{\infty} \frac{1}{2^i}$, понеже n не зависи от i . Известно е, че геометричният ред $\sum_{i=1}^{\infty} \frac{1}{2^i}$ е сходящ, поради което $f_4(n) \asymp n^2$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n)$$

(v) Да сравним $f_1(n)$ с $f_{12}(n)$. Очевидно $f_{12}(n) = n^2 \sum_{k=1}^{n^2} \frac{1}{k}$. От лекции знаем, че $\sum_{k=1}^m \frac{1}{k} \asymp \lg m$, откъдето $f_{12}(n) \asymp n^2 \lg n^2 \asymp n^2 \lg n$. Тогава $f_1(n) < f_{12}(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n)$$

(vi) Да сравним $f_{12}(n)$ с $f_2(n)$. Тъй като $\binom{n}{2} \asymp n^2$, в сила е $f_2(n) \asymp n^3$. Тогава $f_{12}(n) < f_2(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n)$$

(vii) Да сравним $f_2(n)$ с $f_3(n)$. От курса Дискретни Структури знаем, че $\sum_{i=0}^n \binom{n}{i}^2 = \binom{2n}{n}$. А на лекция по ДАА сме показали, че $\binom{2n}{n} \asymp \frac{4^n}{\sqrt{n}}$. Тогава $f_3(n) \asymp \frac{4^n}{\sqrt{n}}$, така че $f_2(n) < f_3(n)$.

Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n) < f_3(n)$$

(viii) Да сравним $f_3(n)$ с $f_9(n)$. Както видяхме, $f_3(n) \asymp \frac{4^n}{\sqrt{n}}$, а $f_9(n) = 2^{n^2}$ по условие. Ако логаритмуваме двете страни на двете асимптотични еквивалентности, получаваме

$$\begin{aligned}\lg f_3(n) &\asymp n \\ \lg f_9(n) &\asymp n^2\end{aligned}$$

Тъй като $n < n^2$, за първообразите директно имаме $f_3(n) < f_9(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n) < f_3(n) < f_9(n)$$

(ix) Да сравним $f_9(n)$ с $f_7(n)$. Логаритмувайки, получаваме

$$\begin{aligned}\lg f_9(n) &\asymp n^2 \\ \lg f_7(n) &\asymp 2^n\end{aligned}$$

Тъй като експоненциалната 2^n расте по-бързо, в асимптотичния смисъл, от полиномиалната n^2 , за първообразите е в сила $f_9(n) < f_7(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n) < f_3(n) < f_9(n) < f_7(n)$$

(x) Да сравним $f_7(n)$ с $f_8(n)$. Забелязваме, че

$$(2^{2^n})^4 = 2^{2^2 \cdot 2^n} = 2^{2^{n+2}}$$

Тогава $f_8(n) = (f_7(n))^4$. Тъй като това са неограничено растящи функции, вярно е, че $f_7(n) < f_8(n)$. Дотук имаме

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n) < f_3(n) < f_9(n) < f_7(n) < f_8(n)$$

(xi) Да сравним $f_8(n)$ с $f_{11}(n)$. Логаритмувайки, получаваме

$$\begin{aligned}\lg f_8(n) &\asymp 2^{n+2} = 4 \cdot 2^n \asymp 2^n \\ \lg f_{11}(n) &\asymp n! \asymp \sqrt{n} \frac{n^n}{e^n}\end{aligned}$$

Ако логаритмуваме още веднъж, получаваме

$$\begin{aligned}\lg \lg f_8(n) &\asymp n \\ \lg \lg f_{11}(n) &\asymp n \lg n\end{aligned}$$

Тъй като $n \lg n$ расте по-бързо, в асимптотичния смисъл, от n , в сила е $f_8(n) < f_{11}(n)$. Окончателната наредба е

$$f_{10}(n) \asymp f_6(n) < f_5(n) < f_1(n) \asymp f_4(n) < f_{12}(n) < f_2(n) < f_3(n) < f_9(n) < f_7(n) < f_8(n) < f_{11}(n)$$

Сега ще подредим по асимптотично нарастване прочутите тридесет функции от учебника

CLR [34, зад. 3-3, стр. 61]. Първо една дефиниция, която ползва итерирания логаритъм от Определение 32.

Определение 34: Функция $\lg^* n$ [34, стр. 58]

Тогава

$$\lg^* n = \min \{i \geq 0 \mid \lg^{(i)} n \leq 1\}$$

“ $\lg^* n$ ” се чете *log-star* на английски. На български ще казваме “лог-звезда”.

Да видим стойностите на функцията лог-звезда за някои стойности на аргумента.

$$\lg^* 2 = 1, \text{ понеже } \lg^{(0)} 2 = 2 \text{ и } \lg^{(1)} 2 = \lg(\lg^{(0)} 2) = \lg(2) = 1$$

$$\lg^* 3 = 2, \text{ понеже } \lg^{(0)} 3 = 3 \text{ и } \lg^{(1)} 3 = \lg(\lg^{(0)} 3) = \lg(\lg 3) = 0.6644 \dots$$

$$\lg^* 4 = 2$$

$$\lg^* 5 = 3$$

...

$$\lg^* 16 = 3$$

$$\lg^* 17 = 4$$

...

$$\lg^* 65536 = 4$$

$$\lg^* 65537 = 5$$

...

$$\lg^* 2^{65536} = 5$$

$$\lg^* (2^{65536} + 1) = 6$$

$$\lg^* 2^{2^{65536}} = 6$$

...

$$\lg^* (2^{2^{65536}} + 1) = 7$$

Очевидно, всяко реално число $x \geq 2$ може да се представи като кула от двойки така:

$$x = 2^{2^{\dots^{2^\alpha}}}$$

където α е реално число, такова че $1 < \alpha \leq 2$. Височината на кулата е броят на елементите на тази редица. Ето таблица с кулите на някои цели положителни числа.

число n	кулата от двойки на n	височината на кулата	$\lg^* n$
2	2	1	1
3	$2^{1.5849625007\dots}$	2	2
4	2^2	2	2
5	$2^{2^{1.2153232957\dots}}$	3	3
16	2^{2^2}	3	3
17	$2^{2^{2^{1.0223362884\dots}}}$	4	4
65536	$2^{2^{2^2}}$	4	4
65537	$2^{2^{2^{2^{1.00000051642167\dots}}}}}$	5	5

Лесно се вижда, че $\lg^* n$ е височината на кулата на n .

Задача 23: [34, зад. 3-3, стр. 61]

Подредете по асимптотично нарастване следните тридесет функции.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2^{\lg n}}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Решение: Ще извършим двадесет и девет сравнения на функции. Както обикновено, всички логаритми са с основа 2.

(i) Сравняваме 1 и $n^{\frac{1}{\lg n}}$. Забелязваме, че $n^{\frac{1}{\lg n}} = 2$; ако вземем $\lg$ на двете страни, получаваме $\lg\left(n^{\frac{1}{\lg n}}\right) = \frac{1}{\lg n} \cdot \lg n = 1$ и $\lg 2 = 1$. Щом $n^{\frac{1}{\lg n}} = 2$, в сила е $1 \asymp n^{\frac{1}{\lg n}}$.

(ii) Сравняваме 1 и $\lg(\lg^* n)$. Веднага виждаме, че $1 < \lg(\lg^* n)$, понеже $\lg(\lg^* n)$ е неограничено растяща функция, бивайки композиция на неограничено растящи функции.

(iii) Сравняваме $\lg(\lg^* n)$ и $\lg^* n$. Но за всяка неограничено растяща функция $f(n)$ е вярно, че $\lg f(n) < f(n)$ (Теорема 31). В частност, $\lg(\lg^* n) < \lg^* n$.

(iv) Сравняваме $\lg^* n$ и $\lg^*(\lg n)$. Нека мислим за n като за кула от двойки. Забелязваме, че разликата между височините на n и $\lg n$ е само едно. Тоест, $\lg^*(\lg n) = (\lg^* n) - 1$. Но тогава $\lg^* n \asymp \lg^*(\lg n)$.

(v) Сравняваме $\lg^* n$ и $2^{\lg^* n}$. Но за всяка неограничено растяща функция $f(n)$ е вярно, че $f(n) < 2^{f(n)}$ (леко следствие от Теорема 31). В частност, $\lg^* n < 2^{\lg^* n}$.

(vi) Сравняваме $2^{\lg^* n}$ и $\ln \ln n$. Първо забелязваме, че $\ln \ln n \asymp \lg \lg n$, поради което сравняваме $2^{\lg^* n}$ и $\lg \lg n$. Вземаме логаритмите на двете страни и получаваме съответно $\lg(2^{\lg^* n}) = (\lg^* n) \cdot (\lg 2) = \lg^* n$ и $\lg \lg \lg n = \lg^{(3)} n$. Очевидно $\lg^* n < \lg^{(k)} n$ за всяка константа $k \geq 0$, така че $\lg^* n < \lg^{(3)} n$. Тогава $2^{\lg^* n} < \lg \lg n$. Тогава $2^{\lg^* n} < \ln \ln n$.

(vii) Сравняваме $\ln \ln n$ и $\sqrt{\lg n}$. Първо забелязваме, че $\ln \ln n \asymp \lg \lg n$, поради което сравняваме $\lg \lg n$ и $(\lg n)^{\frac{1}{2}}$. Да вземем логаритмите на двете функции. Имаме съответно $\lg \lg \lg n$ и $\frac{1}{2} \lg \lg n$. Съгласно Теорема 31, $\lg \lg \lg n < \frac{1}{2} \lg \lg n$. Тогава $\lg \lg n < \sqrt{\lg n}$. Тогава $\ln \ln n < \sqrt{\lg n}$.

(viii) Сравняваме $\sqrt{\lg n}$ и $\ln n$. Тъй като $\ln n \asymp \lg n$, ще сравним $\sqrt{\lg n}$ и $\lg n$. Очевидно $\lim_{n \rightarrow \infty} \frac{\sqrt{\lg n}}{\lg n} = 0$. Тогава $\sqrt{\lg n} < \lg n$. Тогава $\sqrt{\lg n} < \ln n$.

(ix) Сравняваме $\ln n$ и $\lg^2 n$. Тъй като $\ln n \asymp \lg n$, ще сравним $\lg n$ и $\lg^2 n$. Очевидно $\lim_{n \rightarrow \infty} \frac{\lg n}{\lg^2 n} = 0$. Тогава $\lg n < \lg^2 n$. Тогава $\ln n < \lg^2 n$.

(x) Сравняваме $\lg^2 n$ и $2^{\sqrt{2} \lg n}$. Да разгледаме логаритмите на двете функции. Те са съответно $2 \lg \lg n$ и $(\sqrt{2} \lg n) \cdot \lg 2 = \sqrt{2} \sqrt{\lg n}$. Но $\lg \lg n < \sqrt{\lg n}$ (Следствие 14). Тогава $2 \lg \lg n < (\sqrt{2} \lg n) \cdot \lg 2$. Тогава $\lg^2 n < 2^{\sqrt{2} \lg n}$.

(xi) Сравняваме $2^{\sqrt{2} \lg n}$ и $(\sqrt{2})^{\lg n}$. Да разгледаме логаритмите на двете функции. Те са съответно $\lg(2^{\sqrt{2} \lg n}) = (\sqrt{2} \lg n) \cdot \lg 2 = \sqrt{2} \sqrt{\lg n}$ и $\lg((\sqrt{2})^{\lg n}) = (\lg n) \cdot (\lg \sqrt{2})$. Но $\sqrt{\lg n} < \lg n$ (Следствие 14). Тогава $\lg(2^{\sqrt{2} \lg n}) < \lg((\sqrt{2})^{\lg n})$. Тогава $2^{\sqrt{2} \lg n} < (\sqrt{2})^{\lg n}$.

(xii) Да сравним $(\sqrt{2})^{\lg n}$ и n . Забелязваме, че $(\sqrt{2})^{\lg n} = 2^{\frac{1}{2} \lg n} = 2^{\lg \sqrt{n}} = \sqrt{n}$. Очевидно е, че $\sqrt{n} < n$. Тогава $(\sqrt{2})^{\lg n} < n$.

(xiii) Да сравним n и $2^{\lg n}$. Веднага виждаме, че $n = 2^{\lg n}$ заради свойствата на логаритъма. Тогава $n \asymp 2^{\lg n}$.

(xiv) Да сравним n и $n \lg n$. Тъй като $\lg n$ е неограничено растяща, в сила е $n < n \lg n$.

(xv) Да сравним $n \lg n$ и $\lg n!$. Теорема 33 казва, че $n \lg n \asymp \lg n!$.

(xvi) Да сравним $n \lg n$ и n^2 . В сила е $\lim_{n \rightarrow \infty} \frac{n \lg n}{n^2} = \lim_{n \rightarrow \infty} \frac{\lg n}{n} = 0$, понеже $\lg n < n$ съгласно Следствие 14. Тогава $n \lg n < n^2$.

(xvii) Да сравним n^2 и $4^{\lg n}$. Но $4^{\lg n} = 2^{2 \lg n} = 2^{\lg n^2} = n^2$ заради свойствата на логаритъма. Тогава $n^2 \asymp 4^{\lg n}$.

(xviii) Да сравним n^2 с n^3 . В сила е $\lim_{n \rightarrow \infty} \frac{n^2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0$. Тогава $n^2 < n^3$.

(xix) Да сравним n^3 с $(\lg n)!$. Вече видяхме (Задача 16, (2.86)), че

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}}$$

ако ползваме апроксимацията на Stirling (Теорема 32). Но $(\lg n)^{\lg n} = n^{\lg \lg n}$, което е елементарно следствие от свойствата на логаритмите. Тогава

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{n^{\lg \lg n}}{n^{\lg e}}$$

Забелязваме, че

$$\lim_{n \rightarrow \infty} \frac{\sqrt{\lg n} \cdot \frac{n^{\lg \lg n}}{n^{\lg e}}}{n^3} = \lim_{n \rightarrow \infty} \sqrt{\lg n} \cdot \frac{n^{\lg \lg n}}{n^{3+\lg e}} = \infty$$

понеже $\lg \lg n$ е неограничено растяща функция, а $3 + \lg e$ е константа. Заклучаваме, че $n^3 < (\lg n)!$.

(xx) Да сравним $(\lg n)!$ и $(\lg n)^{\lg n}$. От (Задача 16, (2.86)) знаем, че

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}}$$

Но

$$\sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}} = \frac{\sqrt{\lg n}}{n^{\lg e}} \cdot (\lg n)^{\lg n}$$

Тогава

$$\lim_{n \rightarrow \infty} \frac{\frac{\sqrt{\lg n}}{n^{\lg e}} \cdot (\lg n)^{\lg n}}{(\lg n)^{\lg n}} = \lim_{n \rightarrow \infty} \frac{\sqrt{\lg n}}{n^{\lg e}} = 0$$

тъй като $\sqrt{\lg n} < n^{\lg e}$ (Следствие 14). Заклучаваме, че $(\lg n)! < (\lg n)^{\lg n}$.

(xxi) Да сравним $(\lg n)^{\lg n}$ и $n^{\lg \lg n}$. Но $(\lg n)^{\lg n} = n^{\lg \lg n}$, което се вижда веднага, ако вземем логаритъм на изразите от двете страни на равенството. Тогава $(\lg n)^{\lg n} \asymp n^{\lg \lg n}$.

(xxii) Да сравним $n^{\lg \lg n}$ и $\left(\frac{3}{2}\right)^n$. Да логаритмуваме двете функции. Получаваме съответно $(\lg n)(\lg \lg n)$ и $n \lg \left(\frac{3}{2}\right)$. Но очевидно $(\lg n)(\lg \lg n) < (\lg n)^2$, а $(\lg n)^2 < n \lg \left(\frac{3}{2}\right)$ съгласно Следствие 14. Тогава $(\lg n)(\lg \lg n) < n \lg \left(\frac{3}{2}\right)$. Тогава $n^{\lg \lg n} < \left(\frac{3}{2}\right)^n$ (Следствие 12).

(xxiii) Да сравним $\left(\frac{3}{2}\right)^n$ и 2^n . Забелязваме, че

$$\lim_{n \rightarrow \infty} \frac{\left(\frac{3}{2}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \left(\frac{3}{4}\right)^n = 0$$

Тогава $\left(\frac{3}{2}\right)^n < 2^n$ (Наблюдение 17).

(xxiv) Да сравним 2^n и $n \cdot 2^n$. Очевидно $2^n < n \cdot 2^n$, понеже n е неограничено растяща функция.

(xxv) Да сравним $n \cdot 2^n$ и e^n .

$$\lim_{n \rightarrow \infty} \frac{n \cdot 2^n}{e^n} = \lim_{n \rightarrow \infty} \frac{n}{\frac{e^n}{2^n}} = \lim_{n \rightarrow \infty} \frac{n}{\left(\frac{e}{2}\right)^n} = 0$$

понеже $n < \left(\frac{e}{2}\right)^n$ (Следствие 14). Тогава $n \cdot 2^n < e^n$ (Наблюдение 17).

(xxvi) Да сравним e^n и $n!$. От Задача 18 знаем, че $e^n < n!$.

(xxvii) Да сравним $n!$ и $(n+1)!$. Но $(n+1)! = (n+1) \cdot n!$. Очевидно $n! < (n+1) \cdot n!$, понеже $n+1$ е неограничено растяща функция. Тогава $n! < (n+1)!$.

(xxviii) Да сравним $(n+1)!$ и 2^{2^n} . Да логаритмуваме двете функции. Получаваме съответно $\Theta((n+1) \lg(n+1))$ и 2^n . Но $\Theta((n+1) \lg(n+1)) \asymp n \lg n$. Вече видяхме, че $n \lg n < n^2$, а $n^2 < 2^n$ съгласно Следствие 14. Тогава $\Theta((n+1) \lg(n+1)) < 2^n$. Тогава $(n+1)! < 2^{2^n}$ (Следствие 12).

(xxix) Да сравним 2^{2^n} и $2^{2^{n+1}}$. Очевидно $2^{2^{n+1}} = 2^{2 \cdot 2^n} = (2^{2^n})^2$. Тогава

$$\lim_{n \rightarrow \infty} \frac{2^{2^n}}{2^{2^{n+1}}} = \lim_{n \rightarrow \infty} \frac{2^{2^n}}{(2^{2^n})^2} = \lim_{n \rightarrow \infty} \frac{1}{2^{2^n}} = 0$$

Съгласно Наблюдение 17, в сила е $2^{2^n} < 2^{2^{n+1}}$.

Окончателната наредба е

$$\begin{array}{cccccccc} 1 & \asymp & n^{\frac{1}{\lg n}} & < & \lg(\lg^* n) & < & \lg^* n & \asymp & \lg^*(\lg n) & < & 2^{\lg^* n} & < \\ \ln \ln n & < & \sqrt{\lg n} & < & \ln n & < & \lg^2 n & < & 2^{\sqrt{2 \lg n}} & < & (\sqrt{2})^{\lg n} & < \\ n & \asymp & 2^{\lg n} & < & n \lg n & \asymp & \lg n! & < & n^2 & \asymp & 4^{\lg n} & < \\ n^3 & < & (\lg n)! & < & (\lg n)^{\lg n} & \asymp & n^{\lg \lg n} & < & \left(\frac{3}{2}\right)^n & < & 2^n & < \\ n \cdot 2^n & < & e^n & < & n! & < & (n+1)! & < & 2^{2^n} & < & 2^{2^{n+1}} \end{array}$$

□

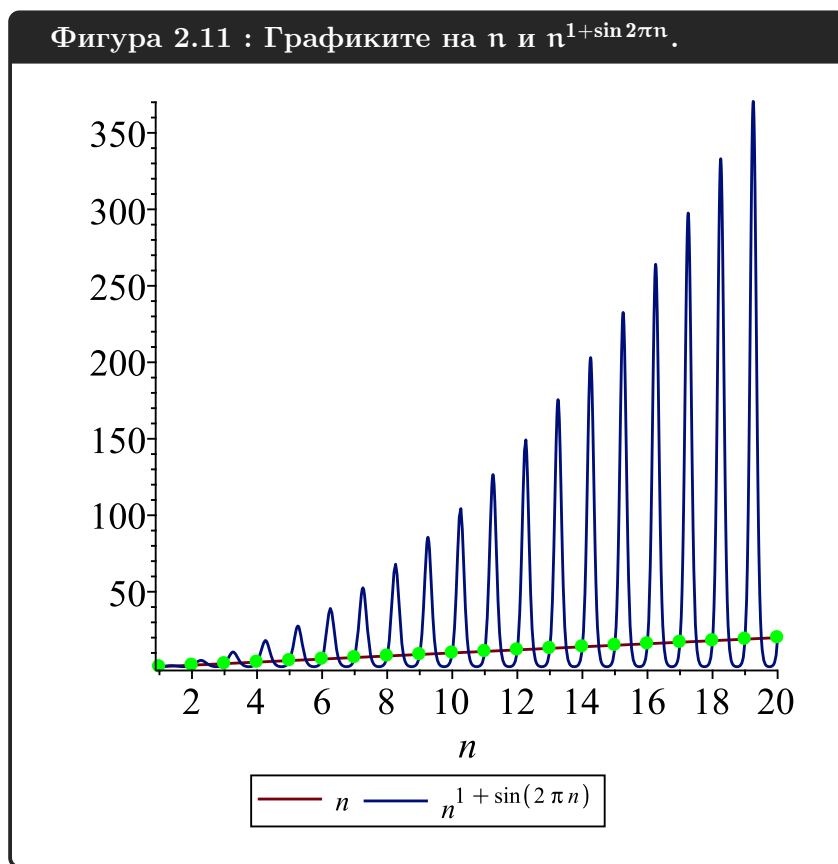
2.3.8 Релациите на асимптотични сравнения при целочислена променлива

Вече приехме, че разглеждаме функции с домейн $\mathbb{R}^+$ (вж. Конвенция 5). Дали нещо ще се промени, ако преминем към домейн $\mathbb{N}^+$? В алгоритмичния анализ функциите са с домейн именно $\mathbb{N}^+$, понеже големината на входа е цяло положително число.

При “добре държащи се” функции като n , n^2 , 2^n и така нататък, нищо не се променя при преминаване от домейн $\mathbb{R}^+$ към домейн $\mathbb{N}^+$. Примерно, $n^2 \asymp n^2 + n$, $2^n > n$, $n 2^n < 3^n$ и когато ги разглеждаме върху $\mathbb{R}^+$, и когато ги разглеждаме върху $\mathbb{N}^+$.

Дори функцията да има не-цели стойности при целочислни аргументи, примерно $\frac{n}{5}$, $\sqrt{n}$ или $\log_2 n$, същото остава в сила, стига функцията да е “добре държаща се”: забележете, че $\frac{n}{5} \geq \sqrt{n}$ е изпълнено и върху $\mathbb{R}^+$, и върху $\mathbb{N}^+$.

Има обаче “лошо държащи се” функции, при които резултатът от асимптотичното сравнение е чувствителен към избора на домейн ($\mathbb{N}^+$ или $\mathbb{R}^+$). Лесно е да конструираме пример за такава функция. Да си припомним функцията $g(n) = n^{1+\sin n}$ на стр. 121. Както стана ясно там, $f(n) = n$ и $g(n) = n^{1+\sin n}$ са асимптотично несравними. Напълно аналогично, функциите n и $n^{1+\sin 2\pi n}$ са асимптотично несравними, когато $n \in \mathbb{R}^+$, понеже $1 + \sin 2\pi n$ осцилира между 0 и 2, също както $1 + \sin n$. Да разгледаме n и $n^{1+\sin 2\pi n}$, когато n е цяло число. Функцията n , която е “добре държаща се”, има същото поведение при $n \in \mathbb{N}^+$. От друга страна, $n^{1+\sin 2\pi n} = n$ сега се държи по съвсем различен начин, понеже $\forall n \in \mathbb{N}^+ : \sin 2\pi n = 0$. Излиза, че върху естествените числа, функциите n и $n^{1+\sin 2\pi n}$ съвпадат. В сила е $n \asymp n^{1+\sin 2\pi n}$, ако $n \in \mathbb{N}^+$. Фигура 2.11 илюстрира всичко това – графиките съвпадат точно върху целите стойности на аргумента.



Друга двойка функции, асимптотичното сравнение на които се променя при целочислен аргумент, е $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$ на стр. 122. Както видяхме там, те съвпадат при $n \in \mathbb{N}^+$.

Ако домейнът е $\mathbb{N}^+$, може да даваме примери за осцилиращи функции не чрез използване на тригонометрични функции, а просто чрез различно дефиниране върху четните и нечетните

числа. Примерно, следните две функции са асимптотично несравними:

$$f(n) = n^2$$

$$g(n) = \begin{cases} n, & \text{ако } n \text{ е четно} \\ n^3, & \text{ако } n \text{ е нечетно} \end{cases}$$

2.3.9 Асимптотичните нотации и релациите на асимптотични сравнения

На пръв поглед, релациите $\asymp$ и така нататък са доста по-удобни и прегледни от съответните асимптотични нотации като Θ и така нататък. Но, както казва Knuth в [91], има случаи, в които е много по-удобно да ползваме асимптотичните нотации, а не релационните символи. А именно, когато нотациите се появяват вътре в някакви изрази (а не вдясно от знака “=”). Knuth говори за това, което ние нарекохме “анонимна функция” (вж. Конвенция 6).

Knuth в [91] дава следния пример:

$$\left(1 + \frac{H_n}{n}\right)^{H_n} = e^{\frac{H_n^2}{n} + O\left(\frac{(\lg n)^3}{n^2}\right)}$$

Същото това нещо, написано не с голямо-О нотация, а с релацията $\leq$, би било тромаво и грозно, защото, за да се използва релационният символ, всичко друго освен $\frac{(\lg n)^3}{n^2}$ би трябвало да се намира от едната му страна.

2.3.10 Релациите $\leq$ и $\geq$ като преднаредби

Както видяхме в Следствие 6, $\asymp$ е релация на еквивалентност, подобно на релацията на равенство върху числата. Да разгледаме $\leq$ и $\geq$. Съгласно Следствие 10, това са фундаменталните релации на асимптотични сравнения. Те не са антисиметрични, защото може за различни функции $f(n)$ и $g(n)$ да е изпълнено $f(n) \leq g(n)$ и $g(n) \leq f(n)$, примерно $n^2 \leq n^2 + n$ и $n^2 + n \leq n^2$. Следователно $\leq$ и $\geq$ не са релации на частична наредба, а са по-общите **релации на преднаредба**. Освен това, както вече видяхме в доказателството на Теорема 24(2.52), $\leq$ и $\geq$ не са пълни, защото може за различни функции $f(n)$ и $g(n)$ да не е изпълнено нито $f(n) \leq g(n)$, нито $g(n) \leq f(n)$. Следният допълнителен материал съдържа достатъчно, за да бъде осмислено това.

Допълнение 18: За преднаредбите

В това допълнение ще обясним в детайли понятието *преднаредба*, което не се изучава в курса по Дискретна Математика, но е важно за осмислянето на релациите $\leq$ и $\geq$. Да си припомним някои дефиниции за релациите по принцип. Сравнете това изложение с изложението в [129, стр. 49–50].

Определение 35: Видови релации

Нека A е произволно множество и $R \subseteq A \times A$ е хомогенна бинарна релация над него. Казваме, че:

- R е *рефлексивна*, ако $\forall a \in A : aRa$.
- R е *ирефлексивна*, още наречена *стриктна*, ако $\forall a \in A : \neg aRa$.
- R е *симетрична*, ако $\forall a, b \in A : aRb \rightarrow bRa$.
- R е *антисиметрична*, ако $\forall a, b \in A : aRb \wedge bRa \rightarrow a = b$.
- R е *пълна*, ако $\forall a, b \in A : a \neq b \rightarrow aRb \vee bRa$.
- R е *транзитивна*, ако $\forall a, b, c \in A : aRb \wedge bRc \rightarrow aRc$.
- R е *преднаредба*, още наричана *квази-наредба*^a, ако е рефлексивна и транзитивна.
- R е *строга преднаредба*, ако е ирефлексивна и транзитивна.
- R е *частична наредба*, ако е антисиметрична преднаредба.
- R е *линейна наредба*, ако е пълна частична наредба.
- R е *релация на еквивалентност*, ако е симетрична преднаредба.

Казваме, че R' е *рефлексивното затваряне* на R , ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е рефлексивна. Казваме, че R' е *симетричното затваряне* на R , ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е симетрична. Казваме, че R' е *транзитивното затваряне* на R , ако R' е най-малката релация, такава че $R \subseteq R'$ и R' е транзитивна.

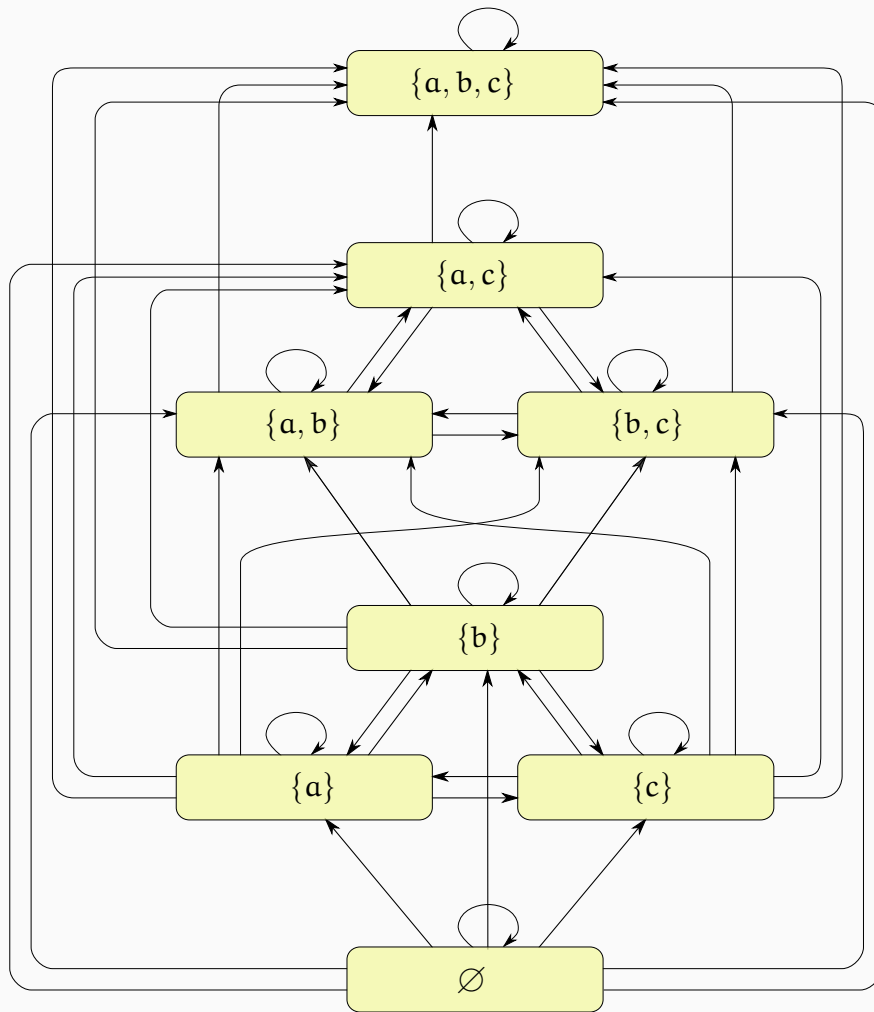
^aНа английски, *preorder* или *quasi-order*.

Пример за “истинска” преднаредба. Да разгледаме един пример за преднаредба, която не е нито релация на еквивалентност, нито на частична наредба; тоест, “истинска” преднаредба. Нека S е крайно множество. Нека $\mathcal{R}' \subseteq 2^S \times 2^S$ е дефинирана така:

$$\forall X \in 2^S \forall Y \in 2^S : (X, Y) \in \mathcal{R}' \leftrightarrow |X| \leq |Y| \quad (2.91)$$

Очевидно $\mathcal{R}'$ е рефлексивна и транзитивна – това следва от свойствата на релацията $\leq$ върху естествените числа. Забележете, че $\mathcal{R}'$ не е нито симетрична, нито антисиметрична. Нека $S = \{a, b, c\}$. Тъй като, примерно, $(\{a, b\}, \{a, b, c\}) \in \mathcal{R}'$ и $(\{a, b, c\}, \{a, b\}) \notin \mathcal{R}'$, то $\mathcal{R}'$ не е симетрична. От друга страна, тъй като, примерно, $(\{a, b\}, \{b, c\}) \in \mathcal{R}'$ и $(\{b, c\}, \{a, b\}) \in \mathcal{R}'$, то $\mathcal{R}'$ не е антисиметрична. Така че $\mathcal{R}'$ е “истинска” преднаредба. Диаграмата на $\mathcal{R}'$ при $S = \{a, b, c\}$ е показана на Фигура 2.12. Забележете, че $\mathcal{R}'$ е и пълна релация, защото за всеки две подмножества на S , поне за едното е вярно, че има мощност поне колкото другото.

Фигура 2.12 : Диаграма на преднаредбата $\mathcal{R}'$ от (2.91).



Следното определение и лема са Lemma 1 в книгата на Birkhoff [20, стр. 21].

Определение 36: Фактор-релация

Нека A е произволно множество и $R \subseteq A \times A$ е някаква преднаредба над него. Нека $\simeq \subseteq A \times A$ ^a е следната релация: $\forall a, b \in A : a \simeq b \leftrightarrow aRb \wedge bRa$. Очевидно $\simeq$ е релация на еквивалентност над A . При това, ако E и F са два класа на еквивалентност на $\simeq$, тогава или $\forall x \in E \forall y \in F : xRy$, или $\forall x \in E \forall y \in F : \neg xRy$. Нека $\mathcal{X}$ е множеството от класовете на еквивалентност на $\simeq$. *Фактор-релацията*^b на R спрямо $\simeq$ е релацията $\mathcal{S} \subseteq \mathcal{X} \times \mathcal{X}$, дефинирана така:

$$\forall P, Q \in \mathcal{X} : (P, Q) \in \mathcal{S} \leftrightarrow \exists a \in P \exists b \in Q : aRb$$

^aОригиналният символ е $\sim$, а не $\simeq$. Тук ползваме $\simeq$, за да няма объркване с релацията за асимптотична еквивалентност от Подсекция 2.3.12.

^bВ книгата на Birkhoff се използва терминът *quotient set*.

Лема 13

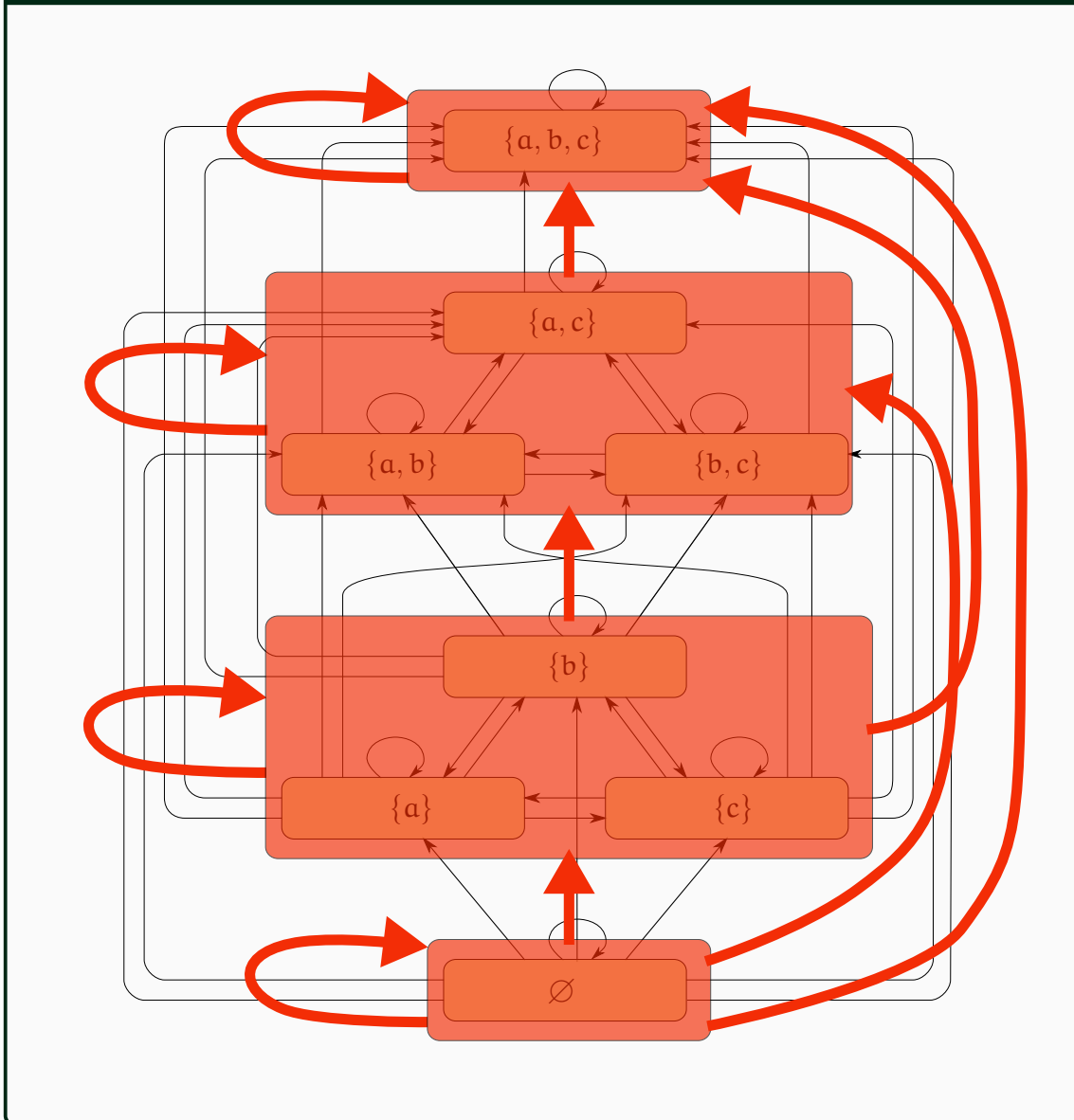
Фактор-релацията е релация на частична наредба.

Нотация 4: Фактор-релацията $R/\simeq$

Означаваме фактор-релацията с $R/\simeq$.

Лесно се вижда, че всяка преднаредба $R \subseteq A \times A$ е пълна тогава и само тогава, когато $R/\simeq$ е линейна наредба (върху класовете на еквивалентност на $\simeq$). Примерно, релацията $\mathcal{R}'$, дефинирана в (2.91) е пълна преднаредба. Фигура 2.13 показва $\mathcal{R}'/\simeq$, която очевидно е линейна. Класовете на еквивалентност на $\simeq$ са очертани с полупрозрачни червени кутии, а диаграмата на $\mathcal{R}'/\simeq$ е нарисувана върху тях с червени стрелки.

Фигура 2.13 : Фактор-релацията $\mathcal{R}'/\simeq$ на преднаредбата $\mathcal{R}'$ от (2.91) е линейна.



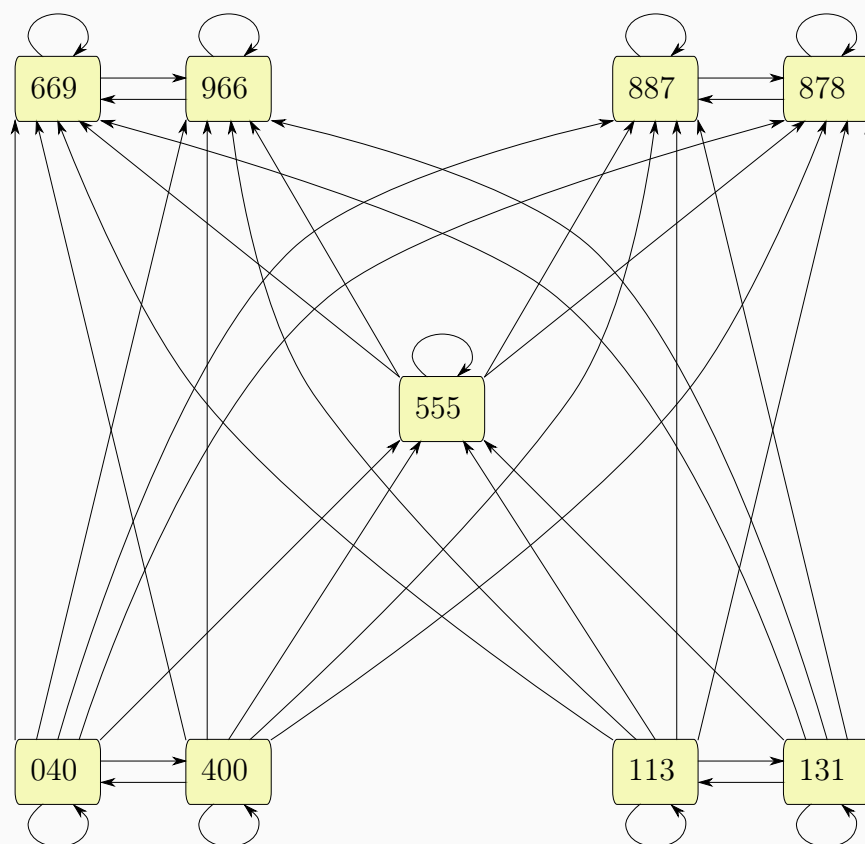
Пример за преднаредба, която не е пълна. Нека V е следното множество от триелементни редици от естествени числа:

$$V = \{040, 400, 113, 131, 555, 779, 977, 886, 868\}$$

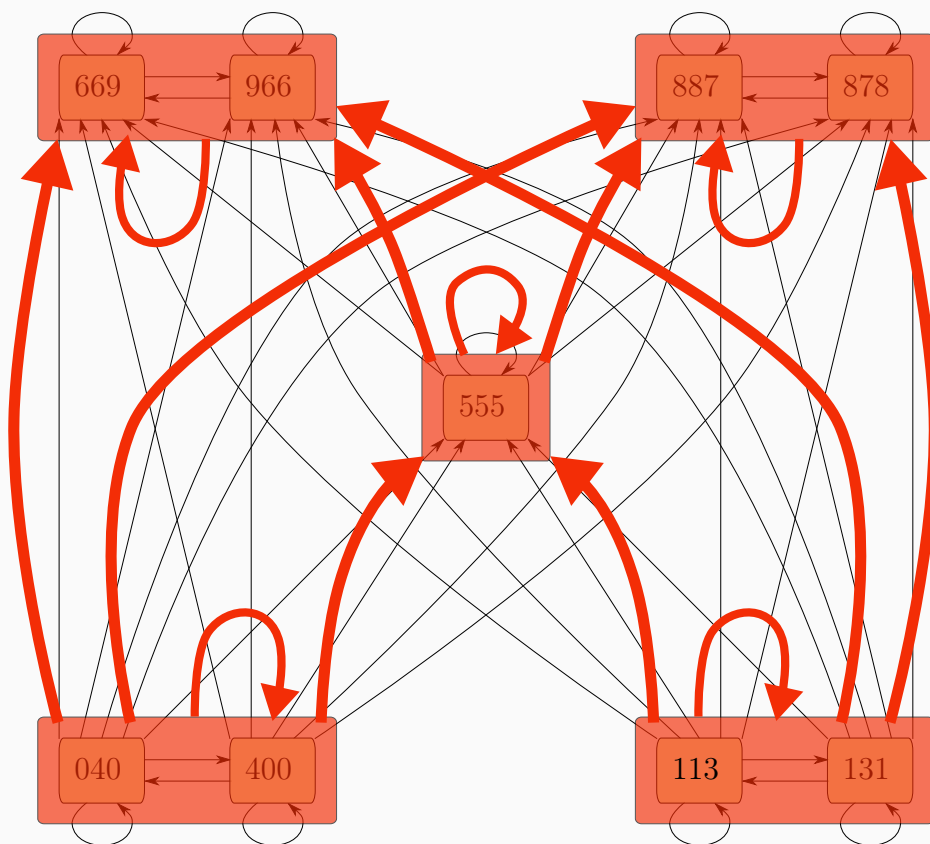
Нека $\mathcal{Q}$ е следната релация над V :

$$\forall \mathbf{x}, \mathbf{y} \in V : (\mathbf{x}, \mathbf{y}) \in \mathcal{Q} \leftrightarrow \sum \mathbf{x} = \sum \mathbf{y} \text{ или } (x_i \leq y_i, \text{ за } i \in \{1, 2, 3\})$$

където $\sum \mathbf{x}$ означава сумата от елементите на редицата $\mathbf{x}$ (аналогично и за $\mathbf{y}$), а x_i е i -ият елемент на $\mathbf{x}$ (аналогично и за $\mathbf{y}$). Диаграмата на $\mathcal{Q}$ е показана на Фигура 2.14. Фигура 2.15 показва фактор-релацията.

Фигура 2.14 : Преднаредбата $\mathcal{Q}$ не е пълна.

Фигура 2.15 : Фактор-релацията $Q/\simeq$ не е линейна.



Наблюдение 20

Релациите $<$ и $>$ са ирефлексивни и транзитивни, а $\leq$ и $\geq$ са рефлексивни и транзитивни. Тогава $<$ и $>$ са строги преднаредби, а $\leq$ и $\geq$ са преднаредби върху множеството на функциите.

Преднаредбите $\leq$ и $\geq$ не са антисиметрични.

Преднаредбите $\leq$ и $\geq$ не са пълни.

Тъй като $\simeq$ е релация на еквивалентност, имаме право да говорим за фактор-релациите $\leq/\simeq$ и $\geq/\simeq$. Техните класове на еквивалентност са (неизброимо безкрайните, неизброимо безкрайно на брой) максимални по включване множества от функции, които са Тета една от друга.

Фактор-релациите $\leq/\simeq$ и $\geq/\simeq$ са “истински” частични наредби, в които има несравними елементи.

Ако ограничим домейна на релациите до ненамаляващите положителни функции, $\leq/\simeq$ има минимален елемент, който е множеството от константите (положителните реални числа). Максимален елемент, естествено, няма – винаги има по-бързо растяща функция. Дуално, $\geq/\simeq$ има максимален елемент, който е множеството от константите, но няма минимален елемент.

2.3.11 Асимптотични сравнения на функции на повече от една променлива

Определение 37 е обобщение на Определение 20 за функции на две променливи.

Определение 37: $\Theta(g(n, m))$

За всяка функция $g(n, m)$:

$$\Theta(g(n, m)) \stackrel{\text{def}}{=} \{f(n, m) \mid \exists c_1, c_2 > 0 \exists n_0, m_0 \forall n \geq n_0 \forall m \geq m_0 : \\ 0 \leq c_1 \cdot g(n, m) \leq f(n, m) \leq c_2 \cdot g(n, m)\}$$

Очевидно е как да обобщим и другите четири асимптотични нотации за функции на две променливи. Също така е очевидно как да обобщаваме и за повече от две променливи.

2.3.12 Релация $\sim$: друга релация на асимптотична еквивалентност

Определение 38: Релация $\sim$ над множеството на функциите

За всички функции $f(n)$ и $g(n)$, $f(n) \sim g(n)$ тогава и само тогава, когато $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$.

Очевидно релацията $\sim$ е рефлексивна, симетрична и транзитивна, поради което е релация на еквивалентност над множеството на функциите, също като $\asymp$. Да си припомним Наблюдение 15. От него следва, че $\sim$ е по-“силна” от $\asymp$ в смисъл, че:

$$\begin{aligned} f(n) \sim g(n) &\rightarrow f(n) \asymp g(n) \\ \neg(f(n) \asymp g(n) &\rightarrow f(n) \sim g(n)) \end{aligned}$$

Поради това доказателството, че за някакви две функции $f(n)$ и $g(n)$ е в сила $f(n) \sim g(n)$, може да е по-трудно от доказателството, че $f(n) \asymp g(n)$. В тези лекции почти няма да ползваме по-силната релация $\sim$ и ще смятаме, че ако сме показали $f(n) \asymp g(n)$, то ние сме намерили най-точната възможна асимптотична близост между $f(n)$ и $g(n)$.

Забележете, че току-що дефинираната $\sim$ и фактор-релацията $\simeq$ от Допълнение 18 (вижте Определение 36) нямат нищо общо.

2.4 Примери за изчисляване на сложност по време на итеративни алгоритми

Много грубо казано, на практика най-често сложността на даден итеративен алгоритъм се оказва $\Theta(n^k)$, където k е максималната вложеност на цикли в него. В тази секция ще видим няколко примера за изчисляване на сложност по време на итеративни алгоритми.

2.4.1 Алгоритми с единичен цикъл

Да разгледаме най-елементарно сумиране на числа от масив.

TRIVIAL SUMMATION($A[1..n]$: масив от цели числа)

```

1  s ← 0
2  for i ← 1 to n
3      s ← s + A[i]
4  return s

```

Може да направим точна сума като (2.21) по начин, аналогичен на тамошния. Но за простота ще разсъждаваме така. Интересува ни само асимптотиката на сложността, а не точната функция. Сложността на алгоритъма е сумата от сложността на изпълнението преди цикъла, сложността на цикъла и сложността на изпълнението след цикъла. От тези сложности, първата и третата са $\Theta(1)$. Да видим каква е втората. Тялото на **for**-цикъла е $s \leftarrow s + A[i]$. Само по себе си, то има сложност $\Theta(1)$. Цикълът очевидно се изпълнява n пъти. Тогава сложността на цялото изпълнение на цикъла е $n \times \Theta(1)$, а за целия алгоритъм имаме

$$\underbrace{\Theta(1)}_{\text{сложността преди цикъла}} + \underbrace{n \times \Theta(1)}_{\text{сложността на цикъла}} + \underbrace{\Theta(1)}_{\text{сложността след цикъла}}$$

Елементарно се показва, че $\Theta(1) + n \times \Theta(1) + \Theta(1) = \Theta(n)$, така че сложността е $\Theta(n)$.

С напълно аналогични разсъждения извеждаме, че всеки алгоритъм по схемата

```

«инициализация»
for i ← c1 to f(n)
    «тяло-на-цикъл»
«завършване»

```

където и *«инициализация»*, и *«тяло-на-цикъл»*, и *«завършване»* имат сложност $\Theta(1)$, а c_1 е някаква константа, има сложност $\Theta(f(n))$. Този резултат остава в сила дори тялото на цикъла да не е просто редица от императивни инструкции, а да има и условни инструкции; дори да има сложна структура от вложени if-then-else, ако всеки изчислителен път през тялото има сложност $\Theta(1)$, резултатът е в сила.

Ако *«инициализация»* има сложност $\phi(n)$, *«тяло-на-цикъл»* има сложност $\sigma(n)$, а *«завършване»* има сложност $\psi(n)$, общата сложност е $\Theta(\phi(n) + \psi(n) + f(n)\sigma(n))$.

2.4.2 Алгоритми с вложени цикли

Да разгледаме алгоритъм по схемата

```

«инициализация»
for i ← c1 to f(n)
    for j ← c2 to g(n)
        «тяло-на-цикъл»
    «завършване»

```

където и *«инициализация»*, и *«тяло-на-цикъл»*, и *«завършване»* имат сложност $\Theta(1)$, а c_1 и c_2 са някакви константи. С разсъждения, напълно аналогични на тези от предната подсекция, извеждаме, че сложността е $\Theta(f(n)) \times x$, където x е сложността на вътрешния цикъл. На свой ред тя е $\Theta(g(n))$, така че общата сложност е $\Theta(f(n)g(n))$.

Ако «инициализация» има сложност $\phi(n)$, «тяло-на-цикъл» има сложност $\sigma(n)$, а «завършване» има сложност $\psi(n)$, общата сложност е $\Theta(\phi(n) + \psi(n) + f(n)g(n)\sigma(n))$.

Лесно се вижда, че алгоритъм по схемата

```

«инициализация»
for i ← c1 to f(n)
    for j ← c2 to g(n)
        for j ← c3 to h(n)
            «тяло-на-цикъл»
«завършване»

```

където и «инициализация», и «тяло-на-цикъл», и «завършване» имат сложност $\Theta(1)$, а c_1 , c_2 и c_3 са някакви константи, има сложност $\Theta(f(n)g(n)h(n))$. Очевидно е как се обобщава това в случай, че «инициализация», «тяло-на-цикъл» и «завършване» имат сложности, зависещи от n , както и за по-голяма вложеност на циклите.

На практика често се срещат итеративни алгоритми с еднократна вложеност на цикъла, в които обаче индексната променлива на вътрешния цикъл се инициализира не с константа, каквато беше c_2 тук, а чрез някаква функция на i , индексната променлива на външния цикъл. Пример за това е SELECTION SORT на стр. 105, при който индексната променлива j на вътрешния цикъл се инициализира с $i + 1$. Ако имаме задача да анализираме такъв алгоритъм, най-сигурно е да направим сума като (2.24) – от нея асимптотиката на сложността се вижда веднага. Не можем със сигурност да твърдим, че сложността е $\Theta(n^2)$ само заради това, че имаме вложен цикъл, ако нямаме теоретичен резултат, покриващ конкретния случай.

Ще разгледаме два примера за двукратна вложеност на цикъла. В единия от тях наистина сложността се оказва $\Theta(n^3)$, което е често срещано при цикъл-в-цикъл-в-цикъл с линейни горни граници за индексните променливи, но при другия не е така.

Първи примерен алгоритъм. Този алгоритъм не прави нищо интересно и го ползваме само като пример за анализ на сложността. Входът е масив, който не се ползва от алгоритъма; в някакъв смисъл, той е запис на n в унарна бройна система.

ALGX($A[1..n]$): масив от цели числа)

```

1  s ← 0
2  for i ← 1 to n
3      for j ← i + 1 to n
4          for k ← i + j - 1 to n
5              s++
6  return s

```

Ще изследваме стойността s , която връща ALGX. Ще я разгледаме като сума, ще опростим сумата до еквивалентен израз-произведение, откъдето асимптотиката ще стане очевидна, и тази асимптотика ще се окаже и сложността на ALGX по време. Тъй като извеждането е дълго, ще го обособим като лема.

Лема 14

ALGX връща

$$s = \begin{cases} \frac{k(k+1)(4k-1)}{6}, & \text{ако } n = 2k \\ \frac{k(k+1)(4k+5)}{6}, & \text{ако } n = 2k + 1 \end{cases}$$

Доказателство: От кода на ALGX имаме

$$s = \sum_{i=1}^n \sum_{j=i+1}^n \sum_{k=i+j-1}^n 1$$

понеже при всяко изпълнение на тялото към s се добавя единица, а s се инициализира с 0.

Да оценим най-вътрешната сума $\sum_{k=i+j-1}^n 1$. Лесно се вижда, че долната граница $i+j-1$ може да надхвърли горната граница n . Ако се случи това, сумата е 0, защото индексната променлива взема стойности от празното множество. По-подробно написано, за всяко цяло t ,

$$\sum_{i=t}^n 1 = \begin{cases} n - t + 1, & \text{ако } t \leq n \\ 0, & \text{в противен случай} \end{cases}$$

Това точно съответства на факта, че в алгоритъма най-вътрешният цикъл не се изпълнява изобщо, ако стойността, с която се инициализира k , надхвърля n . И така,

$$\sum_{k=i+j-1}^n 1 = \begin{cases} n - i - j + 2, & \text{ако } i + j - 1 \leq n, \text{ тоест } j \leq n - i + 1 \\ 0, & \text{в противен случай} \end{cases}$$

Тогава

$$s = \sum_{i=1}^n \sum_{j=i+1}^{n-i+1} (n + 2 - (i + j))$$

Но вътрешната сума е 0, ако $i+1 > n-i+1$, тоест, ако $2i > n$, тоест, ако $i > \lfloor \frac{n}{2} \rfloor$. Следователно,

максималната стойност на i , която трябва да разгледаме, е $\lfloor \frac{n}{2} \rfloor$:

$$\begin{aligned}
 s &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} (n+2-(i+j)) = \\
 &= (n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} 1 - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i \left(\sum_{j=i+1}^{n-i+1} 1 \right) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} j = \\
 &= (n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-i+1-(i+1)+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-i+1-(i+1)+1) - \\
 &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\sum_{j=1}^{n-i+1} j - \sum_{j=1}^i j \right) = \\
 &= (n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-2i+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-2i+1) - \\
 &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\frac{(n-i+1)(n-i+2)}{2} - \frac{i(i+1)}{2} \right) = \\
 &= (n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - 2(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i - (n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 - \\
 &= \frac{1}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} ((n+1)(n+2) - i(2n+3) + i^2 - i^2 - i) = \\
 &= (n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - (3n+5) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 - \\
 &= \frac{(n+1)(n+2)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 + \frac{(2n+4)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i = \\
 &= \left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) - (3n+5) \frac{\left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}{2} + 2 \frac{\left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \left(2 \left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}{6} - \\
 &= \frac{1}{2} \left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) + (n+2) \frac{\left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}{2} = \\
 &= \frac{\left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2)}{2} - \frac{\left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) (2n+3)}{2} + \frac{\left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \left(2 \left\lfloor \frac{n}{2} \right\rfloor + 1 \right)}{3}
 \end{aligned}$$

Ако n е четно, тоест $n = 2k$ за някое $k \in \mathbb{N}^+$, то $\lfloor \frac{n}{2} \rfloor = k$, така че

$$\begin{aligned}
 s &= \frac{k(2k+1)(2k+2)}{2} - \frac{k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} = \\
 &= \frac{k(k+1)(4k+2) - k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} = \\
 &= k(k+1) \left(-\frac{1}{2} + \frac{2k+1}{3} \right) = \frac{k(k+1)(4k-1)}{6}
 \end{aligned}$$

Ако n е нечетно, тоест $n = 2k + 1$ за някое $k \in \mathbb{N}$, то $\lfloor \frac{n}{2} \rfloor = k$, така че

$$\begin{aligned} s &= \frac{k(2k+2)(2k+3)}{2} - \frac{k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} = \\ &= \frac{k(k+1)(4k+6) - k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} = \\ &= k(k+1) \left(\frac{1}{2} + \frac{2k+1}{3} \right) = \frac{k(k+1)(4k+5)}{6} \end{aligned}$$

□

От Лема 14 веднага следва, че сложността по време на ALGX е $\Theta(n^3)$, тъй като $n \asymp k$.

Втори примерен алгоритъм. Този алгоритъм също не прави нищо интересно и също го ползваме само като пример за анализ на сложността. Отново входът не се ползва и в някакъв смисъл е запис на n в унарна бройна система.

$\text{ALGY}(A[1..n])$: масив от цели числа)

```

1  s ← 0
2  for i ← 1 to n
3      for j ← i to n
4          for k ← n + i + j - 3 to n
5              s++
6  return s
```

Да изследваме стойността s , която връща ALGY . Има три вложени **for**-цикли, всеки с горна граница n за съответната индексна променлива, тялото на най-вътрешния цикъл се изпълнява в $\Theta(1)$, така че със сигурност $s = O(n^3)$. Само че сега $s \neq \Theta(n^3)$, за разлика от предния алгоритъм. Защо? За всяко достатъчно голямо n е вярно, че ред 5 се изпълнява само за три стойности на наредената тройка (i, j, k) , а именно

$$(i, j, k) \in \{(1, 1, n-1), (1, 1, n), (1, 2, n-1)\}$$

по простата причина, че най-вътрешният цикъл на редове 5–4 изпълнява тялото си само при $n + i + j - 3 \leq n$, тоест, $i + j \leq 3$, тоест,

- $i = 1$ и $j = 1, 2$;
- $i = 2$ и $j = 1$.

И така, ALGY връща $s = 3$.

Дали обаче върната стойност дава асимптотиката на сложността по време? В примерите, които разглеждахме дотук, това беше така. В примера с ALGY това не е така. Ред 4 съдържа проверка, която не може да стане в нула време. Трябва поне единица от дискретното време на алгоритъма, за да бъде установено, че $n + i + j - 3 > n$ и да се премине към следващото изпълнение на втория **for**-цикл на редове 3–5. Ерго, изпълнението на ALGY не е по-бързо от изпълнението на следния алгоритъм.

$\text{ALGZ}(A[1..n])$: масив от цели числа)

```

1  for i ← 1 to n
```

```

2   for j ← i to n
3       прави нещо в константно време

```

Както стана ясно от изложението горе, **ALGZ** има сложност по време $\Theta(n^2)$, което е истинската сложност на **ALGY**.

Тук може да има възражение, че **ALGY** връща 3 за всяко достатъчно голямо n , така че той е еквивалентен на алгоритъм, който просто връща 3 и работи в $\Theta(1)$. Това възражение не е валидно. Ние изследваме сложността на **дадения** алгоритъм, а не на друг, който е еквивалентен нему и е по-бърз. Това заслужава да се натърти.

Забележка 1

Когато се търси сложността на даден алгоритъм, трябва да се разглежда именно този алгоритъм, буквално, а не някаква негова еквивалентна като изход версия, но с подобрена ефикасност. Алгоритъмът е синтактичен обект и се разглежда буквално. Кодът на алгоритъма е алгоритъмът.

2.4.3 Двоично търсене, итеративен вариант

Разглеждаме Алгоритъм 4. Добре известно е, че сложността му в най-лошия случай е логаритмична. Това може да се покаже полужормално с едно изречение: на всяка итерация на цикъла алгоритъмът решава един бит от индекса на **key** в масива—ако **key** изобщо се среща в масива—а индексът има размер, който е $\Theta(\lg n)$. Дали тази аргументация е задоволително формална и прецизна или не, зависи от изискванията. Сега приемаме, че не е задоволително формална. Ето по-формална аргументация.

Лема 15

Сложността по време на Алгоритъм 4 е $O(\lg n)$.

Доказателство: Следното твърдение е инвариант за **while**-цикъла (редове 3–10).

Инвариант 3: Цикълът на **ALG BSEARCH ITER**

При достигане номер t на ред 3 е вярно, че:

$$h - \ell + 1 \leq \frac{n}{2^{t-1}} \quad (2.92)$$

База. Нека $t = 1$. Тогава $\ell = 1$ и $h = n$ заради редове 1 и 2. В сила е

$$n - 1 + 1 \leq \frac{n}{2^{1-1}} \leftrightarrow n \leq \frac{n}{1} \quad \checkmark$$

Поддръжка. Нека изпълнението е на ред 3 и предстои поне още едно изпълнение на **while**-цикъла. Игнорираме възможността $A[\text{mid}] = \text{key}$, защото, ако това е вярно, изпълнението стига до ред 6 и спира, поради което ред 3 няма да бъде достигнат повече. И така, разглеждаме само следните два случая:

- $\text{key} < A[\text{mid}]$. Тогава изпълнението достига ред 8, където h става $\text{mid} - 1$, което всъщност е $\left\lfloor \frac{\ell+h}{2} \right\rfloor - 1$, изразено чрез старото h . ℓ не се променя и изпълнението се връща на

ред 3. Трябва да покажем, че по отношение на новите ℓ и h (новото ℓ е старото ℓ , но така се казва), неравенство (2.92) остава в сила. Започваме от индуктивното предположение (2.92) за старите стойности:

$$\begin{aligned}
 h - \ell + 1 &\leq \frac{n}{2^{t-1}} && // \text{ делим на 2} \\
 \frac{h - \ell}{2} + \frac{1}{2} &\leq \frac{n}{2^t} \\
 \frac{h + \ell - 2\ell}{2} + \frac{1}{2} &\leq \frac{n}{2^t} \\
 \frac{h + \ell}{2} - \ell + \frac{1}{2} &\leq \frac{n}{2^t} \\
 \frac{h + \ell}{2} - \ell + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^t} \\
 \frac{h + \ell}{2} - 1 - \ell + \frac{1}{2} + 1 &\leq \frac{n}{2^t} \\
 \frac{h + \ell}{2} - 1 - \ell + 1 &\leq \frac{n}{2^t} && // \text{ понеже } \frac{1}{2} > 0 \\
 \underbrace{\left\lfloor \frac{h + \ell}{2} \right\rfloor - 1 - \ell + 1}_{\text{новото } h} &\leq \frac{n}{2^t} && // \text{ понеже } \lfloor m \rfloor \leq m, \forall m \in \mathbb{R}^+
 \end{aligned}$$

- $\text{key} > A[\text{mid}]$. Тогава изпълнението достига ред 10, където ℓ става $\text{mid} + 1$, което всъщност е $\left\lfloor \frac{\ell + h}{2} \right\rfloor + 1$, изразено чрез старото ℓ . h не се променя и изпълнението се връща на ред 3. Трябва да покажем, че по отношение на новите ℓ и h (новото h е старото h), неравенство (2.92) остава в сила. Започваме от индуктивното предположение (2.92) за старите стойности:

$$\begin{aligned}
 h - \ell + 1 &\leq \frac{n}{2^{t-1}} && // \text{ делим на 2} \\
 \frac{h - \ell}{2} + \frac{1}{2} &\leq \frac{n}{2^t} \\
 \frac{2h - h - \ell}{2} + \frac{1}{2} &\leq \frac{n}{2^t} \\
 h - \frac{h + \ell}{2} + \frac{1}{2} &\leq \frac{n}{2^t} \\
 h - \frac{\ell + h}{2} + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^t} \\
 h - \frac{\ell + h}{2} - \frac{1}{2} + 1 &\leq \frac{n}{2^t} \\
 h - \left(\frac{\ell + h}{2} + \frac{1}{2} \right) + 1 &\leq \frac{n}{2^t} \\
 h - \left(\left\lfloor \frac{\ell + h}{2} \right\rfloor + 1 \right) + 1 &\leq \frac{n}{2^t} && // \text{ понеже } \left\lfloor \frac{\ell + h}{2} \right\rfloor + 1 \geq \frac{\ell + h}{2} + \frac{1}{2}, \forall \ell, h \in \mathbb{N}^+ \\
 \underbrace{h - \left(\left\lfloor \frac{\ell + h}{2} \right\rfloor + 1 \right) + 1}_{\text{новото } \ell} &&&
 \end{aligned}$$

Доказахме инварианта (2.92). Да наречем “ $t_{\max}$ ” стойността на t при достигането на ред 3, след което тялото на цикъла се изпълнява за последен път[†]. Очевидно в този момент стой-

[†]Ред 3 се достига още един път.

ностите на ℓ и h са равни. Замествайки t с $t_{\max}$ и $h - \ell$ с 0 в инварианта (2.92), получаваме

$$0 + 1 \leq \frac{n}{2^{t_{\max}-1}} \leftrightarrow 2^{t_{\max}} \leq 2n \leftrightarrow t_{\max} \leq 1 + \log_2 n$$

Щом тялото на цикъла се изпълнява $O(\lg n)$ пъти и всяко изпълнение отнема $\Theta(1)$ време, то е вярно, че сложността по време на алгоритъма е $O(\lg n)$. $\square$

Да си припомним, че разглеждаме сложността по време в **най-лошия** случай.

Лема 16

Сложността по време на Алгоритъм 4 е $\Omega(\lg n)$.

Доказателство: Следното твърдение е инвариант за **while**-цикъла (редове 3–10).

Инвариант 4: Цикълът на ALG BSEARCH ITER

При достигане номер t на ред 3 е вярно, че:

$$\frac{n}{2^t} - 4 < h - \ell \quad (2.93)$$

База. Нека $t = 1$. Тогава $\ell = 1$ и $h = n$ заради редове 1 и 2. В сила е

$$\frac{n}{2^{1-1}} - 4 < n - 1 \leftrightarrow \frac{n}{1} - 4 < n - 1 \quad \checkmark$$

Поддръжка. Нека изпълнението е на ред 3 и предстои поне още едно изпълнение на **while**-цикъла. Игнорираме възможността $A[\text{mid}] = \text{key}$, защото, ако това е вярно, изпълнението стига до ред 6 и спира, поради което ред 3 няма да бъде достигнат повече. И така, разглеждаме само следните два случая:

- $\text{key} < A[\text{mid}]$. Тогава изпълнението достига ред 8, където h става $\text{mid} - 1$, което всъщност е $\left\lfloor \frac{\ell+h}{2} \right\rfloor - 1$, изразено чрез старото h . ℓ не се променя и изпълнението се връща на ред 3. Трябва да покажем, че по отношение на новите ℓ и h (новото ℓ е старото ℓ), неравенство (2.93) остава в сила. Започваме от индуктивното предположение (2.93) за старите стойности:

$$\begin{aligned} \frac{n}{2^t} - 4 &< h - \ell && // \text{ делим на 2} \\ \frac{n}{2^{t+1}} - 2 &< \frac{h - \ell}{2} \\ \frac{n}{2^{t+1}} - 2 &< \frac{h + \ell - 2\ell}{2} \\ \frac{n}{2^{t+1}} - 2 &< \frac{h + \ell}{2} - \ell \\ \frac{n}{2^{t+1}} - 4 &< \frac{h + \ell}{2} - 2 - \ell \\ \frac{n}{2^{t+1}} - 4 &< \underbrace{\left\lfloor \frac{h + \ell}{2} \right\rfloor - 1}_{\text{новото } h} - \ell && // \text{ понеже } m - 2 \leq \lfloor m \rfloor - 1, \forall m \in \mathbb{R}^+ \end{aligned}$$

- $\text{key} > A[\text{mid}]$. Тогава изпълнението достига ред 10, където ℓ става $\text{mid} + 1$, което всъщност е $\left\lfloor \frac{\ell+h}{2} \right\rfloor + 1$, изразено чрез старото ℓ . h не се променя и изпълнението се връща

на ред 3. Трябва да покажем, че по отношение на новите ℓ и h (новото h е старото h), неравенство (2.93) остава в сила. Започваме от индуктивното предположение (2.93) за старите стойности:

$$\begin{aligned}
 \frac{n}{2^t} - 4 &< h - \ell \quad // \text{ делим на } 2 \\
 \frac{n}{2^{t+1}} - 2 &< \frac{h - \ell}{2} \\
 \frac{n}{2^{t+1}} - 2 &< \frac{2h - h - \ell}{2} \\
 \frac{n}{2^{t+1}} - 2 &< h - \frac{h + \ell}{2} \\
 \frac{n}{2^{t+1}} - 4 &< h - \frac{h + \ell}{2} - 2 \\
 \frac{n}{2^{t+1}} - 4 &< h - \left(\frac{h + \ell}{2} + 2 \right) \\
 \frac{n}{2^{t+1}} - 4 &< h - \underbrace{\left(\left\lfloor \frac{h + \ell}{2} \right\rfloor + 1 \right)}_{\text{новото } \ell} \quad // \text{ понеже } m + 2 \geq \lfloor m \rfloor + 1, \forall m \in \mathbb{R}^+
 \end{aligned}$$

Доказахме инварианта (2.93). Да наречем “ $t_{\max}$ ” стойността на t при достигането на ред 3, след което тялото на цикъла се изпълнява за последен път[†]. Очевидно в този момент стойностите на ℓ и h са равни. Замествайки t с $t_{\max}$ и $h - \ell$ с 0 в Инвариант (2.93), получаваме

$$\frac{n}{2^{t_{\max}}} - 4 < 0 \leftrightarrow \frac{n}{2^{t_{\max}}} < 4 \leftrightarrow n < 4 \cdot 2^{t_{\max}} \leftrightarrow \log_2 n < 2 + t_{\max} \leftrightarrow t_{\max} > -2 + \log_2 n$$

Очевидно сложността по време на алгоритъма в най-лошия случай е $\Omega(\lg n)$. □

Теорема 37: Сложността на двоичното търсене е логаритмична

Сложността по време на алгоритъма за двоично търсене (Алгоритъм 4) в най-лошия случай е $\Theta(\lg n)$.

Доказателство: Следва веднага от Лема 15 и Лема 16. □

2.5 Винаги ли е лоша високата сложност

В заключение ще разискваме доколко бавно е синоним на лошо в света на алгоритмите. Наистина, ние търсим колкото е възможно по-бързи алгоритми и свикваме да мислим, че бавен алгоритъм е лош алгоритъм, а бърз алгоритъм е добър алгоритъм. Има важни изчислителни задачи, за които не са известни бързи алгоритми, а има и задачи, за които е доказано, при определени допускания, че няма бързи алгоритми. От конвенционалната гледна точка, липсата на бързи алгоритми—независимо от това дали се дължи на човешкото незнание или е иманентна—е нещо лошо; ако мислим, че бърз=добър и бавен=лош, то високата сложност е нещо като проклетие.

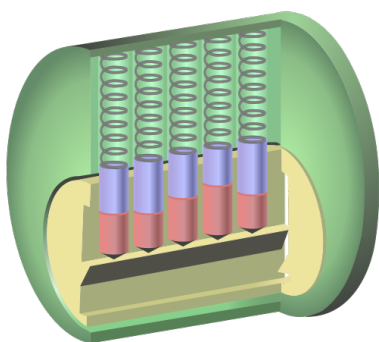
Има обаче друга гледна точка, от която високата сложност е благословия. Това е гледната точка на сигурността. Първо да разгледаме съвсем обикновен пример: ключ от секретна

[†]Ред 3 се достига още един път.

брава като този

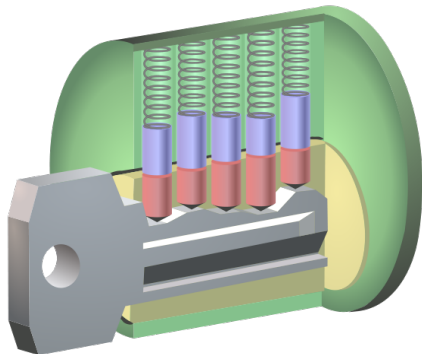


Добре известно е, че секретна брава се отваря при завъртането на цилиндъра в нейната ключалка. Цилиндърът стои плътно в гнездо в ключалката. Ако в ключалката няма ключ, този цилиндър не може да бъде завъртян, защото в него има няколко щифта, обикновено не повече от 6 на брой, сложени в странични напречни отвори, също с цилиндрична форма. Тези отвори са наредени в редица и продължават в гнездото. Всеки щифт е едновременно и в цилиндъра, и в гнездото отстрани. Всеки щифт е срязан на някаква височина, така че всъщност той се състои от два щифта, допрени плътно. Ако няма ключ в ключалката, едно от парчетата на всеки щифт стои едновременно и в цилиндъра, и в гнездото, и така щифтовете пречат на цилиндъра да се върти.



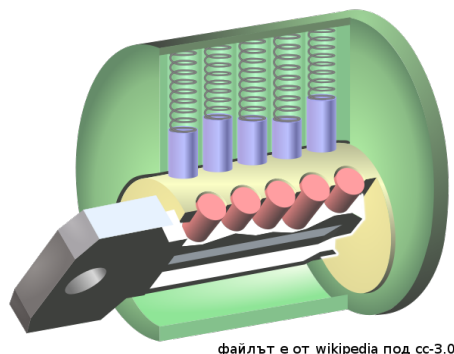
файлът е от wikipedia под cc-3.0

При вкарване на ключ в ключалката, назъбената му повърхност минава точно под щифтовете и ги избутва навън, на различни разстояния съгласно височините на зъбите. Правилният ключ размества щифтовете в отворите им така, че всички срезове на всички щифтове се оказват подравнени точно с границата между цилиндъра и гнездото

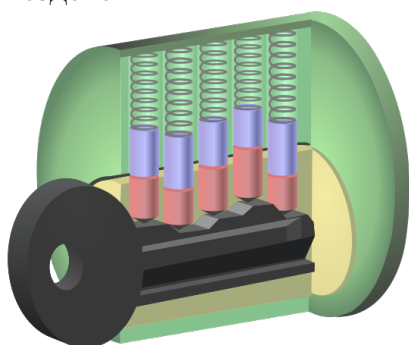


файлът е от wikipedia под cc-3.0

и вече нищо не пречи на цилиндъра да бъде завъртян, което води до издърпване на езика на бравата и вратата се отваря.



Ако обаче в ключалката бъде сложен неправилен ключ, поне един щифт ще пречи на завъртането на цилиндъра, понеже срезът му няма да е подравнен с границата между цилиндъра и гнездото



и бравата няма да бъде отворена.

Броят на щифтовете за дадена ключалка не е тайна. Всеки щифт е срязан на една измежду няколко възможни позиции, обикновено не повече от 10 различни позиции. Бройката на възможните позиции също не е тайна. Тайната на ключа е, точно на каква височина измежду всички възможни височини е срязан всеки щифт. Нека броят на щифтовете е n и има m възможности за различни височини на срязване на всеки щифт. Тогава, от информационна гледна точка, ключът е елемент от $\{1, 2, \dots, m\}^n$. С други думи, редицата от височините, на които са срязани щифтовете, е пълно описание на ключа (заедно с информацията за профила му и за разстоянията между щифтовете, но да игнорираме тези детайли). Всички възможни ключове са най-много краен брой, а именно не повече от m^n . Например, ако $m = 10$ и $n = 6$, ключовете са най-много 1 000 000.

Сигурността при физическите ключалки идва оттам, че атака срещу ключалката, която се състои в изработване на всички възможни ключове и последователното им пробване, е абсолютно непрактична. Виждаме, че в някакъв смисъл **сигурността идва от високата сложност на атаката, която се състои в опитване на всички възможности**; в случая, сложността на изработване на стотици хиляди ключове-проби и последователното им опитване.

В компютърната сигурност, високата сложност дава сигурност по доста аналогичен начин. Тази материя е далеч отвъд началния курс по алгоритми. Тук само ще споменем, че в криптографията[†], която е от огромно значение за компютърната сигурност, в съвременния си вид се основава на използване на ключове (но чисто информационни, а не метални парчета). Съвременната криптография е основана на едно допускане, известно като *Принцип*

[†]Криптография, съвсем накратко, е науката за предаване на информация в среда, в която има подслушвач, по такъв начин, че подслушвачът да не може да разбере какво се предава, въпреки че вижда ясно това, което се предава, и е наясно, че то носи важна информация.

на *Kerckhoffs*: противникът знае всички детайли от криптографската система—алгоритми и протоколи—и единственото, което **не знае**, е ключа[†]. Наистина, използваните в криптографията протоколи не са тайна – несериозно е да се мисли, че система, която е замислена и конструирана от множество хора и се използва с години от много повече хора, може да остане тайна за добре мотивиран противник.

Най-общо казано, криптирането на информацията работи така. Даден участник в комуникацията, която трябва да остане тайна от подслушвача, прилага определен алгоритъм с два входа: съобщението, което трябва да се предаде, и ключа. Изходът от този алгоритъм е стринг, който е неразбираем при прочитане. Процесът на превръщане на разбираемото съобщение в неразбираем стринг се нарича *шифриране*. Приемащата страна (или страни, ако съобщението се праща масово) прилага същия или друг алгоритъм с два входа: полученото неразбираемо съобщение и ключа, и изходът е първоначалното разбираемо съобщение. Процесът на превръщане на неразбираемия стринг в началното разбираемо съобщение се нарича *дешифриране*. Подслушвачът има достъп само до междинния, шифриран вариант на съобщението, от който не се научава нищо. Подслушвачът знае въпросните алгоритми, но не знае ключа. Ако подслушвачът не разполага с някаква информация за ключа и не разполага с алтернативни методи, единственото, което може да направи, е да започне да генерира на сляпо ключ след ключ и да опитва последователно да ги прилага, разчитайки, че ще налучка правилния ключ и ще разбере съобщението. Отново, **сигурността идва от високата сложност на атаката, която се състои в опитване на всички възможности**.

Важен дял от криптографията е криптографията с отворен ключ. При нея, всеки потребител има два ключа, публичен и секретен, като публичния ключ се излага—както показва и името—публично, а секретният се пази в тайна. Публичното излагане на секретния ключ би било фатално за сигурността. Ако потребител А иска да прати съобщение на потребител В, тя взема публичния ключ на В, шифрира с него съобщението си и го праща на В. На свой ред В, използвайки секретния си ключ, дешифрира съобщението. Съобщението може да се дешифрира само чрез секретния ключ на В. Подслушвачът вижда шифрирания вариант, но от него не научава нищо, защото не разполага със секретния ключ на В. Интересното е, че за всеки потребител, двата ключа, публичен и секретен, са взаимно зависими, но сложността на задачата за намиране на секретния ключ от публичния е огромна. Още веднъж, **сигурността идва от високата алгоритмична сложност на атаката**.

Криптографията с отворен ключ има решаващо значение за (почти) сигурното комуникиране в несигурна, публично достъпна среда. Ахилесовата пета на другата криптография—тази само с един ключ—е разпространението на ключа. Компютърните услуги като Интернет банкиране са немислими в масовия си вариант без криптография с публичен ключ.

Задълбочено изложение на връзката между високата сложност на дадени изчислителни задачи и криптографията с отворен ключ има в статиите на Fortnow [52] и Impagliazzo [81]. Изключително детайлно въведение в криптографията е култовата книга *Applied Cryptography* на Bruce Schneier [137].

[†]Аналогично, при секретните брави цялата сигурност е в информацията за височините на зъбите на ключа, при допускането, че противникът знае всичко останало като формата на профила на ключа, броя на щифтовете и т. н., и разполага с необходимата техника, за да си направи физически ключ от описанието му.

Лекция 3

Рекурентни уравнения.

Резюме: Въвеждаме понятието рекурентно уравнение в контекста на анализа на алгоритми. Разглеждаме особеностите на решаването на рекурентни уравнения, когато се иска само асимптотиката на решението. Разглеждаме шест начина за решаване на рекурентни уравнения: чрез развиване, чрез дървото на рекурсията, по индукция, чрез Мастър теоремата, чрез теоремата на Акра-Bazzi и чрез характеристични уравнения.

3.1 Фундамент

3.1.1 Определение

Рекурентно уравнение е уравнение, в което вляво от знака за равенство има символ за функция на някаква променлива-естествено число, например “ $T(n)$ ” или “ a_k ”, а вдясно има валиден израз[†], който съдържа поне една поява на същия символ за функция, но с по-малки стойности на променливата, например “ a_{k-1} ” или “ $T(\frac{n}{2})$ ”. Алтернативна дефиниция е: уравнение, което изразява общия член на някаква редица от числа чрез предишни членове.

За една или повече *начални стойности* на променливата, има други уравнения, наречени *начални условия*, които определят въпросната функция върху началните стойности.

Можем да мислим за рекурентните уравнения като за рекурсивни алгоритми. Наистина, всяко рекурентно уравнение е рекурсивен алгоритъм, който има вход-число и изход-число и вика себе си върху достатъчно големите стойности на входа. Типичен пример е рекурентното уравнение на числата на Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{за } n \geq 2 \end{aligned} \tag{3.1}$$

По правило, рекурентните уравнения са твърде елементарни алгоритми, защото нямат цикли и вложеност на *if*-овете. Важно е да различаваме алгоритъма-рекурентно уравнение от другия алгоритъм – онзи, чиято сложност се описва от уравнението (вижте Наблюдение 21).

Добре дефинирано рекурентно уравнение ще наричаме рекурентно уравнение, което задава коректен (рекурсивен) алгоритъм, който алгоритъм пресмята елементи на някаква безкрайна числова редица. На читателя остава да съобрази, че следното уравнение **не е добре**

[†]Синтактичните правила за това, кои изрази вдясно са валидни, остават неназовани. Първо, тези правила не са универсални, и второ, би било прекален педантизъм да ги излагаме строго формално като граматики. Това само би попречило на изложението.

дефинирано заради “недостиг” на начални условия:

$$T(0) = 1$$

$$T(n) = 2T(n-3) \text{ за } n \geq 1$$

Въпросът дали дадено рекурентно уравнение е добре дефинирано, в частност дали началните условия са достатъчни и смислени, може да е нетривиален. Следното рекурентно уравнение не е добре дефинирано

$$T(1) = 1$$

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} + 17 \right\rfloor\right) + n \text{ за } n > 1$$

поради това, че итераторът (вижте Определение 6) $n \mapsto \left\lfloor \frac{n}{2} + 17 \right\rfloor$, започвайки от произволно естествено число n , има фиксирана точка или 33, или 34; ерго, началната стойност 1 не е достижима от никое число освен самото 1. С други думи, началното условие $T(1) = 1$ в случая е безполезно! Съответната програма ще “гърми” за всеки вход, различен от 1[†].

И така, всяко добре дефинирано рекурентно уравнение определя еднозначно числова редица. Отношението между рекурентни уравнения и числови редици **не е биективно**: за една и съща числова редица има безброй много рекурентни уравнения, такива че тази редица е семантиката на всяко от тези уравнения. Рекурентни уравнения, които определят една и съща числова редица, ще наричаме *еквивалентни*. Убедете се сами, че рекурентното уравнение (3.2) е еквивалентно на (3.1), защото и на двете съответства редицата на Fibonacci 0, 1, 1, 2, 3, 5, 8, ...

$$T(0) = 0$$

$$T(1) = 1$$

$$T(2) = 1$$

$$T(n) = 2T(n-1) - T(n-3) \quad \text{за } n \geq 3 \quad (3.2)$$

Забележка

Подчертаваме, че не трябва да се бъркат двете понятия

- рекурентно уравнение, което е понятие от синтактичното ниво
- функцията, която се реализира от някакво рекурентно уравнение – това е понятие от семантичното ниво. Числовата редица, която съответства на рекурентното уравнение, е практически същото нещо като функцията, която то реализира.

3.1.2 Обобщения

Има различни обобщения на понятието “рекурентно уравнение”, което току-що въведохме. Рекурентните уравнения могат да бъдат на повече от една променлива, например биномният

[†] Става дума за препълване на стека, който операционната система осигурява на съответния процес. Опитайте следния фрагмент на C:

```
int recfun(int n) {if (n == 1) return 1; else return 2*recfun(floor(n/(2.0)) + 17) + n;}
върху n = 1 и n > 1.
```

коэффициент може да се дефинира чрез рекурентното уравнение на две променливи

$$\binom{n}{k} = \begin{cases} 0, & \text{ако } k > n \\ 1, & \text{ако } k = 0 \text{ или } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{в противен случай} \end{cases}$$

или числото на Stirling от втори род може да се дефинира чрез рекурентното уравнение на две променливи

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \begin{cases} 0, & \text{ако } k > n \\ 0, & \text{ако } k = 0 \text{ и } n > 0 \\ 1, & \text{ако } k = n \\ \left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k \cdot \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}, & \text{в противен случай} \end{cases}$$

Разбира се, когато променливите са повече от една, няма голям смисъл да казваме, че рекурентното уравнение задава числова редица. Ако променливите са две, то уравнението задава двумерен безкраен обект; например, уравнението на биномния коефициент задава триъгълника на Pascal (вижте Таблица 12.1).

Може да бъдат дадени няколко взаимно зависими рекурентни уравнения – това е проявление на *взаимна рекурсия*. Пример за такива рекурентни уравнения има в [96, стр. 9] (за краткост изпускаме началните условия):

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 2T_1\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2T_1\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \\ T_1(n) &= T\left(\left\lfloor \frac{n}{2} + 1 \right\rfloor\right) + T_1\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2T_1\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2T_2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2 \\ T_2(n) &= T_1\left(\left\lceil \frac{n}{2} \right\rceil\right) + T_1\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 2T_2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + 2T_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \end{aligned}$$

Друг пример за взаимна рекурентност има [на тази страница](#) на специализирано списание за числа на Fibonacci, където е дадена задачата, да се докаже, че H_{n-1} и G_n са последователни числа на Fibonacci $\forall n \geq 1$, ако F_n са числата на Fibonacci:

$$\begin{aligned} G_1 &= 1 \\ H_0 &= 0 \\ G_n &= F_{n+1}G_{n-1} + F_nH_{n-2}, \quad n \geq 2 \\ H_n &= F_{n+1}G_n + F_nH_{n-1}, \quad n \geq 1 \end{aligned}$$

3.1.3 Рекурентните уравнения са рекурсивни алгоритми

Връзката между рекурентните уравнения и рекурсивните алгоритми е трикова. От една страна, всяко рекурентно уравнение **е** рекурсивен алгоритъм, който за всеки достатъчно голям вход-число връща изход-число след пресмятане, което включва рекурсия (викане/викания на себе си върху по-малко/по-малки число/числа). От друга страна, сложността на рекурсивните алгоритми по правило се описва от рекурентни уравнения. Както вече казахме, сложността на MERGESORT (Секция 6.3) се описва от (6.7).

При изследването на сложността на рекурсивни алгоритми чрез рекурентни уравнения е важно да се прави разлика между алгоритъма, чиято сложност изследваме, и рекурентното уравнение, което описва тази сложност. На свой ред то също е рекурсивен алгоритъм, но

е съвършено различен обект от първоначалния алгоритъм. Като пример да разгледаме следното рекурентно уравнение, където $n \in \mathbb{N}$:

$$T(n) = \begin{cases} 1, & \text{ако } n = 0 \\ nT(n-1) + 1, & \text{в противен случай} \end{cases} \quad (3.3)$$

То е много близко до алгоритъм FACTORIAL на стр. 25. Сега да разгледаме следния алгоритъм. Той не прави нищо полезно, но ще ни послужи за илюстрация.

```

ALGX(A[1 .. n])
1  if n = 0
2      return 1
3  else
4      s ← A[1]
5      for i ← 1 to n
6          s = s + ALGX(A[1 .. n - 1])
7      return s

```

Сложността на ALGX се описва от (3.3). Обаче (3.3) е съвършено различен обект от ALGX. Ако $n = 100$, то никоя практическа имплементация $ALGX(A[1 .. n])$ няма приключи работата си **никога** (вижте Допълнение 12), докато $T(100)$ ще бъде пресметнато **мигновено** върху съвременен компютър[†]. Заслужава си да обособим това наблюдение по следния начин.

Наблюдение 21

Ако е даден рекурсивен алгоритъм ALGX, чиято сложност по време се описва от рекурентно уравнение Y , то Y също е рекурсивен алгоритъм. Обаче ALGX и Y са съвършено различни обекти. Най-малкото, входът на ALGX може да не е число, а масив или друг обект, докато входът на Y —това е променливата на уравнението—е число, имащо смисъла на големина на входа на ALGX. Освен това, по правило ALGX върху вход с големина n работи несравнимо по-бавно от Y с вход числото n .

3.1.4 Класификации на рекурентните уравнения

Тук ще игнорираме началните условия и ще дискутираме “същинското” рекурентно уравнение. Рекурентните уравнения може да се класифицират по много признаци.

- **Брой на появите на функционалния символ вдясно.** Функционалният символ може да се появява точно един път вдясно, например

$$T(n) = 2T(n-1) + 1 \quad (3.4)$$

а може да се появява много пъти с различни стойности на аргумента си, например

$$T(n) = T(n-1) + T(n-3) + T(n-5) \quad (3.5)$$

[†]Стига софтуерът да може да поддържа работа с цели числа с произволна точност (на английски, *arbitrary-precision arithmetic*), за да не се стигне до препълване на променливите заради огромните числа-факториели.

или дори

$$T(n) = T(n-1) + T(n-2) + \dots + T(1) + T(0), \quad (3.6)$$

което се нарича *рекурентно уравнение с пълна история*.

Наблюдение 22

В следното рекурентно уравнение, появите вдясно са една, а не две:

$$T(n) = T(n-1) + T(n-1) + 1$$

Тривиално е да се съобрази, че това рекурентно уравнение всъщност е (3.4). Броят на появите вдясно, за който говорим, е по отношение на **различни** аргументи – както е, например, в (3.5).

Класификацията по брой появи вдясно не е от особена важност, но обикновено рекурентните уравнения с една поява вдясно се решават по-лесно в сравнение с тези с няколко появи. Примерно, Мастър теоремата (Подсекция 3.2.4) е приложима само за рекурентни уравнения с една поява вдясно.

- **Каква е функцията на намаляването** За всяка поява на функционалния символ вдясно, *функцията на намаляването*, неформално казано, е тази математическа операция, която е записана там. Има неограничено много възможности за това.

- ♦ Функцията на намаляването е изваждане на константа като, например, в (3.3), в (3.4) и в (3.5).
- ♦ Функцията на намаляването може да е изваждане, но не на константа. Примерно, $T(n) = T(n - \lfloor \lg n \rfloor)$. Екстремален вариант е уравнение с пълна история като (3.6).
- ♦ Функцията на намаляването може да е деление с константа, например

$$T(n) = 2T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 4T\left(\left\lfloor \frac{n}{7} \right\rfloor\right) + 1$$

- ♦ Може да е деление, но с нарастваща функция, например

$$T(n) = 2T\left(\left\lfloor \frac{n}{\lg n} \right\rfloor\right) + 1$$

- ♦ Може да е коренуване

$$T(n) = nT(\lfloor \sqrt{n} \rfloor) + 1$$

- ♦ Може да е логаритмуване

$$T(n) = T(\lfloor \lg n \rfloor) + 1$$

- ♦ Може да е “смесен” вид намаляване, например:

$$T(n) = T(n-2) + T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 3T(\lfloor \sqrt{n} \rfloor) + 50T(\lfloor \lg n \rfloor) - 10$$

Възможностите наистина са неограничени.

При изследването на сложността на естествени практически алгоритми, смесено намаляване не се среща, а функцията на намаляването е или изваждане на константа, или деление на константа.

- ◆ При алгоритмите, изградени по схемата **Разделяй-и-Владей**, сложността се описва с рекурентни уравнения, при които функцията на намаляването е деление на константа. Ще видим много примери за алгоритми, чиято сложност се описва от такива рекурентни уравнения.
- ◆ Намаляване чрез изваждане на константа се среща често в рекурентните уравнения, описващи сложността на алгоритми, изградени по схемата **Пълно Изчерпване**. Алгоритъм по тази схема е, примерно, алгоритъм за намиране на оптимално (по някакъв критерий, който в момента не назоваваме) подмножество X , който алгоритъм изчислява систематично за всеки елемент a две неща: цената на оптимално решение, в което $a \in X$, и цената на оптимално решение, в което $a \notin X$. На прост български, за всеки елемент алгоритъмът опитва да го сложи в подмножеството и да не го сложи в подмножеството и взема по-доброто решение.

В тези лекционни записки не разглеждаме алгоритми по схемата **Пълно Изчерпване**, понеже те имат (поне) експоненциална сложност, което ги прави абсолютно непрактични дори за скромни стойности на размера на входа (вижте Допълнение 12). Заради това рекурентни уравнения от типа на $T(n) = 2T(n-1) + n$ няма да описват сложността на алгоритми, които разглеждаме.

Следните определения и класификации са в сила само за рекурентни уравнения, при които функцията на намаляването е изваждане на константа. Тези уравнения се наричат още *диференчни уравнения*, на английски *difference equations*. Задълбочено изложение за диференчните уравнения има в книгата “An Introduction to Difference Equations” на Elaydi [43].

- ◆ *Ред* (на английски, *order*) на рекурентното уравнение е най-голямото число, което се изважда от n вдясно. В следните три примера, числото, което определя реда на уравнението, е в червено. И така, това рекурентно уравнение

$$T(n) = 2T(n - 1) + 1$$

е от първи ред, това

$$T(n) = 3T(n - 1) + 4T(n - 2) + n$$

е от втори ред, а това

$$T(n) = T(n - 1) + T(n - 2) + \dots + T(n - k) + n^2$$

е от k -ти ред. Понякога се казва *рекурентно уравнение с крайна история*, за да се подчертае, че k е константа, а не, да речем, $\lfloor \lg n \rfloor$.

Наблюдение 23: В някакъв смисъл, намаляването винаги е изваждане

Ако допуснем k да не е константа, а някаква нарастваща функция на n , то **за всяко** рекурентно уравнение може да твърдим, че функцията на намаляването е изваждане. Например, $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$ може да се запише като $T(n) = T(n - \lfloor \frac{n}{2} \rfloor) + 1$; $S(n) = S(\lfloor \sqrt{n} \rfloor) + 1$ може да се запише като $S(n) = S(n - (n - \lfloor \sqrt{n} \rfloor)) + 1$, и така нататък. От това съображение следва, че класифицирането на рекурентните уравнения по функцията на намаляването е—говорейки теоретично—безсмислено, защото, в някакъв смисъл, тя винаги е изваждане. Но на практика това класифициране е доста удобно.

Подчертаваме, че редът на рекурентното уравнение **не е** същото като броят на появите на функционалния символ вдясно. Например, следното рекурентно уравнение

$$T(n) = T(n - 2) + 2T(n - 5) + 1$$

е от пети ред, макар да има само две появи вдясно. За да се убедим, че е така, да мислим за уравнението като

$$T(n) = 0T(n - 1) + 1T(n - 2) + 0T(n - 3) + 0T(n - 4) + 2T(n - 5) + 1$$

- ♦ *Линейно рекурентно уравнение* е рекурентно уравнение от вида

$$T(n) = f_1(n)T(n - 1) + f_2(n)T(n - 2) + \dots + f_n(n)T(0)$$

където $f_1, \dots, f_n$ са функции, не непременно константи, независещи от $T(n)$. Например, следните три рекурентни уравнения са линейни:

$$A(n) = A(n - 1)$$

$$B(n) = nB(n - 2) + n^2B(n - 4) + \left\lfloor \frac{n \sin n}{\sqrt{n^3 + \lg n}} \right\rfloor B(n - 5)$$

$$C(n) = \sum_{k=0}^{n-1} 2^k C(k)$$

докато тези четири **не са** линейни:

$$A(n) = A(n - 1)A(n - 2) + 1$$

$$B(n) = \left\lceil \frac{1}{1 + B(n - 1)} \right\rceil$$

$$C(n) = \lfloor \sqrt{C(n - 2)} \rfloor$$

$$D(n) = \lfloor \lg(D(n - 1) + D(n - 2)) \rfloor$$

- ♦ Различаваме рекурентни уравнения с константни коефициенти, например

$$P(n) = 1P(n-1)P(n-2) + 3P(n-3)$$

$$S(n) = \left[\sum_{k=1}^{\infty} \frac{1}{k^2} \right] S(n-2) + n$$

$$T(n) = 2T(n-1) + 5T(n-3)$$

от рекурентни уравнения с променливи коефициенти, например

$$P(n) = \lfloor \lg n \rfloor P(n-1)$$

$$S(n) = nS(n-1)$$

$$T(n) = \left\lceil \frac{2+n^2}{3+n} \right\rceil T(n-2)$$

- Различаваме *хомогенни* и *нехомогенни* линейни рекурентни уравнения. При хомогенните, дясната страна е сума от събираеми, всяко от които е от един и същи вид, като всяко от тях задължително съдържа функцията от лявата страна. Примерно,

$$T(n) = c_1T(n-1) + c_2T(n-2) + \dots + c_kT(n-k) \quad (3.7)$$

където k и $c_1, \dots, c_k$ са константи. При нехомогенните има събираеми, които са от съвсем друг вид, които не съдържат функцията от лявата страна. Примерно,

$$T(n) = c_1T(n-1) + c_2T(n-2) + \dots + c_kT(n-k) + f(n) \quad (3.8)$$

където k и $c_1, \dots, c_k$ са константи, където $f(n)$ е функция, която зависи от n и не зависи от $T(n)$. Казваме, че $f(n)$ е *нехомогенната част*.

- Различаваме *дискретни* от *континуални* рекурентни уравнения (Подсекция 3.1.5).

3.1.5 Винаги ли променливата в рекурентно уравнение е целочислена

Това, че функцията-семантика на уравнението може да има стойности, които не са цели числа, е доста очевидно. Например рекурентното уравнение (вж. [141, стр. 56])

$$a_0 = 1$$

$$a_n = \frac{1}{1 + a_{n-1}}$$

определя редицата от дроби $1, \frac{1}{2}, \frac{2}{3}, \frac{3}{5}, \frac{5}{8}, \dots$, която е свързана с числата на Fibonacci. Но променливата n тук е отново целочислена и пак става дума за числова редица.

Ние по правило приемаме, че променливата е целочислена, поради което я записваме с “ n ” или “ k ”. Ако уравнението описва сложност на алгоритъм, това има смисъл, защото размерите на входа са естествени числа. Дали нулата е възможен размер на вход за конкретната ни задача не е съществено; ако ли не, то размерите на входа са цели положителни числа и толкова. Същественото е, че размерите на входа не са дробни, ирационални или отрицателни.

Ако обаче гледаме на уравнението като на самостоятелен обект, нищо не пречи променливата да взема “истински” реални стойности, стига да има някаква долна граница върху тях.

Като пример, в книгата на Flajolet и Sedgewick се посочва (вж. [141, стр. 46]), че методът на Newton за пресмятане на квадратен корен от дадено положително реално β е рекурентно уравнение с $a_0 = 1$:

$$a_n = \frac{1}{2} \left(a_{n-1} + \frac{\beta}{a_{n-1}} \right), \quad n > 0$$

Забележете, че и уравнението $F_n = F_{n-1} + F_{n-2}$ (сравнете с (3.1)) не е безсмислено при реално n , стига началните условия да се предефинират така, че да не могат да бъдат прескочени, от което и (достатъчно голямо) n да стартира процеса на изчислението назад. Една възможност е началните условия да са върху полуотворения интервал $[0, 2)$, например

$$F_n = \begin{cases} n, & \text{ако } n \in [0, 2) \\ F_{n-1} + F_{n-2}, & \text{ако } n > 2 \end{cases} \quad (3.9)$$

Математически погледнато, този алгоритъм е безукорно ефективен. Нещо повече, тъй като реалната променлива намалява с вадене на цели константи, алгоритъмът на (3.9) може да бъде имплементиран ефикасно чрез мемоизация (Секция 12.10).

Наблюдение 24

Ако променливата в рекурентно уравнение е реално число, нямаме право да казваме, че рекурентното уравнение задава числова редица. В такъв случай то задава континуум от стойности, или, по-прецизно казано, реална функция.

Защо изобщо разглеждаме рекурентни уравнения върху реални променливи, при положение, че (в тези лекционни записки) се интересуваме използването на рекурентни уравнения само за изследване на сложността на алгоритми, чиито входове са с целочислени размери? Причината е, че функцията на намаляването не е задължително изваждане на цялочислена константа. Ако функцията на намаляването е деление, дори да е с цяло число, имаме две възможности.

- Да настояваме променливата да е задължително цяло число, което налага да ползваме точна долна или точна горна граница върху резултата от делението. Примерно, рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad (3.10)$$

описващо сложността по време на MERGESORT (Секция 6.3), може да бъде написано по-прецизно като

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad (3.11)$$

Това е смислено, защото в общия случай масивът не може да бъде разделен на две равни части. Най-доброто, което можем да постигнем, е разликата в големините на двете части да най-много единица, което функциите $\lfloor \cdot \rfloor$ и $\lceil \cdot \rceil$ ни гарантират. Но подходът от (3.11) е тромав. На практика, ако нотациите $\lfloor \cdot \rfloor$ и $\lceil \cdot \rceil$ се срещат в рекурентно уравнение, което искаме да решим, първата ни работа е да се освободим от тях, апроксимирайки решението с друго подходящо рекурентно уравнение без тях.

- Да допуснем, че променливата е реална. Това налага началните условия да са върху цял интервал, а не само върху няколко дискретни точки. В примера с уравнение (3.10), началните условия може да бъдат тези:

$$T(n) = n \text{ за } n \in [1, 2)$$

Решението има асимптотика $\Theta(n \lg n)$, също като уравнение (3.11).

Според [99], при първата възможност говорим за *discrete recurrences*, а при втората възможност, за *continuous recurrences*.

Забележе, че при континуалните рекурентни уравнения е безсмислено да говорим за пълна история (вж. (3.6)).

3.1.6 Решаване на рекурентни уравнения

Нека е дадено някакво рекурентно уравнение, в което функционалният символ е T , а името на променливата е n , тоест влияе от знака за равенство има “ $T(n)$ ”. Както стана ясно, то реализира някаква функция; иначе казано, определя някаква числова редица. Да решим това рекурентно уравнение означава да намерим друг, еквивалентен израз (формула) за същата функция, който обаче е **затворена формула**. Например, ако рекурентното уравнение е:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + n \text{ за } n > 0 \end{aligned} \tag{3.12}$$

то решение е $T(n) = \frac{n(n+1)}{2}$ (вижте Задача 24). Какво точно е затворена формула зависи от това, какви операции са позволени. Решението “ $\frac{n(n+1)}{2}$ ” е валидно, ако са позволени операциите събиране с единица, умножение и деление на две.

Да разгледаме друг пример:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= nT(n-1) \text{ за } n > 1. \end{aligned} \tag{3.13}$$

Както е добре известно, решението е $T(n) = n!$, но това решение е валидно само ако можем да ползваме факториела. Ако обаче са разрешени само събиране, изваждане, умножение, деление, коренуване и степенуване, то уравнението (3.13), колкото и да е странно, няма решение. При изброените ограничения, $T(n) = \prod_{i=1}^n i$ или $T(n) = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$ не са валидни решения, защото не са затворени формули. Забележете, че тези два израза използват нотации, касаещи умножение, но не са затворени формули, ако символът “ $\prod$ ” или многоточието не са разрешени.

Тук ще допускаме, че всички “нормални” функции като изброените, плюс логаритмуването и факториелът, са допустими и че решение, което ги ползва, е валидно. Също така, решението може да съдържа биномни коефициенти, числа на Стирлинг от втори род, суми и произведения. Важното е да няма рекурсия – функционалният символ не трябва да се среща вдясно.

Ползете от това да решим дадено рекурентно уравнение са поне две.

- По правило решението е по-бърз алгоритъм за изчисляване на функцията, която съответства на рекурентното уравнение, отколкото алгоритъмът, който е зададен от рекурентното уравнение. Например, $\frac{n(n+1)}{2}$ е по-бърз алгоритъм за решаване на функцията,

която рекурентното уравнение (3.12) реализира, в сравнение с алгоритъма, зададен от самото (3.12). Ако $n = 1\,000\,000\,000$, директното изпълнение на (3.12) ще се извърши в около $2\,000\,000\,000$ стъпки[†], докато $\frac{1\,000\,000\,000(1\,000\,000\,000+1)}{2}$ се изчислява с една инкрементация, едно умножение и едно деление на две.

Има обаче и изключение. Да разгледаме (3.13). Дори $T(n) = n!$ да е валидно решение, алгоритъмът, който то задава, не е съществено по-бърз от рекурсивния алгоритъм на $T(1) = 1, T(n) = nT(n-1)$ за $n > 1$. Така че в този случай е спорно дали решението ни дава (съществено) по-бърз алгоритъм.

Друго потенциално изключение е добре известното решение

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \quad (3.14)$$

на рекурентното уравнение на числата на Fibonacci (3.1). При нереалистичното допускане, че повдигането на n -та степен става за единица време, наистина (3.14) задава алгоритъм със сложност по време $\Theta(1)$. Обаче при по-реалистичен изчислителен модел, който отчита наличието на ирационалните “ $\sqrt{5}$ ” в израза, повдигането на n -та степен е алгоритъм със сложност по време $\Theta(n)$, тъй като всяко двете от събираеми $\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$ и $\frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$ има на свой ред $\Theta(n)$ събираеми, които трябва да присъстват в явен вид, преди да се извършат канцелиранятията, от които появите на $\sqrt{5}$ изчезват, за да остане само цяло число. Забележете, че ако апроксимираме $\sqrt{5}$ с рационално число, ще загубим точност, колкото и добра да е апроксимацията, освен ако нямаме никакви математически гаранции за не-загуба на точност. Повече разсъждения за това има в Допълнение 56.

- Ако решението е достатъчно просто, много бързо можем да преценим каква е асимптотиката на нарастването на $T(n)$. Например, в (3.12) имаме $T(n) = \Theta(n^2)$, което се вижда веднага от решението $T(n) = \frac{n(n+1)}{2}$.

По отношение на числата на Fibonacci, от (3.1) е трудно да се прецени каква е асимптотиката на F_n , докато от (3.14) веднага се вижда, че $F_n \asymp \left(\frac{1+\sqrt{5}}{2} \right)^n$, тоест, горе-долу, $F_n \approx 1.6^n$, понеже $\left(\frac{1+\sqrt{5}}{2} \right)$ е по-малко от единица по абсолютна стойност.

Методите за решаване на рекурентни уравнения ще разискваме в детайли в Секция 3.2. Тук само ще споменем, че универсален **формален** метод за решаване на всякакви рекурентни уравнения **няма**.

3.1.7 Рекурентните уравнения в изследването на алгоритми

Рекурентните уравнения се използват широко в теорията на алгоритмите, защото сложността по време $T(n)$ на даден рекурсивен алгоритъм ALGX се описва с подходящо рекурентно уравнение, което се съставя много лесно. Лявата страна е “ $T(n)$ ”, което има смисъл на времето за работата на алгоритъма върху вход с размер n , а дясната страна е израз, който описва както работата на алгоритъма в рекурсивните викания, така и работата на алгоритъма извън рекурсивните викания.

[†]Един милиард за движението надолу до достигане на началното условие и още един милиард за движението нагоре, от началното условие до $T(1\,000\,000\,000)$.

В дясната страна, за всяко рекурсивно викане има по едно $T(f(n))$, където $f(n)$ е функцията на намаляването в това викане; примерно, $f(n) = n - 1$ или $f(n) = \frac{n}{2}$.

Освен това, има събираемо, което наричаме *нехомогенната част* което отразява работата на алгоритъма извън рекурсивните викания.

Наблюдение 25

Всяко рекурентно уравнение, което описва сложност на алгоритъм, има **положителна** нехомогенна част, защото алгоритъмът няма как да не върши поне константна работа извън рекурсивните викания.

При изследването на алгоритмите не търсим точен израз за сложността. Както се каза в минала лекция, това не е нито възможно, нито е необходимо. Интересува ни само асимптотиката. Намерим ли решение от вида $T(n) = \Theta(g(n))$, намерили сме сложността. Това е огромно облекчение. Ето три последици от това, че търсим само асимптотиката.

По правило игнорираме началните условия. По правило, асимптотиката на решението не зависи от началните условия. Примерно, уравнението $T(0) = 0$, $T(n) = T(n-1) + n$ за $n > 0$ има точно решение $T(n) = \frac{n(n+1)}{2}$ (Задача 24), но решението остава $T(n) \asymp n^2$ за всяка положителна стойност на $T(0)$. Ерго, можем да изпуснем началното условие и да търсим решението на $T(n) = T(n-1) + n$ за неотрицателни n . Каквото и да е началното условие, в сила е $T(n) \asymp n^2$.

Има обаче рекурентни уравнения, чиито решения зависят от началните условия дори и в асимптотичния смисъл. Като пример да разгледаме уравнението

$$T(n) = 8T(n-1) - 20T(n-2) + 16T(n-3) \quad (3.15)$$

Уравнение (3.15) не може да описва сложност на рекурсивен алгоритъм, защото, първо, няма нехомогенна част (Наблюдение 25), и, второ, изваждането $-20T(n-2)$ вдясно няма директна алгоритмична интерпретация[†]. За момента да игнорираме това. (3.15) е решимо с метода с характеристичното уравнение (Подсекция 3.2.6), но, тъй като още не сме разгледали този метод, да използваме Maple (tm). Да кажем, че началните условия са $T(0) = 1$, $T(1) = 20$ и $T(2) = 100$.

```
% C:\Program Files\Maple 2018\bin.X86_64_WINDOWS>cmaple
| \~ / | Maple 2018 (X86 64 WINDOWS)
._ | \ | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \ MAPLE / All rights reserved. Maple is a trademark of
 < ____ > Waterloo Maple Inc.
 | Type ? for help.
> rsolve({T(n) = 8*T(n-1)-20*T(n-2)+16*T(n-3), T(0)=1,T(1)=20,T(2)=100}, {T});

                                     n      n      n
                                {T(n) = (3 n + 3) 2  + 6 4  - 8 2 }

> quit
memory used=4.6MB, alloc=40.3MB, time=0.16
```

[†]Ако четем буквално: правим **минус** двадесет рекурсивни викания върху вход с размер $n-2$. Това няма никакъв смисъл.

Събираемостта 6×4^n е доминиращото, така че $T(n) \asymp 4^n$. Сега да решим (3.15) с Maple (tm) при начални условия $T(0) = 1$, $T(1) = 2$ и $T(2) = 4$.

```
% C:\Program Files\Maple 2018\bin.X86_64_WINDOWS>cmaple
| \~/|      Maple 2018 (X86 64 WINDOWS)
._| \ | |   | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  /   All rights reserved. Maple is a trademark of
 <_ _ _ _ _>   Waterloo Maple Inc.
      |        Type ? for help.
> rsolve({T(n) = 8*T(n-1)-20*T(n-2)+16*T(n-3), T(0)=1,T(1)=2,T(2)=4}, {T});
```

$$\{T(n) = 2^n\}$$

```
> quit
memory used=2.9MB, alloc=8.3MB, time=0.09
```

Сега Maple (tm) казва, че решението е $T(n) = 2^n$, и то точно, а не като асимптотика. Да решим (3.15) с Maple (tm) трети път, сега с начални условия $T(0) = 1$, $T(1) = 10$ и $T(2) = 12$.

```
% C:\Program Files\Maple 2018\bin.X86_64_WINDOWS>cmaple
| \~/|      Maple 2018 (X86 64 WINDOWS)
._| \ | |   | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  /   All rights reserved. Maple is a trademark of
 <_ _ _ _ _>   Waterloo Maple Inc.
      |        Type ? for help.
> rsolve({T(n) = 8*T(n-1)-20*T(n-2)+16*T(n-3), T(0)=1,T(1)=10,T(2)=12}, {T});
```

$$\{T(n) = -3 \cdot 2^n - 6 \cdot 4^n + (10n + 10) \cdot 2^n\}$$

```
> quit
memory used=4.6MB, alloc=40.3MB, time=0.55
```

Това решение е още по-странно. То е отрицателно заради коефициента -6 пред водещото събираемо 4^n , така че нашата теория за асимптотичните нотации не е приложима. Какво става? Да опитаме да решим (3.15) с Maple (tm), без да фиксираме начални условия.

```
% C:\Program Files\Maple 2018\bin.X86_64_WINDOWS>cmaple
| \~/|      Maple 2018 (X86 64 WINDOWS)
._| \ | |   | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  /   All rights reserved. Maple is a trademark of
 <_ _ _ _ _>   Waterloo Maple Inc.
      |        Type ? for help.
> rsolve({T(n) = 8*T(n-1)-20*T(n-2)+16*T(n-3)}, {T});
memory used=3.8MB, alloc=40.3MB, time=0.14
```

$$\{T(n) = -(-T(0) + T(1) - 1/4 T(2)) \cdot 4^n - (-2 T(0) + 1/2 T(1)) \cdot 2^n + (-2 T(0) + 3/2 T(1) - 1/4 T(2)) (n + 1) \cdot 2^n\}$$


```
> quit
memory used=5.2MB, alloc=40.3MB, time=0.17
```

Вече нещата са ясни: коефициентът $T(0) - T(1) + \frac{1}{4}T(2)$ пред 4^n може да е положителен, нула или отрицателен в зависимост от началните условия, дори когато всички те са положителни.

И така, има рекурентни уравнения от видове, които ще разглеждаме, чиито решения зависят от началните условия като асимптотика. Измъкваме се от тази клопка, казвайки, че за уравненията, които ще разглеждаме в курса, винаги има положителни стойности за началните условия, такива че асимптотиката, която нашите формални методи дават, е истинската. Примерно, при рекурентните уравнения от вида от Подсекция 3.2.6 винаги има положителни стойности, за които асимптотиката на решението е такава, каквато е според Следствие 17.

Разбира се, има примери за рекурентни уравнения, чиито решения са много по-чувствителни към началните условия, отколкото са уравненията от вида от Подсекция 3.2.6. Примерно,

$$T(n) = T(n-1) \cdot T(n-2)$$

или дори

$$T(n) = T(n-1)^{T(n-2)}$$

Но ние няма да разглеждаме такива уравнения.

Игнорираме функциите $\text{floor}()$ и $\text{ceiling}()$. За простота ще игнорираме появите на $\lfloor \cdot \rfloor$ и $\lceil \cdot \rceil$ вдясно, освен когато решаваме рекурентно уравнение по индукция (Подсекция 3.2.3). Тъй като се интересуваме само от асимптотиката на решението, ако разглеждаме рекурентно уравнение с целочислена променлива, чиято дясна страна съдържа функциите $\lfloor \cdot \rfloor$ (floor) или $\lceil \cdot \rceil$ (ceiling), просто ще игнорираме тези функции. По този начин преминаваме към реална променлива, което не променя асимптотиката на решението. Примерно, както вече отбелязахме в Подсекция 3.1.5, ако трябва да решим

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

ще решим наместо него това уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{3.16}$$

чието решение има същата асимптотика.

Изключение от това правило има, когато решаваме рекурентното уравнение по индукция (Подсекция 3.2.3). Ако ползваме индукция, не е допустимо просто да изпуснем нотациите $\lfloor \cdot \rfloor$ и $\lceil \cdot \rceil$. В такъв случай се освобождаваме от тях, извършвайки подходящи апроксимации отдолу или отгоре.

Нехомогенната част е в най-простия възможен вид. И още едно опростяване, което не променя асимптотиката на функцията, което ще покажем с пример. Ако уравнение (3.16) описва работата на алгоритъм, който прави две рекурсивни викания и извършва линейна работа извън рекурсивните викания, събираемост, които отразява тази линейна работа—става дума за нехомогенната част—би трябвало да е “ $\Theta(n)$ ”, така че (3.16) би трябвало да

е:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Но асимптотиката на решението не се променя, ако вместо “ $\Theta(n)$ ” напишем просто “ n ”, тоест най-простият за записване представител на класа от функции $\Theta(n)$. И така, записът, който ще ползваме, е като този в (3.16), а именно:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

3.2 Методи за решаване на рекурентни уравнения

3.2.1 Чрез развиване

Средството, което можем да опитаме винаги, за да решим дадено рекурентно уравнение, е да започнем да го развиваме, търсейки някаква закономерност. Ето много прост пример с уравнение, което вече видяхме (3.12).

Задача 24

Решете рекурентното уравнение

$$T(0) = 0$$

$$T(n) = T(n-1) + n \text{ за } n > 0 \quad (3.17)$$

чрез развиване.

Решение: Разсъждаваме така. Нека n е достатъчно голямо число. Тогава е вярно, че $T(n-1) = T((n-1)-1) + (n-1)$, тоест, че $T(n-1) = T(n-2) + n-1$. Ако заместим “ $T(n-1)$ ” в дясната страна на (3.17), с “ $T(n-2) + n-1$ ”, получаваме

$$T(n) = T(n-2) + (n-1) + n \quad (3.18)$$

Но за достатъчно големи n , $T(n-2) = T((n-2)-1) + (n-2)$, тоест $T(n-2) = T(n-3) + (n-2)$. Ако сега заместим дясната страна в (3.18), получаваме

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad (3.19)$$

По правило, ако след 3-4 такива последователни развивания не видим такава закономерност, няма смисъл да опитваме повече. В случая рекурентното уравнение е достатъчно просто, така че закономерност се появи. Очевидно е, че ако продължим по същия начин, след краен брой развивания ще достигнем до

$$T(n) = T(0) + 1 + 2 + \dots + (n-2) + (n-1) + n \quad (3.20)$$

В случая това наистина е очевидно, защото аргументът намалява с единица и рано или късно ще стане нула, а събираемите вдясно, ако се пишат систематично, започват от числото, с единица по-голямо от аргумента на T вдясно и завършват с n , като нарастват с единица.

Сега прилагаме два известни факта. Първо, че $T(0) = 0$, което знаем от началното условие,

и второ, че $1 + 2 + \dots + n = \frac{n(n+1)}{2}$. Замествайки в (3.20), веднага получаваме

$$T(n) = \frac{n(n+1)}{2}$$

Това е и решението. □

Методът с развиването (на английски, *unfolding*) **не е формален**, защото можем да развием началния израз не повече от константен брой пъти и неизбежно използваме интуиция, за да напишем уравнение, в което участват началните условия. Например, при прехода от (3.19) към (3.20) ползваме интуиция. Така че, ако се иска формално решение, методът с развиването не може да се използва. Тогава трябва да ползваме някой от формалните методи като индукция (Подсекция 3.2.3), Мастър теоремата (Подсекция 3.2.4), теоремата на Акра-Bazzi (Подсекция 3.2.5) или метода с характеристичното уравнение (Подсекция 3.2.6). Измежду тези методи единственият универсален е индукцията, защото всички други искат рекурентното уравнение да има определена форма. Но преди да ползваме индукция, трябва да знаем какво да докажем по индукция и за тази цел може да ползваме неформалния метод на развиването (или с дървото на рекурсията, вижте Подсекция 3.2.2).

Методът с развиването има своите ограничения. Опитайте да решите рекурентното уравнение на числата на Фибоначи (3.1) с развиване. Практически е невъзможно да получите (3.14), ако използвате само здрав разум без никаква теория. **Теоретично**, има някаква ненулева вероятност след известен брой развивания на (3.1) да забележите, че закономерността е $\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$, но това иска наистина **нечовешка интуиция**, защото никъде в развиването няма да видите $\sqrt{5}$ и изобщо развиването ще съдържа само цели коефициенти.

Прилагайки метода на развиването, често ползваме следния резултат, ако търсим само асимптотиката на решението, а не точното решение. Забележете, че няма нужда k да е цяло число.

Лема 17

Нека $k > 0$ е константа. Итераторът $n \mapsto n - k$ се изпълнява $\Theta(n)$ пъти, докато n не стане $\Theta(1)$.

Вземаме за очевидно, че съществува реален интервал $[a, b]$, където $0 < a < b$ и $a \leq b - k$, такъв че за всяко достатъчно голямо n съществува $c \in [a, b]$, такова че итерирането

$$n \mapsto n - k \mapsto n - 2k \mapsto n - 3k \mapsto \dots$$

завършва в c . С други думи, че итерирането е

$$\underbrace{n \mapsto n - k \mapsto n - 2k \mapsto n - 3k \mapsto \dots \mapsto n - (m-1)k}_{\text{извън } [a, b]} \mapsto \underbrace{n - mk}_{\text{в } [a, b]}$$

за някое m , такова че $n - mk \leq b$, но $n - (m-1) \cdot k > b$.

Твърди се, че тове итериране се извършва $\Theta(n)$ пъти. Но това следва директно от очевидния факт, че итерирането се извършва точно m пъти, а

$$\begin{aligned} n &\leq km + b \\ n &\geq km - k + b \end{aligned}$$

така че $m \asymp n$. □

Следствие 15

Нека $k > 1$ е константа.

- ❶ Итераторът $n \mapsto \frac{n}{k}$ се изпълнява $\Theta(\lg n)$ пъти, докато n не стане $\Theta(1)$.
- ❷ Итераторът $n \mapsto \sqrt[k]{n}$ се изпълнява $\Theta(\lg \lg n)$ пъти, докато n не стане $\Theta(1)$.

Ще докажем ❶. Разглеждаме итерирането

$$n \mapsto \frac{n}{k} \mapsto \frac{n}{k^2} \mapsto \frac{n}{k^3} \mapsto \dots \mapsto c$$

където c е елемент от предварително избран, подходящ за k , реален интервал, какъвто беше $[a, b]$ в доказателството на Лема 17. Нека $\ell = \log_k n$. Заместваме n с k^ℓ . Тогава итерирането става

$$k^\ell \mapsto \frac{k^\ell}{k} \mapsto \frac{k^\ell}{k^2} \mapsto \frac{k^\ell}{k^3} \mapsto \dots \mapsto c$$

което е същото като

$$k^\ell \mapsto k^{\ell-1} \mapsto k^{\ell-2} \mapsto k^{\ell-3} \mapsto \dots \mapsto c$$

Сега прилагаме Лема 17 за степенните показатели и заключаваме, че итерирането се изпълнява $\Theta(\ell)$ пъти. Но знаем, че $\ell = \log_k n$, така че изпълнението става $\Theta(\lg n)$ пъти.

❷ може да се докаже чрез още едно прилагане на същата идея, но може и директно. Разглеждаме итерирането

$$n \mapsto \sqrt[k]{n} \mapsto \sqrt[k]{\sqrt[k]{n}} \mapsto \sqrt[k]{\sqrt[k]{\sqrt[k]{n}}} \mapsto \dots \mapsto c$$

където c е елемент от предварително избран, подходящ за k , реален интервал. Нека $\ell = \log_k (\log_k (n))$. Заместваме n с k^{k^ℓ} . Тогава итерирането става

$$k^{k^\ell} \mapsto \sqrt[k]{k^{k^\ell}} \mapsto \sqrt[k]{\sqrt[k]{k^{k^\ell}}} \mapsto \sqrt[k]{\sqrt[k]{\sqrt[k]{k^{k^\ell}}}} \mapsto \dots \mapsto c$$

което е същото като

$$k^{k^\ell} \mapsto k^{k^{\ell-1}} \mapsto k^{k^{\ell-2}} \mapsto k^{k^{\ell-3}} \mapsto \dots \mapsto c$$

Сега прилагаме Лема 17 за степенните показатели на степенните показатели и заключаваме, че итерирането се изпълнява $\Theta(\ell)$ пъти. Но знаем, че $\ell = \log_k (\log_k (n))$, така че изпълнението става $\Theta(\lg \lg n)$ пъти. $\square$

Следствие 16

Лема 17 и Следствие 15 ни дават директно следните резултати.

- Уравнението $T(n) = T(n-1) + 1$ има решение $T(n) \asymp \Theta(n)$.
- Уравнението $T(n) = T\left(\frac{n}{2}\right) + 1$ има решение $T(n) \asymp \Theta(\lg n)$.
- Уравнението $T(n) = T(\sqrt{n}) + 1$ има решение $T(n) \asymp \Theta(\lg \lg n)$.

Ако съобразим, че току-що изведените резултати са за височините на дървета с разклоненост 1, а при умножение с 2 на функцията вдясно същите дървета стават свършени двоични дървета със същите съответни височини, имаме следното.

- Уравнението $T(n) = 2T(n-1) + 1$ има решение $T(n) \asymp \Theta(2^n)$.
- Уравнението $T(n) = 2T\left(\frac{n}{2}\right) + 1$ има решение $T(n) \asymp \Theta(n)$.
- Уравнението $T(n) = 2T(\sqrt{n}) + 1$ има решение $T(n) \asymp \Theta(\lg n)$.

Ето един значително по-сложен пример за прилагане на метода с развиването.

Задача 25

Решете уравнението

$$T(n) = n^2 T\left(\frac{n}{2}\right) + 1$$

чрез развиване.

Решение: В сила е

$$\begin{aligned}
 T(n) &= n^2 T\left(\frac{n}{2}\right) + 1 \\
 &= n^2 \left(\frac{n^2}{4} T\left(\frac{n}{4}\right) + 1 \right) + 1 \\
 &= \frac{n^4}{2^2} T\left(\frac{n}{2^2}\right) + n^2 + 1 \\
 &= \frac{n^4}{2^2} \left(\frac{n^2}{2^4} T\left(\frac{n}{2^3}\right) + 1 \right) + n^2 + 1 \\
 &= \frac{n^6}{2^6} T\left(\frac{n}{2^3}\right) + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^6}{2^6} \left(\frac{n^2}{2^6} T\left(\frac{n}{2^4}\right) + 1 \right) + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^8}{2^{12}} T\left(\frac{n}{2^4}\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^8}{2^{12}} \left(\frac{n^2}{2^8} T\left(\frac{n}{2^5}\right) + 1 \right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
 &= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + \frac{n^2}{2^0} + \frac{n^0}{2^0} \\
 &= \frac{n^{10}}{2^{5 \cdot 4}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{4 \cdot 3}} + \frac{n^6}{2^{3 \cdot 2}} + \frac{n^4}{2^{2 \cdot 1}} + \frac{n^2}{2^{1 \cdot 0}} + \frac{n^0}{2^{0 \cdot (-1)}} \\
 &\dots \\
 &= \underbrace{\frac{n^{2i}}{2^{i(i-1)}} T\left(\frac{n}{2^i}\right)}_A + \underbrace{\frac{n^{2(i-1)}}{2^{(i-1)(i-2)}} + \frac{n^{2(i-2)}}{2^{(i-2)(i-3)}} + \dots + \frac{n^8}{2^{4 \cdot 3}} + \frac{n^6}{2^{3 \cdot 2}} + \frac{n^4}{2^{2 \cdot 1}} + \frac{n^2}{2^{1 \cdot 0}} + \frac{n^0}{2^{0 \cdot (-1)}}}_{B: \text{сума с } i \text{ събираеми}}
 \end{aligned}$$

Полученият израз е общият вид на дясната страна в процеса на развиването; с други думи, дясната страна след $i + 1$ развивания. Максималната стойност на i е $\lg n$ съгласно Следствие 15. Изразът вдясно е сума от два израза, които означаваме с A и B . Първо ще пресметнем A с $\lg n$ наместо i , като имаме предвид, че $T(1) = \Theta(1)$.

$$A = \frac{n^{2 \lg n}}{2^{\lg^2 n - \lg n}} T(1) \asymp \frac{2^{\lg n} n^{2 \lg n}}{2^{\lg^2 n}} = \frac{n \cdot n^{2 \lg n}}{2^{\lg^2 n}}$$

Но

$$2^{\lg^2 n} = 2^{\lg n \cdot \lg n} = 2^{\lg(n^{\lg n})} = n^{\lg n} \quad (3.21)$$

Тогава

$$A \asymp \frac{n \cdot n^{2 \lg n}}{n^{\lg n}} \asymp n^{1 + \lg n} \quad (3.22)$$

Сега да разгледаме В. Очевидно, можем да го представим като сума по следния начин:

$$\begin{aligned}
 B &= \sum_{j=1}^{\lg n} \frac{n^{2((\lg n)-j)}}{2^{((\lg n)-j)((\lg n)-j-1)}} \\
 &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{n^{2 \lg n}}{2^{(\lg^2 n - j \lg n - j \lg n + j^2 - \lg n + j)}} \\
 &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{(n^{2 \lg n}) (2^{2j \lg n}) (2^{\lg n})}{(2^{\lg^2 n}) (2^{j^2+j})} \quad (3.23)
 \end{aligned}$$

Но

$$2^{2j \lg n} = 2^{\lg(n^{2j})} = n^{2j} \quad (3.24)$$

и

$$2^{\lg n} = n \quad (3.25)$$

Прилагаме (3.21), (3.24) и (3.25) върху (3.23) и получаваме

$$\begin{aligned}
 B &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{(n^{2 \lg n}) (n^{2j}) (n)}{(n^{\lg n}) (2^{j^2+j})} \\
 &= \sum_{j=1}^{\lg n} \frac{n^{1+\lg n}}{2^{j^2+j}} \\
 &= (n^{1+\lg n}) \sum_{j=1}^{\lg n} \frac{1}{2^{j^2+j}} \\
 &\leq (n^{1+\lg n}) \sum_{j=1}^{\infty} \frac{1}{2^{j^2+j}} \quad // \text{ тъй като знаем, че } \sum_{j=1}^{\infty} \frac{1}{2^j} = 1 \\
 &\leq n^{1+\lg n} \quad (3.26)
 \end{aligned}$$

От (3.22) и (3.26) следва, че

$$T(n) \asymp n^{1+\lg n}$$

□

Допълнение 19: Няколко рекурентни уравнения чрез развиване

Задача 26

Решете рекурентното уравнение

$$T(n) = T(n-2) + 2 \lg n$$

чрез развиване.

Решение Да развием уравнението:

$$\begin{aligned}
 T(n) &= T(n-2) + 2 \lg n \\
 &= T(n-4) + 2 \lg(n-2) + 2 \lg n \\
 &= T(n-6) + 2 \lg(n-4) + 2 \lg(n-2) + 2 \lg n \\
 &= \dots \\
 &= \Theta(1) + \dots + 2 \lg(n-4) + 2 \lg(n-2) + 2 \lg n \\
 &\asymp \lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \dots + \lg 3^2 + \lg 1^2 \\
 &= \lg(n^2 \cdot (n-2)^2 \cdot (n-4)^2 \cdot \dots \cdot 3^2 \cdot 1) \\
 &= \lg \underbrace{(n \cdot n \cdot (n-2) \cdot (n-2) \cdot (n-4) \cdot (n-4) \cdot \dots \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 1)}_{n \text{ множителя}}
 \end{aligned}$$

Нека

$$\begin{aligned}
 T(n) &= \lg \underbrace{(n \cdot n \cdot (n-2) \cdot (n-2) \cdot (n-4) \cdot (n-4) \cdot \dots \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 1)}_{n \text{ множителя}} \\
 X(n) &= \lg \underbrace{(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1)}_{n \text{ множителя}} \\
 Y(n) &= \lg \underbrace{((n+1) \cdot n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 4 \cdot 3 \cdot 2)}_{n \text{ множителя}}
 \end{aligned}$$

Очевидно

$$X(n) \leq T(n) \leq Y(n)$$

Но $X(n) = \lg n!$ и $Y(n) = \lg(n+1)!$. Тогава $X(n) \asymp n \lg n$ и $Y(n) \asymp n \lg n$. Следователно, $T(n) \asymp n \lg n$. $\square$

Задача 27

Решете уравнението

$$T(n) = \frac{n}{n+1} T(n-1) + 1$$

чрез развиване.

Решение В сила е

$$\begin{aligned}
 T(n) &= \frac{n}{n+1}T(n-1) + 1 \\
 &= \frac{n}{n+1} \left(\frac{n-1}{n}T(n-2) + 1 \right) + 1 \\
 &= \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1} + 1 \\
 &= \frac{n-1}{n+1} \left(\frac{n-2}{n-1}T(n-3) + 1 \right) + \frac{n}{n+1} + 1 \\
 &= \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
 &= \frac{n-2}{n+1} \left(\frac{n-3}{n-2}T(n-4) + 1 \right) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
 &= \frac{n-3}{n+1}T(n-4) + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1
 \end{aligned} \tag{3.27}$$

Ако продължим така до $T(1)$, развиваме (3.27) до

$$\begin{aligned}
 T(n) &= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \dots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
 &= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \dots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + \frac{n+1}{n+1} \\
 &= \underbrace{\frac{\Theta(1)}{n+1}}_{O(1)} + \frac{1}{n+1} \sum_{i=3}^{n+1} i \\
 &\asymp \frac{1}{n+1} \cdot \frac{(n-1)(n+4)}{2} \\
 &\asymp \Theta(n)
 \end{aligned}$$

И така, $T(n) \asymp n$. □

Задача 28

Решете рекурентното уравнение

$$T(n) = \frac{n}{n+1}T(n-1) + \lg n$$

чрез развиване.

Решение В сила е

$$\begin{aligned}
 & \frac{n}{n+1}T(n-1) + \lg n = \\
 & \frac{n}{n+1} \left(\frac{n-1}{n}T(n-2) + \lg(n-1) \right) + \lg n = \\
 & \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1} \lg(n-1) + \lg n = \\
 & \frac{n-1}{n+1} \left(\frac{n-2}{n-1}T(n-3) + \lg(n-2) \right) + \frac{n}{n+1} \lg(n-1) + \lg n = \\
 & \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1} \lg(n-2) + \frac{n}{n+1} \lg(n-1) + \lg n = \\
 & \dots = \\
 & \underbrace{\frac{2}{n+1}T(1)}_{O(1)} + \frac{3}{n+1} \lg 2 + \frac{4}{n+1} \lg 3 + \dots + \frac{n-1}{n+1} \lg(n-2) + \frac{n}{n+1} \lg(n-1) + \lg n \asymp \\
 & \frac{(2+1) \lg 2}{n+1} + \frac{(3+1) \lg 3}{n+1} + \dots + \frac{(n-1+1) \lg(n-1)}{n+1} + \frac{(n+1) \lg n}{n+1} = \\
 & \frac{1}{n+1} \left(\sum_{k=2}^n k \lg k + \sum_{k=2}^n \lg k \right)
 \end{aligned}$$

Но $\sum_{k=2}^n k \lg k \asymp n^2 \lg n$ съгласно Теорема 34, а $\sum_{k=2}^n \lg k = \lg n! \asymp n \lg n$ съгласно Теорема 33, откъдето $\frac{1}{n+1} (\sum_{k=2}^n k \lg k + \sum_{k=2}^n \lg k) \asymp n \lg n$, така че $T(n) \asymp n \lg n$. $\square$

Задача 29

Решете рекурентното уравнение

$$T(n) = nT(n-1) + 1$$

чрез развиване.

Решение

$$\begin{aligned}
T(n) &= nT(n-1) + 1 \\
&= n((n-1)T(n-2) + 1) + 1 \\
&= n(n-1)T(n-2) + n + 1 \\
&= n(n-1)((n-2)T(n-3) + 1) + n + 1 \\
&= n(n-1)(n-2)T(n-3) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)((n-3)T(n-4) + 1) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2) + n(n-1) + n + 1 \\
&\dots \\
&= \frac{n!}{(n-i)!} T(n-i) + \frac{n!}{(n-i+1)!} + \frac{n!}{(n-i+2)!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&\dots \\
&= \frac{n!}{1!} T(1) + \frac{n!}{2!} + \frac{n!}{3!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&= n! \times \underbrace{\left(\frac{\Theta(1)}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{(n-1)!} + \frac{1}{n!} \right)}_A
\end{aligned}$$

Твърдим, че A е ограничена отгоре от константа. Наистина, редът $\sum_{i=1}^{\infty} \frac{1}{i!}$, тъй като геометричният ред $\sum_{i=1}^{\infty} \frac{1}{2^i}$ е сходящ, а $i! > 2^i$ за всяко $i > 3$. И така, $T(n) \asymp n!$. $\square$

Задача 30

Решете рекурентното уравнение

$$T(n) = nT(n-1) + 2^n$$

чрез развиване.

Решение

$$\begin{aligned}
T(n) &= nT(n-1) + 2^n \\
&= n((n-1)T(n-2) + 2^{n-1}) + 2^n \\
&= n(n-1)T(n-2) + n2^{n-1} + 2^n \\
&= n(n-1)((n-2)T(n-3) + 2^{n-2}) + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)T(n-3) + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)((n-3)T(n-4) + 2^{n-3}) + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2)2^{n-3} + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&\dots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!2^{n-i+1}}{(n-i+1)!} + \frac{n!2^{n-i+2}}{(n-i+2)!} + \dots + \frac{n!2^{n-1}}{(n-1)!} + \frac{n!2^n}{n!} \\
&\dots \\
&= \frac{n!}{1!}T(1) + \frac{n!2^2}{2!} + \frac{n!2^3}{3!} + \dots + \frac{n!2^{n-1}}{(n-1)!} + \frac{n!2^n}{n!} \\
&= n! \times \left(\Theta(1) + \sum_{k=2}^n \frac{2^k}{k!} \right)
\end{aligned}$$

Но редът $\sum_{k=1}^{\infty} \frac{2^k}{k!}$ е сходящ, тъй като, по критерия на d'Alambert:

$$\frac{\frac{2^{k+1}}{(k+1)!}}{\frac{2^k}{k!}} = \frac{2}{k+1} < \epsilon, \text{ за всяко } \epsilon > 0 \text{ и за всяко достатъчно голямо } k.$$

Следва, че $\sum_{k=2}^n \frac{2^k}{k!}$ е ограничена отгоре от константа. Следва, че $T(n) \asymp n!$. □

Задача 31

Решете рекурентното уравнение

$$T(n) = 2T(n-1) + n!$$

чрез развиване.

Решение

$$\begin{aligned}
T(n) &= 2T(n-1) + n! \\
&= 2(2T(n-2) + (n-1)!) + n! \\
&= 4T(n-2) + 2(n-1)! + n! \\
&= 4(2T(n-3) + (n-2)!) + 2(n-1)! + n! \\
&= 8T(n-3) + 4(n-2)! + 2(n-1)! + n! & (3.28) \\
&= 8(2T(n-4) + (n-3)!) + 4(n-2)! + 2(n-1)! + n! \\
&= 16T(n-4) + 8(n-3)! + 4(n-2)! + 2(n-1)! + n! \\
&\dots & (3.29) \\
&= \underbrace{2^{n-1}T(1) + 2^{n-2}2! + 2^{n-3}3! + \dots + 2^2(n-2)! + 2(n-1)!}_{A} + n! & (3.30)
\end{aligned}$$

Ще докажем, че асимптотиката на дясната страна е $\Theta(n!)$. За целта е достатъчно да докажем, че е $A = O(n!)$, понеже другото събираемо е $n!$. Това може да се докаже чрез Лема 18.

Лема 18

$$\forall n \geq 2 : 2(n-1)! + 2^2(n-2)! + 2^3(n-3)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! < 2n!.$$

Доказателство: По индукция по n .

База: $n = 2$. Лявата страна е $2(2-1)! = 2$. Дясната страна е $2 \cdot 2! = 4$. ✓

Индуктивно предположение: Да допуснем, че за някое $n \geq 2$ е вярно, че

$$2(n-1)! + 2^2(n-2)! + 2^3(n-3)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! < 2n!$$

Индуктивна стъпка: Ще покажем, че

$$2n! + 2^2(n-1)! + 2^3(n-2)! + \dots + 2^{n-2}3! + 2^{n-1}2! + 2^n1! < 2(n+1)!$$

Но това е същото като

$$2n! + 2(2(n-1)! + 2^2(n-2)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!) < 2 \cdot (n+1) \cdot n!$$

което е същото като

$$\begin{aligned} 2n! + 2(2(n-1)! + 2^2(n-2)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!) &< 2 \cdot n \cdot n! + 2n! \leftrightarrow \\ 2(2(n-1)! + 2^2(n-2)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!) &< 2 \cdot n \cdot n! \leftrightarrow \\ 2(n-1)! + 2^2(n-2)! + \dots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! &< n \cdot n! \end{aligned}$$

Това следва веднага от индуктивното предположение и допускането, че $n \geq 2$. □

Има алтернативно, по-просто и елегантно доказателство, посочено на автора на лекционните записки от Добромир Кралчев. Представяме израза от (3.30) така:

$$n! \left(1 + \underbrace{\frac{2}{n} + \frac{2^2}{n(n-1)} + \frac{2^3}{n(n-1)(n-2)} + \dots + \frac{2^{n-2}}{n(n-1) \cdot \dots \cdot 3} + \frac{2^{n-1}}{n(n-1) \cdot \dots \cdot 2}}_B \right)$$

Ако покажем, че $B = O(1)$, то ние сме готови с доказателството. Ще покажем, че $B = O(1)$. Но във всяко събираемо на B , множителите-двойки в числителя са точно колкото множителите в знаменателя, поради което можем да напишем B така:

$$\frac{1}{\frac{n}{2}} + \frac{1}{\frac{n}{2} \cdot \frac{n-1}{2}} + \underbrace{\frac{1}{\frac{n}{2} \cdot \frac{n-1}{2} \cdot \frac{n-2}{2}} + \dots + \frac{1}{\frac{n}{2} \cdot \frac{n-1}{2} \cdot \dots \cdot \frac{2}{2}}}_C$$

тоест, $B = \frac{1}{\frac{n}{2}} + \frac{1}{\frac{n}{2} \cdot \frac{n-1}{2}} + C$. Всяко събираемо на C е $O\left(\frac{1}{n^3}\right)$ и C има $O(n)$ събираеми, поради което $C = O\left(\frac{1}{n^2}\right)$, от където веднага следва, че $B = O(1)$.

Така или иначе, показахме, че за A от (3.30) е вярно, че $A = O(n!)$. От това следва, че $T(n) \asymp n!$. □

3.2.2 Чрез дървото на рекурсията

При този метод си представяме работата на рекурсивен алгоритъм, чиято сложност по време се описва от рекурентното уравнение, което изследваме. Представяме си дървото на рекурсията на този алгоритъм, съобразяваме колко работа се извършва във всеки връх от дървото **извън рекурсивните викания** и сумираме тези количества по нива в дървото. След което търсим закономерност в сумите по нивата.

Методът с дървото е прилича на метода с развиването в това, че не е формален. За да даде резултати, трябва човекът, който го ползва, да прояви достатъчно добра интуиция, за да види закономерност. Сега ще илюстрираме метода с дървото на рекурсията с два примера.

Лесно уравнение: $T(n) = 2T(n/2) + n$. Ще решим рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad (3.31)$$

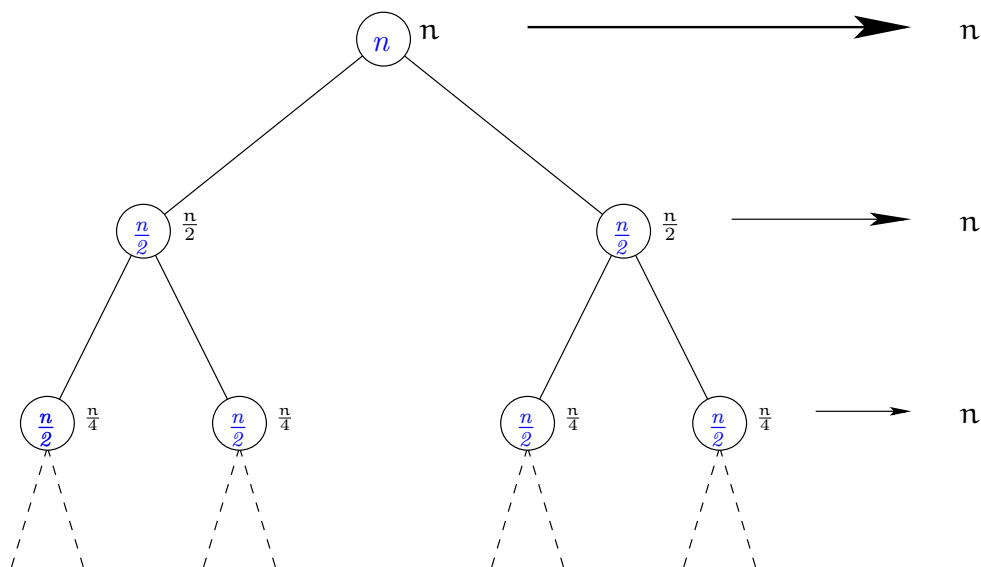
Това уравнение може да бъде решено с Мастър Теоремата (Теорема 41), но тук с учебна цел ще го решим с метода с дървото на рекурсията.

Представяме си рекурсивен алгоритъм, който, разбива входа с размер n на две части, всяка с размер $n/2$ (игнорираме това, че n може да не е четно), ако n е достатъчно голямо, вика себе си върху всяка от тези части и върши линейна работа извън рекурсивните викания. Допускайки, че n е достатъчно голямо, при вход с размер n се правят двете рекурсивни викания и се върши допълнителна работа n ; във всяко от тези две викания се правят две рекурсивни викания върху части с размер $n/4$ и се върши допълнителна работа $n/2$; във всяко от тези четири викания се правят две рекурсивни викания върху части с размер $n/8$ и се върши допълнителна работа $n/4$; и така нататък.

Най-горната част на дървото на рекурсията е изобразена на Фигура 3.1. Всеки изобразен връх представлява едно изпълнение на алгоритъма, което прави рекурсивни викания. Листата, които отговарят на спирачката на рекурсията, не са нарисувани. Във всеки връх в синьо е написан размерът на входа за това изпълнение на алгоритъма, а до всеки връх в черно е написана нехомогенната част за това изпълнение на алгоритъма (работата, която алгоритъмът извършва извън двете рекурсивните викания). Върховете от едно и също ниво в дървото са нарисувани на едно и също ниво на фигурата.

Ясно е, че дървото е свършено двоично дърво (Определение 53). Височината очевидно е $\Theta(\lg n)$, понеже делим големината на входа на 2, докато не достигнем началните условия. Началните условия не са споменати, така че не може да се каже точно каква е височината, но можем да вземем височина точно $\lg n + 1$ за простота. Това не променя асимптотиката. Тъй като дървото е свършено, нивата са пълни, като горните $\lg n$ нива се състоят от вътрешните върхове, а най-долното ниво е от листата.

Сумата от нехомогенните части във всяко ниво от вътрешни върхове е точно n и това може да се докаже тривиално по индукция. Игнорираме нивото от листата. Заключаваме, че в нивата от вътрешните върхове сумата от нехомогенните части е $n \lg n$. От това следва, че $T(n) \asymp n \lg n$. Това е и решението на (3.31), получено с метода с дървото на рекурсията.



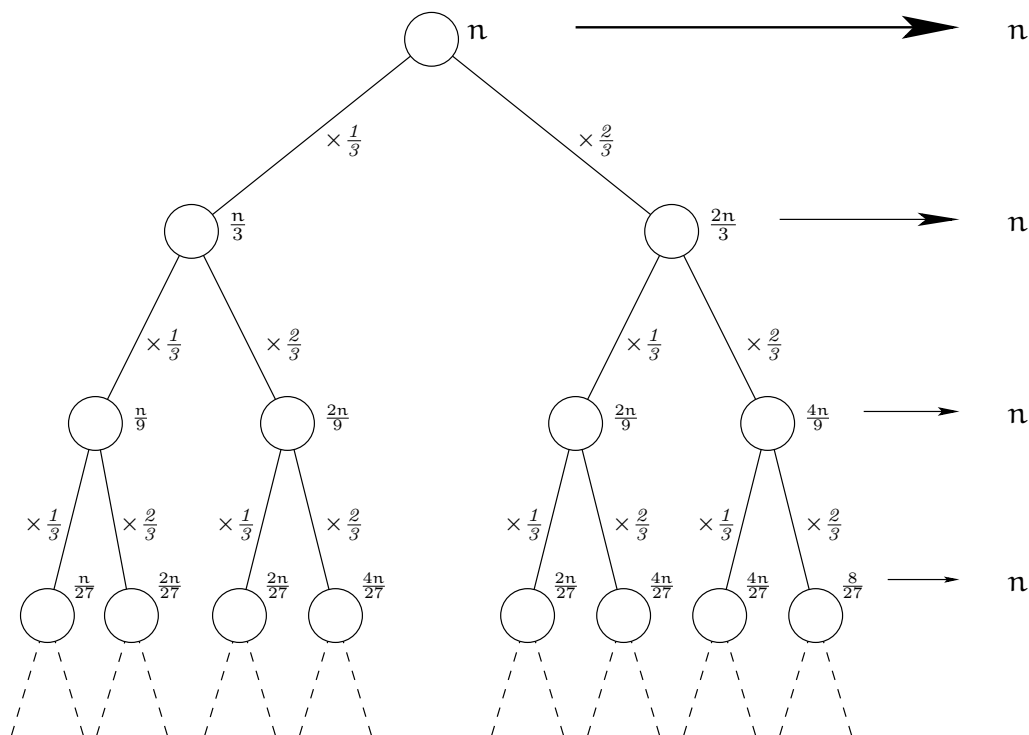
Фигура 3.1: Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$. За всяко от изображените нива, сумата от стойностите на върховете в него е n .

По-трудно уравнение: $T(n) = T(n/3) + T(2n/3) + n$. Ще решим рекурентното уравнение

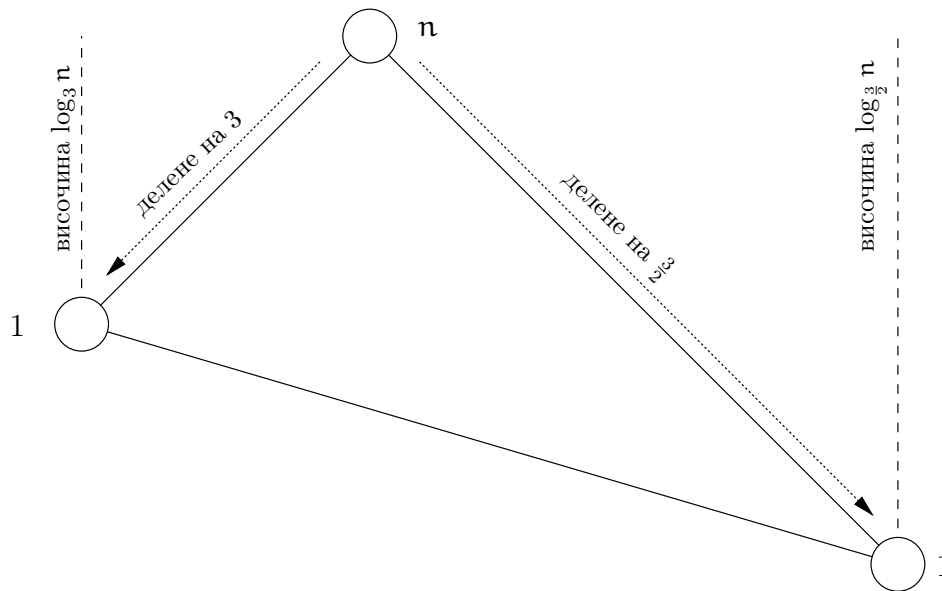
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \quad (3.32)$$

използвайки метода с дървото на рекурсията. Това уравнение не може да бъде решено с Мастър Теоремата (Теорема 41).

Най-горната част на дървото на рекурсията е изобразена на Фигура 3.2.



Фигура 3.2: Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. За всяко от изображените нива, сумата от стойностите на върховете в него е n .

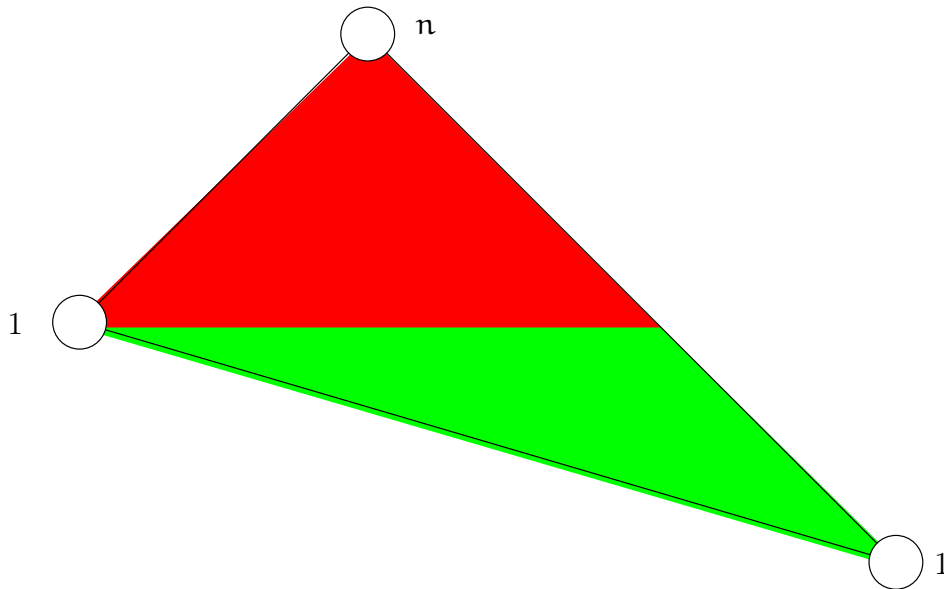


Фигура 3.3: Схема на дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. Най-лявото листо е най-близо до корена, а най-дясното листо е най-отдалечено от корена. Височината на най-лявото листо е $\log_3 n$, а на най-дясното е $\log_{\frac{3}{2}} n$.

Всеки връх, изобразен на Фигура 3.2, представлява едно изпълнение на алгоритъма, което прави рекурсивни викания. Листата, които отговарят на спирачката на рекурсията, не са нарисувани. Ребрата са маркирани с множителите, с които входът намалява. До всеки връх е написана нехомогенната част за това изпълнение на алгоритъма: това е работата, която алгоритъмът извършва извън двете рекурсивните викания. Върховете от едно и също ниво в дървото са нарисувани на едно и също ниво на фигурата.

Коренът е първоначалното изпълнение. В него алгоритъмът извършва работа n извън двете рекурсивни викания, затова вдясно от корена е написано “ n ”. Едното рекурсивно викане се върши върху вход, който е с големина $\frac{1}{3}$ от големината на началния, а другото върху вход, който е с големина $\frac{2}{3}$ от големината на началния. Затова двете ребра, свързващи корена с лявото и дясното дете, са маркирани съответно с “ $\times \frac{1}{3}$ ” и “ $\times \frac{2}{3}$ ”. До всяко от двете деца е написана работата, която се извършва от съответното извикване на алгоритъма извън рекурсивните викания. Тази работа е същата като големината на входа за съответното викане, затова записаните стойности са $\frac{n}{3}$ и $\frac{2n}{3}$. И така нататък.

Да мислим за дървото като за наредено двоично дърво (различаваме лявото от дясното дете на всеки вътрешен връх; вижте Подсекция 5.1.1). Това, че разглеждаме дървото като наредено дърво, има значение единствено за забелязването на закономерност. Нехомогенната част, асоциирана с лявото дете на даден връх, е нехомогенната част, асоциирана с родителя, умножена по $\frac{1}{3}$. Нехомогенната част, асоциирана с дясното дете, е нехомогенната част, асоциирана с родителя, умножена по $\frac{2}{3}$. С други думи, наляво има деление на 3, а надясно, деление на $\frac{3}{2}$. Листата на дървото са очевидно на различни нива, така че ни трябва някаква систематика в описването на дървото, за да видим закономерност. По начина, по който разглеждаме дървото, най-близкото до корена листо е крайт на пътя, който започва от корена и във всеки вътрешен връх продължава наляво, а най-отдалеченото от корена листо е крайт на пътя, който започва от корена и във всеки вътрешен връх продължава надясно. Образно казано, дървото е “криво” и има фôрмата, показана на Фигура 3.3. Допускаме, че началното условие е $n = 1$, така че листата са асоциирани с единици. Дървото е нарисувано по начин, който подсказва, че с деление на 3 достига 1 по-бързо, отколкото с деление на $\frac{3}{2}$.

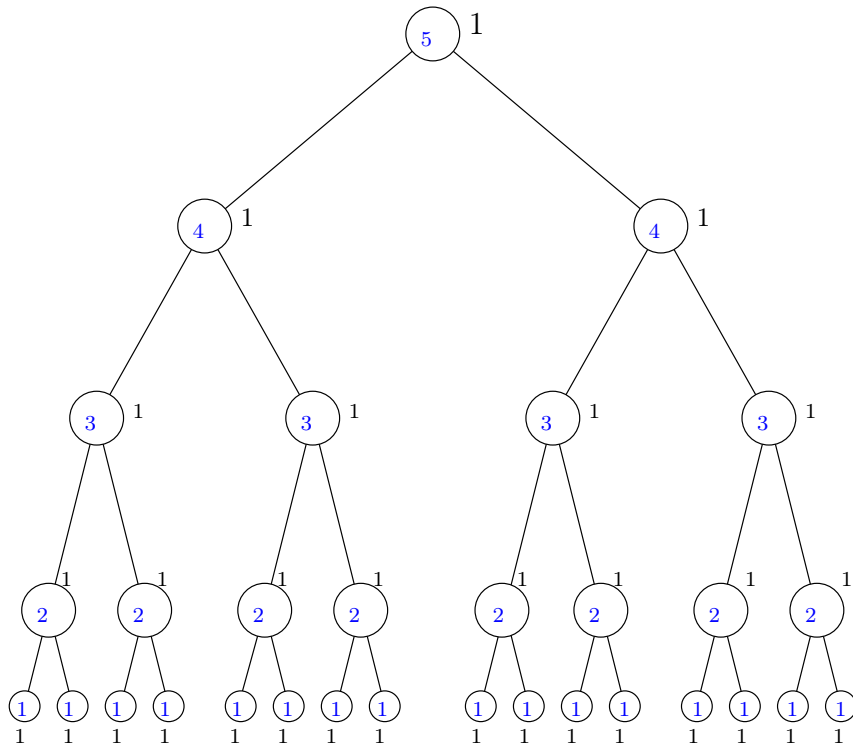


Фигура 3.4: Друга схема на дървото при $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. Червеното поддърво е свършено, понеже нивата му са пълни. Следващите нива, нарисувани в зелено, са непълни.

Тривиално е да се докаже, че нива с номер от 0 (коренът) до $\log_3 n$ (най-лявото дете) са пълни[†] и във всяко от тях, сумата от стойностите е n . Фигура 3.4 илюстрира пълните нива и останалите: пълните нива са в червено, останалите са в зелено. Да намерим асимптотиката на решението на рекурентното уравнение е същото като да оценим сумата от асоциираните стойности по всички върхове на дървото. За червеното поддърво (Фигура 3.4), тази сума е височината, умножена по n , защото по всяко негово ниво, сумата е n . Височината на червеното поддърво е $\log_3 n$, така че сумата на стойностите по неговите върхове е $n \log_3 n$. Това е долна граница за сумата по всички върхове на дървото. Но очевидно $n \log_3 n$ е горна граница за същата сума, понеже $\log_3 n$ е височината на цялото дърво и никое ниво няма сума повече от n . Но $n \log_3 n \asymp n \lg n$ и $n \log_3 n \asymp n \lg n$. Щом $T(n) \leq n \lg n$ и $T(n) \geq n \lg n$, следва, че $T(n) \asymp n \lg n$.

Методът с дървото на рекурсията в обобщение. От тези примери стана ясно, че методът с дървото на рекурсията е подобен на метода с развиването, само че тук групираме събираемите по различен начин, търсейки закономерност. В примерите, които видяхме, изчислихме сумата от нехомогенните части (припомнете си Наблюдение 25) във всички извиквания. И по-точно казано, не ги изчислихме, а само ги оценихме като асимптотика. Истинската работа, която върши алгоритъмът, е пропорционална на сумата от нехомогенните части и броя на върховете на дървото. Но само сумата от нехомогенните части дава асимптотиката, така че се фокусираме върху нея. Ерго, когато решаваме рекурентно уравнение, което отразява работата на рекурсивен алгоритъм и използваме метода с дървото на рекурсията, напълно достатъчно е да намерим асимптотиката на сумата от нехомогенните части. Подобно на развиването, този метод също е неформален и приложимостта му зависи от интуицията на човека, който го прилага. Ако човек не открие някаква закономерност в сумите, методът не е приложим.

[†]В смисъл, че в ниво k има 2^k върха.



Фигура 3.5: Дървото на рекурсията на $T(1) = 1$, $T(n) = 2T(n-1) + 1$, когато $n = 5$. Във върховете, в син цвят са написани съответните стойности на n .

Еквивалентни рекурентни уравнения може да имат съвсем различни дървета на рекурсията. Да разгледаме рекурентното уравнение

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1) + 1, \quad n \geq 2 \end{aligned} \quad (3.33)$$

Съответното дърво на рекурсията е показано на Фигура 3.5 при $n = 5$. На ниво 0 сумата е 1, на ниво 1 сумата е 2, и така нататък, на ниво 4 сумата е 16. Сумата от стойностите, асоциирани с върховете, е 31. Лесно се вижда, а и лесно се доказва формално, че решението на (3.33) е $T(n) = 2^n$.

Сега да преобразуваме (3.33) ето така:

$$T(n) = 2T(n-1) + 1 \leftrightarrow T(n) = T(n-1) + T(n-1) + 1$$

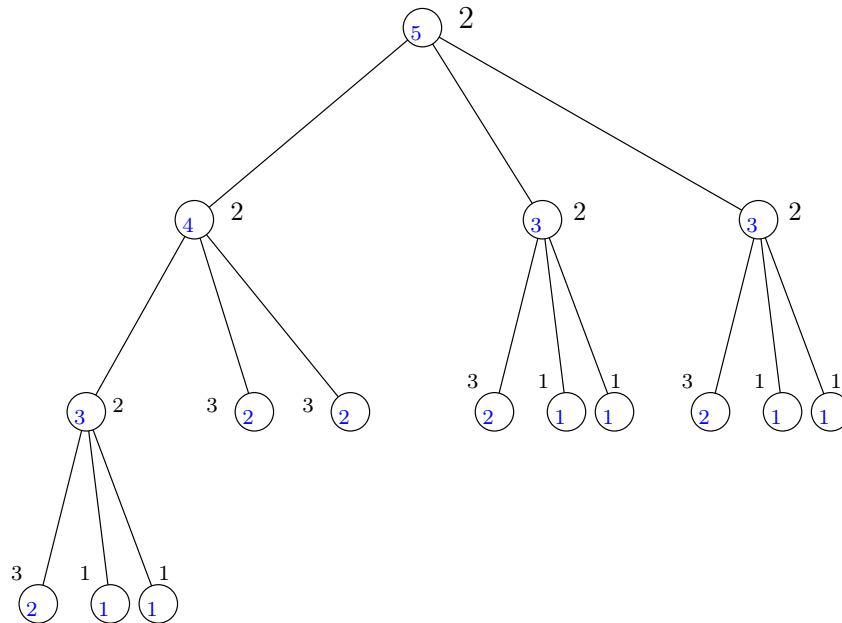
$$\text{Очевидно } T(n-1) = 2T(n-2) + 1$$

$$\text{Тогава } T(n) = T(n-1) + 2T(n-2) + 1 + 1 = T(n-1) + 2T(n-2) + 2$$

Тъй като в случая искаме точно рекурентно уравнение с началните условия, налага се да добавим и начално условия $T(2) = 3$. Рекурентното уравнение става:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 3 \\ T(n) &= T(n-1) + 2T(n-2) + 2, \quad n \geq 3 \end{aligned} \quad (3.34)$$

Рекурентни уравнение (3.33) и (3.34) са еквивалентни. Но техните съответни дървета на рекурсията нито са еднакви, нито са изоморфни. Фигура 3.6 показва дървото на рекурсията,



Фигура 3.6: Дървото на рекурсията на $T(1) = 1$, $T(2) = 3$, $T(n) = T(n-1) + 2T(n-2) + 2$. Във върховете, в син цвят са написани съответните стойности на n .

съответстващо на (3.34), за $n = 5$. Дървото от Фигура 3.6 е неподходящо за откриване на закономерност както за прецизното решение $T(n) = 2^n - 1$, така и за асимптотичното решение $T(n) \asymp 2^n$.

3.2.3 По индукция

Ще демонстрираме доказването на асимптотиката на нарастването на функцията от дадено рекурентно уравнение с пример. Да разгледаме това рекурентно уравнение

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad (3.35)$$

Ще докажем, че $T(n) = \Theta(n \lg n)$ по индукция по n . За тази цел ще докажем поотделно, че $T(n) = O(n \lg n)$ и $T(n) = \Omega(n \lg n)$. Както вече споменахме на стр. 200, скобите $\lfloor \cdot \rfloor$ и $\lceil \cdot \rceil$ не може просто да се изпускат, ако решаваме рекурентно уравнение по индукция. Ние ще се освободим от $\lceil \cdot \rceil$, но с подходящи неравенства, а не чрез игнориране.

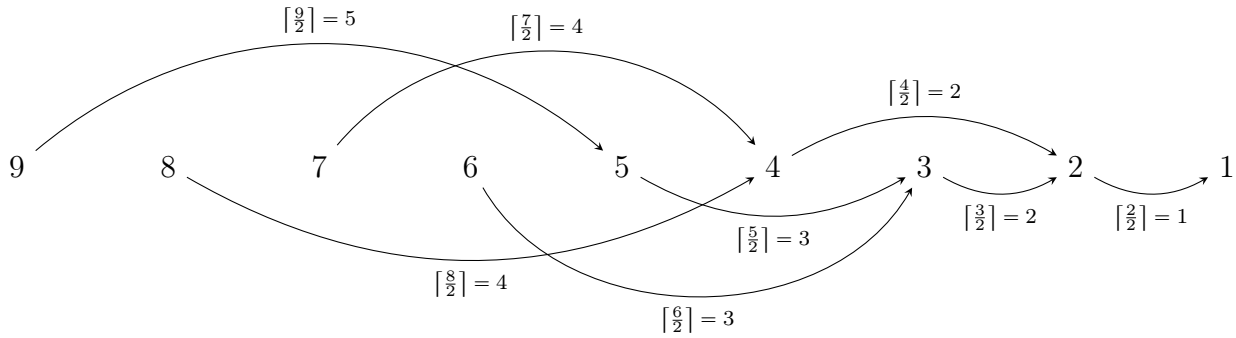
Част i: Ще докажем, че $T(n) = O(n \lg n)$. По дефиниция, това е същото като да докажем, че съществуват положителна константа c и положително n_0 , такова че за всяко $n \geq n_0$:

$$T(n) \leq cn \lg n \quad (3.36)$$

В (3.35) не са споменати начални условия. Близко до ума е да вземем начално условие $T(1) = a$, където $a \in \mathbb{R}^+$. Тогава n_0 е 1 и трябва да докажем, че съществува c , такова че за всяко $n \geq 1$ е изпълнено $T(n) \leq cn \lg n$. Базата е $n = 1$. Но има проблем! В базовия случай твърдението става $T(1) \leq c \cdot 1 \cdot \lg 1$, тоест, $a \leq 0$. Кое е невъзможно, понеже a е положителна константа.

Това не е провал на доказателството, а е несъществена техническа пречка, произтичаща от факта, че $\lg 1 = 0$. Измъкваме се така: вземаме друго начално условие. Най-лесно е да

вземем $T(2) = \alpha$. Стойността 2 на аргумента не е случайна – итераторът $n \mapsto \lceil \frac{n}{2} \rceil$ достига 2, стартирайки от произволно $n > 2$:



$T(2) = \alpha$ не е единствената възможност. Началните условия може да са тези: $T(3) = \alpha_1$ и $T(4) = \alpha_2$, въз основа на това, че двете числа 3 и 4 също не да бъдат прескочени от итератора $n \mapsto \lceil \frac{n}{2} \rceil$.

И така, като начално условие вземаме $T(2) = \alpha$. Ще докажем със силна индукция по n , че съществува $c > 0$, такова че за всяко $n \geq 2$ е изпълнено $T(n) \leq cn \lg n$.

Базата е за $n = 2$. В базата трябва да посочим положително c , такова че $T(2) \leq c \cdot 2 \cdot \lg 2$. Тоест $\alpha \leq 2c$, тоест $c \geq \frac{\alpha}{2}$.

Индуктивното предположение е, че за всяко $k \in \{2, \dots, n-1\}$:

$$T(k) \leq ck \lg k$$

В частност, при $k = \lceil \frac{n}{2} \rceil$, предположението е

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \quad (3.37)$$

Изпълняваме индуктивната стъпка. Ще покажем (3.36), използвайки индуктивното предположение (3.37). От (3.35) и (3.37) е в сила:

$$\begin{aligned} T(n) &\leq 2c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n && // \quad \forall n \geq 2: \left\lceil \frac{n}{2} \right\rceil \leq \left(\frac{n}{2} + 1 \right) \\ &\leq 2c \left(\frac{n}{2} + 1 \right) \lg \left\lceil \frac{n}{2} \right\rceil + n && // \quad \forall n \geq 2: \left\lceil \frac{n}{2} \right\rceil \leq \frac{3}{4}n \end{aligned} \quad (3.38)$$

$$\begin{aligned} &\leq 2c \left(\frac{n}{2} + 1 \right) \lg \left(\frac{3}{4}n \right) + n && (3.39) \\ &= c(n+2) \left(\lg n + \lg \frac{3}{4} \right) + n \\ &= cn \lg n + \textcolor{red}{cn \lg \frac{3}{4}} + 2c \lg n + 2c \lg \frac{3}{4} + n \end{aligned}$$

Иска се да докажем, че полученото $cn \lg n + \textcolor{red}{cn \lg \frac{3}{4}} + 2c \lg n + 2c \lg \frac{3}{4} + n$ не надхвърля $cn \lg n$, за някаква константа c . Това е вярно, ако $\textcolor{red}{cn \lg \frac{3}{4}} + 2c \lg n + 2c \lg \frac{3}{4} + n \leq 0$. Но

$$\textcolor{red}{cn \lg \frac{3}{4}} + 2c \lg n + 2c \lg \frac{3}{4} + n = \left(c \lg \frac{3}{4} + 1 \right) n + 2c \lg n + 2c \lg \frac{3}{4}$$

Асимптотиката на нарастването се определя от събираемото $(c \lg \frac{3}{4} + 1)n$. Ако константата $c \lg \frac{3}{4} + 1$ е отрицателна, тогава това събираемо е отрицателно за всички достатъчно големи стойности на n , откъдето веднага заключаваме, че

$$cn \lg n + cn \lg \frac{3}{4} + 2c \lg n + 2c \lg \frac{3}{4} + n \leq cn \lg n$$

за всички достатъчно големи стойности на n .

За да бъде изпълнено $c \lg \frac{3}{4} + 1 < 0$, трябва да е вярно, че $c > \frac{1}{\lg \frac{4}{3}}$. Числено, дясната страна е по-малка от 2.41. Следователно, всяко $c > 2.41$ “върши работа” за индуктивната стъпка.

Накрая си спомняме, че от базата имаме $c \geq \frac{a}{2}$, където a “идва” от началното условие $T(2) = a$, и $n_0 = 2$. Тогава за нашето доказателство “върши работа” всяко c , по-голямо от $\max\{\frac{a}{2}, 2.41\}$. В заключение, за всяко $c > \max\{\frac{a}{2}, 2.41\}$ и всяко $n \geq 2$ е вярно, че $T(n) \leq cn \lg n$. ✓

Наблюдение 26

В (3.38) ние замествахме $\lceil \frac{n}{2} \rceil$ с $\frac{3}{4}n$. Бихме могли да използваме всяко αn , стига да е изпълнено $\frac{1}{2} < \alpha < 1$.

- Защо трябва да е изпълнено $\frac{1}{2} < \alpha$? Ако това не е изпълнено, нямаме право да твърдим, че неравенството “ $\leq$ ” между (3.38) и (3.39) е в сила.
- А защо трябва да е изпълнено $\alpha < 1$? Да допуснем, че $\alpha = 1$; с други думи, замествахме $\lceil \frac{n}{2} \rceil$ с n . Тогава (3.39) става:

$$c(n+2)(\lg n) + n = cn \lg n + 2c \lg n + n$$

Очевидно, това е по-голямо от $cn \lg n$ за всички достатъчно големи n , така че нямаме доказателство.

Част ii: Ще докажем, че $T(n) = \Omega(n \lg n)$. По дефиниция, това е същото като да докажем, че съществуват положителна константа d и положително n_1 , такова че за всяко $n \geq n_1$:

$$T(n) \geq dn \lg n \quad (3.40)$$

И тук вземаме начално условие $T(2) = a$. Трябва да се докаже, че съществува d , такова че за всяко $n \geq 2$ е изпълнено $T(n) \geq dn \lg n$, като базата е $n = 2$. В базата трябва да посочим положително d , такова че $T(2) \geq d \cdot 2 \cdot \lg 2$, тоест $a \geq 2d$, тоест $d \leq \frac{a}{2}$.

Индуктивното предположение е, че за всяко $k \in \{2, \dots, n-1\}$:

$$T(k) \geq dk \lg k \quad (3.41)$$

В частност, при $k = \lceil \frac{n}{2} \rceil$, предположението е

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) \geq d \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \quad (3.42)$$

От (3.35) и (3.42):

$$\begin{aligned}
 T(n) &\geq 2d \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\
 &\geq 2d \left(\frac{n}{2} \right) \lg \left(\frac{n}{2} \right) + n \\
 &= dn(\lg n - 1) + n \\
 &= dn \lg n + (1 - d)n \\
 &\geq dn \lg n \quad // \text{ при условие, че } (1 - d)n \geq 0
 \end{aligned} \tag{3.43}$$

И така, за всяко d , такава че $0 < d \leq 1$, индуктивната стъпка е в сила.

И накрая си спомняме, че от базата имаме $d \leq \frac{a}{2}$ и $n_0 = 2$. Тогава за нашето доказателство “върши работа” всяко положително $d \leq \min \left\{ \frac{a}{2}, 1 \right\}$. В заключение, за всяко $d \leq \min \left\{ \frac{a}{2}, 1 \right\}$ и всяко $n \geq 2$ е вярно, че $T(n) \geq dn \lg n$. ✓

Това е краят на доказателството. □

Конвенция 9

Оттук насетне, когато решаваме рекурентни уравнения по индукция, за краткост и простота няма да разглеждаме базата. Убедение сме, че ако си направим труда, ще намерим начална стойност на аргумента n_0 и стойност на константата c_0 , такива че базата е в сила, и в крайна сметка твърдението е вярно по отношение на това n_0 и $\max \{c_0, c\}$ или $\min \{c_0, c\}$, в зависимост от това дали доказваме съответно O или Ω , където c е стойност на константата, такава че индуктивната стъпка е в сила.

Преди да видим още примери, две предупреждения за потенциални грешки или проблеми при използването на индукция за доказване на асимптотиката на рекурентни уравнения.

- Първо, доказателството трябва да се извърши спрямо константата, с която е започнало. Да разгледаме отново следното рекурентно уравнение:

$$T(n) = 2T(n - 1) + 1$$

Както вече знаем, решението е $T(n) = \Theta(2^n)$. Сега помислете, къде е грешката в следното доказателство, което “доказва”, че $T(n) = O(n)$. Ще “докажем”, че $T(n) \leq cn$ за някаква константа c . Индуктивното предположение е, че $T(n - 1) \leq c(n - 1)$. Тогава $T(n) \leq 2c(n - 1) + 1$. Тогава $T(n) \leq 2cn - 2c + 1$. Но последният израз казва, че $T(n)$ е ограничена отгоре от константа, умножена по n . Тогава $T(n) = O(n)$, което и искахме да докажем.

Къде е грешката? Грешката е в това, че доказателството не се извършва по отношение на константата, с която е започнато. Твърдението, което се опитваме да докажем е *съществува положителна константа c и стойност на аргумента n_0 , такива че за всяко $n \geq n_0$, $T(n) \leq cn$* . Естествено, това твърдение е лъжа, така че няма как да го докажем, но важното е, че **това** е твърдението. “ $T(n) = O(n)$ ” е просто кратък запис за него. Щом избраният символ за константата е c , доказателството трябва да бъде извършено по отношение на него. $2c$ е **друга** константа. Наистина, $2c$ е константа, но за да имаме доказателство, трябва да го направим по отношение на c .

- Второ, понякога се налага да се засили индуктивното предположение. Ще докажем по

индукция, че решението на

$$T(n) = 2T(n-1) + n \quad (3.44)$$

е $T(n) = \Theta(2^n)$.

Първо ще докажем, че $T(n) = O(2^n)$. Нарочно започваме доказателството с няколко неуспешни опита.

Първи опит. Допускаме, че съществува константа c , такава че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2c2^{n-1} + n \\ &= c2^n + n \\ &\not\leq c2^n \text{ за никоя положителна } c \end{aligned}$$

Доказателството се провали. Това не означава, че твърдението, което доказваме, непременно е грешно. Провал на доказателство **може** да означава, че твърдението не е вярно, но **може** и да означава, че твърдението е вярно, но не може да се докаже по този начин. Ще се опитаме да докажем по-силно твърдение.

Втори опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - b$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + n \\ &= c2^n - 2b + n \\ &\not\leq c2^n - b \text{ за някои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Трети опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^{n-b}$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-b-1}) + n \\ &= c2^{n-b} + n \\ &\not\leq c2^{n-b} \text{ за някои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Четвърти опит. Допускаме, че съществува положителна константа c , такава че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - (n-1)) + n \\ &= c2^n - n + 2 \\ &\not\leq c2^n - n \text{ за никоя положителна } c \end{aligned}$$

Пети опит. Допускаме, че съществуват положителни константи b и c , такива че за всички достатъчно големи n е вярно, че

$$T(n) \leq c2^n - bn$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b(n-1)) + n \\ &= c2^n - 2bn + 2b + n \\ &= c2^n - bn + (1-b)n + 2b \\ &\leq c2^n - bn \text{ за всеки избор на } c > 0 \text{ и } b > 1 \end{aligned}$$

Успех! Чак сега успяхме да формулираме индуктивно предположение, което е доказуемо.

Сега ще докажем, че $T(n) = \Omega(n)$. С други думи, че съществува положителна константа d , такава че за всички достатъчно големи n е изпълнено

$$T(n) \geq d2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\geq 2(d2^{n-1}) + n \\ &= d2^n + n \\ &\geq d2^n \end{aligned}$$

Успех! Видяхме, че в този случай—а това е вярно по принцип при доказателства по индукция на решения на рекурентни уравнения—засилване на индуктивното предположение се иска само за едното от O и Ω доказателствата.

Теорема 38 ползва термина *целочислено разбиване*, на английски *integer partition*.

Определение 39: Целочислено разбиване

Целочислено разбиване на цяло положително число k е изреждане на цели положителни числа в ненарастващ ред, които се сумират до k . Тези цели положителни числа са *дяловете* на разбиването.

Това, че дяловете се изреждат в ненарастващ ред, е само за по-лесно възприемане. Всяка наредба “върши работа”, понеже сумирането е асоциативно и комутативно. Ерго, в същността

си целочисленото разбиване е мултимножество, чиито елементи са дяловете. Но е прието да се описва чрез изреждане на дяловете в ненарастващ ред, като те се конкатенират с “+”. Като пример, числото 4 има точно пет целочислени разбивания, които са

$$4 = 4$$

$$4 = 3 + 1$$

$$4 = 2 + 2$$

$$4 = 2 + 1 + 1$$

$$4 = 1 + 1 + 1 + 1$$

Първото разбиване има само един дял, а именно 4, второто има дялове 3 и 1, и така нататък.

Теорема 38: $T(n) = \sum_{i=1}^{\ell} T(n_i) + \Theta(\ell)$ по индукция

Рекурентното уравнение

$$T(n) = \sum_{i=1}^{\ell} T(n_i) + \Theta(\ell)$$

за произволно целочислено разбиване $n_1 + n_2 + \dots + n_{\ell}$ на $n-1$ има решение $T(n) = \Theta(n)$.

Едно пояснение. Броят ℓ на дяловете на разбиването не е част от дефиницията на рекурентното уравнение в смисъл, че разглеждаме **произволно** разбиване и ℓ само означава броят на дяловете му. Очевидно $1 \leq \ell \leq n-1$, но по отношение на кое да е $T(n_i)$, броят на дяловете не е непременно ℓ , а е число между 1 и $n_i - 1$ и това е всичко, което знаем за него.

Доказателство: По индукция по n .

Част i: Ще докажем, че $T(n) = O(n)$. Ще докажем, че съществуват положителни константи b и c , такива че $T(n) \leq c \cdot n - b$ за всички достатъчно големи n . Нека k е достатъчно голяма положителна константа; в частност, която е по-голяма или равна на скритата мултипликативна константа в израза “ $\Theta(\ell)$ ”. От индуктивното предположение:

$$\begin{aligned} T(n) &\leq \sum_{i=1}^{\ell} (c \cdot n_i - b) + k \cdot \ell \\ &= c \left(\sum_{i=1}^{\ell} n_i \right) - b \cdot \ell + k \cdot \ell \quad // \text{ но } n_1 + n_2 + \dots + n_{\ell} = n - 1 \\ &= c(n - 1) + \ell(k - b) \\ &= c \cdot n - c + \ell(k - b) \\ &\leq c \cdot n - b, \text{ ако } k - b < 0 \text{ и } c > b \end{aligned}$$

□

Част ii: Ще докажем, че $T(n) = \Omega(n)$. Ще докажем, че съществува положителна константа d , такава че $T(n) \geq d \cdot n$ за всички достатъчно големи n . Да допуснем, че k е достатъчно голяма положителна константа; в частност, която е по-голяма или равна на скритата мул-

типлическа константа в израза “ $\Theta(\ell)$ ”. От индуктивното предположение:

$$\begin{aligned}
 T(n) &\geq \sum_{i=1}^{\ell} d \cdot n_i + k \cdot \ell \\
 &= d \left(\sum_{i=1}^{\ell} n_i \right) + k\ell \quad // \text{ но } n_1 + n_2 + \dots + n_{\ell} = n - 1 \\
 &= d(n - 1) + k\ell \\
 &= dn - d + k\ell \\
 &\geq dn, \text{ ако } -d + k\ell \geq 0 \leftrightarrow d \leq k\ell
 \end{aligned}$$

□

Теорема 38 е полезна при оценяване на сложността на алгоритми по схемите **Разделяй-и-Владей** или **Динамично Програмиране**, каквито ще видим в Подсекция 12.8.1 и Подсекция 12.8.2, които се изпълняват върху коренови дървета. $T(n)$ е времето за работата на алгоритъма върху кореново дърво с n върха. $T(n)$ е сумата на следните две събираеми.

1. Сумата от времената за работа на алгоритъма върху поддърветата, вкоренени в децата на корена. Броят на тези деца не е фиксиран и може да е от 1 до $n - 1$. Поради това приемаме, че има ℓ деца, като $1 \leq \ell \leq n - 1$, и вкоренените в тези деца поддървета имат $n_1, \dots, n_{\ell}$ върха, като $n_1 + n_2 + \dots + n_{\ell}$ е целочислено разбиване на $n - 1$,
2. Времето за работата на алгоритъма извън рекурсивните викания. Допускаме, че тя е линейна в ℓ , откъдето и събираемото $+ \ell$.

Забележете, че времето за работата на алгоритъма извън рекурсивните викания е поне линейно в ℓ , защото всяко от извършените ℓ рекурсивни викания връща $\Omega(1)$ информация и всичко това трябва да бъде обработено от текущото рекурсивно викане, което дава долна граница $\Omega(\ell)$ за работата на алгоритъма извън рекурсивните викания.

Теорема 39 е аналог на Случай 3 на Мастър Теоремата (Теорема 41) в смисъл, че асимптотиката на решението е асимптотиката на нехомогенната част, понеже тя доминира над големината на дървото на рекурсията.

Теорема 39: $T(n) = \sum_{i=1}^k T(n/b_i) + n$ по индукция

Нека $k \in \mathbb{N}^+$, $b_1, \dots, b_k \in \mathbb{R}$, $b_1, \dots, b_k > 1$ и $\frac{1}{b_1} + \dots + \frac{1}{b_k} < 1$. Рекурентното уравнение

$$T(n) = T\left(\frac{n}{b_1}\right) + \dots + T\left(\frac{n}{b_k}\right) + n$$

има решение $T(n) = \Theta(n)$.

Доказателство: По индукция по n . Очевидно $T(n) = \Omega(n)$ заради събираемото $+n$, така че ще докажем само $T(n) = O(n)$. Ще докажем, че съществува положителна константа c , такава че $T(n) \leq c \cdot n$ за всички достатъчно големи стойности на n . Нека $\frac{1}{b_1} + \dots + \frac{1}{b_k} = B$.

По условие, $B < 1$. От индуктивното предположение имаме

$$\begin{aligned} T\left(\frac{n}{b_1}\right) &\leq c \frac{n}{b_1} \\ &\dots \\ T\left(\frac{n}{b_k}\right) &\leq c \frac{n}{b_k} \end{aligned}$$

Тогава

$$\begin{aligned} T(n) &= T\left(\frac{n}{b_1}\right) + \dots + T\left(\frac{n}{b_k}\right) + n \\ &\leq c \frac{n}{b_1} + \dots + c \frac{n}{b_k} + n \\ &= cn \left(\frac{1}{b_1} + \dots + \frac{1}{b_k} \right) + n \\ &= cnB + n \end{aligned}$$

Очевидно $cnB + n < cn \leftrightarrow cB + 1 < c$. Но $B < 1$, така че съществува положително c , за което това е вярно: $1 < c - cB \leftrightarrow 1 < c(1 - B) \leftrightarrow c > \frac{1}{1-B}$. $\square$

Допълнение 20: Няколко рекурентни уравнения по индукция

Следното нелеко за решаване рекурентно уравнение е от сборника на Parberry [127, pp. 40, Problem 248].

Задача 32

Решете рекурентното уравнение

$$T(n) = T\left(\left\lfloor \frac{n}{\lg n} \right\rfloor\right) + 1. \quad (3.45)$$

Решение: Уравнението е с единствена поява вдясно, поради което решението му, в асимптотичния смисъл, е броят пъти, който се изпълнява итераторът

$$n \rightarrow \left\lfloor \frac{n}{\lg n} \right\rfloor \quad (3.46)$$

така че n да стане $\Theta(1)$. Но аргументът вдясно намалява по сложен начин и изглежда невероятно да забележим закономерност, ако развиваме и развиваме.

Да опитаме да оценим броя на итериранията на око. Да разгледаме друго рекурентно уравнение $S(n)$, в което аргументът винаги бива делен на едно и също m : $S(n) = S\left(\left\lfloor \frac{n}{m} \right\rfloor\right) + 1$. Ако m е константа, ние знаем от Следствие 16, че решението е $S(n) \asymp \log_m n$. Да видим какво става, когато m не е константа. Дори тогава можем да приложим Следствие 16, стига m да е едно и също нещо при всяко итериране; тоест, винаги да делим на едно и също нещо. С други думи, ако $m = \lg N$, където N е първоначалната

стойност на n и m не се променя при итерирането, то решението е

$$S(n) \asymp \log_m n \asymp \log_{\lg N} n \asymp \quad // \text{ понеже тук } n \text{ е началната ст-ст на аргумента}$$

$$\log_{\lg n} n \asymp \frac{\lg n}{\lg \lg n}$$

Обаче в (3.46), n намалява при итерирането, така че и $\lg n$ намалява, така че делим на все по-малко нещо.

От тези съображения можем да сложим следните граници за $T(n)$:

$$\frac{\lg n}{\lg \lg n} \leq T(n) \leq \lg n \quad (3.47)$$

Това вече е нещо. $\lg \lg n$ е доста близо до константа (макар да е неограничено растяща функция), така че (3.47) ограничава значително възможностите за $T(n)$, въпреки че не дава точно Тета-асимптотиката. Звучи невероятно $T(n) \asymp \lg n$, защото $\lg n$ е асимптотиката при деление само на константа. Да опитаме другата крайност: $T(n) \asymp \frac{\lg n}{\lg \lg n}$.

Ще покажем по индукция по n , че $T(n) \asymp \frac{\lg n}{\lg \lg n}$. За простота изпускаме нотацията " $\lfloor \rfloor$ " от (3.45) и $T(n)$ се определя чрез

$$T(n) = T\left(\frac{n}{\lg n}\right) + 1 \quad (3.48)$$

Част i: Ще докажем, че $T(n) = O\left(\frac{\lg n}{\lg \lg n}\right)$. Тоест, че има положителна константа c и стойност на аргумента n_0 , такива че за всяко $n \geq n_0$,

$$T(n) \leq c \frac{\lg n}{\lg \lg n} \quad (3.49)$$

Индуктивното предположение е

$$T\left(\frac{n}{\lg n}\right) \leq c \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg\left(\frac{n}{\lg n}\right)} \quad (3.50)$$

Замествайки (3.50) в (3.48), получаваме

$$\begin{aligned} T(n) &\leq c \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg\left(\frac{n}{\lg n}\right)} + 1 \\ &= c \frac{\lg n - \lg \lg n}{\lg(\lg n - \lg \lg n)} + 1 \end{aligned} \quad (3.51)$$

Заместваме $\lg n$ с 2^m , така че заместваме $\lg \lg n$ с m , в (3.51) и получаваме

$$\begin{aligned} c \frac{2^m - m}{\lg(2^m - m)} + 1 &\leq c \frac{2^m - m}{\lg 1.9^m} + 1 = c \frac{2^m - m}{m \lg 1.9} + 1 = \frac{c}{1.9} \frac{2^m - m}{m} + 1 = \\ &= \frac{c}{1.9} \frac{2^m}{m} - \frac{c}{1.9} + 1 \leq c \frac{2^m}{m} = c \frac{\lg n}{\lg \lg n} \end{aligned} \quad (3.52)$$

От (3.51) и (3.52) заключаваме, че

$$T(n) \leq c \frac{\lg n}{\lg \lg n}$$

за всяко положително c , да кажем $c = 1$, и всяко достатъчно голямо n .

Част ii: Ще докажем, че $T(n) = \Omega\left(\frac{\lg n}{\lg \lg n}\right)$. Тоест, че има положителна константа d и стойност на аргумента n_0 , такива че за всяко $n \geq n_0$,

$$T(n) \geq d \frac{\lg n}{\lg \lg n} \quad (3.53)$$

Индуктивното предположение е

$$T\left(\frac{n}{\lg n}\right) \geq d \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg\left(\frac{n}{\lg n}\right)} \quad (3.54)$$

Замествайки (3.54) в (3.48), получаваме

$$\begin{aligned} T(n) &\geq d \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg\left(\frac{n}{\lg n}\right)} + 1 \\ &= d \frac{\lg n - \lg \lg n}{\lg(\lg n - \lg \lg n)} + 1 \end{aligned} \quad (3.55)$$

Заместваме $\lg n$ с 2^m , така че заместваме $\lg \lg n$ с m , в (3.55) и получаваме

$$d \frac{2^m - m}{\lg(2^m - m)} + 1 \geq d \frac{2^m - m}{\lg 2^m} + 1 = d \frac{2^m - m}{m} + 1 = d \frac{2^m}{m} - d + 1 \geq d \frac{2^m}{m} \quad (3.56)$$

Това е вярно за, да кажем, $d = 1$. Връщаме се към променливата n . От (3.55) и (3.56) виждаме, че

$$T(n) \geq d \frac{\lg n}{\lg \lg n}$$

за $d = 1$ и всяко достатъчно голямо n . □

Ние вече решихме $T(n) = T(n-2) + 2 \lg n$ в Задача 26, но това беше неформално решение чрез развиване. Сега ще покажем формално вярността на решението.

Задача 33

Докажете, че рекурентното уравнение

$$T(n) = T(n-2) + 2 \lg n$$

има решение $T(n) \asymp n \lg n$.

Решение: По индукция по n .

Part i: Ще докажем, че $T(n) = O(n \lg n)$. Тоест, че има положителна константа c , такава че за всяко достатъчно голямо n ,

$$T(n) \leq cn \lg n \quad (3.57)$$

В сила е

$$\begin{aligned} T(n) &\leq c(n-2) \lg(n-2) + 2 \lg n \quad // \text{от индуктивното предположение} \\ &\leq c(n-2) \lg n + 2 \lg n \\ &= cn \lg n - 2c \lg n + 2 \lg n \\ &\leq cn \lg n, \text{ ако } -2c \lg n + 2 \lg n \leq 0 \leftrightarrow c \geq 1 \end{aligned}$$

Part ii: Ще докажем, че $T(n) = \Omega(n \lg n)$. Тоест, че има положителна константа d , такава че за всяко достатъчно голямо n ,

$$T(n) \geq dn \lg n \quad (3.58)$$

В сила е

$$\begin{aligned} T(n) &\geq d(n-2) \lg(n-2) + 2 \lg n \quad // \text{от индуктивното предположение} \\ &= (dn - 2d) \lg(n-2) + 2 \lg n \\ &= dn \lg(n-2) + 2(\lg n - d \lg(n-2)) \end{aligned} \quad (3.59)$$

Предвид (3.58) and (3.59), целта ни е да покажем, че

$$\begin{aligned} dn \lg(n-2) + 2(\lg n - d \lg(n-2)) &\geq dn \lg n \leftrightarrow \\ dn \lg(n-2) - dn \lg n + 2(\lg n - d \lg(n-2)) &\geq 0 \leftrightarrow \\ \underbrace{d \lg \left(\frac{n-2}{n} \right)^n}_A + \underbrace{2 \lg \frac{n}{(n-2)^d}}_B &\geq 0 \end{aligned} \quad (3.60)$$

Първо ще оценим A при неограничено нарастване на n :

$$\lim_{n \rightarrow \infty} d \lg \left(\frac{n-2}{n} \right)^n = d \lim_{n \rightarrow \infty} \lg \left(1 + \frac{-2}{n} \right)^n = d \lg \lim_{n \rightarrow \infty} \left(1 + \frac{-2}{n} \right)^n = d \lg e^{-2} = -2d \lg e$$

Да разгледаме В при неограничено нарастване на n :

$$\lim_{n \rightarrow \infty} 2 \lg \frac{n}{(n-2)^d} = 2 \lg \lim_{n \rightarrow \infty} \frac{n}{(n-2)^d} \quad (3.61)$$

Забележете, че при всяко $d \in (0, 1)$, (3.61) е $+\infty$. Например, за $d = \frac{1}{2}$, (3.61) става

$$\begin{aligned} 2 \lg \lim_{n \rightarrow \infty} \left(n^{\frac{1}{2}} \frac{n^{\frac{1}{2}}}{(n-2)^{\frac{1}{2}}} \right) &= \\ 2 \lg \lim_{n \rightarrow \infty} \left(n^{\frac{1}{2}} \left(\frac{n}{n-2} \right)^{\frac{1}{2}} \right) &= \\ 2 \lg \left(\left(\lim_{n \rightarrow \infty} n^{\frac{1}{2}} \right) \underbrace{\left(\lim_{n \rightarrow \infty} \left(\frac{1}{1 - \frac{2}{n}} \right)^{\frac{1}{2}} \right)}_1 \right) &= +\infty \end{aligned}$$

Следва, че (3.60) е вярно за всеки избор на d от $(0, 1)$; примерно, $d = \frac{1}{2}$, понеже А по абсолютна стойност е ограничена от константа, а В расте неограничено. $\square$

Наблюдение 27

Доказателството по индукция в **Част ii** е триково. Разгледайте (3.59):

$$dn \lg(n-2) + 2(\lg n - d \lg(n-2))$$

Обикновено се “справяме” с логаритмите на суми или разлики като апроксимираме сумите или разликите с произведения или частни по такъв начин, че неравенството да се запази в желаната посока. Но в този случай, ако апроксимираме $n-2$ вътре в този логаритъм с αn , където α е положителна константа, трябва да е вярно, че $\alpha < 1$, за да е изпълнено $\alpha n < n$. Ето какво става, ако заместим $n-2$ с αn в логаритъма отляво:

$$dn \lg \alpha n + 2(\lg n - d \lg(n-2)) = dn \lg n + dn \lg \alpha + 2(\lg n - d \lg(n-2))$$

За да довършим доказателството, трябва да покажем, че последното е по-голямо или равно на $dn \lg n$. А за да покажем това, трябва да покажем, че събираемото $dn \lg \alpha + 2(\lg n - d \lg(n-2))$ е положително. Но това не е вярно! Тъй като $d > 0$ и $\alpha < 1$, то $dn \lg \alpha < 0$ за всяко $n > 0$. И асимптотичният характер на $dn \lg \alpha + 2(\lg n - d \lg(n-2))$ се определя от $dn \lg \alpha$, понеже линейната функция доминира над логаритмичната за всяко достатъчно голямо n .

И така, ако заместим n с αn , няма да получим валидно доказателство. Налага се да използваме по-изтънчена техника, базирана на математическия анализ.

3.2.4 Чрез Мастър теоремата (МТ)

Този теоретичен резултат на Jon Louis Bentley, Dorothea Haken и James Saxe е от 1978 г. [16]. Преди това, рекурентни уравнения, в които функцията на намаляване е деление, са

били решавани чрез *ad hoc* методи. Например, в класическия учебник по алгоритми на Aho, Hopcroft и Ullman [3] от 1974 г. има следната теорема.

Теорема 40: Решенията на клас рекурентни уравнения (Theorem 2.1 в [3])

Let a , b , and c be nonnegative constants. The solution to the recurrence

$$T(n) = \begin{cases} b, & \text{for } n = 1 \\ aT(n/c) + bn, & \text{for } n > 1 \end{cases}$$

for n a power of c is

$$T(n) = \begin{cases} O(n), & \text{if } a < c, \\ O(n \lg n), & \text{if } a = c, \\ n^{\log_c a}, & \text{if } a > c. \end{cases}$$

Забележете, че авторите на [16] не говорят за “Master theorem”. Гръмкото име “**Master theorem**”, доколкото е известно на автора на тези лекционни записки, се дължи на Cormen, Leiserson и Rivest – авторите на [34]. Без съмнение то идва оттам, че въпросната теорема се явява обобщение на множество теореми като Теорема 40, даващи решения на рекурентни уравнения, в които функцията на намаляване е деление.

От друга страна, няма най-обща теорема, която да дава универсален метод за решаване на всякакви рекурентни уравнения. По отношение на линейните рекурентни уравнения с константни коефициенти, при които намаляването е чрез деление с константа, каквито уравнения третира Теорема 41, е известна значително по-мощна и обща Теорема на Акра-Bazzi [4] (Подсекция 3.2.5). И така, Теорема 41, въпреки името си, е просто едно средство, като са известни много по-мощни средства от него.

Следното изложение е по учебника на Cormen и др. [34, стр. 61].

Теорема 41: Мастър теорема (Theorem 4.1 от [34, стр. 94])

Нека $a \geq 1$ и $b > 1$ са константи и нека $f(n)$ е положителна функция. Нека

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3.62}$$

където $\frac{n}{b}$ има смисъл или на $\left\lfloor \frac{n}{b} \right\rfloor$, или на $\left\lceil \frac{n}{b} \right\rceil$. Тогава асимптотиката на $T(n)$ е следната:

Случай 1 Ако $f(n) = O(n^{\log_b a}/n^\epsilon)$ за някоя положителна константа ϵ , то $T(n) = \Theta(n^{\log_b a})$.

Случай 2 Ако $f(n) = \Theta(n^{\log_b a})$, тогава $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$.
С други думи, $T(n) = \Theta(f(n) \cdot \lg n)$.

Случай 3 Ако са изпълнени следните условия:

1. $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ за някоя положителна константа ϵ , и
2. съществува константа c , такава че $0 < c < 1$ и $\forall n \geq 1 : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$,

то $T(n) = \Theta(f(n))$. □

Накратко ще наричаме тази теорема “МТ”.

Условие 3.2 се нарича *условие за регулярност*.

Няма да даваме доказателство на МТ. Доказателство, базирано на метода с характеристичното уравнение (Подсекция 3.2.6) на **ограничен вариант** на МТ има в Допълнение 23. Читателят може да види пълно доказателство на МТ в [34]. Доказателство има и в статията [99]. Тук само ще дадем няколко пояснения.

Пояснение 1. И в трите случая на МТ сравняваме броя на листата на дървото на рекурсията с функцията $f(n)$. Да се убедим, че броят на листата на дървото наистина е $n^{\log_b a}$. Нека да мислим, че a и b са цели числа[†]. Ако a и b са цели положителни числа, като $b \geq 2$, то (3.62) отговаря на рекурсивен алгоритъм, направен по схемата **Разделяй-и-Владей**, който прави a рекурсивни извиквания, всяко върху вход с големина $\frac{n}{b}$, и освен това извършва $f(n)$ работа (като време) по разделянето на входа и комбинирането на резултатите от извикванията. Дървото на този алгоритъм е показано на Фигура 3.7. Всеки връх на дървото има разклоненост a , и входът намалява с деление на b . Сумите по нивата на дървото на нехомогенните части са $f(n)$, $af(\frac{n}{b})$, $a^2f(\frac{n}{b^2})$, и така нататък. Височината на дървото е $\log_b n$. Броят на върховете от ниво[‡] k очевидно е a^k , така че броят на листата е $a^{\log_b n}$. Лесно се вижда, че $a^{\log_b n} = n^{\log_b a}$.

Броят на листата е пропорционален на големината на дървото, освен ако a не е 1. Ерго, големината на дървото е $\Theta(n^{\log_b a})$, освен ако $a = 1$. Ако $a = 1$, то дървото има точно едно листо и $n^{\log_b a} = n^{\log_b 1} = n^0 = 1$. От друга страна, височината продължава да е $\log_b n$ и размерът на дървото е $\Theta(\log_b n)$. Да игнорираме изключението $a = 1$.

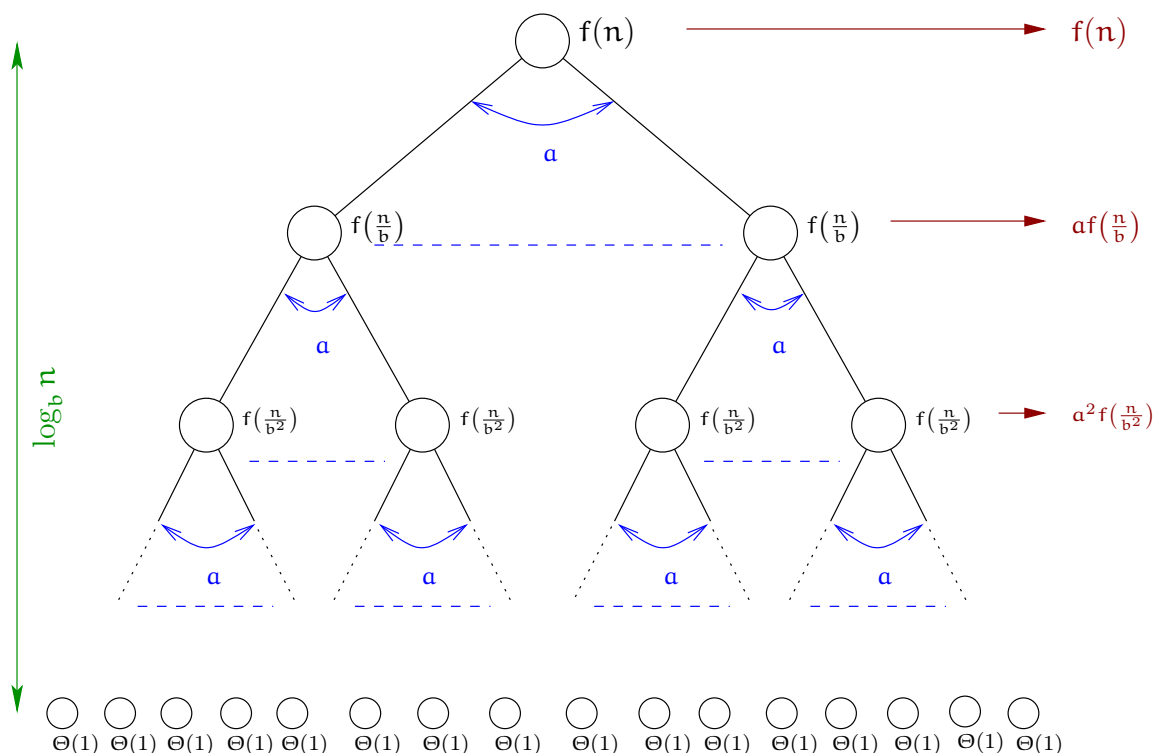
Наблюдение 28

Ако $a > 1$, МТ сравнява големината на дървото с нехомогенната част $f(n)$.

В първия случай на МТ големината на дървото доминира над $f(n)$, така че $T(n)$ е големината на дървото, в асимптотичния смисъл. Във втория случай големината на дървото и $f(n)$ са асимптотично еквивалентни. В третия случай $f(n)$ доминира над големината на дървото, така че $f(n)$ дава директно асимптотиката, ако е изпълнено и условието за регулярност.

[†]В [34] е доказано, че същите съображения остават в сила дори когато a и b не са цели.

[‡]Припомняме си, че върховете от ниво k са точно върховете на разстояние k от корена.



Фигура 3.7: Дървото на рекурсията на алгоритъм, чиято сложност по време се описва от $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, където a и b са цели положителни числа.

Наблюдение 29

Сравняването на големината на дървото с нехомогенната част има смисъл и при други видове рекурентни уравнения. Например, в Задача 29 големината на дървото е “факториелна”, докато нехомогенната част е константа, и съвсем естествено решението е, в асимптотичния смисъл, същото като големината на дървото, а именно $\Theta(n!)$. В някакъв смисъл, това е аналог на първия случай на МТ.

В Задача 30 нещата са аналогични: големината на дървото пак е “факториелна”, докато нехомогенната част е само експоненциална; тъй като експоненциалната функция “бледнее” пред факториела, отново големината на дървото е, в асимптотичния смисъл, същата като решението, а именно $\Theta(n!)$. И това е аналогично на първия случай на МТ.

В Задача 31 обаче големината на дървото е експоненциална, докато нехомогенната част е $n!$ и последната доминира над големината на дървото. В този случай асимптотиката на нехомогенната част е решението, а именно $\Theta(n!)$. В някакъв смисъл, това е аналог на третия случай на МТ.

Решението на $T(n) = 2T(n-1) + 2^n$ е $T(n) \asymp n2^n$, което се извежда елементарно с метода с характеристичното уравнение (Подсекция 3.2.6). Тук и големината на дървото, и нехомогенната част са $\Theta(2^n)$, така че решението $\Theta(n2^n)$ в някакъв смисъл е аналог на втория случай на МТ. И във втория случай на МТ, и тук решението е, асимптотично, произведението от големината на дървото и нейния логаритъм.

Пояснение 2. Трите случая на МТ не са разбиване на всички възможни случаи. Както вече видяхме в Теорема 24(2.52), има функции, които не са асимптотично сравними, така че ако $n^{\log_b a}$ и $f(n)$ не са асимптотично сравними, то МТ не е приложима.

Но дори когато $n^{\log_b a}$ и $f(n)$ са асимптотично сравними, има примери за a , b и $f(n)$, в които $f(n)$ расте прекалено бързо, за да попаднем в първия случай, но прекалено бавно, за да попаднем във втория, и МТ пак не е приложима. Неформално казано, между първия и втория случай на МТ има “празнина” и ако конкретната задача е в някоя от тези “празнини”, не можем да използваме МТ.

Например, нека $a = b = 2$ и $f(n) = \frac{n}{\lg n}$. Тогава $f(n) \neq O(n^{\log_b a}/n^\epsilon)$ за всяка положителна константа ϵ , така че рекурентното уравнение не попада в първия случай. От друга страна, $f(n) \neq \Theta(n^{\log_b a})$, така че не сме и във втория случай. Бидейки попаднали в “празнината” между първия и втория случай, налага се да ползваме други методи, за да решим задачата.

Допълнение 21: Решение на $T(n) = 2T(\frac{n}{2}) + \frac{n}{\lg n}$

Ще решим $T(n) = 2T(\frac{n}{2}) + \frac{n}{\lg n}$. Първо да опитаме с Теорема 41. Използвайки терминологията на Теорема 41, $a = b = 2$, следователно $\log_b a$ е $\log_2 2 = 1$, следователно $n^{\log_b a}$ е $n^1 = n$. Функцията $f(n)$ е $\frac{n}{\lg n}$. Да видим дали може да класифицираме рекурентното уравнение в един от трите случая.

Случай 1 Дали $\frac{n}{\lg n} = O(n^1/n^\epsilon)$ за някоя константа $\epsilon > 0$? Не, защото $\forall \epsilon > 0 : n^\epsilon \neq O(\lg n)$. Детайлно доказателство на този факт има в сборника с решени задачи по алгоритми.

Случай 2 Дали $\frac{n}{\lg n} = \Theta(n^1)$? Не, защото $\frac{n}{\lg n} = o(n^1)$.

Случай 3 Дали $\frac{n}{\lg n} = \Omega(n^1 \cdot n^\epsilon)$ за някоя константа $\epsilon > 0$? Не, и то по същата причина: защото $\frac{n}{\lg n} = o(n^1)$.

Тогава тази задача не може да бъде решена с Теорема 41. Нещо повече: не можем да ползваме и Теорема 42 на стр. 241, защото $\forall t > 0 : \frac{n}{\lg n} \neq \Theta(n^{\log_2 2} \times \lg^t(n))$.

Ще решим задачата с развиване:

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\
 &= 2\left(2T\left(\frac{n}{4}\right) + \frac{\frac{n}{2}}{\lg \frac{n}{2}}\right) + \frac{n}{\lg n} \\
 &= 4T\left(\frac{n}{4}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
 &= 4\left(2T\left(\frac{n}{8}\right) + \frac{\frac{n}{4}}{\lg \frac{n}{4}}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
 &= 8T\left(\frac{n}{8}\right) + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
 &\dots \\
 &= nT(1) + \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \dots + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
 &= nT(1) + n \underbrace{\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{(\lg n) - 2} + \frac{1}{(\lg n) - 1} + \frac{1}{\lg n}\right)}_B
 \end{aligned}$$

Забелязваме, че $B = n \times H_{\lg n}$, защото в скобите се намира $(\lg n)^{\text{тата}}$ парциална сума на хармоничния ред. Добре известно е, че $H_m \asymp \lg m$, следователно $H_{\lg n} = \Theta(\lg \lg n)$, и оттам $B = \Theta(n \lg \lg n)$. От това следва, че $T(n) = \Theta(n \lg \lg n)$.

Аналогично, има примери за a , b и $f(n)$, в които $f(n)$ расте прекалено бързо, за да попаднем във втория случай, но прекалено бавно, за да попаднем в третия. С други думи, между втория и третия случай също има “празнина”, за която МТ не е приложима.

Пояснение 3. Условието $f(n) = O(n^{\log_b a}/n^\epsilon)$ в първия случай на МТ е по-силно от $f(n) = o(n^{\log_b a})$, а условието $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ в третия случай е по-силно от $f(n) = \omega(n^{\log_b a})$. За удобство да направим следните дефиниции.

Определение 40: Асимптотично строго по-бавно/по-бързо и полиномиално по-бавно/по-бързо

Ако $\phi(n) = o(\psi(n))$, казваме, че $\phi(n)$ расте асимптотично строго по-бавно от $\psi(n)$.

Ако $\phi(n) = O(\psi(n)/n^\epsilon)$ за някаква положителна константа ϵ , казваме, че $\phi(n)$ расте полиномиално по-бавно от $\psi(n)$.

Ако $\phi(n) = \omega(\psi(n))$, казваме, че $\phi(n)$ расте асимптотично строго по-бързо от $\psi(n)$.

Ако $\phi(n) = \Omega(\psi(n) \cdot n^\epsilon)$ за някаква положителна константа ϵ , казваме, че $\phi(n)$ расте полиномиално по-бързо от $\psi(n)$.

$f(n) = O(n^{\log_b a}/n^\epsilon)$, където ϵ е положителна константа, казва, че между $f(n)$ и $n^{\log_b a}$ може да бъде “вкарано” n^ϵ , като $f(n)$ е отдолу. С други думи, съществува полиномиално растяща функция, такава че $f(n)$ “изостава” от $n^{\log_b a}$ с нея като множител. Това е значително по-силно от $f(n)$ да “изостава” от $n^{\log_b a}$ на повече от всяка константа като множител. Последното е $f(n) = o(n^{\log_b a})$. Аналогично, $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon)$ казва, че между $f(n)$ и $n^{\log_b a}$ може да бъде “вкарано” n^ϵ , като $f(n)$ е отгоре. С други думи, съществува полиномиално растяща функция, такава че $f(n)$ “изпреварва” $n^{\log_b a}$ с нея като множител. Това е значително по-силно от $f(n)$ да “изпреварва” $n^{\log_b a}$ на повече от всяка константа като множител. Последното е $f(n) = \omega(n^{\log_b a})$.

Наблюдение 30

Ако $\phi(n)$ расте полиномиално по-бавно от $\psi(n)$, то $\phi(n)$ расте асимптотично строго по-бавно от $\psi(n)$. Конверсното не е вярно. Следователно, условието за полиномиално по-бавно нарастване е по-силно от условието за асимптотично строго по-бавно нарастване. Ако $\phi(n)$ расте полиномиално по-бързо от $\psi(n)$, то $\phi(n)$ расте асимптотично строго по-бързо от $\psi(n)$. Конверсното не е вярно. Следователно, условието за полиномиално по-бързо нарастване е по-силно от условието за асимптотично строго по-бързо нарастване.

Наблюдение 30 ни дава право да кажем следното.

$$f(n) = O(n^k/n^\epsilon) \rightarrow f(n) = o(n^k)$$

$$f(n) = o(n^k) \rightarrow f(n) = O(n^k/n^\epsilon)$$

$$f(n) = \Omega(n^k \cdot n^\epsilon) \rightarrow f(n) = \omega(n^k)$$

$$f(n) = \omega(n^k) \rightarrow f(n) = \Omega(n^k \cdot n^\epsilon)$$

Наблюдение 31

Би било грешка да се опитаме да формулираме първия случай на МТ чрез $f(n) = o(n^{\log_b a})$ или третия случай чрез $f(n) = \omega(n^{\log_b a})$.

Пояснение 4. Условието за регулярност в третия случай е по-силно от условието $f(n) = \Omega(n^{\log_b a + \epsilon})$ в смисъл, че

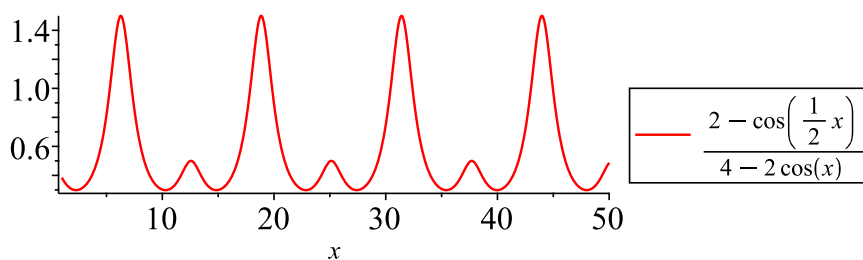
$$\exists c, 0 < c < 1, \forall n \nearrow : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) \rightarrow \exists \epsilon > 0 : f(n) = \Omega(n^{(\log_b a) + \epsilon}) \quad (3.63)$$

$$\exists \epsilon > 0 : f(n) = \Omega(n^{(\log_b a) + \epsilon}) \rightarrow \exists c, 0 < c < 1, \forall n \nearrow : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) \quad (3.64)$$

Ето пример за $f(n)$ като в (3.64), взет от [статията в Wikipedia за МТ](#). Да разгледаме уравнението $T(n) = T(n/2) + n(2 - \cos(n))$. Тогава $a = 1$, $b = 2$ и $f(n) = n(2 - \cos(n))$. Тъй като $f(n)$ е положителна, условията за a , b и $f(n)$ са изпълнени. Да опитае да приложим МТ. В сила е $\log_b a = 0$, така че $n^{\log_b a} = n^0 = 1$, така че със сигурност за някое положително ϵ е изпълнено $f(n) = \Omega(n^{(\log_b a) + \epsilon})$. Да видим как стоят нещата с условието за регулярност. Дали съществува константа c , такава че $0 < c < 1$ и за всички достатъчно големи n е вярно $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$? Очевидно

$$\begin{aligned} a \cdot f\left(\frac{n}{b}\right) &\leq c \cdot f(n) \leftrightarrow \\ 1 \cdot \frac{n}{2} \left(2 - \cos\left(\frac{n}{2}\right)\right) &\leq c \cdot n(2 - \cos(n)) \leftrightarrow \\ 2 - \cos\left(\frac{n}{2}\right) &\leq 2c(2 - \cos(n)) \\ c &\geq \frac{2 - \cos\left(\frac{n}{2}\right)}{2(2 - \cos(n))} \end{aligned}$$

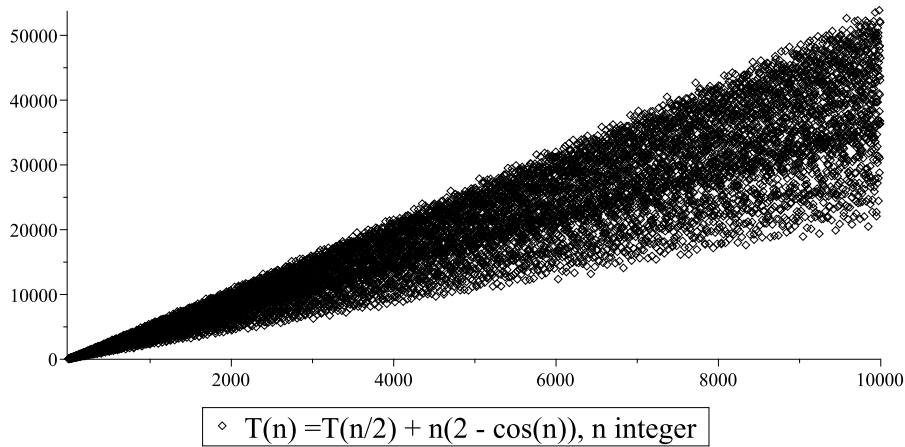
Лесно се вижда, че за всяко цяло положително n_0 съществува цяло $n \geq n_0$, такава че $\frac{2 - \cos\left(\frac{n}{2}\right)}{2(2 - \cos(n))} > 1$. Нагледна аргументация е графиката[†] на $\frac{2 - \cos\left(\frac{x}{2}\right)}{2(2 - \cos(x))}$.



Не съществува константа, по-малка от единица, която е ограничена отдолу от $\frac{2 - \cos\left(\frac{n}{2}\right)}{2(2 - \cos(n))}$. Заключаваме, че третият случай на МТ не е приложим заради това, че условието за регулярност не е изпълнено. От друга страна, числени експерименти навеждат на мисълта, че $T(n) \asymp n$. Ето дискретен плот[‡] на $T(n)$ до $n = 10\,000$.

[†]Фигурата е направена с Maple(TM).

[‡]Фигурата е направена с Maple(TM).



Импликацията (3.63) обаче е вярна: ако условието за регулярност е изпълнено, то $f(n)$ расте полиномиално по-бързо от $n^{\log_b a}$. Доказателство на това се съдържа в Допълнение 22. Възниква резонен въпрос: защо **Случай 3** на МТ не е формулиран само чрез условието за регулярност, щом то е строго по-силно? Според автора на записките, Cormen, Leiserson и Rivest са искали трите случая на МТ да имат сходни формулировки и да е кристално ясно, че във всеки от тях сравняваме $f(n)$ с $n^{\log_b a}$. Както стана ясно, функциите $f(n)$, които растат полиномиално по-бързо от $n^{\log_b a}$, но за които условието за регулярност не е в сила, едва ли биха възникнали в практиката.

Допълнение 22: Условието за регулярност в МТ влече Случай 3

Задача 4.4-3 в [34] е: покажете, че Случай 3 на МТ съдържа излишък в смисъл, че условието за регулярност $af\left(\frac{n}{b}\right) \leq cf(n)$ за някаква положителна константа $c < 1$ влече съществуването на константа $\epsilon > 0$, такава че $f(n) = \Omega(n^{(\log_b a) + \epsilon})$. Сега ще покажем точно това. Нека съществува константа c , такава че $0 < c < 1$ и:

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right)$$

Тъй като $a \geq 1$ и $c < 1$, то $\frac{a}{c} > 1$. За краткост, ще означаваме $\frac{a}{c}$ с буквата s . Тогава в сила е:

$$f(n) \geq sf\left(\frac{n}{b}\right)$$

Тогава

$$f(n) \geq s^2 f\left(\frac{n}{b^2}\right)$$

$$f(n) \geq s^3 f\left(\frac{n}{b^3}\right)$$

...

$$f(n) \geq s^t f\left(\frac{n}{b^t}\right) \tag{3.65}$$

Съществува някаква стойност n_0 на променливата n , за която този процес ще спре. Тогава $\frac{n}{b^t} = n_0$, следователно $t = \log_b \left(\frac{n}{n_0}\right) = \log_b n - \log_b n_0$. Заместяваме t и $\frac{n}{b^t}$ и c в

(3.65) и получаваме:

$$f(n) \geq s^{\log_b n - \log_b n_0} \times f(n_0) = \frac{s^{\log_b n}}{s^{\log_b n_0}} \times f(n_0)$$

Забележете, че $\frac{1}{s^{\log_b n_0}} \times f(n_0)$ е константа. За краткост, да наречем тази константа β . Тогава:

$$f(n) \geq s^{\log_b n} \times \beta$$

Да се върнем към началните имена a и c , като полагаме $z = \frac{1}{c}$, за да е изпълнено $z > 1$. Имаме $s = a \times z$.

$$f(n) \geq a^{\log_b n} \times z^{\log_b n} \times \beta$$

Имайки предвид, че $a^{\log_b n} = n^{\log_b a}$ and $z^{\log_b n} = n^{\log_b z}$, извеждаме:

$$f(n) \geq \beta \times n^{\log_b a} \times n^{\log_b z}$$

Тъй като $z > 1$, то $\log_b z > 0$. Нека $\epsilon = \log_b z$. Извеждаме желанния резултат: за всички достатъчно големи n и някаква константа β :

$$f(n) \geq \beta n^{(\log_b a) + \epsilon} \Rightarrow f(n) = \Omega(n^{(\log_b a) + \epsilon})$$

□

И така, **Случай 3** на МТ би могъл да бъде формулиран само с условието за регулярност, а условието $f(n) = \Omega(n^{\log_b a + \epsilon})$ е излишно. Това е и краят на поясненията върху МТ.

Допълнение 23: Ограничена МТ следва от метода с хар-чното у-ние

Ще покажем, че МТ, но в ограничен вариант, е следствие на метода с характеристичното уравнение (Подсекция 3.2.6). Разглеждаме

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Ще направим субституция на n с b^m . Тоест, $m = \log_b n$.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \leftrightarrow$$

$$T(b^m) = aT\left(\frac{b^m}{b}\right) + f(b^m) \leftrightarrow$$

$$T(b^m) = aT(b^{m-1}) + f(b^m)$$

Ще направим две субституции на функции.

- Ще заменим $T(b^m)$ с някаква $S(m)$. Имаме право да го направим, понеже $T(b^m)$ е функция на m , при положение, че b е константа. Щом $T(b^m)$ става $S(m)$, то $T(b^{m-1})$ става $S(m-1)$.
- С аналогични съображения, заменяме $f(b^m)$ с някаква $g(m)$.

Вече ползвахме тази техника (Лема 17 и Следствие 15): при деление с някаква константа правим субституция с експонента с основа тази константа, при което делението става изваждане в степенния показател.

И така, получихме

$$S(m) = aS(m-1) + g(m)$$

Но това уравнение е решимо с метода с характеристичното уравнение (Подсекция 3.2.6), ако нехомогенната част $g(m)$ е от подходящ вид, а именно

$$g(m) = p_1(m)c_1^m + \dots + p_t(m)c_t^m \quad (3.66)$$

където p_i са полиноми, а c_i са константи, такива че $i \neq j \rightarrow c_i \neq c_j$. Да допуснем, че (3.66) е изпълнено. Тогава решението е елементарно. Хомогенната част

$$S_{\text{хом}}(m) = aS_{\text{хом}}(m-1)$$

дава мултимножество от корени $\{a\}_M$. От нехомогенната част идва мултимножество от корените

$$\underbrace{\{c_1, \dots, c_1\}}_{\deg(p_1)+1}, \dots, \underbrace{\{c_t, \dots, c_t\}}_{\deg(p_t)+1} \}_M$$

За определяне на асимптотиката на решението нас ни интересува само най-голямата константа. БОО, нека най-голямата константа е c_1 и нека $d = \deg(p_1) + 1$ е броят на срещанията на c_1 в мултимножеството от корените. Тогава, съгласно (3.82), решението е

$$S(m) \asymp \begin{cases} a^m, & \text{ако } a > c_1 \\ m^d \cdot a^m, & \text{ако } a = c_1 \\ m^{d-1} \cdot c_1^m. & \text{ако } a < c_1 \end{cases} \quad (3.67)$$

Основанието е ясно:

- ако $a > c_1$, то най-големият елемент на мултимножеството от корените е a и той има кратност 1 и асимптотиката се дава от $m^{1-1} \cdot a^m$, което е a^m ;
- ако $a = c_1$, то най-големият елемент на мултимножеството от корените е $a = c_1$ и той има кратност $d + 1$, а не d , така че асимптотиката се дава от $m^{(d+1)-1} \cdot a^m$, което е $m^d \cdot a^m$;
- ако $c_1 > a$, то най-големият елемент на мултимножеството от корените е c_1 и той има кратност d , така че асимптотиката на решението се дава от $m^{d-1} \cdot c_1^m$.

Забележете, че в последния случай, $d - 1$ е равно на $\deg(p_1)$, а $m^{\deg(p_1)} \cdot c_1^m$ задава асимптотиката на нехомогенната част (помним, че c_1 е най-голямата от константите-основи на експоненти в нехомогенната част). Ерго, $m^{d-1} \cdot c_1^m \asymp g(m)$. Това ни дава

основание да твърдим, че

$$S(m) \asymp \begin{cases} a^m, & \text{ако } a > c_1 \\ m^d \cdot a^m, & \text{ако } a = c_1 \\ g(m). & \text{ако } a < c_1 \end{cases} \quad (3.68)$$

Да се върнем към оригиналната променлива n и функция $T(n)$, помнейки, че $m = \log_b n$, $S(m) = T(n)$ и $g(m) = f(n)$:

$$T(n) \asymp \begin{cases} a^{\log_b n}, & \text{ако } a > c_1 \\ (\log_b n)^d \cdot a^{\log_b n}, & \text{ако } a = c_1 \\ f(n). & \text{ако } a < c_1 \end{cases} \quad (3.69)$$

Предвид това, че $a^{\log_b n} = n^{\log_b a}$, можем да твърдим, че

1. $T(n) \asymp n^{\log_b a}$, ако $a > c_1$,
2. $T(n) \asymp (\log_b n)^d \cdot n^{\log_b a}$, ако $a = c_1$,
3. $T(n) \asymp f(n)$, ако $a < c_1$.

Но това е същото като МТ (Теорема 41), на която вторият случай е засилен чрез обобщението на МТ (Теорема 42), но **при допускането**, че $f(n)$ е такава, че

$$f(n) = p_1(\log_b n) \cdot c_1^{\log_b n} + \dots + p_t(\log_b n) \cdot c_t^{\log_b n} \quad (3.70)$$

Тоест, ако $f(n)$ е сума от функции, всяка от които е произведение от полилогаритмична функция $p_i(\log_b n)$ и полиномиална функция $c_i^{\log_b n} = n^{\log_b c_i}$. За да се убедим, че (3.66) и (3.70) казват едно и също, да си припомним, че $m = \log_b n$ и $g(m) = f(b^m) = f(n)$.

Предвид това, че събираемостта с най-голямата константа c_i задава асимптотиката на $f(n)$, веднага виждаме, че случаите на МТ (Теорема 41) точно отговарят на тези случаи: **Случай 1** отговаря на (1), **Случай 2** отговаря на (2) и **Случай 3** отговаря на (3). Условието за регулярност задължително е изпълнено, ако $f(n)$ е от вида (3.70), така че няма нужда да го споменаваме в (3).

Нещо повече. Дори $f(n)$ да е по-бързо растяща функция от това, което (3.70) казва—с други думи, ако $f(n)$ не е от вида (3.70), но е суперполиномиална—асимптотиката на $T(n)$ пак се дава от $f(n)$. Така че резултата от Допълнение 23 покрива всяка $f(n)$, която би се срещнала на практика в изследване на рекурсивни алгоритми, чиято сложност се описва от $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

В резюме, извеждането чрез метода с характеристичното уравнение на МТ е достатъчно добро за целите на курса. Функциите $f(n)$, които хем остават извън условието, наложено от (3.70), хем не са суперполиномиални, са екзотика, която едва ли ще се срещне при намиране на сложността на алгоритъм, който е полезен на практика, а не е чисто мисловна конструкция.

Ще видим няколко примера за прилагането на МТ.

Задача 34

Решете чрез МТ $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$.

Решение: Използвайки терминологията на МТ, a е 3, b е 4, така че $\log_b a$ е $\log_4 3$, което е приблизително 0.79, и $f(n)$ е $n \lg n$. Със сигурност е вярно, че $n \lg n = \Omega(n^{\log_4 3 + \epsilon})$ за някое $\epsilon > 0$, например $\epsilon = 0.1$. Но трябва да проверим и дали условието за регулярност е в сила, за да видим дали може да приложим третия случай на МТ. Условието за регулярност е:

$$\exists c \text{ такова че } 0 < c < 1 \text{ и } 3 \frac{n}{4} \lg \frac{n}{4} \leq cn \lg n \text{ за всички достатъчно големи } n$$

Очевидно това е вярно за, да речем, $c = \frac{3}{4}$, така че третият случай на МТ е приложим. Съгласно него, $T(n) \asymp n \lg n$. $\square$

Задача 35

Решете чрез МТ $T(n) = T\left(\frac{2n}{3}\right) + 1$.

Решение: Преписваме условието така:

$$T(n) = 1 \cdot T\left(\frac{n}{\frac{3}{2}}\right) + 1$$

за да е по-ясно какви са a и b . Видваме, че a е 1, b е $\frac{3}{2}$, така че $\log_b a$ е $\log_{\frac{3}{2}} 1 = 0$, следователно $n^{\log_b a}$ е $n^0 = 1$. Нехомогенната част $f(n)$ е 1. Очевидно, $1 = \Theta(n^0)$, така че вторият случай на МТ е приложим. Съгласно него, $T(n) \asymp \lg n$. $\square$

Задача 36

Решете чрез МТ $T(n) = 4T\left(\frac{n}{2}\right) + n$.

Решение: Тук $a = 4$, $b = 2$, така че $\log_b a$ е $\log_2 4 = 2$, следователно $n^{\log_b a}$ е n^2 . Нехомогенната част $f(n)$ е n . Сравняваме по асимптотично нарастване n с n^2 . Очевидно, $n = O(n^2/n^\epsilon)$ за някое $\epsilon > 0$, така че първият случай на МТ е приложим. Съгласно него, $T(n) \asymp n^2$. $\square$

Допълнение 24: Едно разширение на Теорема 41**Теорема 42: Разширение на МТ**

При допусканията на Теорема 41, нека

$$f(n) = \Theta(n^{\log_b a} \lg^t n) \tag{3.71}$$

за някое $t \in \mathbb{N}^+$. Тогава $T(n) = \Theta(n^{\log_b a} \lg^{t+1} n)$.

Доказателство: Теорема 41 не е директно приложима, защото рекурентното уравнение с тази функция $f(n)$ при $t \geq 1$ не може да бъде класифицирано в нито един от трите случая. Решаваме задачата с развиване. За простота допусκαем, че n е точна степен на b , тоест $n = b^m$ за някакво цяло число $m > 0$. За читателя остава да обобщи

доказателството за всяко положително n .

Да допуснем, че логаритъмът в (3.71) е с основа b . Забелязваме, че всичко в Θ -нотацията в дясната страна на (3.71) може да се запише така:

$$n^{\log_b a} \log_b^t n = b^{(m \log_b a)} m^t = b^{(\log_b a^m)} m^t = a^m m^t \quad (3.72)$$

Тогава (3.71) е еквивалентно на

$$c_1 a^m m^t \leq f(b^m) \leq c_2 a^m m^t$$

за някакви положителни константи c_1 и c_2 и всички достатъчно големи стойности на m . За простота, ще допуснем, че

$$f(b^m) = a^m m^t \quad (3.73)$$

Не е трудно доказателството да бъде направено по същия начин и без това опростяване.

От условието на МТ имаме $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Използвайки (3.73), записваме:

$$\begin{aligned} T(b^m) &= aT\left(\frac{b^m}{b}\right) + a^m m^t \\ &= aT(b^{m-1}) + a^m m^t \leftrightarrow \\ S(m) &= aS(m-1) + a^m m^t \quad \text{замествайки } T(b^m) \text{ със } S(m) \\ &= a(aS(m-2) + a^{m-1}(m-1)^t) + a^m m^t \\ &= a^2 S(m-2) + a^m(m-1)^t + a^m m^t \\ &= a^2(aS(m-3) + a^{m-2}(m-2)^t) + a^m(m-1)^t + a^m m^t \\ &= a^3 S(m-3) + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\ &\dots \\ &= a^{m-1} S(1) + a^m 2^t + a^m 3^t + \dots + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\ &= a^{m-1} S(1) + a^m(2^t + 3^t + \dots + (m-2)^t + (m-1)^t + m^t) \end{aligned}$$

Но $2^t + 3^t + \dots + (m-2)^t + (m-1)^t + m^t = \Theta(m^{t+1})$; за извеждането на това вижте [61, (6.98) на стр. 288]. Тогава в сила е

$$\begin{aligned} a^{m-1} S(1) + a^m(2^t + 3^t + \dots + (m-2)^t + (m-1)^t + m^t) &= \\ a^{m-1} S(1) + a^m \Theta(m^{t+1}) &= \\ \Theta(a^m m^{t+1}) \end{aligned}$$

Изведохме:

$$S(m) = \Theta(a^m m^{t+1}) \quad (3.74)$$

Да се върнем към началните T и n . (3.74) става

$$T(n) = \Theta(a^{\log_b n} (\log_b n)^{t+1})$$

Имайки предвид, че $a^{\log_b n} = n^{\log_b a}$ и $\log_b n = \Theta(\lg n)$, заключаваме, че:

$$T(n) \asymp n^{\log_b a} \lg^{t+1} n$$



Задача 37

Решете $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$.

Решение: Да опитаме с МТ. Имаме $a = 2$, $b = 2$, така че $\log_b a = \log_2 2 = 1$, следователно $n^{\log_b a} = n^1 = n$. Нехомогенната част е $n \lg n$. Да видим дали можем да класифицираме тази задача в някой от трите случая на МТ.

Опитваме случай 1: Дали $n \lg n = O(n^1/n^\epsilon)$ за някое $\epsilon > 0$? Не, защото $n \lg n = \omega(n^1)$.

Опитваме случай 2: Дали $n \lg n = \Theta(n^1)$? Не, защото $n \lg n = \omega(n^1)$.

Опитваме случай 3: Дали $n \lg n = \Omega(n^1 \cdot n^\epsilon)$ за някое $\epsilon > 0$? Не. Вижте Теорема 29.

Следователно, тази задача не може да се реши с МТ. Ще я решим чрез Теорема 42, съгласно която $T(n) \asymp n \lg^2 n$. □

3.2.5 Чрез теоремата на Akra-Bazzi

МТ има съществено ограничение: тя решава рекурентни уравнения с **една поява вдясно**, които описват **Разделяй-и-Владей** алгоритми, които винаги делят входа **по един и същи начин** (преди да достигнат началните условия). Но това означава, че дори прости **Разделяй-и-Владей** алгоритми, които делят по други начини, например SELECT в Подсекция 6.2.2, не се поддават на решаване с МТ. Рекурентното уравнение (6.6), което описва сложността на SELECT, е решено по индукция в [34, стр. 222], защото [34] не предлага друг формален метод, освен индукцията, за решаване на (6.6).

Сега ще разгледаме една теорема, която е строго по-мощна от МТ и чрез която може да се намери сложността по време на голям клас от естествени **Разделяй-и-Владей** алгоритми, включително и на SELECT. Авторите са Mohamad Akra и Louay Bazzi, а статията [4] е от 1996 г. В тези лекционни записки доказателството на теоремата е изложено в Допълнение 25, понеже е сравнително сложно. Теоремата, както и доказателството ѝ, са представени съгласно статията на Leighton [105][†]. Заслужава да се спомене непубликуваната статия на Neville Campbell [29] от над 300 (триста!) страници, която твърди, че доказателствата на Leighton съдържа дребни грешки от непрецизност, които тя (твърди, че) поправя.

[†]Книгата на Cormen, Leiserson, Rivest [34] споменава теоремата на Akra-Bazzi, но само в края на главата за МТ (стр. 112–113), без да са решени никакви задачи с нея.

Теорема 43: Теорема на Акра-Bazzi (Theorem 1 от [105])

Нека

$$T(x) = \begin{cases} \Theta(1), & \text{ако } 1 \leq x \leq x_0 \\ \sum_{i=1}^k a_i T(b_i x) + g(x), & \text{ако } x > x_0 \end{cases} \quad (3.75)$$

където

1. $x \geq 1$ е реално число,
2. x_0 е константа, такава че $x_0 \geq \frac{1}{b_i}$ и $x_0 \geq \frac{1}{1-b_i}$, за $1 \leq i \leq k$,
3. $a_i > 0$ е константа за $1 \leq i \leq k$,
4. $b_i \in (0, 1)$ е константа за $1 \leq i \leq k$,
5. $k \geq 1$ е константа и
6. $g(x)$ е неотрицателна функция, удовлетворяваща условието за полиномиалното нарастване.
7. p е уникалното число, за което $\sum_{i=1}^k a_i b_i^p = 1$.

Тогава

$$T(x) = x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right)$$

Определение 41: Условието за полиномиалното нарастване от [105]

В контекста на Теорема 43 казваме, че $g(x)$ удовлетворява *условието за полиномиалното нарастване*, ако съществуват положителни константи c_1 и c_2 , такива че за всяко $x \geq 1$, за всяко $1 \leq i \leq k$, за всяко $u \in [b_i x, x]$:

$$c_1 g(x) \leq g(u) \leq c_2 g(x)$$

На английски е *the polynomial growth condition*.

Трудно е да се каже **точно** кои функции удовлетворяват условието за полиномиалното нарастване. Ако f е функция, удовлетворяваща условието за полиномиалното нарастване, то има константи $k \in \mathbb{N}$ и $\alpha, \beta \in \mathbb{R}^+$, такива че $\alpha x^{-k} \leq f(x) \leq \beta x^k$ [29, Lemma 2.34], но конверсното не е непременно вярно [29, стр. 58].

Следните функции:

- полиномите x^α , където α е реална константа,
- логаритмите $\log_b x$, където b е реална константа, като $b > 1$,
- $\lfloor x \rfloor$,
- $\lceil x \rceil$

са примери за функции, удовлетворяващи условието за полиномиалното нарастване [29, Lemma 4.1]. Сумата и произведението на функции, удовлетворяващи условието за полиномиалното нарастване, удовлетворяват условието за полиномиалното нарастване [29, Corollary 4.3]. Също така и частното, стига в знаменателя да има положителна функция. Композицията на функции също запазва условието за полиномиалното нарастване [29, Lemma 4.6].

Една заключителна забележка. Ако $g(x)$ расте суперполиномиално, то тя задава асимптотиката, така че $T(x) \asymp g(x)$. Но това не се получава от Теорема 43, защото в такъв случай $g(x)$ не удовлетворява условието за полиномиалното нарастване.

Допълнение 25: Доказателство на теоремата на Akra-Bazzi

Лема 19: Lemma 1 от [105]

Ако $g(x)$ е неотрицателна функция, удовлетворяваща условието за полиномиалното нарастване, то съществуват положителни константи c_3 и c_4 , такива че за всяко $1 \leq i \leq k$ и всяко $x \geq 1$ е вярно, че

$$c_3 g(x) \leq x^p \int_{b_i x}^x \frac{g(u)}{u^{p+1}} du \leq c_4 g(x)$$

Доказателство: От условието за полиномиалното нарастване имаме:

$$\begin{aligned} \int_{b_i x}^x \frac{g(u)}{u^{p+1}} du &\leq x^p (x - b_i x) \frac{c_2 g(x)}{\min \{(b_i x)^{p+1}, x^{p+1}\}} \\ &= \frac{(1 - b_i) c_2 g(x)}{\min \{1, b_i^{p+1}\}} \\ &\leq c_4 g(x) \end{aligned}$$

където c_4 е константа, такава че

$$c_4 \geq \frac{(1 - b_i) c_2}{\min \{1, b_i^{p+1}\}}$$

за $1 \leq i \leq k$.

Аналогично,

$$\begin{aligned} \int_{b_i x}^x \frac{g(u)}{u^{p+1}} du &\geq x^p (x - b_i x) \frac{c_1 g(x)}{\max \{(b_i x)^{p+1}, x^{p+1}\}} \\ &= \frac{(1 - b_i) c_1 g(x)}{\max \{1, b_i^{p+1}\}} \\ &\geq c_3 g(x) \end{aligned}$$

където c_3 е константа, такава че

$$c_3 \leq \frac{(1 - b_i) c_1}{\max \{1, b_i^{p+1}\}}$$

за $1 \leq i \leq k$.



Ето отново теоремата.

Теорема: Theorem 1 от [105]

Дадено е рекурентното уравнение (3.75)

$$T(x) = \begin{cases} \Theta(1), & \text{ако } 1 \leq x \leq x_0 \\ \sum_{i=1}^k a_i T(b_i x) + g(x), & \text{ако } x > x_0 \end{cases}$$

с ограниченията и имената, посочени след (3.75). Тогава

$$T(x) \asymp x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right)$$

Доказателство: Разбиваме домейна, от който взема стойности x , на интервали

$$I_0 = [1, x_0]$$

$$I_j = (x_0 + j - 1, x_0 + j] \quad \text{за } j \geq 1$$

Ако $x \in I_j$ за някое $j \geq 1$, то за всяко i , такова че $1 \leq i \leq k$ е вярно, че $b_i x \in I_{j'}$ за някое $j' < j$. Това е така, понеже $b_i x > b_i(x_0 + j - 1) \geq b_i x_0 \geq 1$, и освен това $b_i x \leq b_i(x_0 + j) \leq x_0 + j - (1 - b_i)x_0 \leq x_0 + j - 1$. Следователно, стойността на T в даден интервал след нулевия зависи само от стойностите на T от предишни интервали. Следователно, можем да опитаме да докажем теоремата със силна индукция по j .

Първо ще покажем, че има положителна константа c_5 , такава че за всяко $x > x_0$ е вярно, че

$$T(x) \geq c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right)$$

Доказателството е по индукция по интервала I_j , съдържащ x . Базовият случай е $j = 0$. Там твърдението следва от факта, че $T(x) = \Theta(1)$ върху I_0 . В индуктивната стъпка

процедираме така:

$$\begin{aligned}
 T(x) &= \sum_{i=1}^k a_i T(b_i x) + g(x) \quad // \text{от индуктивното предположение} \\
 &\geq \sum_{i=1}^k a_i c_5 (b_i x)^p \left(1 + \int_1^{b_i x} \frac{g(u)}{u^{p+1}} du \right) + g(x) \\
 &= c_5 x^p \sum_{i=1}^k a_i b_i^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \int_{b_i x}^x \frac{g(u)}{u^{p+1}} du \right) + g(x) \quad // \text{от Лема 19} \\
 &\geq c_5 x^p \sum_{i=1}^k a_i b_i^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \frac{c_4 g(x)}{x^p} \right) + g(x) \\
 &= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \frac{c_4 g(x)}{x^p} \right) \left(\sum_{i=1}^k a_i b_i^p \right) + g(x) \quad // \text{защото } \sum_{i=1}^k a_i b_i^p = 1 \\
 &= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du - \frac{c_4 g(x)}{x^p} \right) + g(x) \\
 &= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right) + g(x)(1 - c_4 c_5) \\
 &\geq c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right), \quad \text{ако } (c_4 c_5 < 1)
 \end{aligned}$$

Доказателството, че има положителна константа c_6 , такава че за всяко $x > x_0$ е вярно, че

$$T(x) \leq c_6 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right),$$

е доста подобно. Трябва само да сме сигурни, че c_6 е достатъчно голяма, така че в базовия случай твърдението да е вярно, и освен това $c_3 c_6 \geq 1$.

Това е и края на доказателството. □

Задача 38: от [4]

Решете чрез метода на Акра-Баззи $T(n) = \frac{4}{3}T\left(\frac{n}{2}\right) + 3T\left(\frac{n}{3}\right) + \frac{16}{3}T\left(\frac{n}{4}\right) + \Theta(n^2 \lg \lg n)$.

Решение: Първо да намерим p като решението на

$$\frac{4}{3} \left(\frac{1}{2}\right)^p + 3 \left(\frac{1}{3}\right)^p + \frac{16}{3} \left(\frac{1}{4}\right)^p = 1$$

Намираме $p = 2$.

Тогава,

$$\begin{aligned} T(n) &\asymp n^2 \left(1 + \int_1^n \frac{u^2 \ln \ln u}{u^3} du \right) \\ &= n^2 \left(1 + \int_1^n \frac{\ln \ln u}{u} du \right) \\ &= n^2 (1 + \ln n (\ln \ln n - 1)) \\ &\asymp n^2 \cdot \ln n \cdot \ln \ln n \end{aligned}$$

Задача 39: от [105]

Решете чрез метода на Акга-Bazzi $T(x) = \frac{1}{2}T\left(\frac{x}{2}\right) + \Theta\left(\frac{1}{x}\right)$.

Решение: Това уравнение не може да описва сложност на алгоритъм, но може да се реши с метода на Акга-Bazzi. Първо да намерим p като решението на

$$\frac{1}{2} \left(\frac{1}{2} \right)^p = 1$$

Очевидно $p = -1$. Тогава,

$$\begin{aligned} T(n) &\asymp x^{-1} \left(1 + \int_1^x \frac{\frac{1}{u}}{u^0} du \right) \\ &= \frac{1}{x} \left(1 + \int_1^x \frac{1}{u} du \right) \\ &= \frac{1}{x} (1 + \ln x) \\ &\asymp \frac{\ln x}{x} \end{aligned}$$

3.2.6 Чрез метода с характеристичното уравнение

Теорема 44: Решение на линейно хомогенно рекурентно уравнение

Нека редицата $\mathbf{a} = (a_0, a_1, a_2, \dots)$ е зададена с линейното рекурентно уравнение

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_{r-1} a_{n-(r-1)} + c_r a_{n-r}, \quad n \geq r \quad (3.76)$$

където $c_1, \dots, c_r$ са константи, като $c_r \neq 0$. Нека $\alpha_1, \alpha_2, \dots, \alpha_s$ са различните (комплексни) корени на *характеристичното уравнение*

$$x^r - c_1 x^{r-1} - c_2 x^{r-2} - \dots - c_{r-1} x - c_r = 0 \quad (3.77)$$

Нека α_i има кратност k_i . Тогава

$$a_n = P_1(n) \alpha_1^n + P_2(n) \alpha_2^n + \dots + P_s(n) \alpha_s^n \quad (3.78)$$

където $P_i(n)$ е полином на n от степен $< k_i$. Полиномите $P_i(n)$ имат общо r коефициента, които се определят еднозначно от първите r члена на редицата $\mathbf{a}$. $\square$

Забележете, че $k_1 + k_2 + \dots + k_s = r$, което следва веднага от **основната теорема на алгебрата**.

В терминологията на тези лекционни записки, (3.76) изглежда така:

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \quad (3.79)$$

Но това е хомогенно рекурентно уравнение. Такова рекурентно уравнение не може да описва сложността на рекурсивен алгоритъм, защото алгоритъмът не може да не извършва поне константна работа извън рекурсивните викания. Понеже се интересуваме от анализ на алгоритми, налага се да разгледаме нехомогенни рекурентни уравнения. Преди това, едно определение.

Понятието “мултимножество” спокойно може да бъде избегнато, като вместо това се говори за обикновено множество и кратност на елементите, но авторът на записките намира изразяването чрез “мултимножество” за много ясно и естествено.

Определение 42: Мултимножество: нотация, кратност, обединение, кардиналност

С нотацията “ $\{ \}_M$ ” означаваме мултимножествата. Например, $\{1, 1, 2, 3, 3, 3\}_M$ е мултимножеството, състоящо се от две единици, една двойка и две тройки. За всеки елемент a , с $\#(a, A)$ означаваме броя на появите на a в A , който брой наричаме *кратността на a в A* . $\#(a, A) = 0$, ако a не се среща в A . Например, $\#(1, \{1, 1, 2, 3, 3, 3\}_M) = 2$. Обединението на две мултимножества A и B е:

$$A \cup B = \{x \mid (x \in A \text{ или } x \in B) \text{ и } (\#(x, A \cup B) = \#(x, A) + \#(x, B))\}_M$$

Например, $\{1, 1, 3, 5, 5, 5\}_M \cup \{2, 4, 4, 5, 5, 6\}_M = \{1, 1, 2, 3, 4, 4, 5, 5, 5, 5, 5, 6\}_M$.

Кардиналността на мултимножеството A е броят на елементите му с отчитане на повторенията. Тя се означава с $|A|$. Например, $|\{1, 1, 2, 3, 3, 3\}_M| = 6$.

Теорема 45: Решение на линейно нехомогенно рекурентно уравнение

Разглеждаме уравнението:

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_{r-1} T(n-(r-1)) + c_r T(n-r) + \beta_1^n Q_1(n) + \beta_2^n Q_2(n) + \dots + \beta_m^n Q_m(n) \quad (3.80)$$

където $c_1, \dots, c_r$ са константи, като $c_r \neq 0$, $m \in \mathbb{N}^+$, $\beta_1, \beta_2, \dots, \beta_m$ са две по две различни положителни константи, а $Q_1(n), Q_2(n), \dots, Q_m(n)$ са полиноми от степени съответно $d_1, d_2, \dots, d_m$.

Нека мултимножеството от корените на характеристичното уравнение е A . Очевидно, $|A| = r$. Нека B е мултимножеството от всички β_i , като $\#(\beta_i, B) = d_i + 1$, за $1 \leq i \leq m$. Нека $Y = A \cup B$. Очевидно, $|Y| = r + \sum_{i=1}^m (d_i + 1)$. Нека преименуваме различните елементи на Y като $y_1, y_2, \dots, y_t$ и да дефинираме, че $z_i = \#(y_i, Y)$, за $1 \leq i \leq t$. Тогава общото решение на (3.80) е:

$$\begin{aligned} T(n) = & \gamma_{1,1} y_1^n + \gamma_{1,2} n y_1^n + \gamma_{1,3} n^2 y_1^n + \dots + \gamma_{1,z_1} n^{z_1-1} y_1^n + \\ & \gamma_{2,1} y_2^n + \gamma_{2,2} n y_2^n + \gamma_{2,3} n^2 y_2^n + \dots + \gamma_{2,z_2} n^{z_2-1} y_2^n + \\ & \dots \\ & \gamma_{t,1} y_t^n + \gamma_{t,2} n y_t^n + \gamma_{t,3} n^2 y_t^n + \dots + \gamma_{t,z_t} n^{z_t-1} y_t^n, \end{aligned} \quad (3.81)$$

където $\gamma_{i,j}$, за $1 \leq i \leq t$ и $1 \leq j \leq z_t$, са константи, общо $|Y|$ на брой. В общото решение те са неизвестни константи. Можем да ги намерим от началните условия (като добавим $|B|$ на брой нови начални условия) и тогава ще разполагаме с точното решение.

Следствие 17: Асимптотиката на общото решение от Теорема 45

Понеже ни интересува само асимптотичното нарастване на $T(n)$, точните стойности на константите $\gamma_{i,j}$ са без значение. Нещо повече. Дори само общото решение е прекалено подробно. Асимптотичното нарастване се определя от **точно едно събираемо** в общото решение – събираемостта с максималното y_i , умножено по максималната степен на n , която се среща като множител на y_i^n . Накратко, ако $y_i = \max \{y_1, \dots, y_t\}$, то

$$T(n) \asymp n^{z_i-1} y_i^n \quad (3.82)$$

Забележете, че ако $T(n)$ описва сложността на алгоритъм, най-голямото по абсолютна стойност y_i задължително е положително.

Допълнение 26: Извеждане на решението на линейните рек. у-ния

Ще изведем решението съгласно учебника *An Introduction to Difference Equations* на Elaydi [43, стр. 75]. Първо ще въведем две понятия и после ще видим как с тяхна помощ извеждаме решение.

Първо понятие: оператор. Нека n е дискретна променлива; по-точно, $n \in \mathbb{N}$. Нека $\mathcal{G}$ е множеството от функции

$$\mathcal{G} = \{g \mid g: \mathbb{N} \rightarrow \mathbb{R}\}$$

Оператор, за целите на това изложение, е функция с домейн и кодомейн $\mathcal{G}$; ерго, оператор е функция, изобразяваща функция във функция. Нека x е произволна функция от $\mathcal{G}$. Въвеждаме операторите Δ (оператор разлика) и E (оператор изместване):

$$\Delta x(n) = x(n+1) - x(n) \quad (3.83)$$

$$Ex(n) = x(n+1) \quad (3.84)$$

Примерно, ако $x(n) = n$, резултатът от действието на двата оператора е:

n	0	1	2	3	...	n	...
$x(n)$	0	1	2	3	...	n	...
$\Delta x(n)$	1	1	1	1	...	1	...
$Ex(n)$	1	2	3	4	...	$n+1$	...

Въвеждаме и съответните *итерирани оператори*:

$$\Delta^k x(n) = \begin{cases} x(n), & \text{ако } k = 0 \\ \Delta(\Delta^{k-1} x(n)), & \text{ако } k > 0 \end{cases}$$

$$E^k x(n) = \begin{cases} x(n), & \text{ако } k = 0 \\ E(E^{k-1} x(n)), & \text{ако } k > 0 \end{cases}$$

Ето как изглеждат Δ , Δ^2 , Δ^3 , E , E^2 и E^3 за $x(n) = n$:

n	0	1	2	3	...	n	...
$x(n)$	0	1	2	3	...	n	...
$\Delta x(n)$	1	1	1	1	...	1	...
$\Delta^2 x(n)$	0	0	0	0	...	0	...
$\Delta^3 x(n)$	0	0	0	0	...	0	...
$Ex(n)$	1	2	3	4	...	$n+1$	...
$E^2 x(n)$	2	3	4	5	...	$n+2$	...
$E^3 x(n)$	3	4	5	6	...	$n+3$	...

В някакъв смисъл, операторът Δ е аналог на оператора D (производна) от математическия анализ:

$$Df(y) = \lim_{h \rightarrow 0} \frac{f(y+h) - f(y)}{h}$$

Също както в континуалния анализ, ако $f(y) = y$, то $Df(y) = 1$, а $D^2 f(y) = 0$, в дискретния анализ е вярно, че, ако $x(n) = n$, то $\Delta x(n) = 1$, а $\Delta^2 x(n) = 0$.

И операторът разлика, и операторът изместване са линейни оператори. Ако $a, b \in \mathbb{R}$, а $x, y \in \mathcal{G}$, в сила са:

$$\Delta(ax(n) + by(n)) = a\Delta x(n) + b\Delta y(n)$$

$$E(ax(n) + by(n)) = aEx(n) + bEy(n)$$

По дефиниция, $\mathbf{I}$ е операторът идентитет:

$$\mathbf{I}x(n) = x(n)$$

Вярно е, че

$$\begin{aligned}\mathbf{E} &= \Delta + \mathbf{I} \\ \Delta &= \mathbf{E} - \mathbf{I}\end{aligned}\tag{3.85}$$

Итерираният оператор изместване е лесен за разбиране. Тривиално се показва по индукция, че

$$\forall k \in \mathbb{N} : \mathbf{E}^k x(n) = x(n + k)$$

Итерираният оператор разлика е малко по-триков. За Δ^2 имаме:

$$\begin{aligned}\Delta^2 x(n) &= \Delta(\Delta x(n)) \\ &= \Delta(x(n+1) - x(n)) \\ &= \Delta x(n+1) - \Delta x(n) \\ &= x(n+2) - x(n+1) - x(n+1) + x(n) \\ &= x(n+2) - 2x(n+1) + x(n)\end{aligned}$$

За Δ^3 имаме:

$$\begin{aligned}\Delta^3 x(n) &= \Delta(\Delta^2 x(n)) \\ &= \Delta(x(n+2) - 2x(n+1) + x(n)) \\ &= \Delta x(n+2) - 2\Delta x(n+1) + \Delta x(n) \\ &= (x(n+3) - x(n+2)) - 2(x(n+2) - x(n+1)) + (x(n+1) - x(n)) \\ &= x(n+3) - 3x(n+2) + 3x(n+1) - x(n)\end{aligned}$$

Можем да покажем по индукция, че

$$\Delta^k x(n) = \sum_{i=0}^k (-1)^i \binom{k}{i} x(n + k - i)$$

Действието на оператора Δ върху полиноми е аналогично на действието на оператора $\mathbf{D}$ от континуалния анализ: ако полиномът е от степен $k \geq 1$, то еднократното прилагане на Δ “сваля” степента с единица, k -кратното прилагане на Δ дава $a_k \cdot k!$, където a_k е коефициентът пред старшия член, а оттам $(k+i)$ -кратното на Δ прилагане за $i \geq 1$ дава нула. Наистина, нека

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$$

където $k \geq 1$ и $a_k, \dots, a_0$ са константи; ерго, $\deg p(n) = k$. Тогава

$$\begin{aligned}
 \Delta p(n) &= (a_k(n+1)^k + a_{k-1}(n+1)^{k-1} + \dots + a_1(n+1) + a_0) \\
 &\quad - (a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0) = \\
 &= \underbrace{a_k n^k + a_k k n^{k-1} + \dots + a_k}_{a_k(n+1)^k} + \underbrace{a_{k-1}(n+1)^{k-1} + \dots + a_1(n+1) + a_0}_{\text{от степен } k-1} \\
 &\quad - a_k n^k - a_{k-1} n^{k-1} - \dots - a_1 n - a_0 = \\
 &= a_k k n^{k-1} + \underbrace{\dots + a_k}_{\text{от степен } < k-1} + a_{k-1}(n+1)^{k-1} + \underbrace{\dots + a_1(n+1) + a_0}_{\text{от степен } < k-1} \\
 &\quad - a_{k-1} n^{k-1} - \dots - a_1 n - a_0 = \\
 &\quad \underbrace{\dots}_{\text{от степен } < k-1} = \\
 &= a_k k n^{k-1} + \underbrace{\dots + a_k}_{\text{от степен } < k-1} \\
 &\quad + a_{k-1} n^{k-1} + \underbrace{\dots}_{\text{от степен } < k-1} + \underbrace{\dots + a_1(n+1) + a_0}_{\text{от степен } < k-1} \\
 &\quad - a_{k-1} n^{k-1} - \dots - a_1 n - a_0 = \\
 &\quad \underbrace{\dots}_{\text{от степен } < k-1} = \\
 &= a_k k n^{k-1} + \underbrace{\dots}_{\text{от степен } < k-1}
 \end{aligned}$$

И така, $\Delta p(n)$ е $a_k k n^{k-1}$ плюс полином от степен, по-малка от $k-1$. Аналогично, $\Delta^2 p(n)$ е $a_k k(k-1)n^{k-2}$ плюс полином от степен, по-малка от $k-2$, ако $k \geq 2$. Лесно се вижда, че

$$\Delta^k p(n) = a_k k!$$

и

$$i \geq 1 \rightarrow \Delta^{k+i} p(n) = 0 \quad (3.86)$$

Дефинираме *оператор полином* от степен k така. Нека k е константа и е даден следният оператор, който е полином от степен k на E :

$$p(E) = a_k \cdot E^k + a_{k-1} \cdot E^{k-1} + \dots + a_1 \cdot E + a_0 \cdot I$$

Да разгледаме действието на $p(E)$ върху $b^n y(n)$, където b е константа, а $y(n)$ е някаква функция.

$$\begin{aligned}
 p(E)b^n y(n) &= (a_k \cdot E^k + a_{k-1} \cdot E^{k-1} + \dots + a_1 \cdot E + a_0 \cdot I)b^n y(n) \\
 &= a_k E^k b^n y(n) + a_{k-1} E^{k-1} b^n y(n) + \dots + a_1 E b^n y(n) + a_0 b^n y(n) \\
 &= b^n (a_0 b^k y(n+k) + a_1 b^{k-1} y(n+k-1) + \dots + a_{k-1} b y(n+1) + a_k y(n)) \\
 &= b^n p(b \cdot E)y(n)
 \end{aligned}$$

И така,

$$p(E)b^n y(n) = b^n p(b \cdot E)y(n) \quad (3.87)$$

Аниhilатор за функцията $y(n)$ е оператор полином $N(E)$, такъв че за всяко n е изпълнено

$$N(E)y(n) = 0 \quad (3.88)$$

Може да се каже, че $N(E)$ е аниhilатор за $y(n)$, ако $y(n)$ е решение на (3.88). Следват няколко примера.

- Нека $y(n) = 2^n$. Тогава $N(E) = E - 2$ е аниhilатор за $y(n)$, защото $(E - 2)2^n = 2^{n+1} - 2 \cdot 2^n = 0$.
- И изобщо, ако $y(n) = a^n$, то $N(E) = E - a$ е аниhilатор за $y(n)$, защото $(E - a)a^n = a^{n+1} - a \cdot a^n = 0$.
- Нека $y(n) = n2^n$. Тогава $N(E) = (E - 2)^2$ е аниhilатор за $y(n)$, защото

$$\begin{aligned} (E - 2)^2 n 2^n &= (E^2 - 4E + 4)n 2^n \\ &= (n + 2)2^{n+2} - 4(n + 1)2^{n+1} + 4n 2^n \\ &= 4n 2^n + 8 \cdot 2^n - 8n 2^n - 8 \cdot 2^n + 4n 2^n = 0 \end{aligned}$$

- И изобщо, ако $y(n) = na^n$, то $N(E) = (E - a)^2$ е аниhilатор за $y(n)$.
- Лесно се вижда, че ако $y(n) = n^2 a^n$, то $N(E) = (E - a)^3$ е аниhilатор за $y(n)$.
- И изобщо, ако $y(n) = n^b a^n$, то $N(E) = (E - a)^{b+1}$ е аниhilатор за $y(n)$.
- Лесно се вижда, че ако $y(n) = n^{b_1} a_1^n + n^{b_2} a_2^n$, то $N(E) = (E - a_1)^{b_1+1} (E - a_2)^{b_2+1}$ е аниhilатор за $y(n)$.
- И изобщо, ако $y(n) = n^{b_1} a_1^n + \dots + n^{b_t} a_t^n$, то $N(E) = (E - a_1)^{b_1+1} \cdot \dots \cdot (E - a_t)^{b_t+1}$ е аниhilатор за $y(n)$.

Второ понятие: фундаментално множество от решения. Разглеждаме хомогенни линейни рекурентни уравнения с константни коефициенти и крайна история

$$x(n + k) + c_1 x(n + k - 1) + c_2 x(n + k - 2) + \dots + c_{k-1} x(n + 1) + c_k x(n) = 0 \quad (3.89)$$

където $c_1, \dots, c_k$ са реални константи и $c_k \neq 0$.

Да си припомним следната ключова дефиниция.

Определение 43: Линейно независими функции

Казваме, че функциите $f_1(n), \dots, f_r(n)$ са *линейно независими* за $n \geq n_0$, ако

$$\forall n \geq n_0 : a_1 f_1(n) + \dots + a_r f_r(n) = 0$$

тстк $a_1 = \dots = a_r = 0$.

Задача 40

Покажете, че 2^n , $n2^n$ и $n^2 2^n$ са линейно независими функции за $n \geq 1$.

Решение: Нека за някакви константи a , b и c е изпълнено

$$a2^n + bn2^n + cn^2 2^n = 0, \quad \text{за } n \geq 1$$

Имаме право да разделим на 2^n . Делим на 2^n и получаваме

$$a + bn + cn^2 = 0, \quad \text{за } n \geq 1$$

Това може да е изпълнено само ако $c = 0$, понеже уравнение от втора степен може да има най-много две решения. И така, $c = 0$. Имаме

$$a + bn = 0, \quad \text{за } n \geq 1$$

Аналогично заключаваме, че $b = 0$ и $a = 0$. И така, $a = b = c = 0$. □

Да се върнем на (3.89).

Определение 44: Фундаментално множество от решения

Всяко множество от k линейно независими решения на (3.89) се нарича *фундаментално множество от решения*.

Има сравнително ефикасен начин да се провери дали множество от решения е фундаментално. За целта ще ни помогне понятието *детерминанта на Casorati*^a, на английски *Casoratian*, която е дискретният аналог на детерминантата на Wronski (на английски, *Wronskian*) от континуалната математика.

Определение 45

Нека $x_1(n), \dots, x_r(n) \in \mathcal{G}$. Детерминантата на Casorati за тези функции е

$$W(n) = \det \begin{bmatrix} x_1(n) & x_2(n) & \cdots & x_r(n) \\ \mathbf{E}x_1(n) & \mathbf{E}x_2(n) & \cdots & \mathbf{E}x_r(n) \\ \mathbf{E}^2x_1(n) & \mathbf{E}^2x_2(n) & \cdots & \mathbf{E}^2x_r(n) \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{E}^{r-1}x_1(n) & \mathbf{E}^{r-1}x_2(n) & \cdots & \mathbf{E}^{r-1}x_r(n) \end{bmatrix} \quad (3.90)$$

Задача 41

Разгледайте уравнението

$$x(n+3) - x(n+2) - 22x(n+1) + 40x(n) = 0 \quad (3.91)$$

Докажете, че функциите 2^n , 4^n и $(-5)^n$ са негови решения. Намерете детерминантата на Casorati за тези функции.

Решение: Да се убедим, че 2^n е решение.

$$\begin{aligned} 2^{n+3} - 2^{n+2} - 22 \cdot 2^{n+1} + 40 \cdot 2^n &= 0 \leftrightarrow \\ 8 \cdot 2^n - 4 \cdot 2^n - 2 \cdot 22 \cdot 2^n + 40 \cdot 2^n &= 0 \leftrightarrow \\ \underbrace{(8 - 4 - 44 + 40)}_0 2^n &= 0 \end{aligned}$$

Да се убедим, че 4^n е решение.

$$\begin{aligned} 4^{n+3} - 4^{n+2} - 22 \cdot 4^{n+1} + 40 \cdot 4^n &= 0 \leftrightarrow \\ 64 \cdot 4^n - 16 \cdot 4^n - 4 \cdot 22 \cdot 4^n + 40 \cdot 4^n &= 0 \leftrightarrow \\ \underbrace{(64 - 16 - 88 + 40)}_0 4^n &= 0 \end{aligned}$$

Да се убедим, че $(-5)^n$ е решение.

$$\begin{aligned} (-5)^{n+3} - (-5)^{n+2} - 22 \cdot (-5)^{n+1} + 40 \cdot (-5)^n &= 0 \leftrightarrow \\ -125 \cdot (-5)^n - 25 \cdot (-5)^n + 5 \cdot 22 \cdot (-5)^n + 40 \cdot (-5)^n &= 0 \leftrightarrow \\ \underbrace{(-125 - 25 + 110 + 40)}_0 (-5)^n &= 0 \end{aligned}$$

Сега да намерим детерминантата на Casorati за 2^n , 4^n и $(-5)^n$.

$$\begin{aligned} W(n) &= \det \begin{bmatrix} 2^n & 4^n & (-5)^n \\ 2^{n+1} & 4^{n+1} & (-5)^{n+1} \\ 2^{n+2} & 4^{n+2} & (-5)^{n+2} \end{bmatrix} = \\ &= 2^n 4^{n+1} (-5)^{n+2} + 2^{n+2} 4^n (-5)^{n+1} + 2^{n+1} 4^{n+2} (-5)^n \\ &\quad - 2^{n+2} 4^{n+1} (-5)^n - 2^{n+1} 4^n (-5)^{n+2} - 2^n 4^{n+2} (-5)^{n+1} = 126(-80)^n \end{aligned} \quad \square$$

Лема 20: Lemma 2.13 от [43, стр. 68]

Нека $x_1(n), \dots, x_k(n)$ са решения на (3.89). Нека $W(n)$ е детерминантата на Casorati за тях. Тогава, за всяко $n \geq n_0$:

$$W(n) = (-1)^{k(n-n_0)} c_k^{n-n_0} W(n_0).$$

Доказателство: Ще докажем за частния случай $k = 3$. Доказателството за общия случай може да се направи аналогично.

И така, нека $k = 3$. От (3.90) имаме

$$W(n+1) = \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ x_1(n+3) & x_2(n+3) & x_3(n+3) \end{bmatrix} \quad (3.92)$$

От (3.89) имаме:

$$x_1(n+3) = -c_3x_1(n) - (c_1x_1(n+2) + c_2x_1(n+1))$$

$$x_2(n+3) = -c_3x_2(n) - (c_1x_2(n+2) + c_2x_2(n+1))$$

$$x_3(n+3) = -c_3x_3(n) - (c_1x_3(n+2) + c_2x_3(n+1))$$

Заместваме в (3.92) и получаваме

$$W(n+1) = \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ A & B & C \end{bmatrix} \quad (3.93)$$

където

$$A = -c_3x_1(n) - (c_2x_1(n+1) + c_1x_1(n+2))$$

$$B = -c_3x_2(n) - (c_2x_2(n+1) + c_1x_2(n+2))$$

$$C = -c_3x_3(n) - (c_2x_3(n+1) + c_1x_3(n+2))$$

Както знаем от линейната алгебра, (3.93) може да се запише просто като

$$\begin{aligned} W(n+1) &= \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ -c_3x_1(n) & -c_3x_2(n) & -c_3x_3(n) \end{bmatrix} \\ &= -c_3 \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ x_1(n) & x_2(n) & x_3(n) \end{bmatrix} \\ &= -c_3(-1)^2 \det \begin{bmatrix} x_1(n) & x_2(n) & x_3(n) \\ x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \end{bmatrix} \\ &= -c_3(-1)^2 W(n) \end{aligned}$$

Изразихме $W(n+1)$ чрез $W(n)$:

$$W(n+1) = (-1)^3 c_3 W(n) \quad (3.94)$$

което е линейно рекурентно уравнение от първи ред. С развиване получаваме

$$W(n+1) = (-1)^{3(n-n_0)} \cdot c_3^{n-n_0} \cdot W(n_0) \quad (3.95)$$

Кое е и краят на доказателството за $n = 3$. □

Следствие 18

Тъй като $c_k \neq 0$ по условие, вярно е, че детерминантата на Casorati $W(n)$ е ненулева за всяко $n \geq n_0$ тстк детерминантата на Casorati $W(n_0)$ е ненулева. Ерго, или $W(n)$ е тъждествено нула при $n \geq n_0$, или никога не е нула при $n \geq n_0$.

Ще покажем, че множество от k решения на (3.89) е фундаментално (тоест, линейно

независимо) тстк $W(n)$ никога не е нула. Разглеждаме k решения $x_1(n), \dots, x_k(n)$ на (3.89). Да кажем, че за някакви константи $\alpha_1, \dots, \alpha_k$ е вярно, че

$$\forall n \geq n_0 : \alpha_1 x_1(n) + \dots + \alpha_k x_k(n) = 0$$

В частност е вярно, че

$$\alpha_1 x_1(n+1) + \dots + \alpha_k x_k(n+1) = 0$$

...

$$\alpha_1 x_1(n+k-1) + \dots + \alpha_k x_k(n+k-1) = 0$$

Всичко това може да се запише кратко ето така:

$$X(n)\xi = 0 \tag{3.96}$$

където

$$X(n) = \begin{bmatrix} x_1(n) & \dots & x_k(n) \\ \vdots & \dots & \vdots \\ x_1(n+k-1) & \dots & x_k(n+k-1) \end{bmatrix}, \quad \xi = \begin{bmatrix} \alpha_1 \\ \dots \\ \alpha_k \end{bmatrix}$$

Очевидно $\det X(n) = W(n)$. Съгласно линейната алгебра, (3.96) има само тривиалното решение $\alpha_1 = \dots = \alpha_k = 0$ тстк $X(n)$ е несингулярна, тоест, $\det X(n) \neq 0$, тоест, $W(n) \neq 0$ за всяко $n \geq n_0$. От тези и предишните съображения лесно следва Теорема 46.

Теорема 46 е Theorem 2.15 в [43, стр. 71].

Теорема 46: М-во решения е фундаментално тстк дет. на Casorati е ненулева

Множеството $X = \{x_1(n), \dots, x_k(n)\}$ от решения на (3.89) е фундаментално тстк $\exists n_0 \in \mathbb{N}^a$, такова че $W(n_0) \neq 0$, където $W(n_0)$ е детерминантата на Casorati за X .

“В Теорема 2.15 в [43] има дребна грешка. Авторът казва “for some $n_0 \in \mathbb{Z}^+$ ”, но всъщност не е необходимо $n_0 \neq 0$. И наистина, в Example 2.16, пак на стр. 71, където той показва как се прилага Теорема 2.15, той използва $n_0 = 0$, за да покаже, че дадено множество от решения е фундаментално.

Задача 42

Покажете, че $\{2^n, 4^n, (-5)^n\}$ е множество от фундаментални решения на (3.91).

Решение: От Задача 41 знаем, че $2^n, 4^n$ и $(-5)^n$ са решения на (3.91) и че детерминантата на Casorati за тях е $126(-80)^n$. Ако $n = 0$, тя е 126, което не е нула. Съгласно Теорема 46, $\{2^n, 4^n, (-5)^n\}$ е множество от фундаментални решения на (3.91). $\square$

Теорема 47 е Theorem 2.18 от [43, стр. 72].

Теорема 47: Фундаментално множество от решения съществува

Уравнение (3.89) има фундаментално множество от решения за $n \geq 0$.

Доказателство: Очевидно (3.89) има “единични” решения $x_1(n), x_2(n), \dots, x_k(n)$:

$$x_1(0) = 1, x_1(1) = 0, x_1(2) = 0, \dots, x_1(k-1) = 0$$

$$x_2(0) = 0, x_2(1) = 1, x_2(2) = 0, \dots, x_2(k-1) = 0$$

$$x_3(0) = 0, x_3(1) = 0, x_3(2) = 1, \dots, x_3(k-1) = 0$$

...

$$x_k(0) = 0, x_k(1) = 0, x_k(2) = 0, \dots, x_k(k-1) = 1$$

Веднага се вижда, че $W(0) = 1$. Щом детерминантата на Casorati е ненулева, то от Теорема 46 следва, че множеството от “единичните” решения е фундаментално множество от решения на (3.89). $\square$

Уравнение (3.89) има безброй много фундаментални множества от решения. Сега ще видим как се генерират фундаментални решения от известни решения.

Лема 21

Нека $x_1(n)$ и $x_2(n)$ са решения на (3.89). Тогава всяка линейна комбинация на $x_1(n)$ и $x_2(n)$ е решение на (3.89).

Доказателство: Ако за $n \geq k$ имаме

$$x_1(n+k) + c_1x_1(n+k-1) + c_2x_1(n+k-2) + \dots + c_{k-1}x_1(n+1) + c_kx_1(n) = 0$$

$$x_2(n+k) + c_1x_2(n+k-1) + c_2x_2(n+k-2) + \dots + c_{k-1}x_2(n+1) + c_kx_2(n) = 0$$

то очевидно

$$\begin{aligned} & (x_1(n+k) + x_2(n+k)) + c_1(x_1(n+k-1) + x_2(n+k-1)) + \dots + \\ & c_{k-1}(x_1(n+1) + x_2(n+1)) + c_k(x_1(n) + x_2(n)) = 0 \\ & \alpha x_1(n+k) + \alpha c_1x_1(n+k-1) + \dots + \alpha c_{k-1}x_2(n+1) + \alpha p_kx_2(n) = 0 \end{aligned}$$

за произволна константа α . $\square$

Чрез Лема 21 лесно можем да докажем следния важен резултат.

Теорема 48: Принцип на суперпозицията

Ако $x_1, x_2, \dots, x_r(n)$ са решения на (3.89), то

$$x(n) = \alpha_1x_1(n) + \alpha_2x_2(n) + \dots + \alpha_rx_r(n)$$

също е решение на на (3.89), където $\alpha_1, \alpha_2, \dots, \alpha_r$ са произволни константи. $\square$

Теорема 49: Фундаментално множество от решения и конкретно решение

Нека $\{x_1(n), \dots, x_k(n)\}$ е фундаментално множество от решения на (3.89). Тогава за всяко конкретно решение $x(n)$ на (3.89) съществуват константи $a_1, a_2, \dots, a_k$, такива че

$$x(n) = \sum_{i=1}^k a_i x_i(n)$$

Доказателство: Ползвайки нотацията на (3.96), записваме

$$X(n)\xi = \hat{x}(n)$$

където

$$\hat{x}(n) = \begin{bmatrix} x(n) \\ x(n+1) \\ \vdots \\ x(n+k-1) \end{bmatrix}$$

Тъй като $X(n)$ е инвертируема, в сила е

$$\xi = X^{-1}(n)\hat{x}(n)$$

В частност,

$$\xi = X^{-1}(n_0)\hat{x}(n_0)$$

и

$$\xi = X^{-1}(0)\hat{x}(0)$$

□

Теорема 49 ни дава право да направим следното определение.

Определение 46: Общото решение на хомогенно линейно рекурентно у-ние

Нека $\{x_1(n), \dots, x_k(n)\}$ е фундаментално множество от решения на (3.89). Ще казваме, че $x(n) = \sum_{i=1}^k \alpha_i x_i(n)$, където $\alpha_1, \alpha_2, \dots, \alpha_k$ са произволни константи, е *общото решение* на (3.89).

Всяко (конкретно) решение на (3.89) се получава от общото с подходящ избор на константи $\alpha_1, \alpha_2, \dots, \alpha_k$.

Теорема 50: Решенията на хомог. лин. рек. у-ние образуват линейно пр-ство

Нека S е множеството от всички решения на (3.89). Тогава $(S, +, \cdot)$ е линейно пространство с дименсия k , където $+$ е сумиране на елементи, а $\cdot$ е умножение със скалар.

Доказателство: Следва лесно от Лема 21. Базис може да е фундаменталното множество от решения от Теорема 47. $\square$

Извеждане на решението за хомогенните уравнения. След като въведохме необходимите понятия, да видим как се решава хомогенно линейно рекурентно уравнение от ред k с константни коефициенти. Общият вид на уравнението е (3.89). Ще намерим фундаментално решение и оттам ще намерим общото решение.

Допускаме, че решенията на (3.89) са от вида λ^n , където λ е комплексно число. Замествайки в (3.89), получаваме

$$\lambda^k + c_1\lambda^{k-1} + c_2\lambda^{k-2} + \dots + c_{k-1}\lambda + c_k = 0 \quad (3.97)$$

Уравнение (3.97) е *характеристичното уравнение* на рекурентното уравнение (3.89). Съгласно основната теорема на алгебрата, (3.97) има k на брой комплексни, не непременно различни, корени. Нулата не е корен, тъй като $c_k \neq 0$. Тези корени са *характеристичните корени* и решението се получава чрез тях⁶. Нека характеристичните корени са $\lambda_1, \dots, \lambda_k$.

Случай i Характеристичните корени са два по два различни. Ще покажем, че в този случай множеството $\{\lambda_1^n, \dots, \lambda_k^n\}$ е фундаментално множество от решения. За да се убедим, че е така, да разгледаме детерминантата на Casorati $W(0)$:

$$W(0) = \det \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 & \dots & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 & \dots & \lambda_k^2 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \lambda_1^{k-1} & \lambda_2^{k-1} & \lambda_3^{k-1} & \dots & \lambda_k^{k-1} \end{bmatrix} \quad (3.98)$$

Това е пример за *детерминанта на Vandermonde*, за която има добре известно решение. А именно,

$$W(0) = \prod_{1 \leq i < j \leq k} (\lambda_i - \lambda_j)$$

Щом $i \neq j \rightarrow \lambda_i \neq \lambda_j$, очевидно $W(0) \neq 0$. Следователно, $\{\lambda_1^n, \lambda_2^n, \dots, \lambda_k^n\}$ е фундаментално множество от решения на (3.89). Оттук и Теорема 49 следва, че общото решение е

$$x(n) = \sum_{i=1}^k \alpha_i \lambda_i^n$$

където α_i са константи. Но нас ни интересува само асимптотиката на решението, която се определя от водещото събираемо съгласно Теорема 26. Нека $\lambda_j = \max \{\lambda_1, \lambda_2, \dots, \lambda_n\}$. Ако (3.89) описва сложност на алгоритъм, то $\lambda_j > 0$ със сигурност и решението е

$$x(n) \asymp \lambda_j^n$$

Случай ii По-интересният случай е случаят, в който има кратни корени. Това е и общият случай. **Случай i** е само негов подслучай, който ние разгледахме “за загравяне”. Нека различните корени са $\lambda_1, \lambda_2, \dots, \lambda_r$ с кратности съответно $m_1, m_2, \dots, m_r$. Очевидно $\sum_{i=1}^r m_i = k$.

Можем да запишем (3.89) като

$$(\mathbf{E}^k + c_1 \mathbf{E}^{k-1} + c_2 \mathbf{E}^{k-2} + \dots + c_{k-1} \mathbf{E} + c_k \mathbf{I})x(n) = 0$$

което можем да запишем като

$$(\mathbf{E} - \lambda_1)^{m_1} (\mathbf{E} - \lambda_2)^{m_2} \dots (\mathbf{E} - \lambda_r)^{m_r} x(n) = 0 \quad (3.99)$$

факторизирайки $\boxed{\mathbf{E}^k + c_1 \mathbf{E}^{k-1} + \dots + c_k}$ до $\boxed{(\mathbf{E} - \lambda_1)^{m_1} (\mathbf{E} - \lambda_2)^{m_2} \dots (\mathbf{E} - \lambda_r)^{m_r}}$.

Да разгледаме само

$$(\mathbf{E} - \lambda_i)^{m_i} x(n) = 0 \quad (3.100)$$

за кое да е $i \in \{1, 2, \dots, r\}$. Твърдим, че всяка функция $h(n)$, която е решение на (3.100), е решение и на (3.89). Наистина, ако $h(n)$ е такова решение, то $(\mathbf{E} - \lambda_i)^{m_i} h(n) = 0$. Тогава

$$\begin{aligned} & (\mathbf{E} - \lambda_1)^{m_1} (\mathbf{E} - \lambda_2)^{m_2} \dots (\mathbf{E} - \lambda_r)^{m_r} h(n) = \\ & (\mathbf{E} - \lambda_1)^{m_1} \dots (\mathbf{E} - \lambda_{i-1})^{m_{i-1}} (\mathbf{E} - \lambda_{i+1})^{m_{i+1}} \dots (\mathbf{E} - \lambda_r)^{m_r} \underbrace{(\mathbf{E} - \lambda_i)^{m_i} h(n)}_0 = \\ & (\mathbf{E} - \lambda_1)^{m_1} \dots (\mathbf{E} - \lambda_{i-1})^{m_{i-1}} (\mathbf{E} - \lambda_{i+1})^{m_{i+1}} \dots (\mathbf{E} - \lambda_r)^{m_r} 0 = 0 \end{aligned}$$

Да допуснем, че сме намерили множество от фундаментални решения за уравненията $(\mathbf{E} - \lambda_i)^{m_i} x(n)$, за всяко $i \in \{1, 2, \dots, r\}$. Теорема 51 доказва, че тяхното обединение е множество от фундаментални решения за (3.99). Първо една лема.

Лема 22

Нека $i \in \{1, \dots, r\}$. Множеството

$$G_i = \left\{ \lambda_i^n, \binom{n}{1} \lambda_i^{n-1}, \binom{n}{2} \lambda_i^{n-2}, \dots, \binom{n}{m_i-1} \lambda_i^{n-m_i+1} \right\}$$

е множество от фундаментални решения на (3.100).

Доказателство: Първо да се убедим, че функциите от G_i са решения. Ще покажем, че $\forall t \in \{0, 1, \dots, m_i - 1\} : \binom{n}{t} \lambda_i^{n-t}$ е решение на (3.100). Наистина, имайки предвид, че

$\binom{n}{t}$ е полином на n от степен t , в сила е:

$$(\mathbf{E} - \lambda_i)^{m_i} \binom{n}{t} \lambda_i^{n-t} = \quad // \text{ съгласно (3.87)}$$

$$\lambda_i^{n-t} (\lambda_i \mathbf{E} - \lambda_i)^{m_i} \binom{n}{t} =$$

$$\lambda_i^{n-t+m_i} (\mathbf{E} - \mathbf{I})^{m_i} \binom{n}{t} = \quad // \text{ съгласно (3.85)}$$

$$\lambda_i^{n-t+m_i} \Delta^{m_i} \binom{n}{t} = \quad // \text{ понеже } t < m_i \text{ и съгласно (3.86)}$$

0

Сега ще покажем, че G_i е множество от фундаментални решения на (3.100). Съгласно Следствие 18, достатъчно е да покажем, че $W(0) \neq 0$, където $W(0)$ е следната детерминантата на Casorati:

$$\begin{vmatrix} \lambda_i^0 & \binom{0}{1} \lambda_i^{0-1} & \binom{0}{2} \lambda_i^{0-2} & \cdots & \binom{0}{m_i-2} \lambda_i^{0-m_i+2} & \binom{0}{m_i-1} \lambda_i^{0-m_i+1} \\ \lambda_i^1 & \binom{1}{1} \lambda_i^{1-1} & \binom{1}{2} \lambda_i^{1-2} & \cdots & \binom{1}{m_i-2} \lambda_i^{1-m_i+2} & \binom{1}{m_i-1} \lambda_i^{1-m_i+1} \\ \lambda_i^2 & \binom{2}{1} \lambda_i^{2-1} & \binom{2}{2} \lambda_i^{2-2} & \cdots & \binom{2}{m_i-2} \lambda_i^{2-m_i+2} & \binom{2}{m_i-1} \lambda_i^{2-m_i+1} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \lambda_i^{m_i-2} & \binom{m_i-2}{1} \lambda_i^{m_i-2-1} & \binom{m_i-2}{2} \lambda_i^{m_i-2-2} & \cdots & \binom{m_i-2}{m_i-2} \lambda_i^{m_i-2-m_i+2} & \binom{m_i-2}{m_i-1} \lambda_i^{m_i-2-m_i+1} \\ \lambda_i^{m_i-1} & \binom{m_i-1}{1} \lambda_i^{m_i-1-1} & \binom{m_i-1}{2} \lambda_i^{m_i-1-2} & \cdots & \binom{m_i-1}{m_i-2} \lambda_i^{m_i-1-m_i+2} & \binom{m_i-1}{m_i-1} \lambda_i^{m_i-1-m_i+1} \end{vmatrix}$$

Припомняме си, че биномният коефициент е нула, ако долният индекс е по-голям от горния, поради което елементите над главния диагонал са **нули**. А елементите по главния диагонал са очевидно **единици**, следователно $W(0)$ е:

$$\begin{vmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \lambda_i^1 & 1 & 0 & \cdots & 0 & 0 \\ \lambda_i^2 & \binom{2}{1} \lambda_i^{2-1} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \lambda_i^{m_i-2} & \binom{m_i-2}{1} \lambda_i^{m_i-2-1} & \binom{m_i-2}{2} \lambda_i^{m_i-2-2} & \cdots & 1 & 0 \\ \lambda_i^{m_i-1} & \binom{m_i-1}{1} \lambda_i^{m_i-1-1} & \binom{m_i-1}{2} \lambda_i^{m_i-1-2} & \cdots & \binom{m_i-1}{m_i-2} \lambda_i^{m_i-1-m_i+2} & 1 \end{vmatrix}$$

От линейната алгебра знаем, че такава детерминанта е произведението от елементите по главния диагонал, а то е 1. И така, $W(0) = 1^6$, поради което множеството $G_i = \left\{ \lambda_i^n, \binom{n}{1} \lambda_i^{n-1}, \binom{n}{2} \lambda_i^{n-2}, \dots, \binom{n}{m_i-1} \lambda_i^{n-m_i+1} \right\}$ е фундаментално множество от решения. $\square$

Теорема 51 е Theorem 2.23 от [43, стр. 77].

Теорема 51: Фундаментално множество от решения при кратни корени

Нека G_i има смисъла от Лема 22, за $1 \leq i \leq r$. Множеството $G = \bigcup_{i=1}^r G_i$ е фундаментално множество от решения на (3.99).

Доказателство: Съгласно Лема 22, функциите от G са решения на (3.99). Да разгледаме детерминантата на Casorati $W(0)$. Тъй общият вид на $W(0)$ не е особено прегледен и информативен, ако матрицата трябва да се запише на една страница (вж. [43, стр. 77]), да разгледаме конкретен пример. Да кажем, че $k = 6$, $r = 2$, $m_1 = 4$ и $m_2 = 2$. Тогава

$$W(0) = \det \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ \lambda_1 & 1 & 0 & 0 & \lambda_2 & 1 \\ \lambda_1^2 & 2\lambda_1 & 1 & 0 & \lambda_2^2 & 2\lambda_2 \\ \lambda_1^3 & 3\lambda_1^2 & 3\lambda_1 & 1 & \lambda_2^3 & 3\lambda_2^2 \\ \lambda_1^4 & 4\lambda_1^3 & 6\lambda_1^2 & 4\lambda_1 & \lambda_2^4 & 4\lambda_2^3 \\ \lambda_1^5 & 5\lambda_1^4 & 10\lambda_1^3 & 10\lambda_1^2 & \lambda_2^5 & 5\lambda_2^4 \end{bmatrix} \quad (3.101)$$

Сравнете (3.101) с детерминантата от (3.98). В (3.98) на всяко λ_i съответства точно една колона, в която, отгоре надолу, са записани степените на това λ_i , започвайки от нула. В (3.101), на всяко λ_i съответстват точно m_i колони, оформени като блок; в (3.101) блоковете са очертани в **червено** за λ_1 и **синьо** за λ_2 . Клетката на ред t и колона j в блока, съответстващ на λ_i , съдържа $\binom{t}{j} \lambda_i^t$, ако броенето на колоните започва от нула. Очевидно в най-лявата колона на блока са степените на λ_i , понеже долните индекси на биномните коефициенти са нули и по този начин биномните коефициенти са единици. Това е пример за *обобщена детерминанта на Vandermonde*, за която има добре известно решение [87]. А именно,

$$W(0) = \prod_{1 \leq i < j \leq r} (\lambda_i - \lambda_j)^{m_i \cdot m_j}$$

Тъй като $\lambda_i \neq \lambda_j$ при $i \neq j$, то $W(0) \neq 0$. Тогава G е множество от фундаментални решения на (3.99). $\square$

Следствие 19 на Теорема 51 е математическа обосновка на метода с характеристичното уравнение, що се отнася до хомогенните уравнения.

Следствие 19

Общото решение на (3.99) се дава от

$$x(n) = \sum_{i=1}^r (\alpha_{i,0} + \alpha_{i,1}n + \cdots + \alpha_{i,m_i-1}n^{m_i-1}) \lambda_i^n$$

където $\alpha_{i,j}$ за $1 \leq i \leq r$, $0 \leq j \leq m_i - 1$ са константи.

Извеждане на решението за нехомогенните уравнения. За съжаление, Следствие 19 не е достатъчно, за да решаваме рекурентни уравнения, които описват сложност на алгоритми. Всяко рекурентно уравнение, описващо адекватно сложност на (рекурсивен) алгоритъм, неизбежно има нехомогенна част, която е поне единица, защото алгоритъмът не може да не върши работа поне $\Theta(1)$ извън рекурсивните викания. Ерго, хомогенното уравнение (3.89) не може да описва сложност на алгоритъм. Сега разглеждаме съответно нехомогенно уравнение в общ вид:

$$x(n+k) + c_1x(n+k-1) + c_2x(n+k-2) + \dots + c_{k-1}x(n+1) + c_kx(n) = g(n) \quad (3.102)$$

където $g(n)$ е нехомогенната част. *Съответното хомогенно уравнение е*

$$x(n+k) + c_1x(n+k-1) + c_2x(n+k-2) + \dots + c_{k-1}x(n+1) + c_kx(n) = 0 \quad (3.103)$$

Забележете, че то е същото като (3.89); тук е записано отново за прегледност.

Задача 43

Разгледайте нехомогенното уравнение

$$x(n+2) - x(n+1) - 6x(n) = 5 \cdot 3^n \quad (3.104)$$

1. Намерете общото решение на съответното хомогенно уравнение.
2. Проверете дали 3^n и $(-2)^n$ са решения на (3.104).
3. Проверете дали $x_1(n) = n3^{n-1}$ и $x_2(n) = (1+n)3^{n-1}$ са решения на (3.104).
4. Покажете, че $x_2(n) - x_1(n)$ не е решение на (3.104).
5. Покажете, че не е вярно, че $c \cdot x_1(n)$ за всяка константа c е решение на (3.104).

Решение: По 1: съответното хомогенно уравнение е

$$x(n+2) - x(n+1) - 6x(n) = 0 \quad (3.105)$$

Характеристичното уравнение е

$$\lambda^2 - \lambda - 6 = 0$$

Корените му са $\lambda_1 = 3$ и $\lambda_2 = -2$. Общото му решение, съгласно Следствие 19, е

$$x(n) = \alpha_1 3^n + \alpha_2 (-2)^n$$

Естествено, 3^n и $(-2)^n$ са решения на (3.105):

$$3^{n+2} - 3^{n+1} - 6 \cdot 3^n = 9 \cdot 3^n - 3 \cdot 3^n - 6 \cdot 3^n = 0$$

$$(-2)^{n+2} - (-2)^{n+1} - 6 \cdot (-2)^n = 4 \cdot (-2)^n - (-2)(-2)^n - 6 \cdot (-2)^n = 0$$

По 2: нито 3^n , нито $(-2)^n$ са решения на (3.104). Току-що видяхме, че заместването на всяко от тях в лявата страна на (3.104) дава нула, а дясната страна на (3.104) не е нула.

По 3: $n3^{n-1}$ е решение. Заместваме $x(n)$ с $n3^{n-1}$ в лявата страна на (3.104):

$$\begin{aligned}(n+2)3^{n+1} - (n+1)3^n - 6n3^{n-1} &= 3^{n-1}((2+n)3^2 - (1+n)3^1 - 6n) \\ &= 3^{n-1}(18 + 9n - 3 - 3n - 6n) \\ &= 3^{n-1}(15) \\ &= 5 \cdot 3^n\end{aligned}$$

$(1+n)3^{n-1}$ също е решение. Пак заместваме $x(n)$ вляво, този път с $(1+n)3^{n-1}$:

$$\begin{aligned}(1+n+2)3^{n+1} - (1+n+1)3^n - 6(1+n)3^{n-1} &= 3^{n-1}((3+n)3^2 - (2+n)3^1 - 6(1+n)) \\ &= 3^{n-1}(27 + 9n - 6 - 3n - 6 - 6n) \\ &= 3^{n-1}(15) \\ &= 5 \cdot 3^n\end{aligned}$$

По 4: Вярно е, че $x_2(n) - x_1(n) = 3^{n-1}$. Вече показахме, че 3^n е решение на съответното хомогенно уравнение, а оттук следва, че и $3^{n-1} = \frac{1}{3}3^n$ е негово решение. Тогава няма как 3^{n-1} да е решение на нехомогенното, чиято дясна страна е $5 \cdot 3^n$.

По 5: Заместваме $x(n)$ със $cn3^{n-1}$ в лявата страна на (3.104):

$$\begin{aligned}c(n+2)3^{n+1} - c(n+1)3^n - 6cn3^{n-1} &= c3^{n-1}((2+n)3^2 - (1+n)3^1 - 6n) \\ &= c3^{n-1}(18 + 9n - 3 - 3n - 6n) \\ &= c3^{n-1}(15) \\ &= 5c3^n\end{aligned}$$

Ако $c \neq 1$, то $5c3^n$ не е същото като $5 \cdot 3^n$. □

Задача 43 показва, че, за разлика от хомогенните уравнения (вж. Теорема 50), при нехомогенните уравнения решенията не образуват линейно пространство: нито сумата на две решения е решение, нито произведението на решение с произволен скалар е решение.

Задача 43 ни навежда и на мисълта, че разликата от две решения на нехомогенното уравнение (3.102) е решение на съответното хомогенно уравнение (3.103). Ще оформим това в отделна теорема, въпреки че доказателството е твърде просто.

Теорема 52: Разликата м-у две реш. на нехомогенното е реш. на хомогенното

Разликата между всеки две решения на нехомогенно линейно рекурентно уравнение е решение на съответното хомогенно уравнение.

Доказателство: Нека $x_1(n)$ и $x_2(n)$ са решения на (3.102). Тогава

$$x_1(n+k) + c_1 x_1(n+k-1) + \dots + c_{k-1} x_1(n+1) + c_k x_1(n) = g(n)$$

$$x_2(n+k) + c_1 x_2(n+k-1) + \dots + c_{k-1} x_2(n+1) + c_k x_2(n) = g(n)$$

Изваждаме втория ред от първи и получаваме:

$$(x_1(n+k) - x_2(n+k)) + c_1(x_1(n+k-1) - x_2(n+k-1)) + \dots + c_{k-1}(x_1(n+1) - x_2(n+1)) + c_k(x_1(n) - x_2(n)) = 0$$

Виждаме, че $x_1(n) - x_2(n)$ е решение на (3.103). $\square$

Да кажем, че общото решение на хомогенното (3.103) е $x_h(n)$. Да кажем, че $x_p(n)$ означава кое е да е решение² на нехомогенното (3.102). Ще наричаме $x_p(n)$, “частно решение”. Следната теорема е Theorem 2.40 от [43, стр. 84]. Тя ни казва как може да конструираме всички решения на нехомогенното (3.102).

Теорема 53: Общото решение на нехомогенното уравнение

Всяко решение на нехомогенното (3.102) може да бъде записано като

$$x(n) = x_p(n) + \sum_{i=1}^k \alpha_i x_i(n)$$

където $\{x_1(n), \dots, x_k(n)\}$ е фундаментално множество от решения на хомогенното (3.103), а $x_p(n)$ е частно решение.

Доказателство: Според Теорема 53, разликата $x(n) - x_p(n)$ е решение на хомогенното (3.103). Съгласно Теорема 49, това решение е от вида $\sum_{i=1}^k \alpha_i x_i(n)$, където $x_1(n), \dots, x_k(n)$ са елементите на някое фундаментално множество от решения на въпросното хомогенно уравнение. $\square$

Теорема 53 ни дава право да направим следното определение.

Определение 47: Общото решение на нехомогенно линейно рекурентно у-ние

Нека $x_h(n)$ е общото решение на хомогенното уравнение (3.103). Общото решение на нехомогенното уравнение (3.102) е $x_h(n) + x_p(n)$, където $x_p(n)$ е частно решение на (3.102).

Ние вече знаем как се намира общото решение на хомогенното уравнение. Ако имаме и начин да намерим някое частно решение, то имаме начин да намерим и общото решение на нехомогенното. Методът, който се използва за намиране на частно решение, се нарича *методът на неопределените коефициенти* ([43, стр. 85]). Той не е универсален: трябва нехомогенната част да е от определен вид. А именно, да е линейна комбинация от функции от вида $n^b a^n$, където a и b са константи. Има и други видове функции, за които методът работи, но по отношение на изследването на сложността на рекурсивни алгоритми е достатъчно да разгледаме нехомогенна част, която е линейна комбинация на функции от вида $n^b a^n$. В (3.80) нехомогенната част е точно такава.

Да запишем (3.102) така:

$$\mathbf{p}(\mathbf{E})x(n) = g(n) \quad (3.106)$$

където $\mathbf{p}(\mathbf{E}) = \mathbf{E}^k + c_1\mathbf{E}^{k-1} + c_2\mathbf{E}^{k-2} + \dots + c_{k-1}\mathbf{E} + c_k\mathbf{I}$. Нека $\mathbf{N}(\mathbf{E})$ е аниhilатор за $g(n)$ от (3.106). Тогава прилагаме аниhilатора върху двете страни на уравнението и получаваме

$$\mathbf{N}(\mathbf{E})\mathbf{p}(\mathbf{E})x(n) = 0 \quad (3.107)$$

Тъй като търсим само асимптотиката на решението, а не истинското решение с точните константи, (3.107) е напълно достатъчно. В [43, стр. 85–87] акцентът е върху намиране на точното решение, откъдето идва и името “методът с неопределените коефициенти”. Нас тези коефициенти-константи не ни интересуват, така че третираме (3.107) по начин, напълно аналогичен на начина, по който третирахме (3.99). Уравнение (3.99) бе получено от хомогенното (3.89), прилагайки оператора полином, факторизиран по характеристичните корени. Уравнение (3.107) бе получено от нехомогенното (3.102). Очевидно лявата страна на (3.107) може да бъде факторизирана по корените на алгебрично уравнение, което се получава от композицията на оператора полином, който “идва” от хомогенната част, и аниhilатора, който “идва” от нехомогенната част. Мултимножеството от получените корени е обединението на мултимножествата от

- характеристичните корени на $\mathbf{p}(\mathbf{E})x(n) = 0$ и
- характеристичните корени на $\mathbf{N}(\mathbf{E})x(n) = 0$.

Задача 44

Решете $x(n+2) - x(n+1) - 6x(n) = 5 \cdot 3^n$.

Решение: Това е уравнение (3.104). Съответното хомогенно уравнение е $x(n+2) - x(n+1) - 6x(n) = 0$ с мултимножество от корените $\{3, -2\}_M$, така че операторът полином е $\mathbf{p}(\mathbf{E}) = (\mathbf{E} - 3)(\mathbf{E} + 2)$.

Аниhilаторът на $5 \cdot 3^n$ е $\mathbf{E} - 3$ (петицата е без значение за аниhilатора и оттам, за общото решение). Тогава операторът от лявата страна на уравнение (3.107) в конкретния случай е $(\mathbf{E} - 3)^2(\mathbf{E} + 2)$. Ясно е в какъв смисъл коренът 3 “идва” от нехомогенната част чрез $(\mathbf{E} - 3)x(n) = 0$.

И така, отчитайки и нехомогенната част, цялото мултимножество от корените е $\{2, 3, 3\}_M$. Решението е $x(n) \asymp n3^n$. $\square$

^a Felice Casorati е италиански математик.

^bЧитателят може да възрази, че в общия случай алгебрични уравнения от степен ≥ 5 нямат точни решения в радикали. Това е вярно, но не е съществено от гледна точка на анализа на алгоритми в практиката. Дори характеристичното уравнение да е от степен ≥ 5 , характеристичните корени могат да бъдат апроксимирани числено с каквато точност искаме. Тъй като всеки корен λ участва в решението като λ^n , то, ако решението е $T(n) \asymp \lambda^n$, очевидно никоя апроксимация $\tilde{\lambda}$, която е различна от λ , няма да даде правилната асимптотика на решението, понеже или $\lambda^n < \tilde{\lambda}^n$, или $\lambda^n > \tilde{\lambda}^n$.

Но това е само от гледна точка на теоретичен перфекционизъм. На практика, намирането на $\tilde{\lambda}$ с точност до четвъртия или шестия знак след десетичната точка ще ни даде достатъчно прецизна приблизителна оценка за сложността на алгоритъма, който разглеждаме. Като пример, нека водещото

събираймо в израза за $T(n)$ е $3\lambda^n$ и ние пресметнем с числени методи, че $\lambda \approx 2.0667$, като четирите цифри след десетичната точка са коректни. Тогава изразът $3 \cdot 2.0667^n$ ни дава чудесна представа за това, колко дълго би работила софтуерна имплементация на този алгоритъм за разумно големи стойности на n (за които изобщо има смисъл да пускаме експоненциалния алгоритъм). Това, че 2.0667 е само апроксимация, няма значение. То “върши работа”.

Нещо повече. Дори да намерим λ с точен израз с радикали, да кажем $\lambda = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)$ (намесени са числата на Fibonacci!), то, ако искаме да преценим колко дълго би работил софтуер, имплементиращ алгоритъм със сложност по време $\Theta(\lambda^n)$, ние ще вземем някаква числена апроксимация на λ .

⁶В книгата на Elaydi [43, стр. 76] има очевидна грешка в крайния етап на извеждането на $W(0)$.

²Субскриптът “p” в “ $x_p(n)$ ” идва от “particular”.

Ще разгледаме три примера за решаване на рекурентни уравнения с метода с характеристичното уравнение.

Задача 45

Решете рекурентното уравнение

$$T(n) = T(n-1) + 1$$

Решение: Преписваме рекурентното уравнение така: $T(n) = T(n-1) + 1^n n^0$, за да сме сигурни, че формата му удовлетворява (3.80). Характеристичното уравнение е $x - 1 = 0$ с единствен корен $x_1 = 1$. Тогава мултимножеството от корените на характеристичното уравнение е $\{1\}_M$. Използвайки конвенцията за именуване на (3.80), $m = 1$, $\beta_1 = 1$, и $d_1 = 0$. Така че към мултимножеството $\{1\}_M$ добавяме $\beta_1 = 1$ с кратност $d_1 + 1 = 1$, получавайки $\{1, 1\}_M$. Тогава $T(n) = A 1^n + B n 1^n$ за някакви константи A и B , следователно $T(n) = \Theta(n)$. $\square$

Задача 46

Решете рекурентното уравнение

$$T(n) = 4T(n-3) + 1 \tag{3.108}$$

Решение: Характеристичното уравнение е

$$x^3 - 4 = 0$$

Корените му са

$$x_1 = \sqrt[3]{4}$$

$$x_2 = \sqrt[3]{4} e^{i\frac{2\pi}{3}}$$

$$x_3 = \sqrt[3]{4} e^{i\frac{-2\pi}{3}}$$

Имайки предвид и нехомогенната част, мултимножеството от корените е

$$\{\sqrt[3]{4}, \sqrt[3]{4} e^{i\frac{2\pi}{3}}, \sqrt[3]{4} e^{i\frac{-2\pi}{3}}, 1\}_M$$

Решението, спрямо някакви комплексни константи A , B , C и D е:

$$\begin{aligned} T(n) &= A \left(\sqrt[3]{4}\right)^n + B \left(\sqrt[3]{4}\right)^n e^{\frac{2\pi ni}{3}} + C \left(\sqrt[3]{4}\right)^n e^{-\frac{2\pi ni}{3}} + D 1^n = \\ &= A \left(\sqrt[3]{4}\right)^n + B \left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{2\pi n}{3}\right) + i \sin\left(\frac{2\pi n}{3}\right)\right) + \\ &\quad C \left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{-2\pi n}{3}\right) + i \sin\left(\frac{-2\pi n}{3}\right)\right) + D \\ &= A \left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2\pi n}{3}\right)(B + C) + \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2\pi n}{3}\right)(B - C)i + D \quad (3.109) \end{aligned}$$

Гарантирано A е реално число, а константите B , C и D са такива комплексни числа, че $T(n)$ е цяло положително за всяко цяло положително n , надхвърлящо аргументите на трите неупоменати начални условия, при допускането, че началното условия задава цели положителни числа на T . Причината е много проста: няма как (3.108) да “произведе” дори едно число, което не е цяло положително, ако стартира с цели положителни числа. И така, от (3.109) извеждаме веднага, че $T(n) \asymp \left(\sqrt[3]{4}\right)^n$. $\square$

Допълнение 27: Когато има комплексни характеристични корени

Разсъждаваме върху точното решение на

$$T(n) = 4T(n-3) + 1$$

имайки предвид общото решение (3.109).

Ако вземем $B = C = \frac{1}{2}$, получаваме едно решение:

$$T_1(n) = A \left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2\pi n}{3}\right) + D$$

Ако вземем $B = -\frac{1}{2}i$ and $C = \frac{1}{2}i$, получаваме друго решение:

$$T_2(n) = A \left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2\pi n}{3}\right) + D$$

Съгласно принципът на суперпозицията (Лема 21), ако имаме линейно рекурентно уравнение и знаем, че някакви функции g_i , където $1 \leq i \leq k$, са решения, то всяка тяхна линейна комбинация също е решение. Тогава имаме общо решение

$$T(n) = A_1 \left(\sqrt[3]{4}\right)^n + A_2 \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2\pi n}{3}\right) + A_3 \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2\pi n}{3}\right) + A_4 \quad (3.110)$$

за някакви константи A_1 , A_2 , A_3 и A_4 . Асимптотиката на решението е $T(n) = \Theta\left(\left(\sqrt[3]{4}\right)^n\right)$.

Точното решение на (3.108) зависи от началните условия, които не са дадени. Да кажем,

че началните условия са следните:

$$T(1) = 1$$

$$T(2) = 2$$

$$T(3) = 3$$

$$T(4) = 4$$

Щом началните условия са цели положителни числа, то константите $A_1, \dots, A_4$ в (3.110) трябва да са такива, че $T(n)$ да е цяло положително число за всяко $n \in \mathbb{N}^+$.

Първо да разгледаме общото решение (3.110) и да съобразим какви стойности вземат $\cos\left(\frac{2\pi n}{3}\right)$ и $\sin\left(\frac{2\pi n}{3}\right)$ при цели положителни стойности на n :

n	1	2	3	4	5	6	7	8	9	10
$\cos\left(\frac{2\pi n}{3}\right)$	$-\frac{1}{2}$	$-\frac{1}{2}$	1	$-\frac{1}{2}$	$-\frac{1}{2}$	1	$-\frac{1}{2}$	$-\frac{1}{2}$	1	$-\frac{1}{2}$
$\sin\left(\frac{2\pi n}{3}\right)$	$\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$	$-\frac{\sqrt{3}}{2}$	0	$\frac{\sqrt{3}}{2}$

Забележете какво означава това за общото решение (3.110). Тъй като $\cos\left(\frac{2\pi n}{3}\right)$ и $\sin\left(\frac{2\pi n}{3}\right)$ задават периодични числови редици, и двете с период 3, можем да мислим за (3.110) като за **три** израза:

$$T(n) = \begin{cases} A_1 \left(\sqrt[3]{4}\right)^n - A_2 \frac{\left(\sqrt[3]{4}\right)^n}{2} + A_3 \frac{\left(\sqrt[3]{4}\right)^n \sqrt{3}}{2} + A_4, & \text{ако } n \equiv 1 \pmod{3} \\ A_1 \left(\sqrt[3]{4}\right)^n - A_2 \frac{\left(\sqrt[3]{4}\right)^n}{2} - A_3 \frac{\left(\sqrt[3]{4}\right)^n \sqrt{3}}{2} + A_4, & \text{ако } n \equiv 2 \pmod{3} \\ A_1 \left(\sqrt[3]{4}\right)^n + A_2 \left(\sqrt[3]{4}\right)^n + A_4, & \text{ако } n \equiv 0 \pmod{3} \end{cases} \quad (3.111)$$

За да намерим $A_1, \dots, A_4$, в (3.110) последователно заместваме n с 1, 2, 3 и 4, като за $T(1), \dots, T(4)$ ползваме началните условия, и така получаваме следната система:

$$1 = A_1 4^{1/3} - A_2 \frac{4^{1/3}}{2} + A_3 \frac{4^{1/3} \sqrt{3}}{2} + A_4$$

$$2 = A_1 4^{2/3} - A_2 \frac{4^{2/3}}{2} - A_3 \frac{4^{2/3} \sqrt{3}}{2} + A_4$$

$$3 = A_1 + A_2 4 + A_4$$

$$4 = A_1 4^{4/3} - A_2 2 \cdot 4^{1/3} + A_3 2\sqrt{3} 4^{1/3} + A_4$$

Решението на системата е

$$A_1 = \frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6}$$

$$A_2 = -\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2}$$

$$A_3 = \frac{4^{2/3} \sqrt{3}}{12} - \frac{4^{1/3} \sqrt{3}}{6}$$

$$A_4 = 0$$

Тогава точното решение е

$$T(n) = \left(\frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6} \right) 4^{n/3} + \left(-\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2} \right) 4^{n/3} \cos(2\pi n/3) + \left(\frac{4^{2/3}\sqrt{3}}{12} - \frac{4^{1/3}\sqrt{3}}{6} \right) 4^{n/3} \sin(2\pi n/3) \quad (3.112)$$

Колкото и да е странно, този израз е цяло положително число за всяко $n \in \mathbb{N}^+$.

Имайки предвид, че $\cos\left(\frac{2\pi n}{3}\right)$ и $\sin\left(\frac{2\pi n}{3}\right)$ задават периодични числови редици, и двете с период 3, можем да мислим за (3.112) като за три отделни изрази, всеки с константни коефициенти – също както (3.111).

Задача 47

Решете рекурентното уравнение

$$T(n) = 15T(n-1) - 81T(n-2) + 185T(n-3) - 150T(n-4) + n^2 2^{n+1} + (3n+1)2^{2n} + (3n+7)2^n + 7n^3 + 17$$

Решение: Първо да препишем уравнението по такъв начин, че нехомогенната част да стане от “правилния вид”. В момента тя не е от правилния вид: в степенните показатели се срещат $n+1$ и $2n$, което не е разрешено, и освен това се среща два пъти експоненциална функция с основа 2, което също не е разрешено. И така,

$$T(n) = 15T(n-1) - 81T(n-2) + 185T(n-3) - 150T(n-4) + (2n^2 + 3n + 7)2^n + (3n+1)4^n + (7n^3 + 17)1^n$$

Сега вече нехомогенната част е в правилния вид.

Хомогенното уравнение е

$$T(n) = 15T(n-1) - 81T(n-2) + 185T(n-3) - 150T(n-4)$$

Характеристичното уравнение е

$$x^4 - 15x^3 + 81x^2 - 185x + 150 = 0$$

Мултимножеството от корените е $\{2, 3, 5, 5\}_M$. От нехомогенната част имаме мултимножество $\{1, 1, 1, 1, 2, 2, 2, 4, 4\}_M$. Обединението на мултимножествата е $\{1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 4, 4, 5, 5\}_M$. Най-големият елемент е 5 с кратност 2. Решението е $T(n) \asymp n5^n$. $\square$

Ето пример за рекурентно уравнение, което не може да се реши директно чрез метода с характеристичното уравнение, но методът с характеристичното уравнение помага да получим интуиция за асимптотиката на решението, която интуицията можем да докажем формално по индукция.

Задача 48

Решете рекурентното уравнение

$$R(n) = 6R(n-1) + 11R(n-2) + n^{\sqrt{n}}$$

Решение: Това уравнение не може да се реши с метода с характеристичното уравнение, понеже нехомогенната част не е от правилния вид: $n^{\sqrt{n}}$ не е произведение от полином на n и експоненциална функция на n . Ще решим уравнението с индукция, но преди това трябва да сме наясно какво ще доказваме по индукция.

Хомогенното уравнение е $R(n) = 6R(n-1) + 11R(n-2)$ с решение $R(n) \asymp (3 + 2\sqrt{5})^n$, където $3 + 2\sqrt{5} \approx 7.472135954$. Важното е, че $3 + 2\sqrt{5} > 2$.

Лесно се вижда, че за нехомогенната част $n^{\sqrt{n}}$ е вярно, че $n^{\sqrt{n}} < 2^n$: в сила е $n^{\sqrt{n}} = 2^{\lg(n^{\sqrt{n}})} = 2^{\sqrt{n} \lg n}$, а очевидно $2^{\sqrt{n} \lg n} < 2^n$, понеже $\sqrt{n} \lg n < n$. Щом $n^{\sqrt{n}} < 2^n$ и $2^n < (3 + 2\sqrt{5})^n$, вярно е, че $n^{\sqrt{n}} < (3 + 2\sqrt{5})^n$. Близко е до ума, че в такъв случай асимптотиката се задава от големината на дървото на рекурсията, която е $\Theta((3 + 2\sqrt{5})^n)$. Да докажем това по индукция.

Първо да докажем, че $R(n) = O((3 + 2\sqrt{5})^n)$. Дали има константа $c > 0$, такава че $R(n) \leq c(3 + 2\sqrt{5})^n$ за всички достатъчно големи n ? Индуктивните предположения са

$$R(n-1) \leq c(3 + 2\sqrt{5})^{n-1}$$

$$R(n-2) \leq c(3 + 2\sqrt{5})^{n-2}$$

Искаме да докажем

$$R(n) \leq 6c(3 + 2\sqrt{5})^{n-1} + 11c(3 + 2\sqrt{5})^{n-2} + n^{\sqrt{n}} \quad (3.113)$$

Дали има $c > 0$, за което дясната страна е ограничена отгоре от $c(3 + 2\sqrt{5})^n$? За съжаление, не. Това неравенство

$$6c(3 + 2\sqrt{5})^{n-1} + 11c(3 + 2\sqrt{5})^{n-2} + n^{\sqrt{n}} \leq c(3 + 2\sqrt{5})^n$$

не е вярно, понеже

$$\begin{aligned} & 6c(3 + 2\sqrt{5})^{n-1} + 11c(3 + 2\sqrt{5})^{n-2} = \\ & c \left(6(3 + 2\sqrt{5})^{n-1} + 11(3 + 2\sqrt{5})^{n-2} \right) = \\ & c(3 + 2\sqrt{5})^n \left(\frac{6}{3 + 2\sqrt{5}} + \frac{11}{(3 + 2\sqrt{5})^2} \right) = c(3 + 2\sqrt{5})^n \end{aligned}$$

Убедете се сами, че $\frac{6}{3+2\sqrt{5}} + \frac{11}{(3+2\sqrt{5})^2} = 1$. Доказателството на (3.113) се провали.

Ще засилим твърдението – ще извадим една експоненциална функция от $(3 + 2\sqrt{5})^{n-1}$, но експоненциална функция с основа, по-малка от $3 + 2\sqrt{5}$. И така, сега доказваме, че съществува константа $c > 0$, такава че $R(n) \leq c(3 + 2\sqrt{5})^n - 2^n$ за всички достатъчно големи n . Забележете, че $c(3 + 2\sqrt{5})^n - 2^n$ има само положителни стойности за всички достатъчно големи n , независимо от това колко малка е $c > 0$. Индуктивните предположения са

$$R(n-1) \leq c(3 + 2\sqrt{5})^{n-1} - 2^{n-1}$$

$$R(n-2) \leq c(3 + 2\sqrt{5})^{n-2} - 2^{n-2}$$

Искаме да докажем

$$R(n) \leq 6c(3 + 2\sqrt{5})^{n-1} - 6 \cdot 2^{n-1} + 11c(3 + 2\sqrt{5})^{n-2} - 11 \cdot 2^{n-2} + n^{\sqrt{n}} \quad (3.114)$$

Преобразуваме дясната страна

$$\begin{aligned}
 & 6c(3 + 2\sqrt{5})^{n-1} - 6 \cdot 2^{n-1} + 11c(3 + 2\sqrt{5})^{n-2} - 11 \cdot 2^{n-2} + n^{\sqrt{n}} = \\
 & c(3 + 2\sqrt{5})^n \left(\frac{6}{3 + 2\sqrt{5}} + \frac{11}{(3 + 2\sqrt{5})^2} \right) - 6 \cdot 2^{n-1} - 11 \cdot 2^{n-2} + n^{\sqrt{n}} = \\
 & c(3 + 2\sqrt{5})^n - 6 \frac{2^n}{2} - 11 \frac{2^n}{4} + n^{\sqrt{n}} = \\
 & c(3 + 2\sqrt{5})^n - \frac{23}{4} 2^n + n^{\sqrt{n}}
 \end{aligned}$$

Дали има $c > 0$, такава че

$$\begin{aligned}
 c(3 + 2\sqrt{5})^n - \frac{23}{4} 2^n + n^{\sqrt{n}} &\leq c(3 + 2\sqrt{5})^n - 2^n \leftrightarrow \\
 -\frac{19}{4} 2^n + n^{\sqrt{n}} &\leq 0
 \end{aligned}$$

за всички достатъчно големи n ? Със сигурност да, понеже лявата страна е отрицателна за всички достатъчно големи n : както вече показахме, $2^n > n^{\sqrt{n}}$. Доказахме 3.114.

Сега ще докажем, че $R(n) = \Omega((3 + 2\sqrt{5})^n)$. Ще докажем, че съществува константа $d > 0$, такава че $R(n) \geq d(3 + 2\sqrt{5})^n$ за всички достатъчно големи n . Индуктивните предположения са

$$\begin{aligned}
 R(n-1) &\geq d(3 + 2\sqrt{5})^{n-1} \\
 R(n-2) &\geq d(3 + 2\sqrt{5})^{n-2}
 \end{aligned}$$

Ще докажем, че

$$R(n) \geq 6d(3 + 2\sqrt{5})^{n-1} + 11d(3 + 2\sqrt{5})^{n-2} + n^{\sqrt{n}}$$

Дали има $d > 0$, за което дясната страна е ограничена отдолу от $d(3 + 2\sqrt{5})^n$?

$$\begin{aligned}
 6d(3 + 2\sqrt{5})^{n-1} + 11d(3 + 2\sqrt{5})^{n-2} + n^{\sqrt{n}} &\geq d(3 + 2\sqrt{5})^n \leftrightarrow \\
 d(3 + 2\sqrt{5})^n \left(\frac{6}{3 + 2\sqrt{5}} + \frac{11}{(3 + 2\sqrt{5})^2} \right) + n^{\sqrt{n}} &\geq d(3 + 2\sqrt{5})^n \leftrightarrow \\
 d(3 + 2\sqrt{5})^n + n^{\sqrt{n}} &\geq d(3 + 2\sqrt{5})^n \leftrightarrow n^{\sqrt{n}} \geq 0
 \end{aligned}$$

Да, със сигурност има такава, да кажем $d = 1$. □

Част II

Сортиране

Лекция 4

Въведение. Елементарни сортиращи алгоритми.

Резюме: Разглеждаме задачата СОРТИРАНЕ и нейната важност за решаването на други задачи. Въвеждаме понятието *стабилност на сортиращ алгоритъм*. Разглеждаме два елементарни сортиращи алгоритъма: INSERTION SORT и SELECTION SORT и анализираме тяхната коректност, сложност по време и памет и стабилност. Особено внимание отделяме на коректността, която по правило доказваме чрез инвариант на цикъл.

4.1 За задачата СОРТИРАНЕ

Понятието “преднаредба” е дефинирано и обяснено подробно в Допълнение 18.

Изч. Задача 13: СОРТИРАНЕ

екземпляр: Редица $X = (a_1, a_2, \dots, a_n)$, чиито елементи са от някакво множество A , което е подходящо за сортиране и върху които е дефинирана релация на пълна преднаредба $< \subseteq A \times A$.

решение: пермутация $(a'_1, a'_2, \dots, a'_n)$ на елементите на X , такава че $a'_1 < a'_2 < \dots < a'_n$.

Подчертаваме, че екземплярът на задачата е **редицата** $(a_1, a_2, \dots, a_n)$, а не (мулти)множеството $\{a_1, a_2, \dots, a_n\}_M$.

Елементите $a_1, \dots, a_n$ не са непременно цели числа. “Елементи, подходящи за сортиране” означава елементи, за които е реалистично допускането, че имат големина единица и че може да бъдат сравнявани и местени в паметта в единица време. Може да са реални числа[†], а може да са някакви по-общи структури. На практика най-често възниква необходимостта да се сортират *записи*, всеки от които има *ключ* и някаква *сателитна информация*, като релацията $<$ е дефинирана само върху множеството от ключовете. Ключовете във всеки запис може да са повече от един и в такъв случай говорим за първичен ключ, вторичен ключ и така нататък.

С учебна цел ще разглеждаме основно сортиране на цели числа, а релацията $<$ ще бъде добре познатата “по-малко или равно” $\leq$.

Съществува и алтернативна, по-абстрактна дефиниция на задачата СОРТИРАНЕ, в която екземплярът не е редица, а (мулти)множество. Това може да е много удобно при разглеждане

[†]Не е невъзможно да са дори дори комплексни числа, стига релацията $<$ да е дефинирана смислено.

на долни граници (Лекция 13). Примерно, в [123, стр. 6] задачата е дефинирана по този начин. Да се ограничим само върху реални числа.

Изч. Задача 14: СОРТИРАНЕ, АБСТРАКТНА ДЕФИНИЦИЯ

екземпляр: Мултимножество $\{a_1, a_2, \dots, a_n\}_M$ от реални числа.

решение: За всяка двойка (i, j) , такава че $1 \leq i < j \leq n$, отговор дали $a_i < a_j$.

Изглежда, че авторът на [123] смята, че е екземплярът е множество, тъй като, ако е мултимножество, отговорът на въпроса $a_i \stackrel{?}{<} a_j$ може да не е достатъчно информативен; тогава би трябвало да имаме отговорите на двата въпроса $a_i \stackrel{?}{<} a_j$ и $a_j \stackrel{?}{<} a_i$, за да установим “истинското отношение” между a_i и a_j . От друга страна, той говори за задачата ELEMENT UNIQUENESS [123, Problem 2] (вижте Подсекция 4.2.3) като за частен случай на СОРТИРАНЕ, което означава, че екземплярът е мултимножество – няма никакъв смисъл да питаме дали елементите на множество са уникални, защото знаем, че те са такива; уникалността на елементите е дефинираща характеристика за множествата.

Да оставим настрана въпроса дали екземплярът на СОРТИРАНЕ, АБСТРАКТНА ДЕФИНИЦИЯ е множество или мултимножество. Важното е друго. Наистина можем да гледаме на решението на задачата не като на редица, а като на колекция от отговорите на сравненията на двойките елементи на екземпляра. Може дори да смятаме, че решението е пермутация, но тя не е реализирана чрез масив, а е абстрактна структура данни, която позволява да се правят запитвания (queries): при даден индекс i на елемента a_i от входа се връща броя на елементите от входа, по-малки от него, плюс едно; това е точно позицията, която би имал a_i в сортирания масив. По този начин, правейки запитвания за a_i и a_j , научаваме отговора на въпроса $a_i \stackrel{?}{<} a_j$. Пермутацията да бъде реализирана чрез масив—както е в “класическото” сортиране—е само една възможност за нейната реализация. Тя е добра и смислена, понеже позволява, ползвайки $\Theta(n)$ памет, да можем да отговорим в $\Theta(1)$ време на всеки от общо $\Theta(n^2)$ въпроса. Но абстрактната дефиниция е абстрактна именно защото отказваме да фиксирате реализация на пермутацията; при абстрактните типове данни реализацията е “черна кутия”, в която не знаем какво има.

4.2 Защо сортирането е важно

По правило в практиката сортирането не е самоцелно, а е само първи етап от решаването на някаква задача. Има доста примери за важни задачи, за които разполагаме с ефикасни алгоритми, които използват сортирането като предварителна обработка. Алгоритмичният фолклор казва, че ако съставител на алгоритми е изправен пред непозната задача, едно от първите неща, които трябва да опита, за да направи ефикасен алгоритъм, е да сортира някакви данни и да види дали от това няма да произтече нещо полезно [144, стр. 106].

Наблюдение 32

Предимството на сортираните данни над несортираните се корени в транзитивността на релацията $<$. Ако редицата $(a_1, a_2, \dots, a_n)$ е сортирана и е известно, че $a_i < x$ за някое i и някакво x , то от транзитивността на $<$ заключаваме, че $a_j < x$ за всяко $j < i$.

4.2.1 Двоично търсене

В простия си вариант като задача за разпознаване, ТЪРСЕНЕ се дефинира така.

Изч. Задача 15: ТЪРСЕНЕ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: $a_1, a_2, \dots, a_n$, key: обекти от един и същи тип.

въпрос: Дали има елемент измежду $a_1, a_2, \dots, a_n$, който е равен на key?

На практика по-полезна е задачата във версия задача за търсене. Авторът съжалява за двукратното използване на “търсене” в името на задачата; това изглежда неизбежно в текущата номенклатура.

Изч. Задача 16: ТЪРСЕНЕ, ВЕРСИЯ ЗА ТЪРСЕНЕ

екземпляр: $a_1, a_2, \dots, a_n$, key: обекти от един и същи тип.

решение: Ако има елемент измежду $a_1, a_2, \dots, a_n$, който е равен на key, индексът на един такъв елемент. В противен случай индикация, че такъв елемент няма.

Ако входът на някакъв алгоритъм за ТЪРСЕНЕ е произволна редица $a_1, a_2, \dots, a_n$ и key, то единствената възможност за действие е проверяване дали $a_i \stackrel{?}{=} \text{key}$ по всички $i \in \{1, 2, \dots, n\}$. Това се нарича *последователно търсене*. Най-лошият случай по отношение на сложността е, key да не е нито един от $a_1, a_2, \dots, a_n$. Тогава всеки алгоритъм би трябвало да направи n сравнения в най-лошият случай. Очевидно е, че ако сме пропуснали да проверим дали $a_i \stackrel{?}{=} \text{key}$ за поне едно i , няма как да сме убедени, че key не се среща.

Ако обаче елементите $a_1, a_2, \dots, a_n$ са подходящи за сортиране, уникални и са сортирани, можем да приложим *двоично търсене* (на английски е *binary search*), което е много по-бързо от последователното търсене. Следва описание на итеративния вариант на алгоритъма.

Алгоритъм 4: Alg BSearch Iter

ALG BSEARCH ITER(Сортиран числен масив $A[1..n]$ без повторения, число key)

```

1   $\ell \leftarrow 1$ 
2   $h \leftarrow n$ 
3  while  $\ell \leq h$  do
4       $\text{mid} \leftarrow \lfloor \frac{\ell+h}{2} \rfloor$ 
5      if  $A[\text{mid}] = \text{key}$ 
6          return mid
7      else if  $\text{key} < A[\text{mid}]$ 
8           $h \leftarrow \text{mid} - 1$ 
9      else
10          $\ell \leftarrow \text{mid} + 1$ 
11  return -1
```

Сложността по време е $\Theta(\lg n)$ в най-лошият случай, което изведохме много подробно в Подсекция 2.4.3. Нещо повече, точната сложност като брой сравнения $A[\text{mid}] \stackrel{?}{=} \text{key}$ е $\lfloor \lg n \rfloor + 1$ в най-лошият случай.

Разликата между последователното търсене в линейно време и двоичното търсене в логаритмично време е огромна. Ако, примерно, $n = 7\,000\,000\,000$, с не повече от 33 достъпа до

масива можем да установим, че даден елемент не е във входа, ако данните са сортирани и ползваме двоично търсене. Сравнете 33 със 7 000 000 000!

Теорема 54: Коректността на ALG BSEARCH ITER

За всяко изпълнение на $\text{ALG BSEARCH ITER}(A[1..n], \text{key})$, ако key се среща в A , то ALG BSEARCH ITER връща индекса на key в A , в противен случай ALG BSEARCH ITER връща -1 .

Доказателство: Следното твърдение е инвариант за **while**-цикъла на редове 3–10.

Инвариант 5: Цикълът на ALG BSEARCH ITER

Всеки път, когато изпълнението на ALG BSEARCH ITER е на ред 3, ако $\text{key} \in A[1..n]$, то $\text{key} \in A[\ell..h]$.

База. Разглеждаме първия път, когато изпълнението достигне ред 3. Заради присвояванията на редове 1 и 2, съждението “ако $\text{key} \in A[1..n]$, то $\text{key} \in A[\ell..h]$ ” е тривиално вярно. ✓

Поддръжка. Да допуснем, че изпълнението е на ред 3 и тялото на цикъла ще се изпълни поне още веднъж. Следва, че $A[\lfloor \frac{\ell+h}{2} \rfloor] \neq \text{key}$, понеже, ако $A[\lfloor \frac{\ell+h}{2} \rfloor] = \text{key}$, условието на ред 5 ще се окаже истина и алгоритъмът ще терминира на ред 6 и ново достигане на ред 3 няма да има.

На ред 4, на mid се присвоява $\lfloor \frac{\ell+h}{2} \rfloor$. Тъй като $A[\text{mid}] \neq \text{key}$, следните два случая са изчерпателни.

Случай I $\text{key} < A[\text{mid}]$. Щом $\text{key} < A[\text{mid}]$, то $\text{key} < A[j]$ за всяко $j \in \{\text{mid} + 1, \text{mid} + 2, \dots, n\}$; това е заради транзитивността на релацията $<$ и факта, че $A[1..n]$ е сортиран. Но тогава key не може да се намира в $A[\text{mid}..h]$, така че, ако key се среща в A , key се намира в $A[\ell.. \text{mid} - 1]$.

От друга страна, условието на ред 7 е истина и на ред 8, на h се присвоява $\text{mid} - 1$. Спрямо новото h е вярно, че при следващото достигане на ред 3, ако $\text{key} \in A[1..n]$, то $\text{key} \in A[\ell..h]$. С други думи, инвариантът се запазва.

Случай II $\text{key} > A[\text{mid}]$. Щом $\text{key} > A[\text{mid}]$, то $\text{key} > A[j]$ за всяко $j \in \{\ell, \ell + 1, \dots, \text{mid} - 1\}$; това е заради транзитивността на релацията $>$ и факта, че $A[1..n]$ е сортиран. Но тогава key не може да се намира в $A[\ell.. \text{mid} - 1]$, така че, ако key се среща в A , key се намира в $A[\text{mid} + 1..h]$.

От друга страна, условието на ред 7 е лъжа и на ред 10, на ℓ се присвоява $\text{mid} + 1$. Спрямо новото ℓ е вярно, че при следващото достигане на ред 3, ако $\text{key} \in A[1..n]$, то $\text{key} \in A[\ell..h]$. С други думи, инвариантът се запазва.

Терминация. Да допуснем, че изпълнението е на ред 3 за последен път. Има две причини тялото на цикъла да не се изпълни повече изцяло.

1. $\ell > h$. Съгласно инварианта, ако key се намира в $A[1..n]$, то key се намира в $A[\ell..h]$. В този случай обаче $A[\ell..h]$ е празният масив, така че входният масив не съдържа key . От друга страна, изпълнението отива на ред 11, така че алгоритъмът коректно индикира отсъствието на key във входния масив, връщайки -1 . ✓
2. $A[\lfloor \frac{\ell+h}{2} \rfloor] = \text{key}$. Тогава очевидно изпълнението отива на ред 6 и алгоритъмът коректно връща индекса на key във входния масив. ✓ □

Допълнение 28: Двоичното търсене в рекурсивен вариант

```

ALG BSEARCH REC(Сортиран числен масив  $A[\ell \dots h]$  без повторения, число  $key$ )
1  if  $\ell > h$ 
2      return  $-1$ 
3  else
4       $mid \leftarrow \lfloor \frac{\ell+h}{2} \rfloor$ 
5      if  $A[mid] = key$ 
6          return  $mid$ 
7      else if  $key < A[mid]$ 
8          return
9          ALG BSEARCH REC( $A[\ell \dots mid - 1]$ ,  $key$ )
10     else
11         return
12         ALG BSEARCH REC( $A[mid + 1 \dots h]$ ,  $key$ )

```

Началното викане е $ALG BSEARCH REC(A[1 \dots n], key)$.

Теорема 55: Коректността на $ALG BSEARCH REC$

За всяко изпълнение на $ALG BSEARCH REC(A[\ell \dots h], key)$, ако key се среща в A , то $ALG BSEARCH REC$ връща индекса на key в A , в противен случай $ALG BSEARCH REC$ връща -1 .

Доказателство: Доказателството е по индукция по изпълнението на алгоритъма, както подобава на доказателство за коректност на рекурсивен алгоритъм.

База. Базата на доказателството разглежда базата на рекурсията. Управляващата променлива за рекурсията е разликата $h - \ell$, така че проверката дали изпълнението е в базовия случай става на ред 1, а самата база е ред 2. Наистина, ако ℓ е по-голямо от h , масивът $A[\ell \dots h]$ е празен, така че key не се среща в него и алгоритъмът връща -1 на ред 2.

Разглеждаме работата на алгоритъма на дадено ниво в рекурсията.

Индуктивно предположение. Допускаме, че ако бъде направено кое да е от рекурсивните викания на ред 8 или ред 11, то работи коректно. А именно, връща -1 , ако key не се среща в съответния подмасив, или индекса на key , в противен случай.

Индуктивна стъпка. Следните три възможности са изчерпателни.

Случай I $A[mid] = key$. В този случай, от една страна е вярно, че key се среща в масива, а от друга страна на ред 6, алгоритъмът връща индекса на key . ✓

Случай II $key < A[mid]$. В този случай, $key < A[j]$ за всяко $j \in \{mid + 1, mid + 2, \dots, h\}$; това е заради транзитивността на релацията $<$ и факта, че $A[1 \dots n]$ е сортиран. Но тогава key не може да се намира в $A[mid \dots h]$, така че, ако key се среща в $A[\ell \dots h]$, key се намира в $A[\ell \dots mid - 1]$.

Булевото условие на ред 7 е истина и рекурсивното викане на ред 8 се извършва. Съгласно индуктивното предположение, викането върху $A[\ell \dots mid - 1]$ на ред 8 работи коректно, откъдето следва, че викането върху $A[\ell \dots h]$ работи коректно. ✓

Случай III $key > A[mid]$. В този случай, $key > A[j]$ за всяко $j \in \{\ell, \ell + 1, \dots, mid - 1\}$; това е заради транзитивността на релацията $<$ и факта, че $A[1 \dots n]$ е сортиран. Но тогава key не може да се намира в $A[\ell \dots mid]$, така че, ако key се среща в $A[\ell \dots h]$, key се намира в $A[mid + 1 \dots h]$.

Булевото условие на ред 7 е лъжа, поради което се извършва рекурсивното викане на ред 11. Съгласно индуктивното предположение, викането върху $A[mid + 1 \dots h]$ на ред 11 работи коректно, откъдето следва, че викането върху $A[\ell \dots h]$ работи коректно. ✓ □

Следствие 20

Началното викане $ALG\ BSEARCH\ REC(A[1 \dots n], key)$ работи коректно: ако key се намира в $A[1 \dots n]$, то алгоритъмът връща индекса му, в противен случай връща -1 .

Една забележка за двоичното търсене. Двоичното търсене не е приложимо, ако има елементи с еднаква стойност, които все пак различаваме някак и търсим точно определен такъв. Такива елементи може да се намират в произволен ред един спрямо друг след сортирането. Примерно, елементите имат първичен и вторичен ключ, но са сортирани само по първичен ключ, което влече, че елементите с еднакъв първичен и различен вторичен ключ може да са подредени произволно. Ако търсим точно определена наредена двойка (x, y) от първичен и вторичен ключ, в най-лошия случай трябва да прегледаме последователно всички елементи с първичен ключ x , за да се убедим, че измежду тях нито един не е с вторичен ключ y . С други думи, търсенето се изразжда в последователно търсене, защото наредбата не е линейна. И така, за да работи двоичното търсене, трябва елементите да са подредени с линейна наредба.

4.2.2 Най-близки елементи

Това е оптимизационна задача. Ето я в двата варианта (припомнете си понятията на стр. 13).

Изч. Задача 17: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ОЛЕКОТЕНИЯТ ВАРИАНТ

екземпляр: $a_1, a_2, \dots, a_n$: цели числа; $n \geq 2$

решение: $\min \{|a_i - a_j| : 1 \leq i < j \leq n\}$

Изч. Задача 18: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ТЕЖКИЯТ ВАРИАНТ

екземпляр: $a_1, a_2, \dots, a_n$: цели числа; $n \geq 2$

решение: Двойка елементи (a_i, a_j) , такива че $i \neq j$ и разликата $|a_i - a_j|$ е минимална.

Възможно е масивът да съдържа еднакви елементи. В такъв случай, в първата версия решението е 0, а във втората, наредена двойка (a_i, a_j) за кои да е a_i, a_j , такива че $a_i = a_j$.

Не е задължително елементите да са цели числа, но трябва да бъдат от множество, върху което е дефинирана пълна преднаредба, и трябва да е дефинирано разстояние между всеки два елемента, което да се изчислява в константно време. В горната дефиниция разстоянието е $|a_i - a_j|$.

Наивен алгоритъм за задачата е да се тестват всички двуелементни подмножества така.

ALG CLOSEST PAIR, NAÏVE($A[1..n]$)

```

1  closest ← ∞
2  for i ← 1 to n - 1
3      for j ← i + 1 to n
4          if |A[i] - A[j]| < closest
5              closest ← |A[i] - A[j]|
6              first ← A[i]
7              second ← A[j]
8  return (first, second)
```

Сложността по време на този алгоритъм е $\Theta(n^2)$.

По-бърз алгоритъм е: първо елементите да бъдат сортирани и след това с едно единствено сканиране на сортираната редица, да кажем отляво надясно, най-близката двойка ще бъде намерена. Сега не е необходимо да ползваме нотация за абсолютна стойност около $A[i + 1] - A[i]$, защото след сортирането, $A[i + 1] - A[i]$ е задължително неотрицателно.

ALG CLOSEST PAIR, SOPHISTICATED($A[1..n]$)

```

1  SORT(A)
2  closest ← ∞
3  for i ← 1 to n - 1
4      if A[i + 1] - A[i] < closest
5          closest ← A[i + 1] - A[i]
6          first ← A[i]
7          second ← A[i + 1]
8  return (first, second)
```

Доказателството за коректност се основава на простото наблюдение, че ако минималното разстояние между елементи от входа е δ , то след сортирането има два съседни (в сортираната редица) елемента, разстоянието между които е δ .

Както ще видим в следваща лекция, сортирането може да бъде имплементирано със сложност $\Theta(n \lg n)$, така че бързият алгоритъм има сложност

$$\underbrace{\Theta(n \lg n)}_{\text{сортиране}} + \underbrace{\Theta(n)}_{\text{сканиране}} = \Theta(n \lg n),$$

което е много по-добре от $\Theta(n^2)$.

4.2.3 Уникалност на елементите

Изн. Задача 19: УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ (ELEMENT UNIQUENESS / EU)

екземпляр: $a_1, a_2, \dots, a_n$: цели числа; $n \geq 2$.

въпрос: Вярно ли е, че няма нито два еднакви елемента измежду дадените?

Напълно аналогично на НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, наивният алгоритъм се състои в тестване на всички двуелементни подмножества, което означава $\Theta(n^2)$ тестове в най-лошия случай.

```

ALG ELEMENT UNIQUENESS, NAÏVE( $A[1..n]$ )
1  for  $i \leftarrow 1$  to  $n - 1$ 
2      for  $j \leftarrow i + 1$  to  $n$ 
3          if  $A[i] = A[j]$ 
4              return FALSE
5  return TRUE

```

Ако обаче първо сортираме числата, всяка група елементи, които са два по два еднакви, задължително ще се появи като непрекъсната подредица на сортираната редица. С едно сканиране отляво надясно ще установим дали има еднакви елементи.

```

ALG ELEMENT UNIQUENESS, SOPHISTICATED( $A[1..n]$ )
1  SORT( $A$ )
2  for  $i \leftarrow 1$  to  $n - 1$ 
3      if  $A[i] = A[i + 1]$ 
4          return FALSE
5  return TRUE

```

Отново, сложността на подхода, използващ първо сортиране във време $\Theta(n \lg n)$, е $\Theta(n \lg n)$.

4.2.4 Откриване на еднакви елементи в два масива

Да кажем, че са дадени два масива от уникални числа $A[1..n]$ и $B[1..m]$ и искаме да намерим общите им елементи. В третия том на ТАОСР на Knuth [94, стр. 2] тази задача, само че обобщена за няколко масива, е наречена *Matching items in two or more files*. Наивният подход е всеки елемент от A да бъде търсен в B (или обратното) с последователно търсене. В екстремния случай $n = \Theta(1)$ или $m = \Theta(1)$ този подход води до линеен алгоритъм, но това е тривиализъм. В общия случай обаче, наивният подход води до алгоритъм със сложност $\Theta(mn)$. Следният алгоритъм е по-бърз.

```

ALG MATCHING IN ARRAYS( $A[1..n], B[1..m]$ )
1  SORT( $A$ )
2  SORT( $B$ )
3   $i \leftarrow 1, j \leftarrow 1$ 
4  while  $i \leq n$  and  $j \leq m$  do
5      if  $A[i] = B[j]$ 
6          print " $A[i]$  съвпада с  $B[j]$ "
7           $i++$ ,  $j++$ 
8      else if  $A[i] < B[j]$ 
9           $i++$ 
10     else
11          $j++$ 

```

Ако се използва ефикасно сортиране и $m \asymp n$, сортирането на двата масива става във време $\Theta(n \lg n)$. Това доминира над времето $\Theta(n)$ за втората фаза от алгоритъма (самото търсене), така че сложността на алгоритъма е $\Theta(n \lg n)$.

4.2.5 Мода

Мода е най-често срещана стойност измежду дадени стойности. Забележете, че тази стойност не е непременно уникална – ако няма повторения на елементи, всеки елемент е мода (поради което не е коректно да се казва “най-често срещаната стойност”). Като изчислителна задача, МОДА е задача за търсене.

Мода може да се намери във време $\Theta(n \lg n)$ чрез предварително сортиране, което работи във време $\Theta(n \lg n)$. Ето един ефикасен алгоритъм за намиране на мода в $\Theta(n \lg n)$ чрез сортиране, последвано от едно сканиране на сортираните данни.

```

COMPUTE MODE(A[1 .. n])
1  SORT(A)
2  mode ← A[1]
3  m ← 1
4  for i ← 2 to n
5      if A[i - 1] = A[i]
6          if A[i] = mode
7              m ++
8          else
9              s ++
10             if s > m
11                 mode ← A[i]
12                 m ← s
13         else
14             s ← 1
15  return mode

```

Допълнение 29: Коректността на COMPUTE MODE

Както вече казахме, модата не е непременно уникална. Алгоритъмът връща *най-малката мода*. За краткост, навсякъде до края на Допълнение 29, когато казваме “мода” имаме предвид най-малката мода.

Следното определение е в сила **само в рамките на това доказателство**.

Определение 48: група

Нека $A[1 .. n]$ е масив от числа. *Група в A* е всеки максимален по включване подмасив от еднакви елементи. *Дължината на групата* е броят на елементите ѝ. Ако групата се казва X , дължината ѝ се бележи с $|X|$. *Представител на групата* е стойността на който да е от нейните елементи.

По отношение на даден индекс $i \in \{2, 3, \dots, n\}$, казваме, че $A[i]$ *разширява група*, ако $A[i] = A[i - 1]$, и че $A[i]$ *започва нова група*, ако $A[i] \neq A[i - 1]$. $A[i]$ *разширява група-мода*, ако $A[i]$ разширява група и $A[i - 1]$ е мода на $A[1 .. i - 1]$. $A[i]$ *превръща група-немода в група-мода*, ако $A[i]$ разширява група и $A[i - 1]$ не е мода на $A[1 .. i - 1]$, но $A[i]$ е мода на $A[1 .. i]$. $A[i]$ *разширява група-немода*, ако $A[i]$ разширява група, но не е мода на $A[1 .. i]$. $\square$

Забележете, че ако масивът е сортиран и $A[i]$ започва нова група, то $A[i]$ не може да

е мода при $i \geq 2$.

Наблюдение 33

Нека $A[1 \dots n]$ има точно t различни елемента и е сортиран. Тогава A се състои от точно t групи, чиято сумарна дължина е n и чиито представители се появяват в строго нарастващ ред.

Наблюдение 34

За всяко $i \in \{2, 3, \dots, n\}$, има точно четири, две по две изключващи се, възможности за $A[i]$. Ще бележим тези възможности с буквите $Q_1, \dots, Q_4$.

Q_1 : $A[i]$ разширява група-мода.

Q_2 : $A[i]$ разширява група-немода.

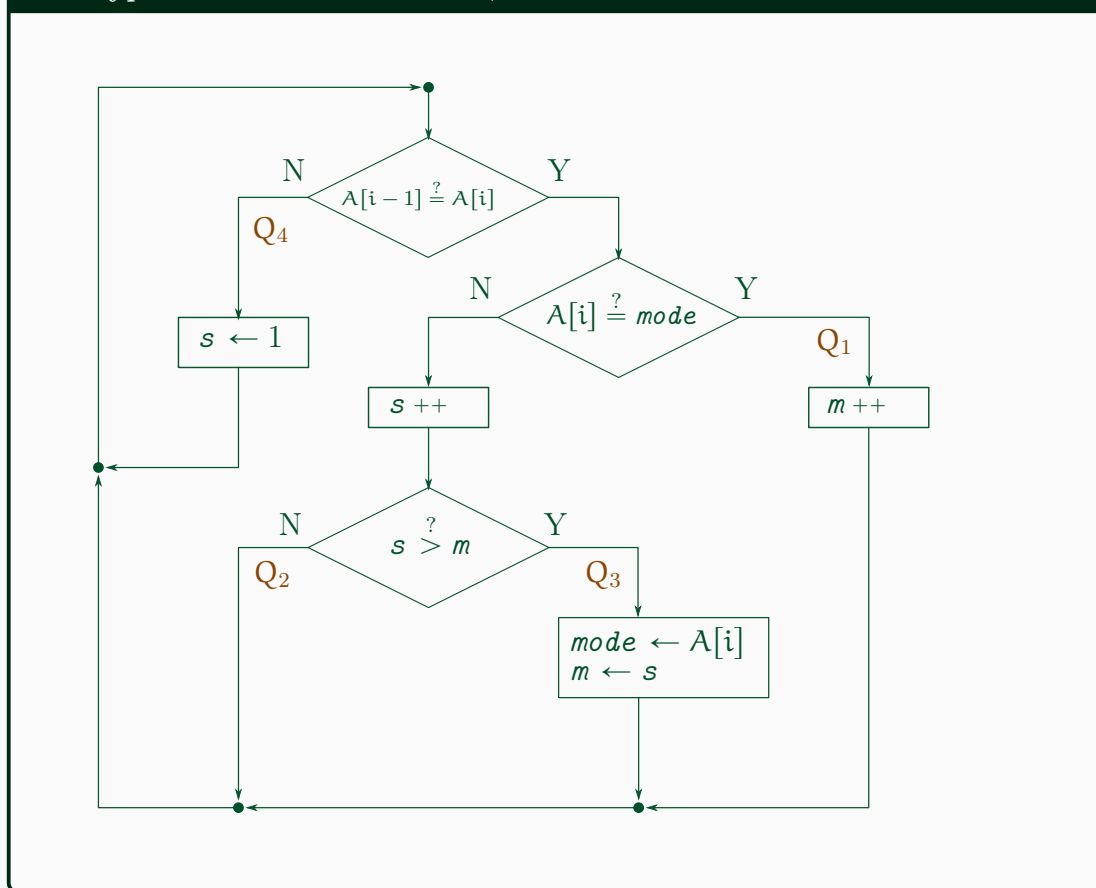
Q_3 : $A[i]$ превръща група-немода в група-мода.

Q_4 : $A[i]$ започва нова група.

Удобно е да мислим за $Q_1, \dots, Q_4$ като за *състояния*. След сортирането, алгоритъмът сканира еднократно масива отляво надясно и след прочитането на всеки пореден елемент преминава неявно в едно от четирите състояния. Неявно, защото никъде не се говори експлицитно за състояния. Но трите вложени **if**-а в някакъв смисъл ни дават именно това: четири различни състояния. Най-вътрешният **if** няма **else**, но това е без значение – можем да мислим, че има празен **else**. При всяко преминаване през тялото на цикъла, в зависимост от истинността на трите булеви условия, точно едно от четирите възможни състояния се реализира (имплицитно). Фигура 4.1 показва блок-схема на тялото на цикъла на COMPUTE MODE, на която ясно се вижда кое преминаване през тялото на цикъла на кое състояние отговаря.

Въвеждането на тези четири състояния няма отношение към доказателството за коректност – то не ги използва никъде. Въвеждаме тези четири състояния, за да обосновем конструкцията на алгоритъма. Иначе казано, те са полезни за **дизайна**, а не за **анализа** на алгоритъма.

Фигура 4.1 : Блок-схема на цикъла на COMPUTE MODE.



Следното твърдение е инвариант на **for**-цикъла (редове 1–14).

Инвариант 6: COMPUTE MODE

При всяко достигане на ред 4 на COMPUTE MODE:

- ① *mode* съдържа стойността на модата на $A[1 \dots i-1]$.
- ② *m* съдържа броя на нейните появи в $A[1 \dots i-1]$.
- ③ Ако $A[i-1]$ не е мода на $A[1 \dots i-1]$, то *s* съдържа дължината на най-дясната група в $A[1 \dots i-1]$.

База. Нека изпълнението е на ред 4 за първи път. Тогава $i = 2$ и подмасивът $A[1 \dots i-1]$ е всъщност $A[1]$. Модата на $A[1]$ очевидно е $A[1]$ и броят на нейните появи е 1. Да видим дали всички съждения на инварианта са верни.

- ① $mode = A[1]$ заради присвояването на ред 2. ✓
- ② $m = 1$ и заради присвояването на ред 3. ✓
- ③ Това твърдение също е вярно. Забележете, че *s* не е инициализирана в този момент, така че съдържанието ѝ е недефинирано, но това няма значение за истинността на ③! ③ е импликация, чийто антецедент е лъжа, понеже $A[i-1]$ е мода на $A[1 \dots i-1]$. Импликацията, чийто антецедент е лъжа, е истина независимо от консеквента. ✓

Поддръжка. Да допуснем, че инвариантът е в сила на някое изпълнение на ред 4, което не е последно.

Случай I $A[i-1] = A[i]$. Условието на ред 5 е истина и изпълнението отива на ред 6.

Случай I.1 $A[i-1]$ е мода на $A[1..i-1]$. Този подслучай е Q_1 : $A[i]$ разширява група-мода.

От индуктивното предположение знаем, че модата на $A[1..i-1]$ се съдържа в $mode$. Следва, че $A[i-1] = mode$. Щом $A[i-1] = mode$ и $A[i-1] = A[i]$, в сила е $A[i] = mode$. Условието на ред 6 е истина и изпълнението отива на ред 7. Да видим дали всички съждения на инварианта остават истина.

- ① Щом $mode$ е мода на $A[1..i-1]$ и $A[i] = mode$, то $mode$ е мода и на $A[1..i]$. При следващото достигане на ред 4, i се инкрементира неявно. Спрямо новото i е вярно, че $mode$ е мода на $A[1..i-1]$. ✓
- ② Изпълнява се ред 7 и m се инкрементира. Сега вече m съдържа броя на появите на модата на $A[1..i]$. При следващото достигане на ред 4, i се инкрементира неявно. Спрямо новото i е вярно, че m съдържа броя на появите на модата на $A[1..i-1]$. ✓
- ③ Знаем, че $A[i-1]$ е мода на $A[1..i-1]$ преди инкрементирането на i . От това и от факта, че $A[i-1] = A[i]$ преди инкрементирането на i следва, че $A[i]$ е мода на $A[1..i]$ преди инкрементирането на i . Забелязваме, че след инкрементирането на i това съждение става “ $A[i-1]$ е мода на $A[1..i-1]$ ”. И така, отново е вярно, че $A[i-1]$ е мода на $A[1..i-1]$. Следователно, антецедентът на импликацията в ③ е лъжа. Ерго, импликацията е истина. ✓

Случай I.2 $A[i-1]$ не е мода на $A[1..i-1]$. Тъй като $A[i-1] = A[i]$, $A[i]$ не е мода на $A[1..i-1]$. Съгласно индуктивното предположение, $mode$ съдържа стойността на модата на $A[1..i-1]$. Тогава $A[i] \neq mode$. Тогава условието на ред 6 е лъжа и изпълнението отива на ред 9.

Първо ще покажем, че част ③ на инварианта се запазва и при двете възможности—лъжа или истина—за булевото условие на ред 10. Тези две възможности отговарят съответно на Q_2 : $A[i]$ разширява група-немода и Q_3 : $A[i]$ превръща група-немода в група-мода. Щом $A[i-1]$ не е мода на $A[1..i-1]$, то част ③ от индуктивното предположение казва, че s съдържа дължината на най-дясната група в $A[1..i-1]$. Става дума за момента, в който изпълнението е на ред 8. Изпълнява се ред 9, където s се инкрементира. Тъй като $A[i-1] = A[i]$, новата стойност на s съдържа дължината на най-дясната група в $A[1..i]$. Независимо от това дали булевото условие на ред 10 е истина или лъжа, s не се променя до следващото достигане на ред 4 (редове 11 и 12 не променят s). При следващото достигане на ред 4, i се инкрементира неявно. Спрямо новата стойност на i е вярно, че s съдържа дължината на най-дясната група в $A[1..i-1]$. И така, от една страна текущото $A[i]$ не е мода на текущия $A[1..i-1]$, а от друга страна s съдържа дължината на най-дясната група в текущото $A[1..i-1]$. Следва, че част ③ на инварианта се запазва в **Случай I.2**. ✓

Сега ще разгледаме части ① и ② на инварианта. Да се върнем към момента, в който изпълнението беше на ред 9. Помним, че променливата s , съгласно индуктивното

предположение, съдържа дължината на най-дясната група на $A[1 \dots i-1]$. След инкрементирането на s на ред 9 новата стойност на s съдържа дължината на най-дясната група в $A[1 \dots i]$, понеже $A[i-1] = A[i]$. Изпълнението отива на ред 10.

Случай I.2.a $A[i]$ не е мода на $A[1 \dots i]$. В текущия контекст, това е същото като да кажем, че дължината на най-дясната група на $A[1 \dots i]$ не е по-голяма от дължина от броя на появите на модата на $A[1 \dots i]$. Но видяхме, че дължината на най-дясната група на $A[1 \dots i]$ се съдържа в променливата s . От друга страна, броя на появите на модата на $A[1 \dots i]$ се съдържа в променлива m от индуктивното предположение. Заключаваме, че $s \leq m$, а **Случай I.2.a** отговаря на Q_2 : $A[i]$ разширява група-немода.

Щом $s \leq m$, то условието на ред 10 е лъжа и изпълнението отива на ред 4. $mode$ и m остават непроменени. Променливата i се инкрементира. Спрямо новата ѝ стойност, $mode$ съдържа модата на $A[1 \dots i-1]$, а m съдържа броя на появите ѝ. Показахме, че части ① и ② на инварианта се запазват.

Случай I.2.b $A[i]$ е мода на $A[1 \dots i]$. В текущия контекст, това е същото като да кажем, че най-дясната група на $A[1 \dots i]$, чиято дължина се съдържа в променливата s , има по-голяма дължина от броя на появите на модата на $A[1 \dots i-1]$, който брой се съдържа в променлива m (от индуктивното предположение). Тогава $s > m$, а s съдържа броя на появите на модата на $A[1 \dots i]$. И така, $A[i-1]$ не е мода на $A[1 \dots i-1]$ в **Случай I.2**, но същата стойност е мода на $A[1 \dots i]$, понеже $A[i]$ е равна на $A[i-1]$ в **Случай I**. Заключаваме, че **Случай I.2.b** отговаря на Q_3 : $A[i]$ превръща група-немода в група-мода.

Щом $s > m$, то условието на ред 10 е истина и се изпълняват редове 11 и 12. Сега $mode$ съдържа модата на $A[1 \dots i]$, а m съдържа броя на появите ѝ. После променливата i се инкрементира. Спрямо новата ѝ стойност, $mode$ съдържа модата на $A[1 \dots i-1]$, а m съдържа броя на появите ѝ. Показахме, че части ① и ② на инварианта се запазват.

Случай II Изпълнението е на ред 5 и $A[i-1] \neq A[i]$. Лесно се вижда, че в този случай $A[i]$ не може да е мода на $A[1 \dots i]$. Следователно, модата на $A[1 \dots i]$ е същата като на $A[1 \dots i-1]$, и броят на появите ѝ в $A[1 \dots i]$ е същият като в $A[1 \dots i-1]$. Съгласно индуктивното предположение, $mode$ съдържа модата на $A[1 \dots i-1]$, а m съдържа броя на появите ѝ в $A[1 \dots i-1]$. Заключаваме, че в този момент $mode$ съдържа модата на $A[1 \dots i]$, а m съдържа броя на появите ѝ в $A[1 \dots i]$.

Условието на ред 5 е лъжа и изпълнението отива на ред 13, след което на ред 14, променливата s става единица и изпълнението отива на ред 4, като i се инкрементира неявно. Спрямо новата стойност на i :

- ① $mode$ е мода на $A[1 \dots i-1]$. ✓
- ② m съдържа броя на появите на модата на $A[1 \dots i-1]$. ✓
- ③ От една страна, $A[i-1]$ не е мода на $A[1 \dots i-1]$, от друга страна най-дясната група на $A[1 \dots i-1]$ има дължина единица, а от трета страна s съдържа единица. ✓ □

Модата не може да бъде изчислена във време, което не е $\Omega(n \lg n)$. Това е доказано в Подсекция 13.3.2.

4.2.6 Медиана

Нека са дадени числа $a_1, a_2, \dots, a_n$, не непременно различни. *Медиана* наричаме всяко a_i , такова че за останалите числа е вярно, че половината са по-малки или равни на a_i , а другата половина са по-големи или равни на a_i . Строго погледнато, тази дефиниция на медиана е смислена само когато n е нечетно. Следното определение е смислено за произволно n .

Определение 49: Медиана

Нека $A[1..n]$ е масив от цели числа. *Медианата* на A е елементът $A\left[\left\lfloor \frac{n+1}{2} \right\rfloor\right]$ след сортиране на A .

Съществува още по-детайлно определение на “медиана”. Тъй като при четно n има два естествени кандидата за медиана, това определение казва, че масивите с четен брой елементи имат две медиани: елементът на позиция $\left\lfloor \frac{n+1}{2} \right\rfloor$ в сортирания масив е *долната медиана*, а този на позиция $\left\lceil \frac{n+1}{2} \right\rceil$ в сортирания масив е *горната медиана*. Вижте [34, стр. 213].

Като изчислителна задача, МЕДИАНА е задача за търсене.

Изч. Задача 20: МЕДИАНА

екземпляр: $a_1, a_2, \dots, a_n$: цели числа.

решение: медианата на $a_1, a_2, \dots, a_n$.

Медиана се пресмята лесно, ако използваме сортиране, понеже самата дефиниция на медиана използва сортиране неявно: медианата е елементът, стоящ в средата на сортираната редица на тези елементи. В следния алгоритъм $(n/2)$ означава целочислено деление.

COMPUTE MEDIAN($A[1..n]$, n е нечетно)

```
1  SORT(A)
2  return  $A[(n/2) + 1]$ 
```

Медианата може да бъде намерена в линейно време чрез алгоритъма, намиращ k -тото по големина число измежду n дадени числа в линейно време, за произволно k . Този алгоритъм може да бъде намерен в [34], а и ние ще го дискутираме в Подсекция 6.2.2.

4.2.7 2SUM, 3SUM, kSUM

Нека навсякъде в тази подсекция k е цяло число, по-голямо или равно на 2.

Дефиниция на задачата. Широко разпространеното разбиране е, че това е задача за разпознаване.

Изч. Задача 21: kSUM, версия за РАЗПОЗНАВАНЕ

екземпляр: Мултимножество от реални числа S .

въпрос: Дали има подмултимножество $\{a_1, \dots, a_k\}_M \subseteq S$, такова че $a_1 + \dots + a_k = 0$?

Естествено обобщение на Задача 21 е следната задача за търсене.

Изч. Задача 22: kSUM, версия за търсене**екземпляр:** Множество от реални числа S .**решение:** Броят на подмножествата $\{a_1, \dots, a_k\} \subseteq S$, такива че $a_1 + \dots + a_k = 0$.

Задачите 2SUM и 3SUM са частни случаи при съответно $k = 2$ и $k = 3$. В тази подсекция ще се фокусираме върху 2SUM и 3SUM, като последната сякаш е най-важна на практика заради приложенията си в изчислителната геометрия.

Как помага сортирането. За всяко k има наивен алгоритъм със сложност $\Theta(n^k)^{\dagger}$, който просто опитва всички подмножества с мощност k . Ако обаче числата от S са предварително сортирани, то можем да “свалим” една единица от степенния показател: за 2SUM има алгоритъм със сложност $\Theta(n)$ (тук **не отчитаме** времето за сортирането!), за 3SUM има алгоритъм със сложност $\Theta(n^2)$, и така нататък.

В Подподсекция 2.1.3.4 видяхме такъв алгоритъм за 2SUM във версия за разпознаване, наречен ALG 2SUM. Там се казваше, че числата са цели, но това не е съществено по никакъв начин. Нищо не се променя, ако числата са реални, при допускането, че сложността на базовите операции върху число е константна.

ALG 2SUM решава вариант на задачата, в който се иска числата a_1 и a_2 да се сумират не непременно до нула, а до някакво число m , което е част от входа. Този вариант обаче не е съществено различен от “класическия” вариант на сумиране до нула. Забележете, че $a_1 + a_2 = m$ е същото като $(a_1 - m/2) + (a_2 - m/2) = 0$, така че привидно по-общият вариант се свежда до “класическия”, ако предварително извадим $m/2$ от всяко число.

Напълно аналогично, алгоритъмът ALG 3SUMCOUNT от Подподсекция 2.1.3.5, който е за задачата във версия за търсене, в който трите числа се сумират до нула и това е заложено в кода, може да бъде използван за намиране на броя на тройките, които се сумират до каквото число m пожелаем: просто изваждаме $m/3$ от всички числа в началото и тогава

$$a_1 + a_2 + a_3 = m \leftrightarrow (a_1 - m/3) + (a_2 - m/3) + (a_3 - m/3) = 0$$

Ако във варианта за търсене, S е мултимножество? В Задача 22 се казва, че S е множество. Ерго, не се допуска екземплярът да има еднакви числа. Възможността да има повтарящи се числа във входа е без значение за задачата във версията за разпознаване, но има отношение към версията за търсене. Лесно се вижда, че ALG 3SUMCOUNT от Подподсекция 2.1.3.5 не работи върху мултимножества: при всяко откриване на триада на ред 6 алгоритъмът инкрементира с единица брояча d на ред 7, а на ред 8 “мръдва” и двата индекса p и q един към друг. Това е коректно само ако елементите от входа са уникални; ако има повторения, може да “изтървем” триади по този начин. Прочее, в екстремния случай, в който входът се състои само от нули, изходът трябва да е $\binom{n}{3}$. Веднага се вижда, че ALG 3SUMCOUNT не работи коректно върху вход само от нули, защото алгоритъмът има сложност по време $\Theta(n^2)$, а изходът трябва да е $\Theta(n^3)$: невъзможно е само с инкрементации на d с единица, които са най-много квадратичен брой, да постигнем изход, чиято големина е кубична. По друг начин казано, всеки коректен алгоритъм за задачата 3SUM във версия за търсене, който акумулира резултата в брояч, който брояч се инкрементира с единица и не се манипулира по никакъв друг начин, има сложност по време $\Omega(n^3)$, ако входът е мултимножество.

[†]Сложността на наивния алгоритъм е $\Theta(n^k)$ само ако смятаме, че k е константа. В противен случай сложността е $\Omega(k \binom{n}{k})$, понеже подмножествата с мощност k са точно $\binom{n}{k}$, а всяко от тях бива тествано във време $\Theta(k)$.

Заслужава да се отбележи, че 3SUM във версия за търсене може да се реши и с квадратичен алгоритъм дори върху екземпляри-мултимножества, ако търсим само бройката на триадите, а не самите триади. Това може да стане, ако след предварителното сортиране “кондензираме” числения масив, като наместо всички появи на едно и също число запишем само числото и броя на неговите появи. Иначе казано, кондензираният масив е от наредени двойки (x, y) , където x е число от входа, а y е броят на неговите появи във входа. След това използваме идеята на ALG 3SUMCOUNT с незначителната разлика, че при откриване на триада икrementираме d така: $d \leftarrow d + \alpha \cdot \beta \cdot \gamma$, където α е броят на срещанията на $A[i]$, β е броят на срещанията на $A[p]$, а γ е броят на срещанията на $A[q]$.

Този трик обаче открива само броя на триадите, в които няма повторения. Към този брой трябва да добавим следните две числа.

- Броят на триадите, състоящи се от три елемента от входа с еднаква стойност. Тази стойност може да е само нула, така че добавяме $\binom{k_0}{3}$, където k_0 е броят на нулите във входа.
- Броят на триадите, състоящи се от два елемента с една и съща стойност s и още един елемент с различна стойност t . Трябва или $s < 0$ и $t > 0$, или $s > 0$ и $t < 0$. Освен това трябва $|s| = |t|/2$. Нека броят на елементите във входа със стойност s е α , а броят на елементите във входа със стойност t е β . Добавяме $\binom{\alpha}{2} \cdot \beta$ за всяка такава двойка s, t , понеже всеки два елемента със стойност s , комбинирани с един елемент със стойност t , са една триада. Откриването на тези двойки може да стане в линейно време върху сортирания, кондензиран масив с идея, подобна на идеята на ALG 2SUM.

Приложения на задачата 3SUM. Редица важни задачи в областта на компютърната геометрия съдържат в себе си задачата 3SUM в смисъл, че съществува редукция на 3SUM до всяка една от тях. Такива задачи са посочени в статията на Gajentaan и Overmars [55]:

- Дадени са прави в равнината. Дали има три от тях, пресичащи се в една и съща точка?
- Дадени са отсечки, непресичащи се по двойки и всяка е успоредна на някоя от координатните оси. Дали съществува права, която ги разделя на две непразни множества?
- Дадени са райони, всеки от които е между две успоредни прави в равнината. Дали обединението на тези райони съдържа даден правоъгълник?
- Дадени са препятствия в равнината, всяко от които е отсечка. Тези отсечки не се пресичат по двойки и всяка е успоредна на една от координатните оси. Дадена е и пръчка-отсечка. Може ли тази пръчка да се придвижи с транскации и ротации от дадена начална позиция до дадена крайна позиция, без да се сблъска с препятствие?

Понятието “редукция” ще въведем чак в Лекция 13, в Секция 13.3. Тук само ще споменем, че ако бъде открит алгоритъм за 3SUM (във версията за разпознаване) със сложност $O(n^{2-\epsilon})$, за колкото и да е малко реално положително ϵ , за всяка от тези задачи също ще има алгоритъм със сложност $O(n^{2-\epsilon})$; но ако се докаже, че няма алгоритъм за 3SUM със сложност $O(n^{2-\epsilon})$, за никое реално положително ϵ , от това следва, че за никоя от тези задачи няма алгоритъм със сложност $O(n^{2-\epsilon})$.

Иначе казано, всяка от тези задачи се решава чрез едно решаване на 3SUM, плюс “още малко”. Какво ще рече това, ще видим в Секция 13.3 и Подсекция 13.3.5.

4.3 Анализ на сортиращи алгоритми. Стабилност.

Анализът на даден сортиращ алгоритъм се състои в това, което изброихме миналата лекция за анализа на произволен алгоритъм: доказателство на коректността и анализ на сложността по време и по памет. В контекста на сортиращите алгоритми има още едно свойство, което представлява интерес. То се нарича *стабилност*. Сега ще обясним в детайли какво означава сортиращ алгоритъм да бъде стабилен с един пример от култовата книга The Art of Computer Programming на Knuth [94]. Knuth означава с N броя на елементите във входа, а сортираната пермутация с “ $K_{p(1)} \leq K_{p(2)} \leq \dots \leq K_{p(N)}$ ”.

Задача 49: задача 2 на стр. 5 в [94]

Да допуснем, че всеки запис R_j в даден файл съдържа два ключа, първичен ключ K_j и вторичен ключ k_j , като върху ключовете е дефинирана линейна наредба $<$. Дефинираме *лексикографска наредба* върху ключовете по стандартния начин:

$$(K_i, k_i) < (K_j, k_j) \text{ ако } K_i < K_j \text{ или } K_i = K_j \text{ и } k_i < k_j$$

Alice взема файла и го сортира първо по първичните ключове, получавайки n групи от записи с еднакви първични ключове във всяка група:

$$K_{p(1)} = \dots = K_{p(i_1)} < K_{p(i_1+1)} = \dots = K_{p(i_2)} < \dots < K_{p(i_{n-1}+1)} = \dots = K_{p(i_n)},$$

където $i_n = N$. След това тя сортира всяка от n на брой групите по вторичния ключ. Bill взема оригиналния файл и първо го сортира по вторичния ключ, а след това, по първичния.

Chris взема оригиналния файл и прави едно единствено сортиране, като обаче ползва и двата ключа с лексикографската наредба.

Дали всеки от тях е получил непременно един и същи резултат?

Отговорът е, със сигурност Alice и Chris ще получат един и същи резултат. Резултатът на Bill ще е гарантирано същият само при условие, че неговото второ сортиране—това по първичния ключ—не променя относителния ред на записи, имащи еднакъв (първичен) ключ, който ред е създаден от вече извършеното първо сортиране. Ето малък пример: нека наредените двойки от ключовете са $(1, 2)$, $(5, 2)$, $(4, 5)$, $(2, 5)$, $(3, 1)$, $(1, 4)$, $(3, 2)$. Сортираната с лексикографска наредба редица е:

$$(1, 2), (1, 4), (2, 5), (3, 1), (3, 2), (4, 5), (5, 2)$$

Това е резултатът, който ще получат Alice и Chris след своите сортирания.

Какво точно ще получи Bill след **първото** сортиране—това по вторичните ключове—зависи от сортиращия алгоритъм, който ползва. Със сигурност

- на първа позиция ще е наредената двойка $(3, 1)$,
- после ще са трите наредени двойки с втори елемент 2, но не е ясно в какъв ред
- после $(1, 4)$,
- и накрая двете наредени двойки с втори елемент 5, но не е ясно в какъв ред.

Примерно, Bill може да получи това след първото сортиране:

$$(3, 1), (5, 2), (3, 2), (1, 2), (1, 4), (4, 5), (2, 5)$$

но може да получи и това:

$$(3, 1), (3, 2), (5, 2), (1, 2), (1, 4), (2, 5), (4, 5)$$

Сега Bill прави **второ** сортиране, този път по първичните ключове. В получената наредба на първите две места са наредените двойки $(1, 2)$ и $(1, 4)$. Но забележете, че при произволно сортиране по първи ключ няма как да знаем дали $(1, 2)$ ще е вляво от $(1, 4)$ или не. От гледна точка на коректно лексикографско сортиране, разбира се, искаме началото да бъде

$$(1, 2), (1, 4), \dots$$

но може да се получи и

$$(1, 4), (1, 2), \dots$$

тъй като при сортирането по първични ключове не “гледа” вторичните ключове. **По отношение на сортирането само по първични ключове, $(1, 2)$ и $(1, 4)$ са еднакви елементи!** Първата фаза—сортирането по вторични ключове—със сигурност слага $(1, 2)$ преди $(1, 4)$, защото $2 < 4$, но втората фаза може потенциално да сложи $(1, 4)$ вляво от $(1, 2)$, ако ги счита за еднакви.

Определение 50: Стабилно сортиране

Стабилно сортиране е сортиране, което не променя взаимното разположение на елементи с еднакви ключове.

При стабилно сортиране, за всеки два елемента с еднакви ключове, които елементи са все пак различни (заради, примерно, вторични ключове), този, който е вляво от другия **преди** сортирането, задължително ще е вляво от него и **след** сортирането.

Наблюдение 35

Всеки нестабилен сортиращ алгоритъм може да бъде направен стабилен, ако се добави първоначалната позиция на всеки елемент като вторичен ключ^a и се сортира по наредените двойки (първичен ключ, вторичен ключ) – като Chris от Задача 49. Ако направим това обаче, може да влошим сложността по памет, понеже $\Theta(n)$ допълнителна памет ще се използва от новите ключове, използвани за стабилизиране. Ако преди това изкуствено стабилизиране сложността по памет е била $\Theta(1)$, ще я влошим до $\Theta(n)$. Следователно, няма как да използваме този трик върху алгоритъм, който е in-place и той да остане in-place.

^aТова е в случай, че поначало има само един ключ. Ако поначало има k ключа, добавяме още един, $(k + 1)$ -ви, най-младши ключ с първоначалните позиции.

4.4 Елементарни Сортиращи Алгоритми

Разделянето на сортиращите алгоритми на елементарни и други, неелементарни, е доста условно. Под елементарни сортиращи алгоритми разбираме такива, които биха хрумнали бързо на човек, който за пръв път се сблъсква със СОРТИРАНЕ. По правило те работят във време $\Theta(n^2)$ в най-лошия случай и не използват никакви изтънчени структури данни или нетривиални идеи.

4.4.1 INSERTION SORT

INSERTION SORT сортира по начин, който далечно напомня на начина, по който картоигращ сортира картите, които държи в ръка (които са в произволна наредба в началото) – плъзгайки поглед отляво надясно, той или тя разглежда последователно картите. При това всяка карта, която е обект на внимание в момента, бива сложена на правилното ѝ място в подредицата от картите вляво. Има една важна разлика между този начин на нареждане и работата на алгоритъма INSERTION SORT: карта може да бъде пъхната между другите карти мигновено, докато ако искаме да “пъхнем” дадено a_i някъде вляво, първо трябва да му направим място, за да не бъде затрит елементът, който вече е там.

Да си припомним псевдокода на INSERTION SORT.

Алгоритъм 5: Insertion Sort

```

INSERTION SORT( $A[1..n]$ : array of integers)
1  for  $i \leftarrow 2$  to  $n$ 
2       $key \leftarrow A[i]$ 
3       $j \leftarrow i - 1$ 
4      while  $j > 0$  and  $A[j] > key$  do
5           $A[j + 1] \leftarrow A[j]$ 
6           $j--$ 
7       $A[j + 1] \leftarrow key$ 

```

Съгласно конвенцията, спомената на стр. 30, оценяването на конюнкцията на ред 4 става отляво надясно, като при това, ако не е вярно, че $j > 0$, то не се проверява дали $A[j] > key$.

Коректността на INSERTION SORT. Доказателството за коректност ще направим чрез инвариант на цикъла. Целта е да докажем, че INSERTION SORT сортира всеки свой вход.

Допълнение 30: Неформално за доказателствата за коректност

Доказателствата за коректност на алгоритми са тежки, защото обектите, за които трябва да изказваме и доказваме твърдения, са динамични. Променливите се менят с работата на алгоритъма заради присвояванията. Ако в доказателството за коректност на INSERTION SORT говорим за $A[i]$, какво имаме предвид? Дали елемента в клетка i на входа, тоест преди началото на алгоритъма? Или елемента, който е бил в клетка i на масива в началото на текущата итерация на външния цикъл? Или имаме предвид клетка i в даден конкретен момент от изпълнението, който момент разглеждаме в доказателството? В общия случай това са различни елементи.

В това допълнение ще дадем интуицията зад формалното доказателство за коректността, а Лема 23 и Теорема 56 ще бъдат сухото и формално доказателство за коректност.

Най-общо казано, схемата на доказателството на коректността на INSERTION SORT е следната. Това, което искаме да докажем е, че алгоритъмът терминира върху всеки вход и винаги, когато терминира, масивът A се състои от началните елементи, но в сортиран вид. Това е прекалено общо и е в сила за всеки сортиращ алгоритъм.

Сега да говорим конкретно за INSERTION SORT. Схемата за доказателството на коректността му е следната. Разглеждаме **само едно изпълнение** на външния цикъл (редове 1–7). По отношение на **това изпълнение** на външния цикъл, ефектът от работата на алгоритъма е следният. Ако в начало му е вярно, че:

- подмасивът $A[1 \dots i - 1]$ се състои от същите елементи като входния подмасив $A[1 \dots i - 1]$, но в сортиран вид
- и подмасивът $A[i \dots n]$ е същият като входния $A[i \dots n]^a$,

то в самия му край—това означава, точно след изпълнението на ред 7, но преди следващото инкрементиране на управляващата променлива i на ред 1—е вярно, че:

- подмасивът $A[1 \dots i]$ се състои от същите елементи като входния $A[1 \dots i]$, но в сортиран вид,
- и подмасивът $A[i + 1 \dots n]$ е същият като входния $A[i + 1 \dots n]$.

Разговорно казано, сортираната зона в A “върви” от левия край надясно и на всяко изпълнение на външния цикъл нараства с една клетка, но задължително се състои от същите елементи, които са били в нея в началото на алгоритъма; а частта извън сортираната зона, която пък намалява с една клетка, също се състои от същите елементи, които са били там в началото.

И всичко това формулирахме без изобщо да разглеждаме вътрешния цикъл. Дали то е приемливо за доказателство за коректност на алгоритъма или не, зависи от това, колко прецизно доказателство искаме. Посоченото доказателство е прекалено недетайлно и неформално и, ако искаме голяма прецизност, то е незадоволително — не е доказано, че ефектът от едно изпълнение на външния цикъл е такъв, какъвто тук се твърди, че е. Това не е доказателство, а схема, по която може да бъде направено детайлно доказателство.

Сега ще обясним как става нарастването на сортираната зона с една позиция. Разглеждаме в детайли **едно изпълнение на външния цикъл**. Допускаме, че в началния му момент подмасивът $A[1 \dots i - 1]$ е сортиран и се състои точно от елементите, които са били на позиции $1, \dots, i - 1$ във входа. В този момент $A[i]$ може да “не си е на мястото” в $A[1 \dots i]$, защото $A[i]$ може да е по-малък от някакви елементи на $A[1 \dots i - 1]$. В първата стъпка от изпълнението на **for**-цикъла, в променливата **key** се слага $A[i]$ и после се изпълнява вътрешният цикъл (редове 4–7). Ефектът от работата на вътрешния цикъл е следният. Нека t_1 е моментът на първоначалното достигане на ред 4, тоест самото начало на изпълнението на вътрешния цикъл, а t_2 е моментът на достигане на ред 7, тоест веднага след излизането от вътрешния цикъл. В момента t_2 :

- $A[1 \dots j]$ е същият като $A[1 \dots j]$ в момента t_1 и се състои точно от тези елементи на $A[1 \dots i - 1]$ в момента t_1 , които са по-малки или равни на **key**, в сортиран вид;
- $A[j + 2 \dots i]$ е същият като $A[j + 1 \dots i - 1]$ в момента t_1 и се състои точно от тези елементи на $A[1 \dots i - 1]$ в момента t_1 , които са по-големи от **key**, в сортиран вид;

- клетка $A[j + 1]$ може да се смята за празна, защото стойността, която е била в нея в момента t_1 , е запазена другаде. А именно,
 - ◆ в $A[j + 2]$, ако вътрешният цикъл е бил изпълнен поне веднъж, или
 - ◆ в *key*, в противен случай.

Разговорно казано, ако гледаме само $A[1 \dots i]$, в момента t_2 , $j + 1$ е “правилното място” на елемента, който се е намирал в $A[i]$ в момента t_1 . “Правилното място” е по отношение на елементите, които са по-малки или равни на него. Те се намират в $A[1 \dots j]$ в момента t_1 и не са били местени от вътрешния цикъл⁶. Вътрешният цикъл пресмята това $j + 1$, като заедно с това премества с единица надясно точно тези елементи на $A[1 \dots i]$, които са по-големи от елемента $A[i]$ в момента t_1 , без да ги размества взаимно; тоест, мести ги като блок.

Това вече е много по-детайлно обяснение, но все още не е достатъчно детайлно – все още не сме видели как действа една итерация на вътрешния цикъл. Сега разглеждаме в детайли **едно изпълнение на вътрешния цикъл**. Нека t_1 и t_2 имат същото значение, каквото имаха досега, и нека t' е моментът на някое достигане на ред 4, което не е последното, а t'' е следващото достигане на ред 4 (очевидно $t_1 < t' < t'' < t_2$). Ако в t' :

- $A[j + 2 \dots i]$ е сортиран и всеки елемент от него е по-голям от *key* и
- $A[1 \dots j]$ е сортиран и е същият като в t_1 и
- клетка $A[j + 1]$ може да се счита за празна

то в t'' :

- $A[j + 1 \dots i]$ е сортиран и всеки елемент от него е по-голям от *key* и
- $A[1 \dots j - 1]$ е сортиран и е същият като в t_1 и
- клетка $A[j]$ може да се счита за празна

Това вече е пълна схема на доказателство, което може да бъде извършено по индукция. Естественият ред на формалното доказателство по отношение на вложените цикли е, отвътре навън.

^аТова е абсолютно очевидно от кода на алгоритъма, но за пълнота го формулираме отделно.

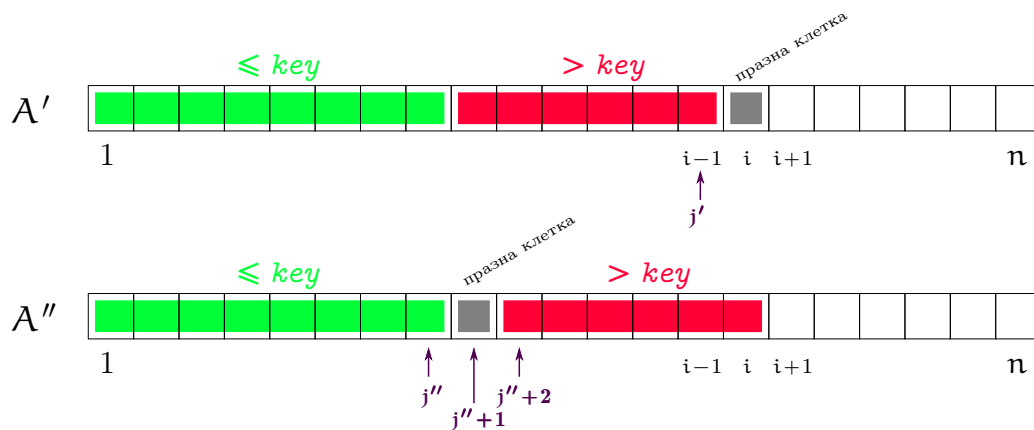
^бЗабележете, че $j + 1$ е правилното място за $A[i]$ по отношение на елементите, които са по-малки или равни на него. По отношение на елементите, които са по-големи от него, правилното място му е j . Тъй като алгоритъмът мести елементите, които са по-големи от $A[i]$, надясно, а оставя на място тези, които са по-малки или равни, правилното място за $A[i]$ е именно $j + 1$.

Лема 23: Коректността на вътрешния цикъл на INSERTION SORT

Разглеждаме едно единствено изпълнение на **for**-цикъла на INSERTION SORT (редове 1–7). Нека t' е моментът от работата на алгоритъма, в който изпълнението е на ред 4 за първи път, а t'' е моментът от работата на алгоритъма, в който изпълнението е на ред 4 за последен път. С A' и j' означаваме съответно масива A и променливата j в момента t' , а с A'' и j'' означаваме съответно масива A и променливата j в момента t'' . Допускаме, че в момента t' подмасивът $A'[1..i-1]$ е сортиран, променливата key съдържа $A'[i]$, а променливата j' съдържа $i-1$. Тогава в момента t'' е вярно, че:

- ① $A''[1..j''] = A'[1..j']$ и неговите елементи са точно тези елементи на $A'[1..i-1]$, които са по-малки или равни на key ;
- ② $A''[j''+2..i] = A'[j''+1..i-1]$ и неговите елементи са точно тези елементи на $A'[1..i-1]$, които са по-големи от key .

Доказателство: Приемаме за очевидно, че стойността на i не се мени между моментите t' и t'' . Образно, лемата твърди следното (променливата key не е нарисувана):



Следното твърдение е инвариант за **while**-цикъла по отношение на едно единствено изпълнение на **for**-цикъла.

Инвариант 7: Вътрешният цикъл на INSERTION SORT

Всеки път, когато изпълнението е на ред 4:

- ① $A[1..j] = A'[1..j]$;
- ② $A[j+2..i] = A'[j+1..i-1]$;
- ③ $\forall x \in A[j+2..i] : x > key$.

База. Разглеждаме момента t' : първото достигане на ред 4.

Ще покажем ①. Текущият масив A е същият като A' , така че наистина $A[1..j] = A'[1..j]$. ✓

Ще покажем ②. По допускане, $j = i-1$ в момента t' . От една страна, $A[j+2..i]$ е $A[i+1..i]$, който подмасив е празен. От друга страна $A'[j+1..i-1]$ е $A'[i..i-1]$, който подмасив също е празен. ✓

Ще покажем ③. То е вярно в празния смисъл, защото $A[j+2..i]$ е празен. ✓

Поддръжка. Да допуснем, че инвариантът е в сила в даден момент t_1 , в който изпълнението е на ред 4 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че $j > 0$ и $A[j] > key$ в момента t_1 . Нека t_2 означава момента на следващото достигане на ред 4. Очевидно $t' \leq t_1 < t_2 \leq t''$.

Ще покажем, че ❶ е в сила в момента t_2 . В момента $t_1 + 1$ се извършва присвояването на ред 5, което не променя нищо в подмасива $A[1..j-1]$. От индуктивното предположение ❶ знаем, че $A[1..j] = A'[1..j]$ в момента t_1 . Заклучаваме, че $A[1..j-1] = A'[1..j-1]$ след присвояването на ред 5. После j се декрементира на ред 6. Спрямо новата стойност на j е вярно, че $A[1..j] = A'[1..j]$ и това остава в сила в момента t_2 . ✓

Ще покажем, че ❷ е в сила в момента t_2 . От индуктивното предположение ❷ знаем, че в момента t_1 е вярно, $A[j+2..i] = A'[j+1..i-1]$. От индуктивното предположение ❶ следва, че в момента t_1 е вярно, че $A[j] = A'[j]$. Тогава след присвояването $A[j+1] \leftarrow A[j]$ на ред 5 е вярно, че $A[j+1..i] = A'[j..i-1]$. После j се декрементира на ред 6. Спрямо новата стойност на j е вярно, че $A[j+2..i] = A'[j+1..i-1]$. ✓

Ще покажем, че ❸ е в сила в момента t_2 . Допуснали сме, че $\forall x \in A[j+2..i] : x > key$ в момента t_1 . Знаем, че $A[j] > key$ в момента t_1 . След присвояването на ред 5 е вярно, че $\forall x \in A[j+1..i] : x > key$. Следва декрементирание на j на ред 6. Спрямо новата стойност на j , в сила е $\forall x \in A[j+2..i] : x > key$. ✓

Терминация. Да разгледаме момента t'' . Както се разбрахме, текущият масив се нарича A'' . В сила е $\neg(j'' > 0 \text{ and } A''[j''] > key)$, което, съгласно закона на De Morgan, е еквивалентно на $j'' \leq 0$ or $A''[j''] \leq key$. Следните два случая са изчерпателни и взаимно изключващи се.

Случай 1: $j'' \leq 0$. Тъй като j намалява с единица, то не може да стане отрицателно, следователно $j'' = 0$. Заместваме j с 0 в части ❷ и ❸ на инварианта и получаваме:

$$\text{❷ } A''[2..i] = A'[1..i-1];$$

$$\text{❸ } \forall x \in A''[2..i] : x > key.$$

От ❷ и ❸ следва, че всички елементи на $A'[1..i-1]$ са по-големи от key . Тогава $A'[1..i-1]$ няма елементи, по-малки или равни на key . От друга страна, масивите $A''[1..j'']$ и $A'[1..j'']$, за които става дума в ❶, са празни, понеже $j'' = 0$. Заклучаваме, че ❶ е вярно, в празния смисъл. ✓

Да се убедим, че ❷ също е вярно. Част ❷ на инварианта става $A''[j''+2..i] = A'[j''+1..i-1]$, щом $A = A''$ и $j = j''$. Остава да се убедим, че елементите на този масив са точно тези елементи на $A'[1..i-1]$, които са по-големи от key . Съгласно ❷ и ❸, всички елементи на $A'[1..i-1]$ са по-големи от key . Но $A'[1..i-1] = A'[j''+1..i-1]$, щом $j'' = 0$. Заклучаваме, че елементите на $A''[j''+2..i] = A'[j''+1..i-1]$ са точно тези елементи на $A'[1..i-1]$, които са по-големи от key . Ерго, ❷ е вярно. ✓

Случай 2: $j'' > 0$, но $A''[j''] \leq key$. Първо ще докажем ❷. Част ❷ на инварианта става $A''[j''+2..i] = A'[j''+1..i-1]$, щом $A = A''$ и $j = j''$. Остава да се убедим, че елементите на този масив са точно тези елементи на $A'[1..i-1]$, които са по-големи от key . Но това следва директно от тези факти:

- $\forall x \in A''[j''+2..i] : x > key;$
- $A''[j''] \leq key;$

- $A''[j''] = A'[j'']$, което следва от част ❶ на инварианта;
- масивът $A'[1 \dots i-1]$ е сортиран.

Сега ще докажем ❶. Това, че $A''[1 \dots j''] = A'[1 \dots j'']$, се получава директно от част ❶ на инварианта при заместването на j с j'' и на A с A'' . Сега ще докажем, че елементите на този масив са точно тези елементи на $A'[1 \dots i-1]$, които са по-малки или равни на key . Но това следва директно от вече доказвания факт, че елементите на $A'[1 \dots i-1]$, които са по-големи от key , са елементите на $A'[j''+1 \dots i-1]$. $\square$

Теорема 56: Коректността на INSERTION SORT

INSERTION SORT е сортиращ алгоритъм.

Доказателство: Да наречем входа A_0 . Следното твърдение е инвариант за **for** цикъла:

Инвариант 8: Външният цикъл на INSERTION SORT

Всеки път, когато изпълнението на INSERTION SORT е на ред 1, текущият подмасив $A[1 \dots i-1]$ се състои от същите елементи като $A_0[1 \dots i-1]$, но в сортиран вид.

База. При първото достигане на ред 1 е вярно, че $i = 2$. Текущият подмасив $A[1 \dots i-1]$ се състои от един единствен елемент $A_0[1]$ и е тривиално сортиран. ✓

Поддръжка. Да допуснем, че твърдението е изпълнено при някое достигане на ред 1, което не е последното. Нека този момент от изпълнението е $\hat{t}$. Нека $\tilde{A}$ е името на A в момента $\hat{t}$. Индуктивното допускане, изразено чрез $\tilde{A}$, е:

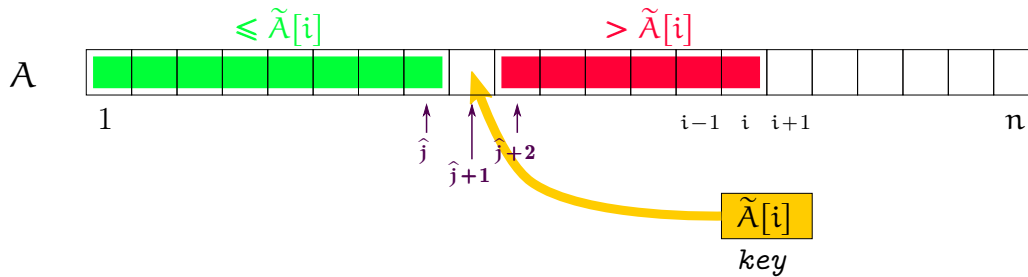
$\tilde{A}[1 \dots i-1]$ се състои от същите елементи като $A_0[1 \dots i-1]$, но в сортиран вид.

Изпълнява се ред 2 и key получава стойност $\tilde{A}[i]$. Изпълнява се ред 3 и j получава стойност $i-1$. После се изпълнява **while**-цикълът (редове 4-6). Нека $\hat{t}$ е моментът, в който **while** цикълът терминира; тоест, изпълнението е на ред 4 за последен път. Нека $\hat{A}$ и $\hat{j}$ означават съответно масива A и променливата j в момента $\hat{t}$. Може да приложим Лема 23, защото нейните три предпоставки са изпълнени, а именно $\tilde{A}[1 \dots i-1]$ е сортиран, key съдържа $\tilde{A}[i]$, а j в началото на изпълненията на **while**-а съдържа $i-1$. Лема 23 ни казва, че

1. $\hat{A}[1 \dots \hat{j}] = \tilde{A}[1 \dots \hat{j}]$ и неговите елементи са точно тези елементи на $\tilde{A}[1 \dots i-1]$, които са по-малки или равни на key ;
2. $\hat{A}[\hat{j}+2 \dots i] = \tilde{A}[\hat{j}+1 \dots i-1]$ и неговите елементи са точно тези елементи на $\tilde{A}[1 \dots i-1]$, които са по-големи от key .

Но $\tilde{A}[1 \dots i-1]$ е сортиран, така че и $\hat{A}[1 \dots \hat{j}]$, и $\hat{A}[\hat{j}+1 \dots i-1]$ са сортирани. Директно следва, че и $\hat{A}[1 \dots \hat{j}]$, и $\hat{A}[\hat{j}+2 \dots i]$ са сортирани.

Щом key съдържа $\tilde{A}[i]$, очевидно е, че след присвояването $A[j+1] \leftarrow key$ на ред 7, текущият масив $A[1 \dots i]$ се състои от елементите на $\tilde{A}[1 \dots i]$, в сортиран вид. Образно:



Но по допускане, $\tilde{A}[1..i-1]$ се състои от елементите на $A_0[1..i-1]$, в сортиран вид, а очевидно $\tilde{A}[i] = A_0[i]$. Заклучаваме, че текущият масив $A[1..i]$ се състои от елементите на $A_0[1..i]$, в сортиран вид.

При следващото достигане на ред 1, i се инкрементира. Спрямо новото i е вярно, че текущият масив $A[1..i]$ се състои от елементите на $A_0[1..i]$, в сортиран вид. ✓

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на $n+1$. Заместваме i с $n+1$ в инварианта и получаваме “текущият подмасив $A[1..(n+1)-1]$ се състои от същите елементи като $A_0[1..(n+1)-1]$, но в сортиран вид”. Кое е същото като “текущият подмасив $A[1..n]$ се състои от същите елементи като $A_0[1..n]$, но в сортиран вид”. □

Сложност по време на INSERTION SORT. Вече показахме в (2.23) на стр. 105, че сложността по време в най-лошия случай на INSERTION SORT е $\Theta(n^2)$.

Сложност по памет на INSERTION SORT. INSERTION SORT очевидно ползва $\Theta(1)$ допълнителна памет.

Стабилност на INSERTION SORT. INSERTION SORT е стабилен сортиращ алгоритъм. Прецизното доказателство остава за упражнение. Тук само ще споменем, че алгоритъмът е стабилен заради използването на строго неравенство $>$ на ред 4; **while** цикълът престава да се изпълнява за първата (максималната) стойност на j , за която $A[j]$ става равен на *key* и заради това всички двойки равни елементи остават в същия относителен ред, в какъвто са били в началото. Ако наместо строго равенство ползвахме $\geq$ на това място, алгоритъмът нямаше да е стабилен.

Допълнение 31: INSERTION SORT чрез суопове

INSERTION SORT може да се реализира и така: за i от 2 до n , i -ият елемент се разменя с $(i-1)$ -вия, докато има нужда. Ето псевдокод.

INSERTION SORT, SWAP($A[1..n]$: array of integers)

```

1  for  $i \leftarrow 2$  to  $n$ 
2       $j \leftarrow i$ 
3      while  $j > 1$  and  $A[j-1] > A[j]$  do
4          swap( $A[j-1], A[j]$ )
5           $j--$ 
```

Този вариант е по-бавен от практическа гледна точка. Като асимптотика, INSERTION SORT, SWAP има сложност по време $\Theta(n^2)$ в най-лошия случай, също като Алгоритъм 5, но софтуерната му реализация би работила отчетливо по-бавно от софтуерната

реализация на Алгоритъм 5. Причината е, че суоповите са бавни – всеки суоп се състои от три местения на елементи, така че поредица от k суопове иска $3(k - 1)$ местения на елементи, докато вариантът, изложен на стр. 294, прави само $k + 1$ местения: слага $A[i]$ в *key*, премества k елемента в A “нагоре по веригата” и слага на съдържанието на *key* в “освободеното място” на дъното на веригата. $3(k - 1)$ е значително повече от $k + 1$ и при големи размери на входа, разлика ще се усети, ако говорим за софтуерната реализация.

От друга страна, вариантът със суоповите се верифицира по-лесно, както предстои да се убедим. Най-малкото, ако сортиращ алгоритъм променя масива само със суопове, не се налага да обосноваваме, че масивът остава същият след произволна поредица от суопове – това е очевидно. Ерго, ако сортиращ алгоритъм мени масива само със суопове, напълно достатъчно е да докажем, че изходът е сортиран.

Лема 24: Вътрешният цикъл на INSERTION SORT, SWAP

По отношение на едно изпълнение на външния цикъл (редове 1–5) на INSERTION SORT, SWAP, ако наречем A' масива при първото достигане на ред 3 и допуснем, че в този момент $A'[1 \dots i - 1]$ е сортиран и $j = i$, то ефектът от изпълнението на вътрешния цикъл (редове 3–5) е следният:

- текущият $A[1 \dots j - 1]$ е същият като $A'[1 \dots j - 1]$ и се състои точно от елементите на $A'[1 \dots i - 1]$, по-малки или равни на $A'[i]$;
- текущият елемент $A[j]$ е същият като $A'[i]$;
- текущият $A[j + 1 \dots i]$ е същият като $A'[j \dots i - 1]$ и се състои точно от елементите на $A'[1 \dots i - 1]$, по-големи от $A'[i]$.

Доказателство: Следното твърдение е инвариант за вътрешния цикъл.

Инвариант 9: Вътрешният цикъл на INSERTION SORT, SWAP

Всеки път, когато изпълнението е на ред 3:

- текущият $A[1 \dots j - 1]$ е същият като $A'[1 \dots j - 1]$;
- текущият елемент $A[j]$ е същият като $A'[i]$;
- текущият $A[j + 1 \dots i]$ е същият като $A'[j \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$.

База: При първото достигане, $j = i$ по допускане. Първата част от инварианта е “текущият $A[1 \dots i - 1]$ е същият като $A'[1 \dots i - 1]$ ”, което е тривиално вярно. Втората част от инварианта е “текущият елемент $A[i]$ е същият като $A'[i]$ ”, което е тривиално вярно. Масивът $A[j + 1 \dots i]$ е празен при $i = j$, също така и $A'[j \dots i - 1]$ е празен при $i = j$, откъдето следва, че третата част от инварианта е истина, в празния смисъл. ✓

Поддръжка: Да допуснем, че инвариантът е изпълнен за някое достигане на ред 3, което не е последното. Щом не е последното, то е вярно, че $j > 1$ и $A[j - 1] > A[j]$. Да видим как се запазва първата част на инварианта. След swap-а на ред 4 вече не е вярно, че текущият $A[1 \dots j - 1]$ е същият като $A'[1 \dots j - 1]$. Това, което е вярно, е

че текущият $A[1 \dots j - 2]$ е същият като $A'[1 \dots j - 2]$. Но след декрементацията на j на ред 5, изразено в новата стойност на j , отново е вярно, че $A[1 \dots j - 1]$ е същият като $A'[1 \dots j - 1]$.

Да видим как се запазва втората част от инварианта. След swap-а ред 4 вече не е вярно, че $A[j] = A'[i]$. Това, което е вярно, е че $A[j - 1] = A'[i]$. Но след декрементацията на j на ред 5, изразено в новата стойност на j , отново е вярно, че $A[j] = A'[i]$.

Да видим как се запазва третата част от инварианта. След swap-а ред 4 продължава да е вярно, че текущият $A[j + 1 \dots i]$ е същият като $A'[j \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$. Нещо повече, сега вече е вярно, че $A[j \dots i]$ е същият като $A'[j - 1 \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$. Но след декрементацията на j на ред 5, изразено в новата стойност на j , отново е вярно, че текущият $A[j + 1 \dots i]$ е същият като $A'[j \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$. С това доказателството на поддръжката приключи.

Терминация: При последното достигане на ред 4 е вярно, че $j \leq 1$ или $A[j - 1] \leq A[j]$. Първо да допуснем, че $j \leq 1$. Тъй като j се декрементира със стъпка 1, виждаме, че всъщност $j = 1$. Замествайки в инварианта j с 1, получаваме

- текущият $A[1 \dots 0]$ е същият като $A'[1 \dots 0]$;
- текущият елемент $A[1]$ е същият като $A'[i]$;
- текущият $A[2 \dots i]$ е същият като $A'[1 \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$.

Масивът $A[1 \dots 0]$ е празен, така че в празния смисъл е вярно, че се състои точно от елементите на $A'[1 \dots i - 1]$, по-малки или равни на $A'[i]$. Ерго, първото твърдение на лемата е вярно.

Второто твърдение тук е точно като второто твърдение на лемата, така че и второто твърдение на лемата е вярно.

Да разгледаме третото твърдение. Щом $A[2 \dots i] = A'[1 \dots i - 1]$ и всеки елемент на $A[2 \dots i]$ е по-голям от $A'[i]$ и $A[1] = A'[i]$, то всички елементи на $A'[1 \dots i - 1]$, по-големи от $A'[i]$, се намират именно в $A[2 \dots i]$. Тогава третото твърдение тук е същото като третото твърдение на лемата.

Сега да допуснем, че $A[j - 1] \leq A[j]$. От първото твърдение на инварианта знаем, че текущият $A[1 \dots j - 1]$ е същият като $A'[1 \dots j - 1]$. От второто знаем, че $A[j] = A'[i]$. Щом $A[j - 1] \leq A'[i]$ и освен това $A'[1 \dots i - 1]$ е сортиран, веднага следва, че текущият $A[1 \dots j - 1]$ се състои точно от елементите на $A'[1 \dots i - 1]$, по-малки или равни на $A'[i]$. Първото твърдение на лемата е вярно.

Второто твърдение на лемата е същото като второто твърдение на инварианта.

Това, че текущият $A[j + 1 \dots i]$ е същият като $A'[j \dots i - 1]$ и всеки негов елемент е по-голям от $A'[i]$, знаем от инварианта. Имайки предвид, че елементите на $A[1 \dots j - 1]$ са по-малки или равни на $A'[i]$ и $A[j] = A'[i]$, получаваме и това, че текущият $A[j + 1 \dots i]$ съдържа точно тези елементи от $A'[1 \dots i - 1]$, които са по-големи от $A'[i]$. И третото твърдение на лемата “излезе”. $\square$

Теорема 57: Коректността на INSERTION SORT, SWAP

Алгоритъмът INSERTION SORT, SWAP е коректен сортиращ алгоритъм.

Доказателство: Следното твърдение е инвариант за външния цикъл.

Инвариант 10: Външният цикъл на INSERTION SORT, SWAP

При всяко достигане на ред 1, текущият $A[1..i-1]$ се състои точно от елементите на входния $A[1..i-1]$, но в сортиран вид.

База: При първото достигане, $i = 2$. Твърдението е тривиално вярно.

Поддръжка: Да допуснем, че изпълнението е на ред 1 и предстои поне още едно изпълнение на тялото на цикъла. На ред 2, j става i . Следва изпълнението на вътрешния цикъл. След приключването му, съгласно Лема 24,

- текущият $A[1..j-1]$ е същият като $A'[1..j-1]$ и се състои точно от елементите на $A'[1..i-1]$, по-малки или равни на $A'[i]$;
- текущият елемент $A[j]$ е същият като $A'[i]$;
- текущият $A[j+1..i]$ е същият като $A'[j..i-1]$ и се състои точно от елементите на $A'[1..i-1]$, по-големи от $A'[i]$.

Но от това следва, че текущият $A[1..i]$ се състои от елементите на входния $A[1..i]$, само че в сортиран вид. Следва инкрементация на i с единица. Спрямо новата стойност на i е вярно, че текущият $A[1..i-1]$ се състои от елементите на входния $A[1..i-1]$, само че в сортиран вид. Кое е точно инвариантът.

Терминация: При термиране на алгоритъма, $i = n + 1$. Замествайки i с $n + 1$ в инварианта, получаваме “текущият $A[1..n]$ се състои точно от елементите на входния $A[1..n]$, но в сортиран вид.” $\square$

4.4.2 SELECTION SORT

Да си припомним псевдокода на SELECTION SORT.

SELECTION SORT($A[1..n]$: array of integers)

```

1  for  $i \leftarrow 1$  to  $n - 1$ 
2      for  $j \leftarrow i + 1$  to  $n$ 
3          if  $A[j] < A[i]$ 
4              swap( $A[i], A[j]$ )
```

Коректността на SELECTION SORT. Теорема 58 обосновава коректността на SELECTION SORT. Това, че SELECTION SORT само размества елементите на входа и в края на алгоритъма елементите в масива са точно тези, които са били в началото, е очевидно: единствените промени в A стават на ред 4 чрез размени (swaps). Това отличава тази реализация на SELECTION SORT от реализацията на INSERTION SORT, която вече разглеждахме. При SELECTION SORT не беше абсолютно очевидно, че елементите в края са точно тези, които са били в началото, защото неговата реализация ползва не размени, а присвоявания. Поради това при анализа на INSERTION SORT се наложи да обосновем защо нито един от оригиналните елементи не се “губи”, бивайки презаписан с нещо друго. При анализа на SELECTION SORT нямаме такива притеснения.

Лема 25

По отношение на едно изпълнение на външния **for** цикъл (редове 1–4) на **SELECTION SORT**, изпълнението на вътрешния **for** цикъл (редове 2–4) има следния ефект: $A[i]$ е най-малък елемент в $A[i \dots n]$.

Доказателство:

По отношение на едно изпълнение на външния **for** цикъл, следното твърдение е инвариант за вътрешния **for** цикъл:

Инвариант 11: Вътрешният цикъл на SELECTION SORT

Всеки път, когато изпълнението достигне ред 2, текущият $A[i]$ е минимален елемент в $A[i \dots j - 1]$.

База. Първият път, когато изпълнението на вътрешния **for** цикъл е на ред 2 е вярно, че $j = i + 1$. Тогава $A[i]$ е тривиално най-малък елемент в $A[i \dots (i + 1) - 1]$.

Поддръжка. Да допуснем, че твърдението е вярно в някой момент, когато изпълнението е на ред 2 и предстои вътрешният **for** цикъл да бъде изпълнен още веднъж. Следните два случая са изчерпателни.

Случай 1. $A[j] < A[i]$. Условието на ред 3 е **TRUE** и размяната на ред 4 се случва. Съгласно допускането, $A[i]$ е минимален елемент в $A[i \dots j - 1]$. Тъй като $A[j] < A[i]$, съгласно транзитивността на релацията $<$, $A[j]$ е минималният елемент в подмасива $A[i \dots j]$ преди размяната. Тогава $A[i]$ е минималният елемент в $A[i \dots j]$ след размяната. След размяната j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минималният елемент в $A[i \dots j - 1]$.

Случай 2. $A[j] \nless A[i]$. Тогава условието на ред 3 е **FALSE** и размяната на ред 4 не се случва. Съгласно предположението, $A[i]$ е минимален елемент в $A[i \dots j - 1]$. Тъй като $A[i] \leq A[j]$, очевидно $A[i]$ е минимален елемент в $A[i \dots j]$. После j нараства с единица и изпълнението отива на ред 2. По отношение на новата стойност на j е вярно, че $A[i]$ е минимален елемент в $A[i \dots j - 1]$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 2 за последен път. Очевидно j е равно на $n + 1$. Заместваме $n + 1$ на мястото на j в инварианта и получаваме “текущият $A[i]$ е минимален елемент в $A[i \dots (n + 1) - 1]$ ”. $\square$

Теорема 58: Коректността на SELECTION SORT

SELECTION SORT е сортиращ алгоритъм.

Доказателство:

Да наречем оригиналния масив $A'[1 \dots n]$. Следното е инвариант за външния **for** цикъл (редове 1–4):

Инвариант 12: Външният цикъл на SELECTION SORT

Всеки път, когато изпълнението на **SELECTION SORT** е на ред 1, текущият подмасив $A[1 \dots i - 1]$ се състои от $i - 1$ на брой най-малки елементи на $A'[1 \dots n]$ в сортиран вид.

База. При първото достигане на ред 1 е вярно, че $i = 1$. Текущият подмасив $A[1 \dots i - 1]$ е празен и се състои от нула на брой най-малки елементи от $A'[1 \dots n]$ в сортиран вид.

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 1, което не е последното. Нека наричаме масива A в този момент с името A'' . Съгласно Лема 25, ефектът на вътрешния **for** цикъл е, че поставя на i -тата позиция най-малка стойност от $A''[i \dots n]$. От друга страна, съгласно индуктивното допускане, $A''[1 \dots i - 1]$ се състои от $i - 1$ на брой най-малки елементи на $A'[1 \dots n]$ в сортиран вид. Заключаваме, че в края на изпълнението на външния **for** цикъл, текущият $A[1 \dots i]$ се състои от i на брой най-малки елемента от $A'[1 \dots n]$ в сортиран вид. После i нараства с единица и изпълнението отива на ред 1. По отношение на новата стойност на i е вярно, че текущият $A[1 \dots i - 1]$ съдържа $i - 1$ на брой най-малки елемента от $A'[1 \dots n]$ в сортиран вид.

Терминация. Да разгледаме момента, в който изпълнението е на ред 1 за последен път. Очевидно i е равно на n . Заместваме n на мястото на i в инварианта и получаваме “текущият подмасив $A[1 \dots n - 1]$ се състои от $n - 1$ на брой най-малки елемента на $A'[1 \dots n]$ в сортиран вид”. Но тогава текущият $A[n]$ е максимален елемент от $A'[1 \dots n]$. С това доказателството за коректност на SELECTION SORT приключва. $\square$

Сложност по време на SELECTION SORT. Вече показахме в (2.26) на стр. 106, че сложността по време в най-лошия случай на SELECTION SORT е $\Theta(n^2)$.

Сложност по памет на SELECTION SORT. SELECTION SORT очевидно ползва $\Theta(1)$ допълнителна памет.

Стабилност на SELECTION SORT. В този си вид SELECTION SORT не е стабилен. За да се убедим в това, да разгледаме малък подходящ пример. Нека входът е

2, 3, 4, 2, 1

За да различаваме двойките, да ги означим така:

2', 3, 4, 2'', 1

Очевидно след първото изпълнение на външния **for** цикъл единицата ще отиде на най-лява позиция, бивайки разменена с 2':

1, 3, 4, 2'', 2'

и крайният резултат ще е

1, 2'', 2', 3, 4

Лекция 5

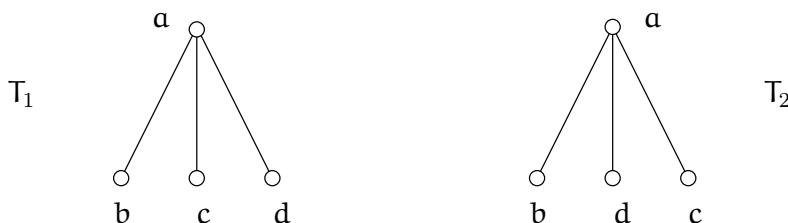
Двоична пирамида. HEAPSORT. Приоритетна опашка.

Резюме: Въвеждаме понятието попълнено двоично дърво и двоична пирамида. Показваме два начина да се строи двоична пирамида: наивен начин във време $\Theta(n \lg n)$ и бързото построяване във време $\Theta(n)$ на Floyd. Демонстрираме сортиращия алгоритъм HEAPSORT и го анализираме. Въвеждаме понятието приоритетна опашка като абстрактен тип данни и разискваме имплементация на приоритетна опашка чрез двоична пирамида.

5.1 Двоични дървета и пирамиди

5.1.1 Попълнено двоично дърво

Тъй като теорията на графите е сравнително нова област, няма универсално приета пълна единна терминология и всяко сериозно изложение, използващо графови понятия, трябва да започва с изчерпателни определения, за да е ясно точно какво се има предвид. Терминологията на английски ще изложим според американския Национален Институт за Стандарти и Технологии [NIST](#). Техните определения за дървета са изложени на [тази страница](#). Не допускаме дървета без върхове. Разглеждаме само коренови дървета, така че когато кажем “дърво” (tree), имаме предвид кореново дърво (rooted tree). В контекста на структури от данни по правило дърветата са *наредени*[†] (на английски, *ordered trees*), което означава, че за всеки вътрешен връх, неговите деца са наредени с линейна наредба. Следните две дървета T_1 и T_2 са различни като наредени дървета, но са едно и също дърво, ако ги разглеждаме като ненаредени дървета:



Ако T е кореново дърво с корен r , за всеки връх u в него с $T[u]$ бележим поддървото, вкоренено в u . Формално,

- ако $u = r$, то $T[u]$ е самото T ,

[†]За разлика от “чистата” теория на графите, в която по правило кореновите дървета не са наредени.

- иначе, $T[u]$ е това от двете дървета на $T - (u, \text{parent}(u))$, което съдържа u ,

като $T[u]$ е кореново дърво с корен u .

Определение 51: Височина, дълбочина, нива

Височината на връх u в кореново дърво е максималното разстояние между u и кое да е листо в $T[u]$. Височината на дървото е височината на корена. Дълбочината на връх u в кореново дърво е разстоянието между него и корена. Ниво в дърво е множеството от всички върхове, които са на една и съща дълбочина. Номер на ниво в дърво е дълбочината на кой да е връх от това ниво. Съседни нива са нива, чиито дълбочини се различават с единица.

Двоично дърво, на английски *binary tree*, в теорията на графите е кореново дърво, наредено или не, в което всеки връх има не повече от две деца. Пълно двоично дърво, на английски *full binary tree*, е двоично дърво, в което всеки вътрешен връх има точно две деца. За целите на тези лекции обаче, под “двоично дърво” ще разбираме дърво, което задължително е наредено, но и нещо повече: различаваме ляво от дясно дете. С други думи, ако има връх с точно едно дете, това дете е или ляво, или дясно, и ние различаваме тези възможности. Следните две дървета D_1 и D_2 са различни като двоични дървета, защото детето в D_1 е ляво, а в D_2 е дясно, но са едно и също дърво, ако ги разглеждаме като подредени дървета, защото детето е само едно:



Определение 52: Попълнено двоично дърво, не-индуктивно определение

Попълнено двоично дърво е наредено двоично дърво, в което:

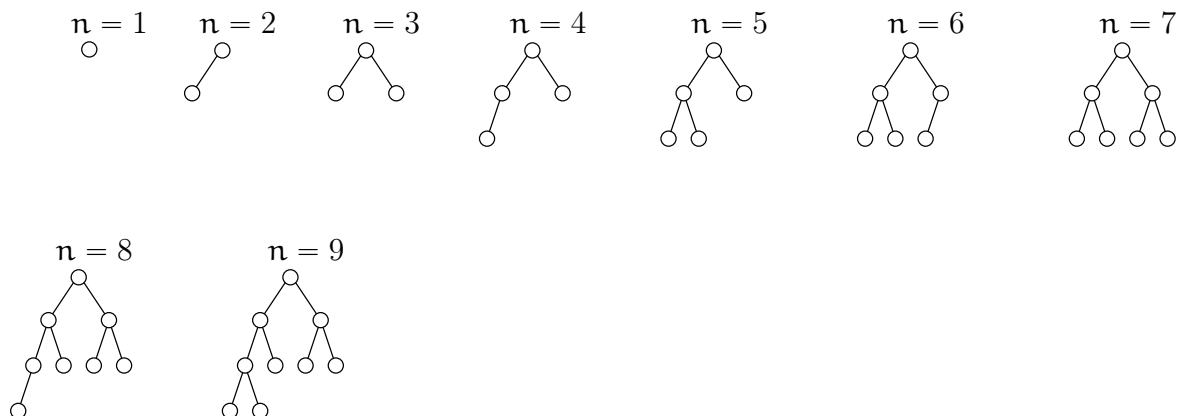
- всички нива, с изключение може би на последното, имат максималния възможен брой^a върхове^b.
- листата на последното ниво са максимално вляво. □

^a Ниво номер k може да има най-много 2^k върха в себе си.

^b Следствие от това е, че всички листа са или в едно ниво, или в две нива, които са съседни.

На английски терминът е *complete tree*.

За всяко $n \in \mathbb{N}^+$ има едно единствено попълнено дърво с n върха:



Определение 53: Съвършено двоично дърво

Не-индуктивното определение е следното. *Съвършено двоично дърво* (на английски, *perfect binary tree*) е попълнено двоично дърво, в което всички листа са на едно и също ниво.

Индуктивното определение е следното.

- Един единствен връх u без ребра е съвършено двоично дърво с корен u , множество от листа $\{u\}$, множество от вътрешни върхове $\emptyset$ и височина 0.
- Нека T' и T'' са съвършени коренови дървета с корени съответно u' и u'' , множества от листа съответно W' и W'' , множества от вътрешни върхове съответно Z' и Z'' и една и съща височина $h \geq 0$. Нека r е връх извън T' и T'' . Тогава

$$(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\})$$

е съвършено кореново дърво с корен r , множество от листа $W' \cup W''$, множество от вътрешни върхове $Z' \cup Z'' \cup \{r\}$ и височина $h + 1$.

Двете определения са еквивалентни.

Определение 54: Попълнено двоично дърво, индуктивно определение**База:**

1. $(\{u\}, \emptyset)$ е попълнено двоично дърво с корен u , множество от листа $\{u\}$, множество от вътрешни върхове $\emptyset$ и височина 0. u няма деца.
2. $(\{u, v\}, \{(u, v)\})$ е попълнено двоично дърво с корен u , множество от листа $\{v\}$, множество от вътрешни върхове $\{u\}$ и височина 1. v е лявото дете на u , а дясното дете липсва.

Индуктивна стъпка:

3. Нека T' е съвършено кореново дърво с корен u' , множество от листа W' , и височина $h - 1 \geq 0$. Нека T'' е попълнено кореново дърво с корен u'' , множество от листа W'' , и височина $h - 1$. Нека r е връх извън T' и T'' . Тогава

$$(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}),$$

където u' е лявото дете на r , а u'' е дясното дете на r , е попълнено кореново дърво с корен r , множество от листа $W' \cup W''$ и височина h .

4. Нека T' е попълнено кореново дърво с корен u' , множество от листа W' , и височина $h - 1 \geq 1$. Нека T'' е съвършено кореново дърво с корен u'' , множество от листа W'' , и височина $h - 2$. Нека r е връх извън T' и T'' . Тогава

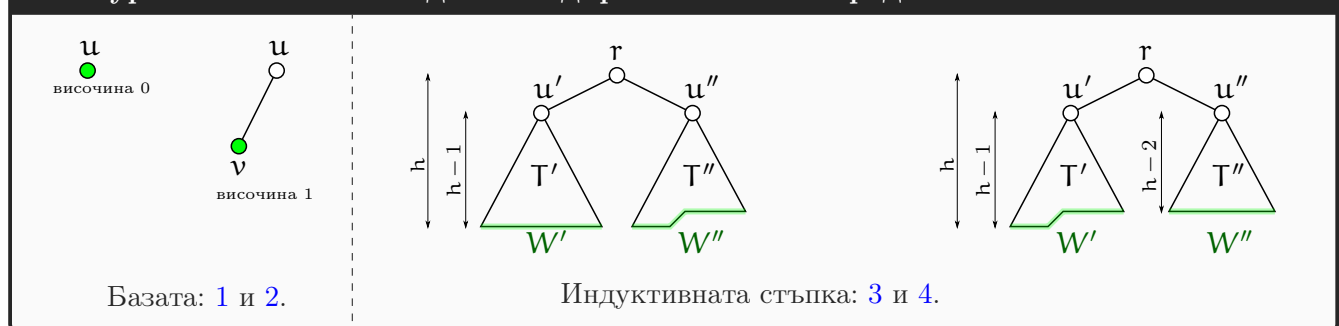
$$(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}),$$

където u' е лявото дете на r , а u'' е дясното дете на r , е попълнено кореново дърво с корен r , множество от листа $W' \cup W''$ и височина h .

Определение 52 е еквивалентно на Определение 54.

Фигура 5.1 илюстрира Определение 54. Листата са в зелено.

Фигура 5.1 : Попълнено двоично дърво съгласно Определение 54.



Доказателството за еквивалентност на Определение 52 и Определение 54 остават за читателя.

Както обикновено, пишейки “ $\lg n$ ”, имаме предвид “ $\log_2 n$ ”.

Теорема 59: Височината на попълнено двоично дърво

За всяко попълнено двоично дърво с n върха, за височината h е изпълнено $h = \lceil \lg n \rceil$.

Доказателство: Чрез структурна индукция съгласно Определение 54. В базовите случаи 1 и 2 твърдението е вярно: имаме $0 = \lfloor \lg 1 \rfloor$ и $1 = \lfloor \lg 2 \rfloor$. ✓

В случай 3, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h - 1 = \lfloor \lg p \rfloor$ и $h - 1 = \lfloor \lg q \rfloor$. Но $\lfloor x \rfloor$ е най-голямото **цяло число, ненадхвърлящо x** , за всяко x . Вярно е, че:

$$h - 1 \leq \lg p < h$$

$$h - 1 \leq \lg q < h$$

Тогава,

$$2^{h-1} \leq p < 2^h$$

$$2^{h-1} \leq q < 2^h$$

Но p , q и h са цели числа и 2^h също е цяло число. Имаме право да кажем, че

$$2^{h-1} \leq p < 2^h - \frac{1}{2}$$

$$2^{h-1} \leq q < 2^h - \frac{1}{2}$$

Тогава

$$2 \times 2^{h-1} \leq p + q < 2 \times 2^h - 1 \quad \leftrightarrow$$

$$2^h \leq p + q < 2^{h+1} - 1 \quad \leftrightarrow$$

$$2^h < p + q + 1 < 2^{h+1}$$

Но $n = p + q + 1$. Тогава,

$$2^h < n < 2^{h+1} \quad \leftrightarrow$$

$$h < \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓

В случай 4, нека T' има p върха и T'' има q върха. Съгласно индуктивното предположение, $h - 1 = \lfloor \lg p \rfloor$ и $h - 2 = \lfloor \lg q \rfloor$. Тогава

$$h - 1 \leq \lg p < h \tag{5.1}$$

$$h - 2 \leq \lg q < h - 1 \tag{5.2}$$

Ще докажем, че $\lg(q + 1) < h$. Наистина, q е число от вида $2^k - 1$ за някое $k \geq 1$, защото T'' е съвършено дърво, така че $q + 1 = 2^k$, така че $\lg(q + 1) = k$ и от (5.2) имаме

$$\lg q < h - 1 \leftrightarrow (\lg q) + 1 < h \leftrightarrow \lg 2q < h$$

Но $q + 1 \leq 2q$ при $q \geq 1$, следователно

$$\lg(q + 1) < h \tag{5.3}$$

Ще докажем, че $h - 1 \leq \lg(q + 1)$.

- Ако $q = 1$, $h - 2 = 0$, така че $h - 1 = 1$; от друга страна, $\lg(1 + 1) = 1$. ✓

- Нека $q \geq 2$. От (5.2) знаем, че $h - 2 \leq \lg q$. Но $h - 2$ е цяло число. От друга страна, $\lg q$ е дробно число, а $\lg(q + 1)$ е цяло число, тъй като $q + 1 = 2^k$. Очевидно $\lg q < \lg(q + 1)$. Тогава $h - 1 \leq \lg(q + 1)$, понеже $h - 1$ е следващото цяло число след $h - 2$. ✓

И така,

$$h - 1 \leq \lg(q + 1) < h \quad (5.4)$$

От (5.1) и (5.4) имаме

$$h - 1 \leq \lg p < h$$

$$h - 1 \leq \lg(q + 1) < h$$

Тогава,

$$2^{h-1} \leq p < 2^h$$

$$2^{h-1} \leq q + 1 < 2^h$$

Оттук имаме

$$2^h \leq p + q + 1 < 2^{h+1} \quad \leftrightarrow$$

$$2^h \leq n < 2^{h+1}$$

Тогава,

$$h \leq \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓

□

Теорема 60: Броят на листата на попълнено двоично дърво

Всяко попълнено двоично дърво T има точно $\lfloor \frac{n}{2} \rfloor$ листа.

Доказателство: Чрез структурна индукция съгласно Определение 54. За двата базови случая твърдението е вярно: в случай 1, $n = 1$ и наистина има $\lfloor \frac{1}{2} \rfloor = 1$ листа, а в случай 2, $n = 2$ и наистина има $\lfloor \frac{2}{2} \rfloor = 1$ листа. ✓

Нека T е конструирано чрез случай 3. Нека T' има p върха и T'' има q върха. Очевидно $n = p + q + 1$. И T' , и T'' са попълнени дървета и индуктивното предположение за тях казва, че те имат съответно $\lfloor \frac{p}{2} \rfloor$ и $\lfloor \frac{q}{2} \rfloor$ листа. Наготово използваме факта, че T' има нечетен брой върхове, понеже е свършено двоично дърво. Нека $p = 2k + 1$. Съгласно Определение 54, множеството от листата на T се разбива на множествата от листата на T' и T'' , така че броят на листата на T е:

$$\begin{aligned} \left\lfloor \frac{p}{2} \right\rfloor + \left\lfloor \frac{q}{2} \right\rfloor &= \left\lfloor \frac{2k+1}{2} \right\rfloor + \left\lfloor \frac{q}{2} \right\rfloor = k + 1 + \left\lfloor \frac{q}{2} \right\rfloor = \left\lfloor k + 1 + \frac{q}{2} \right\rfloor = \left\lfloor \frac{2k+2+q}{2} \right\rfloor = \\ &= \left\lfloor \frac{2k+1+q+1}{2} \right\rfloor = \left\lfloor \frac{p+q+1}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor \quad \checkmark \end{aligned}$$

Нека T е конструирано чрез случай 4. Нека T' има p върха и T'' има q върха. Очевидно $n = p + q + 1$. И T' , и T'' са попълнени дървета и индуктивното предположение за тях казва, че те имат съответно $\lfloor \frac{p}{2} \rfloor$ и $\lfloor \frac{q}{2} \rfloor$ листа. Наготово използваме факта, че T'' има нечетен брой

върхове, понеже е съвършено двоично дърво. Нека $q = 2k + 1$. Съгласно Определение 54, множеството от листата на T се разбива на множествата от листата на T' и T'' , така че броят на листата на T е:

$$\begin{aligned} \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil &= \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{2k+1}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + k + 1 = \left\lceil \frac{p}{2} + k + 1 \right\rceil = \left\lceil \frac{p + 2k + 2}{2} \right\rceil = \\ &= \left\lceil \frac{p + 2k + 1 + 1}{2} \right\rceil = \left\lceil \frac{p + q + 1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad \checkmark \end{aligned} \quad \square$$

Следствие 21

Всяко попълнено двоично дърво има точно $\left\lfloor \frac{n}{2} \right\rfloor$ вътрешни върхове.

Доказателство: Следва от Теорема 60 и факта, че $\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor = n$ за всяко цяло n . $\square$

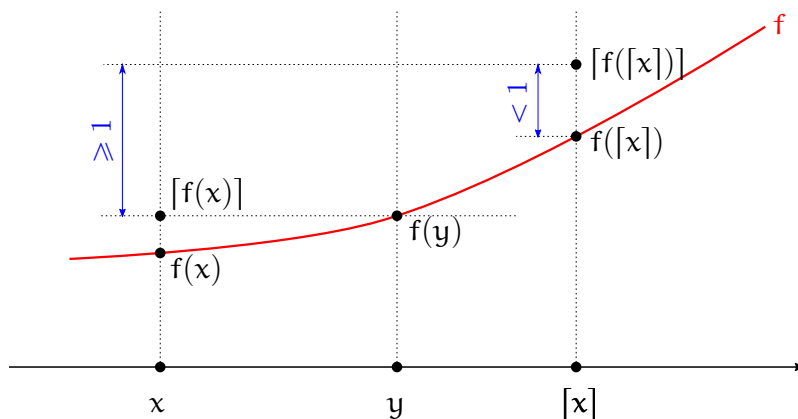
Следните равенства са от книгата Concrete Mathematics [61, стр. 71].

Теорема 61: $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$ и $\lfloor f(\lceil x \rceil) \rfloor = \lfloor f(x) \rfloor$

Нека $f(x)$ е произволна реална непрекъсната строго растяща функция със следното свойство: ако $f(x)$ е цяло, то x е цяло. Тогава $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$ и $\lfloor f(\lceil x \rceil) \rfloor = \lfloor f(x) \rfloor$.

Доказателство: БОО, ще докажем само $\lfloor f(\lceil x \rceil) \rfloor = \lfloor f(x) \rfloor$. Ако $x = \lfloor x \rfloor$, няма какво да се доказва. Да допуснем, че $x \neq \lfloor x \rfloor$. Тогава $x < \lfloor x \rfloor$. Но тогава $f(x) < f(\lfloor x \rfloor)$, защото f е строго растяща. Тогава $\lfloor f(x) \rfloor \leq \lfloor f(\lfloor x \rfloor) \rfloor$, понеже $\lfloor \cdot \rfloor$ е ненамаляваща.

Да допуснем, че $\lfloor f(x) \rfloor < \lfloor f(\lfloor x \rfloor) \rfloor$. Ключовото наблюдение е, че тогава съществува реално y , такова че $x < y < \lfloor x \rfloor$ и $f(y) = \lfloor f(x) \rfloor$, понеже $\lfloor f(x) \rfloor < f(\lfloor x \rfloor)$ и f е непрекъсната. А $\lfloor f(x) \rfloor < f(\lfloor x \rfloor)$, понеже $\lfloor f(\lfloor x \rfloor) \rfloor - \lfloor f(x) \rfloor \geq 1$ (това са цели числа), докато $\lfloor f(\lfloor x \rfloor) \rfloor - f(\lfloor x \rfloor) < 1$.



Но $\lfloor f(x) \rfloor$ е цяло число. Следователно, $f(y)$ е цяло число. Тогава, заради свойството на f , числото y е цяло. Но това е невъзможно – не съществува цяло число, което е строго между x и $\lfloor x \rfloor$. ⚡

От полученото противоречие следва, че $\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$. $\square$

Следствие 22

$$\forall x \in \mathbb{R}^+ \quad \forall b \in \mathbb{N}^+ : \left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \quad \text{и} \quad \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil$$

Доказателство: Прилагаме Теорема 61 с $f(x) = \frac{x}{b}$. □

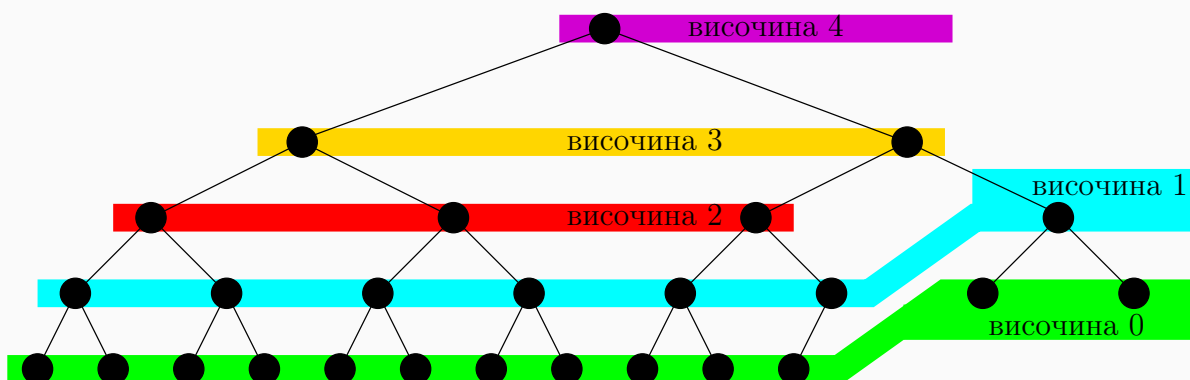
Следствие 23

$$\forall x \in \mathbb{R}^+ \forall a \in \mathbb{N}^+ \forall b \in \mathbb{N}^+ : \left\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \text{ и } \left\lceil \frac{\lceil \frac{x}{a} \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil$$

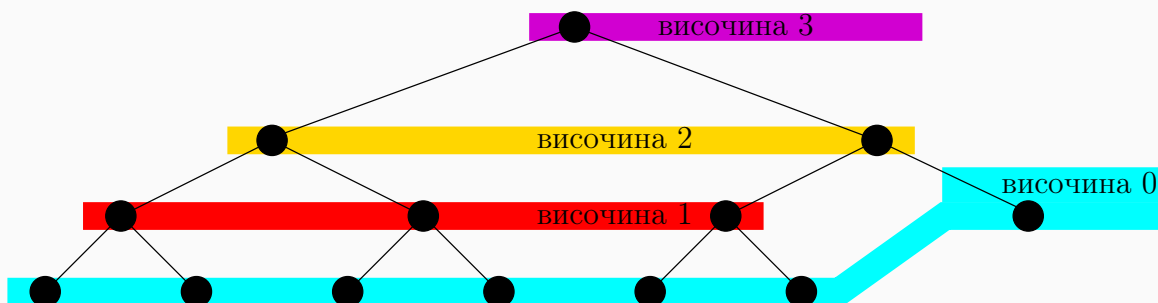
Доказателство: Прилагаме Следствие 22 с $\frac{x}{a}$ наместо x . □

В Лема 26 и Теорема 62 става дума за височини на върхове. Да си припомним, че *височината на връх u* е максималното разстояние между u и кое да е листо в поддървото $T[u]$. Пример за височините на върховете на попълнено двоично дърво има на Фигура 5.2. Отгоре е показано попълнено двоично дърво с височина 4, а под него е показано дървото, което се получава от T чрез изтриване на листата.

Фигура 5.2 : Височини на върхове в попълнено двоично дърво.



Височините на върховете в попълнено двоично дърво с височина 4.



Попълненото дърво, което се получава след изтриване на всички върхове с височина 0, тоест, листата. Височината на всеки връх намалява с едно.

Лема 26

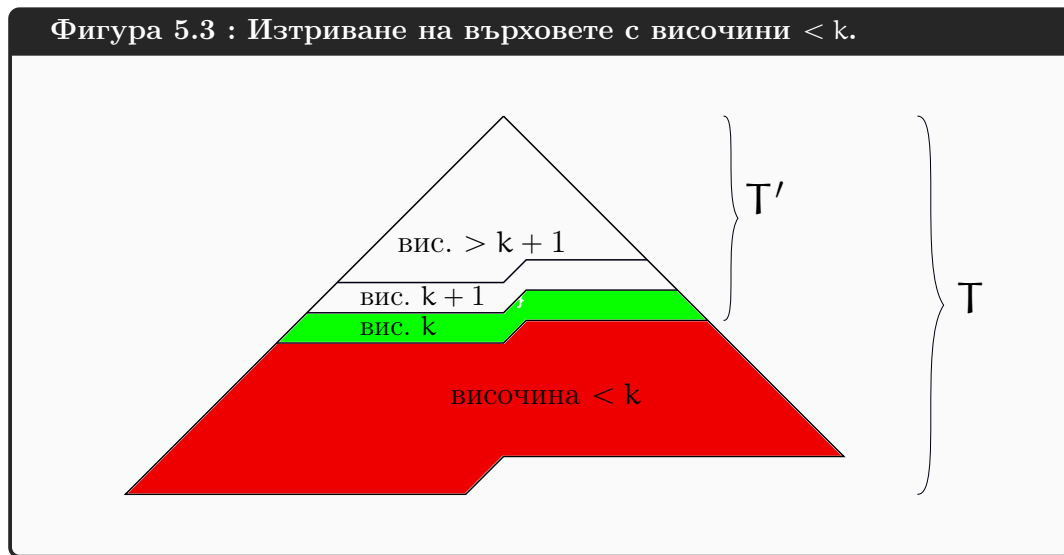
Във всяко попълнено двоично дърво T има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върхове с височина $\geq k$.

Доказателство: Доказателството е по индукция по k , по отношение на произволно, но фиксирано, попълнено двоично дърво T с височина h и n върхове. Забележете, че k взема стойности от **крайно** множество, а именно $\{0, 1, \dots, h\}$.

База. $k = 0$. Върховете с височина ≥ 0 са точно върховете на T , които са n на брой. В сила е $n = \left\lfloor \frac{n}{2^0} \right\rfloor$. ✓

Индуктивно предположение. Нека за някаква височина $k < h$ е вярно, че броят на върховете с височина $\geq k$ е $\left\lfloor \frac{n}{2^k} \right\rfloor$.

Индуктивна стъпка. Ще докажем, че броят на върховете с височина $\geq k+1$ е $\left\lfloor \frac{n}{2^{k+1}} \right\rfloor$. Да изтрием всички върхове с височина $< k$ от T . Нека полученото дърво е T' и нека n' е броят на неговите върхове. Всички върхове в T' са точно върховете с височина $\geq k$ в T .



Съгласно индуктивното предположение, $n' = \left\lfloor \frac{n}{2^k} \right\rfloor$. Съгласно Следствие 21, има точно $\left\lfloor \frac{n'}{2} \right\rfloor$ вътрешни върхове в T' . Ключовото наблюдение е, че вътрешните върхове на T' са точно върховете с височина $\geq k+1$ в T . Следователно, има $\left\lfloor \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rfloor$ върхове с височина $\geq k+1$ в T .

Съгласно Следствие 23, $\left\lfloor \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2 \cdot 2^k} \right\rfloor$, което е равно на $\left\lfloor \frac{n}{2^{k+1}} \right\rfloor$. □

Теорема 62: Броят на върховете с височина k в попълнено дърво

Във всяко попълнено двоично дърво T има точно $\left\lfloor \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rfloor$ върхове с височина k .

Доказателство: Съгласно Лема 26, има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geq k$. Върховете с височина k са точно листата в поддървото на T , индуцирано от върховете с височина $\geq k$.

Съгласно Теорема 60, тези листа са точно $\left\lfloor \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rfloor$. □

5.1.2 Адреси в попълнено двоично дърво

Следното определение задава процедура, която дава адреси-стрингове на върховете на попълнено двоично дърво, използвайки Определение 54. С “ε” означаваме празния стринг.

Определение 55: Адреси на върхове в попълнено двоично дърво

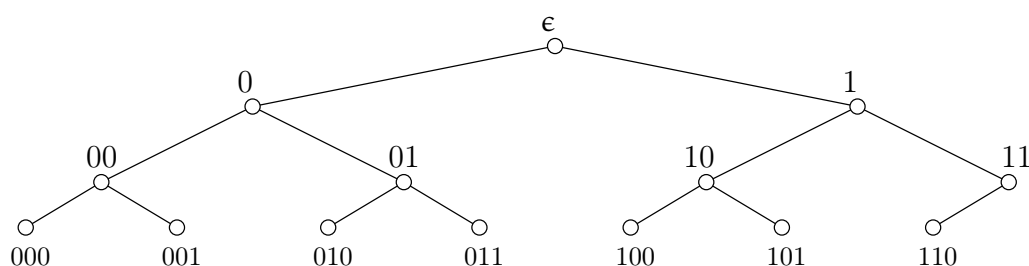
Всеки връх x в попълненото дърво има *адрес*, който бележим с $\text{addr}(x)$. Адресите се конструират от следната процедура от две фази.

фаза I: В базовия случай 1, $\text{addr}(u) = \epsilon$. В базовия случай 2, $\text{addr}(u) = \epsilon$ и $\text{addr}(v) = 0$. В случаи 3 и 4:

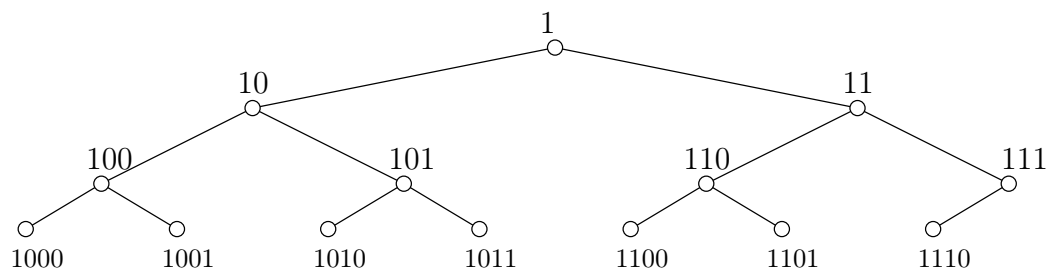
- $\text{addr}(u) = \epsilon$,
- за всеки връх x от T' , новият адрес на x се получава от стария с конкатенация на нула в левия край.
- за всеки връх x от T'' , новият адрес на x се получава от стария с конкатенация на единица в левия край.

фаза II: За всеки връх x от T , новият адрес се получава от стария чрез конкатенация на единица в началото. □

Ето пример за конструиране на адресите от **фаза I** за попълненото дърво с 14 върха:



Продължавайки със същия пример, ето окончателните адреси след **фаза II**:



Според Определение 55 адресите на върховете в попълнените двоични дървета са стрингове. В изложението надолу понякога ще злоупотребяваме с това и ще третираме адресите като числа—а именно, числата, записани с тези стрингове.

Лема 27

Нека T е произволно попълнено двоично дърво и u е произволен връх в него. Ако u не е листо:

- ако u има две деца v и w , където v е лявото дете, в сила е $\text{addr}(v) = 2 \times \text{addr}(u)$ и $\text{addr}(w) = 2 \times \text{addr}(u) + 1$.
- ако u има точно едно дете v , в сила е $\text{addr}(v) = 2 \times \text{addr}(u)$.

Ако u не е корен и v е родителят на u , то $\text{addr}(v) = \left\lfloor \frac{\text{addr}(u)}{2} \right\rfloor$.

Доказателство, част I: Първо ще докажем твърдението за адресите на децата. Ще докажем твърдението по дълбочината на u .

Базовият случай е: u има дълбочина нула, тоест u е коренът. Ако u има две деца v и w , като v е лявото, $\text{addr}(u) = 1$, $\text{addr}(v) = 10$ и $\text{addr}(w) = 11$, така че твърдението е вярно. Ако u има точно едно дете, очевидно става дума за дърво, построено чрез базова конструкция 2 на Определение 54, и съгласно Определение 55, $\text{addr}(u) = 1$ и $\text{addr}(v) = 10$, следователно твърдението пак е вярно.

Нека u има дълбочина d и u има поне едно дете. Ако u има точно две деца v и w , като v е лявото, $\text{addr}(v) = \text{addr}(u)0$ и $\text{addr}(w) = \text{addr}(u)1$, следователно твърдението е вярно, когато разглеждаме адресите като числа, записани в двоична позиционна бройна система.

Доказателство, част II: Ще докажем твърдението за адреса на родителя на всеки връх, който не е коренът. Но това твърдение е очевидно, имайки предвид **част I**. Адресът на u , изразен чрез адресите на децата си, е

$$\text{addr}(u) = \frac{\text{addr}(v)}{2} \quad \text{и} \quad \text{addr}(u) = \frac{\text{addr}(w) - 1}{2}$$

като, ако u има само едно дете, очевидно само първият израз е в сила. Тъй като $\frac{\text{addr}(v)}{2}$ е задължително четно, а $\frac{\text{addr}(w)}{2}$, ако съществува, е задължително нечетно, исканото твърдение следва веднага. $\square$

За всяко $n \in \mathbb{N}^+$, $\text{bin}(n)$ означава стринга-представяне на n в двоична позиционна бройна система (с водеща единица). Нека $n, k \in \mathbb{N}^+$ и $m = \lfloor \log_2 n \rfloor + 1$. Тогава $\text{bin}_k(n)$ се дефинира така:

$$\text{bin}_k(n) = \begin{cases} 0^{k-m} \text{bin}(n), & \text{ако } k > m \\ \text{bin}(n), & \text{ако } k = m \\ \text{суфиксът с дължина } k \text{ на } \text{bin}(n), & \text{ако } k < m. \end{cases}$$

Забележете, че в случая m е дължината на представянето на n в двоична позиционна бройна система. Също така забележете, че $\text{bin}(n)$ и $\text{bin}_k(n)$ са стрингове над азбуката $\{0, 1\}$, а не числа.

Нека x и y са произволни два върха от едно и също ниво и връх c е общият предшественик с най-голяма дълбочина на x и y . Казваме, че x е вляво от y , ако x е връх от $T[a]$ и y е връх от $T[b]$, където a е лявото дете, а b е дясното дете на c .

Лема 28

Нека $T = (V, E)$ е произволно попълнено двоично дърво с n върха. Нека височината на T е h . Нека V_d е подмножеството от върховете на дълбочина d , за $0 \leq d \leq h$. Тогава

- За $0 \leq d \leq h - 1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина $d + 1$ и водеща единица.
- Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина $h + 1$ и водеща единица от 10^h до $\text{bin}(n)$ в лексикографската наредба.
- Нещо повече, във всяко ниво върховете са лексикографски сортирани по адреси отляво надясно в T .

Доказателство: Нека $\text{addr}'(x)$ е адресът на x след **фаза I**, за всеки връх $x \in V$. В термините на $\text{addr}'(x)$, твърдения (а) и (б) са еквивалентни съответно на:

- (а') За $0 \leq d \leq h-1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина d .
- (б') Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина h от 0^h до $\text{bin}_h(n)$ в лексикографската наредба.

Това ще докажем чрез структурна индукция съгласно Определение 54 на попълнено дърво, използвайки Определение 55 на адреси на върховете на попълнено дърво.

База. В случай 1 на Определение 54, единственият връх има адрес ϵ . В случай 2,

- в ниво 0 единственият връх наистина има адрес ϵ , който има дължина 0,
- в последното ниво 1 наистина адресите са от 0 до $\text{bin}_1(2) = 0$. ✓

Индуктивна стъпка Да разгледаме случай 3.

- На ниво 0, коренът r наистина получава адрес ϵ .
- Да разгледаме произволно ниво d , където $1 \leq d \leq h-1$. Тривиално е да се докаже, че върховете от ниво d в T са 2^d на брой и че тяхната наредба отляво надясно се състои от върховете на ниво $d-1$ в T' , последвани от върховете от ниво $d-1$ на T'' . Тъй като $0 \leq d-1 \leq h-2$, ниво $d-1$ не е последно ниво нито в T' , нито в T'' . Тогава, съгласно част (а') на индуктивното допускане, адресите на върховете от ниво $d-1$ и в T' , и в T'' са булевите стрингове с дължина $d-1$, които и в T' , и в T'' са сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d-1$ в T' и на 1 вляво на всеки адрес от ниво $d-1$ в T'' получаваме—по отношение на T —всички булеви стрингове с дължина d , сортирани лексикографски.
- Да разгледаме последното ниво h в T . В случай 3 то се състои от всички 2^{h-1} върхове от ниво $h-1$ на T' , последвани вдясно от върховете от ниво $h-1$ на T'' .

Да разгледаме последното ниво $h-1$ в T' . Нека T' има p върха и T'' има q върха. Съгласно част (б') на индуктивното предположение, отляво надясно адресите в ниво $h-1$ на T' са

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \quad \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{\text{bin}_{h-1}(p)}_{\text{дължина } h-1}$$

Но $p = 2^{h-1+1} - 1 = 2^h - 1$, защото T' е свършено дърво, така че $\text{bin}_{h-1}(p) = 1^{h-1}$ единици. Тогава адресите в ниво $h-1$ на T' са, отляво надясно,

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \quad \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{11\dots 1}_{\text{дължина } h-1}$$

тоест всички булеви стрингове с дължина $h-1$.

Да разгледаме последното ниво $h-1$ в T'' . Съгласно част (а') на индуктивното предположение, отляво надясно адресите в ниво $h-1$ на T'' са

$$\underbrace{00\dots 0}_{\text{дължина } h-1}, \quad \underbrace{00\dots 01}_{\text{дължина } h-1}, \dots, \underbrace{\text{bin}_{h-1}(q)}_{\text{дължина } h-1}$$

Следователно, в цялото дърво T , адресите на върховете от ниво h са, отляво надясно

$$\underbrace{00\dots 0}_{\text{дължина } h}, \underbrace{00\dots 01}_{\text{дължина } h}, \dots, \underbrace{011\dots 1}_{\text{дължина } h}, \underbrace{10\dots 0}_{\text{дължина } h}, \underbrace{10\dots 01}_{\text{дължина } h}, \dots, \underbrace{1\text{bin}_{h-1}(q)}_{\text{дължина } h}$$

Тъй като $n = p + q + 1$ и $p = 2^h - 1$, то $n = 2^h + q$. Но тъй като $q \leq p$, дължината на $\text{bin}(q)$ също е по-малка от h и $\text{bin}_{h-1}(q)$ се получава от $\text{bin}(q)$ с добавяне на ≥ 0 на брой нули вляво. Имайки предвид, че $\text{bin}(2^h) = 10^h$, заключаваме, че стрингът $1\text{bin}_{1-h}(q)$ е равен на $\text{bin}(n)$. Заключаваме, че адресите от последното ниво на T' отляво надясно, следвани от адресите от последното ниво на T'' отляво надясно, представляват редицата

$$\underbrace{00\dots 0}_{\text{дължина } h}, \underbrace{00\dots 01}_{\text{дължина } h}, \dots, \underbrace{011\dots 1}_{\text{дължина } h}, \underbrace{10\dots 0}_{\text{дължина } h}, \underbrace{10\dots 01}_{\text{дължина } h}, \dots, \underbrace{\text{bin}(n)}_{\text{дължина } h}$$

Да разгледаме случай 4.

- На ниво 0, коренът r наистина получава адрес ϵ .
- Да разгледаме произволно ниво d в T , където $1 \leq d \leq h - 1$. Аналогично на предния случай, върховете от ниво d в T са 2^d на брой и тяхната наредба отляво надясно се състои от върховете на ниво $d - 1$ в T' , последвани от върховете от ниво $d - 1$ на T'' . Тъй като $0 \leq d - 1 \leq h - 2$, ниво $d - 1$ не е последното ниво в T' , но е последното ниво в съвършеното дърво T'' при $d - 1 = h - 2$.

Адресите на върховете от ниво $d - 1$ в T' са булевите стрингове с дължина $d - 1$ и са сортирани отляво надясно съгласно част (а') на индуктивното предположение. Ако $d - 1 < h - 2$, адресите на върховете от ниво $d - 1$ в T'' са булевите стрингове с дължина $d - 1$ и са сортирани отляво надясно съгласно част (а') на индуктивното предположение.

Да разгледаме случая $d - 1 = h - 2$ по отношение на T'' . Сега част (б') на индуктивното предположение е приложима. Съгласно нея, адресите на върховете от ниво $d - 1$ в T'' са булевите стрингове от 0^{h-2} до $\text{bin}_{h-2}(q)$ в лексикографската наредба, където q е броят на върховете в T'' , и те са сортирани отляво надясно лексикографски. Но тъй като T'' е съвършено дърво с височина $h - 2$, вярно е, че $q = 2^{h-2+1} - 1 = 2^{h-1} - 1$, следователно $\text{bin}_{h-2}(q) = 1^{d-1}$.

Заключаваме, че за $1 \leq d \leq h - 1$, адресите на върховете от ниво $d - 1$ в T' са всички булеви стрингове с дължина $d - 1$, сортирани отляво надясно, а също така адресите на върховете от ниво $d - 1$ в T'' са всички булеви стрингове с дължина $d - 1$ сортирани отляво надясно. Очевидно след слагане на 0 вляво на всеки адрес от ниво $d - 1$ в T' и на 1 вляво на всеки адрес от ниво $d - 1$ в T'' получаваме—по отношение на дървото T —всички булеви стрингове с дължина d , сортирани лексикографски отляво надясно.

- Да разгледаме последното ниво h в T . То е последното ниво $h - 1$ на T' , така че част (б') на индуктивното предположение е приложима. Съгласно нея, адресите на върховете от ниво $h - 1$ в T' са булевите стрингове от 0^{h-1} до $\text{bin}_{h-1}(p)$, където p е броят на върховете в T' , и те са сортирани отляво надясно лексикографски. Тогава във **фаза I** тези върхове получават адреси от 0^h до $0\text{bin}_{h-1}(p)$, сортирани отляво надясно лексикографски.

Да си припомним, че $n = p + q + 1$, където q е броят на върховете в съвършеното дърво T'' . Тъй като T'' е съвършено и с височина $h - 2$, вярно е, че $q = 2^{h-1} - 1$, следователно $q + 1 = 2^{h-1}$, следователно $\text{bin}(q + 1) = 10^{h-1}$. От друга страна, $p \geq q + 1$, така че

$\text{bin}(p) = 1\sigma$ за някой булев стринг σ с дължина $h - 1$. Тогава $\text{bin}(p + q + 1) = 10\sigma$. С други думи, $\text{bin}(n) = 10\sigma$. Но тогава $\text{bin}_h(n) = 0\sigma = 0\text{bin}_{h-1}(p)$.

Докажем, че наистина адресите на върховете от последното ниво на T са лексикографски сортираните стрингове от 0^h до $\text{bin}_h(n)$. $\square$

5.1.3 Двоична пирамида

Сега допускаме, че всеки връх има асоциирана стойност, наречена *ключ* (key). В общия случай тази стойност е реално число, но за простота в примерите, които ще разгледаме, ключовете са цели положителни числа.

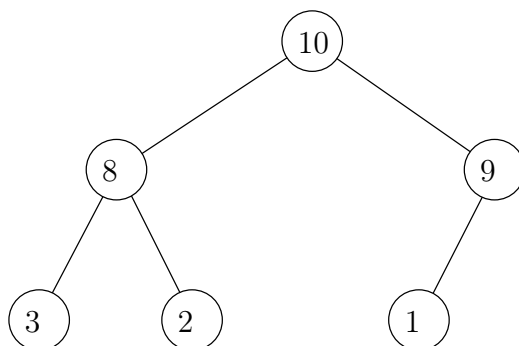
Определение 56: Двоична пирамида (binary heap)

Двоична пирамида, на английски *binary heap*, е попълнено двоично дърво, в което:

- или ключът на всеки връх е по-голям или равен на ключовете на неговите деца и тогава пирамидата е *максимална пирамида* (max heap),
- или ключът на всеки връх е по-малък или равен на ключовете на неговите деца и тогава пирамидата е *минимална пирамида* (min heap).

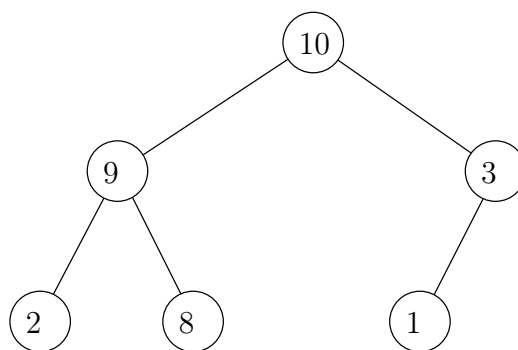
Тук ще разглеждаме максимални пирамиди и казвайки “пирамида” ще имаме предвид “максимална пирамида”. Но изложението може да бъде променено лесно, така че да се отнася за минимални пирамиди. Също така, ще изпускаме “двоични”, понеже не разглеждаме пирамиди, които не са двоични.

Ето пример за пирамида:



Очевидно формата на дървото е фиксирана от броя на върховете, но подредбата на ключовете не е фиксирана[†]. Най-големият ключ 10 непременно е в корена и 9 непременно е в дете на корена, но известно вариране е възможно. Ето друг пример за пирамида със същите ключове:

[†]В Допълнение 32 дискутираме броя на различните пирамиди с n върха.

**Наблюдение 36**

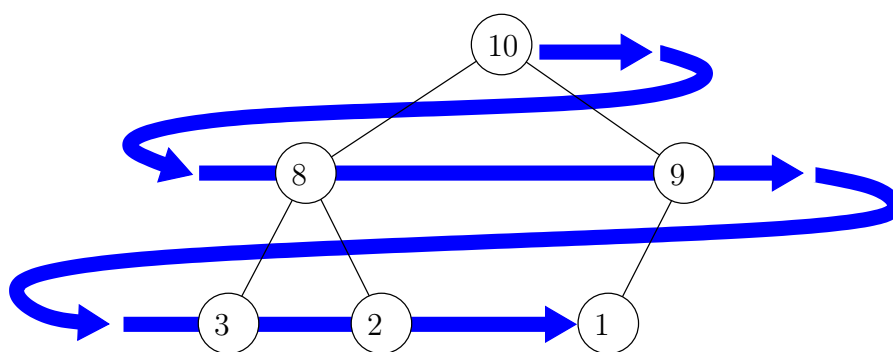
Попълнено двоично дърво с ключове е пирамида тогава и само тогава, когато за всяко листо u , по уникалния път, свързващ корена с u , стойностите на ключовете са ненарастващи в посока от корена към u .

5.1.4 Реализиране на пирамида чрез масив**Определение 57**

Нека $T = (V, E)$ е попълнено двоично дърво с n върха. *Линеаризация* на T е масивът $A[1..n]$, в който за всяко i , $1 \leq i \leq n$, $A[i]$ е ключът на върха, чийто адрес е i .

За да съкратим изложението, често игнорираме разликата между “пирамида” и “линеаризация на пирамида” и, казвайки просто “пирамида”, имаме предвид “линеаризация на пирамида”. С други думи, самият масив-линеаризация бива наричан “пирамида”.

Съгласно Лема 28, съществува биекция между множеството от адресите и множеството $\{1, 2, \dots, n\}$, така че определението е смислено. И така, пирамидите може да се реализират много ефикасно чрез масиви. Линеаризацията на първия пример е $[10, 8, 9, 3, 2, 1]$. Линеаризирането можем да си представяме като обхождане на дървото по нива, във всяко ниво, отляво надясно, и последователно записване на ключовете в масива:



Лема 28 гарантира, че в нивата няма “празнини”, така че в масива няма “дупки”.

При тази реализация се избягва разходът на памет, който е неизбежен при дърветата по принцип. Ако искаме да представим двоично дърво в общия случай—не непременно попълнено—трябва по някакъв начин да укажем за всеки връх кой е родителят, или кои са децата. Това ни струва допълнителна памет. Докато пирамида може да се реализира без никаква допълнителна памет, като при това обхождането е много ефикасно като сложност по време. Последното твърдение се основава на факта (виж Наблюдение 37), че ако масивът

$A[1..n]$ е линеаризация на пирамида, то за всеки елемент $A[i]$ можем в $\Theta(1)$ време да определим дали $A[i]$ е корен или не, ако не е, то кой е родителят му, дали е вътрешен връх или листо, и ако е вътрешен връх, кое е лявото му дете $A[j]$, ако такова съществува, и дясното му дете $A[k]$, ако такова съществува.

Следното наблюдение е просто приложение на Лема 27.

Наблюдение 37

Нека T е пирамида с n елемента и масивът $A[1..n]$ е нейната линеаризация. Тогава за всеки елемент $A[i]$:

1. елементът, който отговаря на родителя на $A[i]$, е $A[\lfloor \frac{i}{2} \rfloor]$, ако върхът на T , който отговаря на $A[i]$, не е коренът.
2. елементът, който отговаря на лявото дете на $A[i]$, е $A[2 \times i]$, ако върхът на T , отговарящ на $A[i]$, има ляво дете,
3. елементът, който отговаря на дясното дете на $A[i]$, е $A[2 \times i + 1]$, ако върхът на T , отговарящ на $A[i]$, има дясно дете. $\square$

Наблюдение 37 ни дава основание да дефинираме следните изчислителни примитиви[†], чиито имена са достатъчно информативни:

$$\begin{aligned} \text{PARENT}(i) \\ \text{return (if } i \geq 2 \text{ then } \lfloor \frac{i}{2} \rfloor \text{ else } i) \end{aligned} \quad (5.5)$$

$$\begin{aligned} \text{LEFT}(i) \\ \text{return (if } 2i \leq n \text{ then } 2i \text{ else UNDEFINED)} \end{aligned} \quad (5.6)$$

$$\begin{aligned} \text{RIGHT}(i) \\ \text{return (if } 2i + 1 \leq n \text{ then } 2i + 1 \text{ else UNDEFINED)} \end{aligned} \quad (5.7)$$

$$\begin{aligned} \text{LEVEL}(i) \\ \text{return } \lfloor \log_2 i \rfloor \end{aligned}$$

$$\begin{aligned} \text{ISLEAF}(i) \\ \text{return (if } i > \lfloor \frac{n}{2} \rfloor \text{ then YES else NO)} \end{aligned}$$

$$\begin{aligned} \text{ISINTERNALVERTEX}(i) \\ \text{return not ISLEAF}(i) \end{aligned}$$

$$\begin{aligned} \text{ISROOT}(i) \\ \text{return (if } i = 1 \text{ then YES else NO)} \end{aligned}$$

[†]“Примитив” е, неформално казано, лесна за възприемане функция с малка сложност, която се използва често от алгоритмите, които ни интересуват.

Наблюдение 38

Забележете, че $\text{PARENT}(1) = 1$. С други думи, коренът е родител на себе си съгласно (5.5). Това не следва по никакъв начин от Наблюдение 37! Можеше да възприемем алтернативния подход, а именно, че коренът няма родител, но изложението е по-кратко и елегантно, ако всеки елемент има родител, включително и коренът.

Определение 58: Итерирани родители и предшественици в пирамида

За всяко $t \geq 1$, дефинираме, че

$$\text{PARENT}^t(i) = \begin{cases} i, & \text{ако } t = 0 \\ \text{PARENT}(\text{PARENT}^{t-1}(i)), & \text{ако } t > 0 \end{cases}$$

и

$$\text{ANCESTORS}(i) = \{\text{PARENT}^t(i) \mid t \geq 1\}$$

Казваме, че $A[j]$ е *предшественик* на $A[i]$, ако $j \in \text{ANCESTORS}(i)$. По това определение никой елемент на линейаризацията не е предшественик на себе си, освен $A[1]$. Предвид последното, ясно е, че $\text{ANCESTORS}(i)$ е крайно множество, въпреки че “ $t \geq 1$ ” означава “ $t \in \mathbb{N}^+$ ”.

Определение 59: Пирамидална инверсия в масив

Нека $A[1..n]$ е масив от ключове. *Пирамидална инверсия* в A наричаме всяка наредена двойка индекси $\langle i, j \rangle$, такива че $i < j$, $A[i]$ е предшественик на $A[j]$ и $A[i] < A[j]$. Наредената двойка $\langle A[i], A[j] \rangle$ също наричаме пирамидална инверсия. *Директна пирамидална инверсия* е всяка пирамидална инверсия $\langle i, j \rangle$, такава че $i = \text{PARENT}(j)$. Пирамидални инверсии, които не са директни, се казват *индиректни*.

Ако от контекста е ясно, че става дума за пирамидални инверсии, може да изпускаме прилагателното “пирамидални” и да говорим просто за *инверсии*.

Например, на Фигура 5.10 е показано попълнено дърво с ключове (неговата линейаризация е на Фигура 5.11 на най-горния ред), което би било пирамида, ако не беше най-дясното листо на последното ниво, нарисувано в червено и с ключ 35. В дървото има точно три пирамидални инверсии, които са между това листо и негови предшественици; а именно, предшествениците с ключове 11, 9 и 6. Директната инверсия е с предшественика с ключ 6.

Наблюдение 39

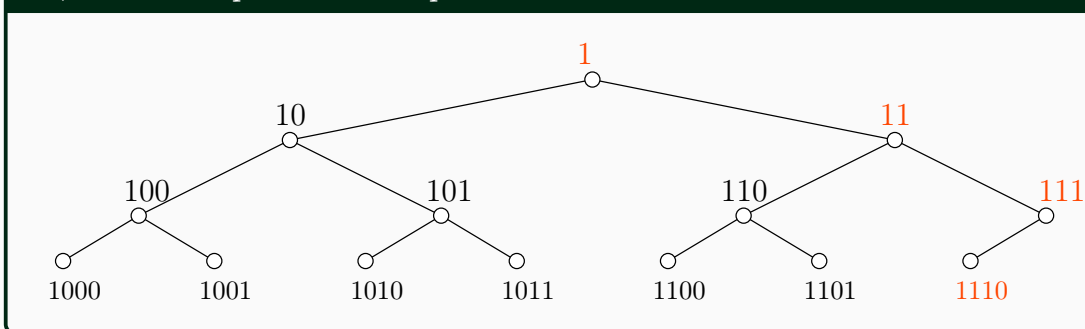
Масивът от ключове A е пирамида тогава и само тогава, когато няма нито една пирамидална инверсия. □

Допълнение 32: За броя на пирамидите на n върха

В това допълнение ще разгледаме интересния въпрос, колко от всички $n!$ пермутации на n различни числа са пирамиди; по-точно казано, линейаризации на пирамиди. Изложението следва изложението в [94, стр. 152].

Да разгледаме отново попълненото двоично дърво с адресите от Подсекция 5.1.2, което за удобство показваме отново на Фигура 5.4.

Фигура 5.4 : Попълнено дърво с 14 върха и техните адреси. Специалните върхове са в червено.



Дървото има 14 върха. Да запишем броевете на върховете на поддърветата, вкоренени във всички върхове на дървото, в нарастващ ред по адресите на тези върхове. Ще записваме броевете в двоична позиционна бройна система:

1110, 111, 110, 11, 11, 11, 10, 1, 1, 1, 1, 1, 1, 1

За да е напълно ясно: цялото дърво (тоест, поддървото с корен върха, чийто адрес е 1) има четиринадесет върха, което в двоична позиционна бройна система е 1110. Поддървото, вкоренено в следващия връх (с адрес 10) има седем върха, така че записваме 111. И така нататък. С червен цвят са записани броевете на върховете в тези поддървета, чиито корени лежат на пътя между корена на дървото и последния връх (чийто адрес е $\text{bin}(n)$). Тези върхове са наречени в [94], *специалните върхове*, а дърветата, вкоренени в тях, *специалните поддървета*. В [94, стр. 157, упр. 20 и 21] е показано, че ако $\text{bin}(n) = b_k b_{k-1} \dots b_1 b_0$, където $k = \lfloor \lg n \rfloor$ и $b_k = 1$, то размерите^a на специалните поддървета, записани в двоична позиционна бройна система, са

$$1b_{k-1} \dots b_1 b_0, \quad 1b_{k-2} \dots b_1 b_0, \quad \dots \quad 1b_1 b_0, \quad 1b_0, \quad 1$$

а размерите на неспециалните поддървета числа от вида $2^m - 1$, защото те са свършени двоични дървета, и броевете на неспециалните поддървета от всяка срещаща се големина са съответно

$$\begin{aligned} & \left\lfloor \frac{n-1}{2} \right\rfloor \text{ числа } 1, \\ & \left\lfloor \frac{n-2}{4} \right\rfloor \text{ числа } 3, \\ & \left\lfloor \frac{n-4}{8} \right\rfloor \text{ числа } 7, \\ & \dots \\ & \left\lfloor \frac{n-2^{k-1}}{2^k} \right\rfloor \text{ числа } 2^k - 1 \end{aligned}$$

Знаейки размерите на поддърветата, можем да изчислим H_n ⁶: броя на начините да разположим ключовете $\{1, 2, \dots, n\}$ в пирамида, като използваме резултата от [94, стр.

67, упр. 20]:

$$H_n = \frac{n!}{s_1 \times s_2 \times \cdots \times s_n} \quad (5.8)$$

където $\{s_1, s_2, \dots, s_n\}_M$ е мултимножеството от всички размери на поддървета (вж. Определение 42). Примерно, ако елементите са 14, каквото е дървото на Фигура 5.4, начините да сложим ключовете $\{1, 2, \dots, 14\}$, така че дървото да е пирамида, са на брой

$$H_{14} = \frac{14!}{14 \times 6 \times 2 \times 1 \times 1^6 \times 3^3 \times 7^1} = 2745600$$

Сравнете H_{14} с $14! = 87178291200$.

Редицата H_n за $n \geq 1$ е редица A056971 в онлайн енциклопедията на целочислените редици. Точният израз (5.8) не дава представа за асимптотиката на H_n . Асимптотиката на H_n се разглежда в [80]. Основният резултат е твърде сложен, но грубо приближение е следното. Ако $h_n = \log\left(\frac{n!}{H_n}\right)$, то $h_n \asymp n$. Оттук—това е грубо приближение—асимптотиката на H_n е приблизително $\frac{n!}{c^n}$ за някаква константа c .

^aТова са общо $k = \lfloor \lg n \rfloor$ числа, колкото е височината на попълненото двоично дърво с n върха съгласно Теорема 59.

^bДа не се бърка с n -тата парциална сума на хармоничния ред! Случайно съвпадение на имена.

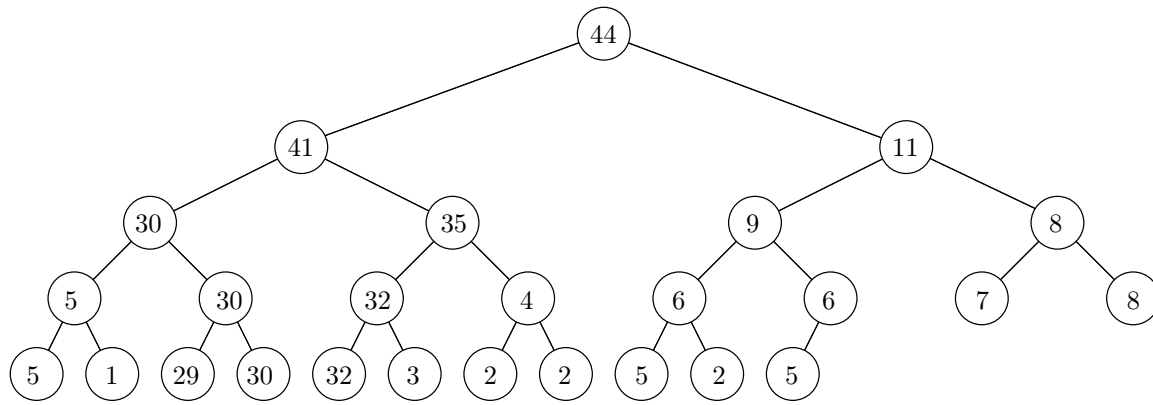
5.1.5 Подпирамида

Нека T е пирамида с n върха, реализирана с масив $A[1..n]$. За всеки връх $u \in V(T)$, *подпирамидата с корен u* е $T[u]$. По отношение на масива A , който престава пирамидата, u е индекс на елемент. Нотацията $A[u]$ означава реализацията на $T[u]$ в масива A . Очевидно, в общия случай $A[u]$ не е непрекъснат подмасив, а се състои от няколко подмасива—на брой колкото е височината на съответното поддърво—всеки от които е непрекъснат подмасив на A (вижте Фигура 5.8).

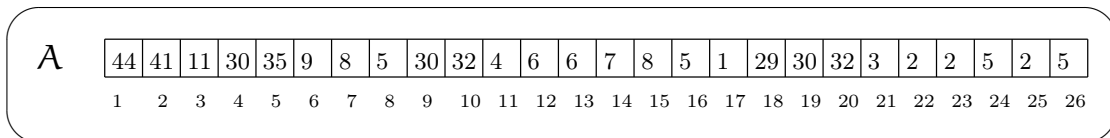
Наблюдение 40

Ако $A[1..n]$ е пирамида, то за всяко $k \in \{1, 2, \dots, n\}$, $A[k]$ е пирамида. □

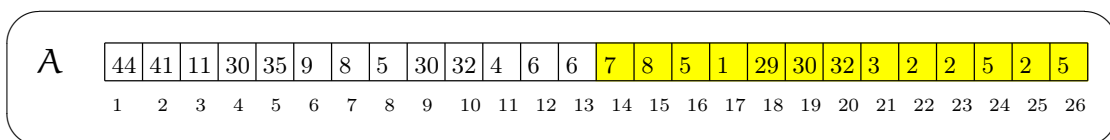
Като пример за пирамида и нейната линеаризация, да разгледаме пирамидата, показана на Фигура 5.5. Фигура 5.6 показва нейната линеаризация. Фигура 5.7 показва листата на пирамидата. На Фигура 5.8 е показана подпирамидата $A[3]$. Връх 3 е третият връх в масива (дясното дете на корена, ако разсъждаваме в термините на дървото). Фигура 5.9 показва листата на $A[3]$ в червено.



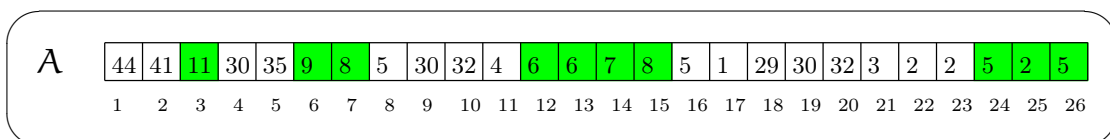
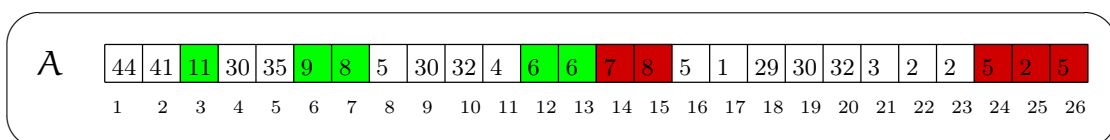
Фигура 5.5: Пирамида с 26 върха.



Фигура 5.6: Линеаризацията на пирамида от Фигура 5.5.



Фигура 5.7: Листата на пирамидата от Фигура 5.5 са отбелязани с жълто.

Фигура 5.8: Подпирамидата $A[3]$.Фигура 5.9: Листата на $A[3]$ в червено.

Използвайки дефинираните примитиви, следният алгоритъм обхожда $T[u]$. “Обхожда” означава “извежда ключовете на $T[u]$ в *preorder*[†]” (а не по нива).

TRAVERSE BINARY SUBHEAP($A[1..n]$: пирамида, i : число от $\{1, 2, \dots, n\}$)

```

1  print i
2  left ← LEFT(i)
3  right ← RIGHT(i)
4  if left ≤ n
5      TRAVERSE BINARY SUBHEAP(A, left )
6  if right ≤ n
7      TRAVERSE BINARY SUBHEAP(A, right )

```

5.2 Построяване на пирамида

Даден е масив $A[1..n]$ от ключове. За простота допускаме, че ключовете се естествени числа. Числата са подредени произволно. Искаме да разместим числата така, че масивът да стане пирамида. Дори числата да са две по две различни, има много начини да направим пирамида от дадени числа. За подробна дискусия на броя на пирамидите вижте Допълнение 32.

5.2.1 Наивно построяване на пирамида

Най-естественият начин да бъде превърнат произволен масив от числа в пирамида чрез разместване е чрез последователно добавяне на елементи в частично построена пирамида.

NAIVE BUILD HEAP, OUTLINE($A[1..n]$: масив от цели числа)

```

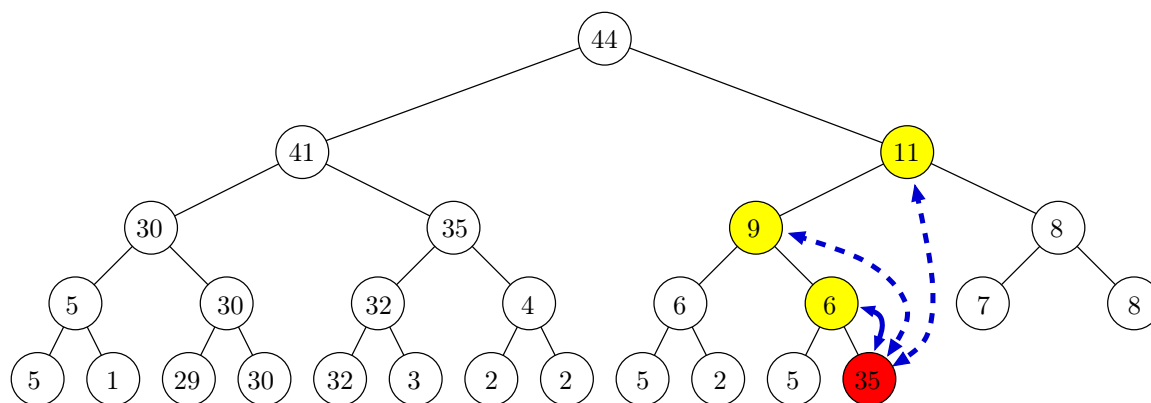
1  for i ← 2 to n
2      добави  $A[i]$  към досега построената пирамида  $A[1..i-1]$ ,
      така че  $A[1..i]$  да стане пирамида

```

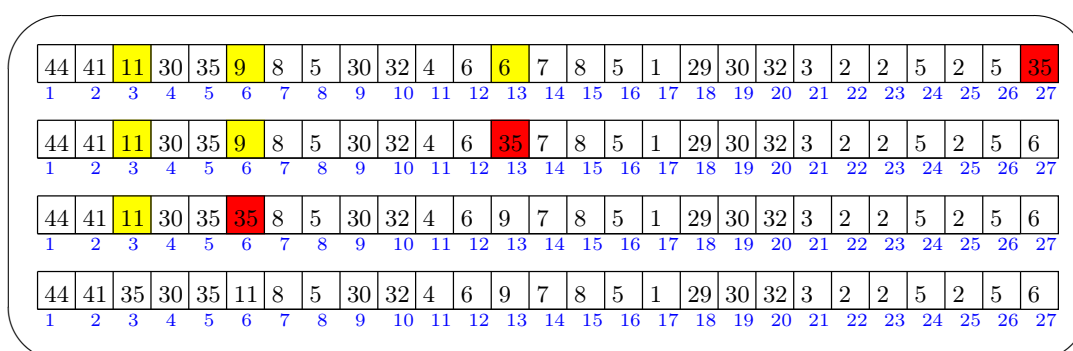
Това описание е прекалено общо, но идеята е ясна: “добави към $A[1..i-1]$ ” не е просто увеличаване на индекса i . Дори $A[1..i-1]$ да е пирамида, $A[1..i]$ не е непременно пирамида, защото може да има пирамидални инверсии $\langle \text{PARENT}^t(i), i \rangle$. Това е илюстрирано на Фигура 5.10, на която структурата е показана като дърво, а не като масив. Дървото съдържа пирамидата от Фигура 5.5 плюс още един връх, оцветен в червено. Ключът на този нов връх е 35 и той участва в пирамидални инверсии с три от своите предшественици. Съществуват много начини да бъдат разместени елементите на $A[1..i]$, така че да няма пирамидални инверсии. Най-естественият и прост начин е, да ликвидираме пирамидалните инверсии, в които участва новодобавеният $A[i]$, без да разместваме никакви елементи, които не участват в тези инверсии. Да мислим за обекта като за дърво. Тогава въпросните инверсии са между новодобавения $A[i]$ и върхове по пътя от него към корена, които върхове индуцират подпът (тоест, по пътя от $A[i]$ към корена, върховете, участващи в инверсии, са непрекъснатата редица). Очевидният подход тогава е да разменяме $A[i]$ с $A[\text{PARENT}(i)]$, докато има инверсии.

Всяка такава размяна ликвидира една инверсия и не въвежда нови инверсии, защото ключът, който застава “отгоре”, е по-голям от този, който преди е бил там преди размяната, следователно този елемент, който застава отгоре, остава не по-малък от другото си дете, ако има такова.

[†]За обхождания на дървета, включително в *preorder*, вижте примерно [142].



Фигура 5.10: Почти-пирамидата, получена от пирамидата от Фигура 5.5 с един добавен връх (в червено). Има точно три пирамидални инверсии между новодобавения елемент и негови предшественици. Пирамидалните инверсии са показани със сини стрелки, като с непрекъснатата линия е показана единствената директна пирамидална инверсия, а с прекъснати линии – двете индиректни пирамидални инверсии.



Фигура 5.11: Линеаризацията на почти-пирамидата от Фигура 5.5 е на най-горния ред. Елементът, който “не си е на мястото”, е накрая в червено. Елементите, които образуват пирамидални инверсии с него, са в жълто. Следващите три реда отгоре надолу съответстват на трите изпълнения на **while**-цикъла, всяко от които “избутва” 35 наляво в масива.

Алгоритъм 6: Naive Build Heap

NAIVE BUILD HEAP($A[1..n]$: масив от ест. числа)

```

1  for  $i \leftarrow 2$  to  $n$ 
2     $k \leftarrow i$ 
3    while  $k > 1$  and  $A[\text{PARENT}(k)] < A[k]$  do
4      swap( $A[k], A[\text{PARENT}(k)]$ )
5       $k \leftarrow \text{PARENT}(k)$ 
```

Фигура 5.11 илюстрира едно изпълнение на **for**-цикъла на NAIVE BUILD HEAP, в **while**-цикълът разменя елемента 35 с елементи вляво, докато има нужда.

Сега ще докажем подробно коректността на наивното построяване на пирамида. Тъй като алгоритъмът е итеративен, ще направим доказателството с инвариант на цикъла. Преди това обаче ще дадем една дефиниция, която значително опростява изложението.

Определение 60: Почти-пирамида

Почти-пирамида е попълнено двоично дърво, чиито ключове удовлетворяват изискването за пирамида може би с изключение на най-дясното листо на последния ред.

Иначе казано, линеаризацията на почти-пирамидата е такава, че всички пирамидални инверсии са между последния елемент и негови предшественици. Фигура 5.10 и най-горният ред на Фигура 5.11 показват почти-пирамида и нейната линеаризация, като добре илюстрират факта, че всички инверсии са между последния елемент и негови предшественици, които са “плътно подредени” над него (в дървото). Това си заслужава да бъде формулирано в самостоятелно наблюдение.

Наблюдение 41: Инверсиите в почти-пирамида

Нека $A[1..n]$ е линеаризация на почти-пирамида. Тогава всички пирамидални инверсии са от вида $\langle i, n \rangle$, където $i \in \text{ANCESTORS}(n)$. Нещо повече, ако $\langle i, n \rangle$ е инверсия, то за всяко k , такова че $i < k < n$ и $k \in \text{ANCESTORS}(n)$, е вярно, че $\langle k, n \rangle$ е инверсия.

Лема 29: Вътрешният цикъл на NAIVE BUILD HEAP

Спрямо едно единствено изпълнение на **for**-цикъла на NAIVE BUILD HEAP (редове 1–5), нека t е моментът непосредствено преди изпълнението на **while** цикъла (редове 3–5). Нека наричаме A' масива A в момента t . Да допуснем, че $A'[1..i]$ реализира почти-пирамида. Тогава в момента на приключване на **while** цикъла е вярно, че текущият $A[1..i]$ реализира пирамида.

Доказателство: В момента t е вярно, че $k = i$ заради присвояването на ред 2. Разглеждаме поотделно два случая за момента t .

Първо допусκαе, че в момента t в $A'[1..i]$ няма инверсии. Тогава $A'[1..i]$ е пирамида. От друга страна, **while** цикълът не се изпълнява изобщо, понеже $A[\text{PARENT}(k)] \geq A[k]$, така че изпълнението отива на ред 1. Спрямо новото i е вярно, че текущият $A[1..i-1]$ е същият масив като $A'[1..i]$ спрямо предишното i . Ерго, текущият $A[1..i-1]$ е пирамида, което трябваше да докажем.

Сега допусκαе, че в момента t има инверсии. Съгласно Наблюдение 41, всички инверсии са от вида $\langle x, i \rangle$, където $x \in \text{ANCESTORS}(i)$. Дефинираме a като минималното x , такова че $\langle x, i \rangle$ е инверсия. Следното твърдение е инвариант за **while** цикъла.

Инвариант 13: Вътрешният цикъл на NAIVE BUILD HEAP

При всяко достигане на ред 3, множеството от пирамидалните инверсии в текущия $A[1..i]$ е $\{\langle b, k \rangle \mid b \in \text{ANCESTORS}(k) \text{ и } b \geq a\}$.

База. Когато изпълнението достигне до ред 3 за първи път, $A'[1..i]$ реализира почти-пирамида по допускане от условието на Лема 29. Тъй като $k = i$ заради присвояването на ред 2, твърдението на инварианта става “множеството от пирамидалните инверсии в текущия $A[1..i]$ е $\{\langle b, i \rangle \mid b \in \text{ANCESTORS}(i) \text{ и } b \geq a\}$ ”, което следва веднага от Наблюдение 41 и дефиницията на a . ✓

Поддръжка. Допускаме, че твърдението е вярно за някое достигане на ред 3, което не е последното. На ред 4 става размяна на $A[k]$ с $A[\text{PARENT}(k)]$, с което множеството от

пирамидалните инверсии в текущия $A[1 \dots i]$ става $\{\langle b, k \rangle \mid b \in \text{ANCESTORS}(\text{PARENT}(k)) \text{ и } b \geq a\}$. Но след изпълнението на ред 5, спрямо новото k , отново е вярно, че множеството от пирамидалните инверсии в текущия $A[1 \dots i]$ е $\{\langle b, k \rangle \mid b \in \text{ANCESTORS}(k) \text{ и } b \geq a\}$.

Терминация. При последното достигане на ред 3 е вярно поне едно от двете: $k \leq 1$ или $A[\text{PARENT}(k)] \geq A[k]$.

- Ако $k \leq 1$, то $\text{ANCESTORS}(k) = \emptyset$, така че множеството от пирамидалните инверсии в текущия $A[1 \dots i]$ е празното множество. С други думи, инверсии няма.
- Нека $A[\text{PARENT}(k)] \geq A[k]$. От това и транзитивността на релацията на наредба на ключовете следва, че за всяко $b \in \text{ANCESTORS}(k)$, $\langle b, k \rangle$ не е инверсия. Но съгласно инварианта, множеството от пирамидалните инверсии в текущия $A[1 \dots i]$ е

$$\{\langle b, k \rangle \mid b \in \text{ANCESTORS}(k) \text{ и } b \geq a\}$$

Заклучаваме, че това множество е празното множество[†]. С други думи, пирамидални инверсии няма. $\square$

Теорема 63: Коректността на наивното построяване на пирамида

За всеки вход A , NAIVE BUILD HEAP построява пирамида от него.

Доказателство: Следното твърдение е инвариант на **for**-цикъла (редове 1–5).

Инвариант 14: Цикълът на NAIVE BUILD HEAP

Всеки път, когато изпълнението е на ред 1, $A[1 \dots i]$ реализира почти-пирамида.

База. При първото достигане на ред 1 е вярно, че $i = 2$. Съгласно Определение 60, всеки двueleментен масив реализира почти-пирамида, така че инвариантът е верен. $\checkmark$

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 1, което не е последното. На ред 2, k приема стойност i . Прилагаме Лема 29. Съгласно лемата, при допускането, че при влизането в **while**-а е изпълнено текущият $A[1 \dots i]$ да реализира почти-пирамида и k да е равно на i , вярно е, че при излизането от **while**-а, $A[1 \dots i]$ реализира пирамида. Тогава при следващото достигане на ред 1, променливата i се инкрементира и спрямо новото i е вярно, че $A[1 \dots i]$ реализира почти-пирамида.

Терминация. При последното достигане на ред 1 е вярно, че $i = n + 1$. Твърдението “ $A[1 \dots n + 1]$ реализира почти-пирамида” е формално некоректно, защото говори за несъществуващия елемент $A[n + 1]$ [‡]. Ако обаче игнорираме тази несъществена подробност, това твърдение ни дава желанния резултат, защото казва, че в $A[1 \dots n]$ няма пирамидални инверсии. Тоест, $A[1 \dots n]$ реализира пирамида. $\square$

Сега ще изследваме сложността по време на наивното построяване на пирамида. Външният цикъл се изпълнява $\Theta(n)$ пъти без оглед на конкретния вход. Вътрешният цикъл се изпълнява, в най-лошия случай, $\lceil \lg i \rceil$ пъти за всяко i , понеже височината на дървото $A[1 \dots i]$ е

[†]А за да бъде то празно, трябва да е вярно, че за всяко $b \in \text{ANCESTORS}(k)$, $b < a$.

[‡]Тази формална непрецизност можеше да бъде избягната, ако поначало бяхме присъединили към входния $A[1 \dots n]$ още един елемент $A[n + 1]$ със стойност $-\infty$. Този елемент би имал роля, аналогична на т. нар. пазач от Определение 61.

$\lfloor \lg i \rfloor$. Тогава сложността се определя от сумата

$$\sum_{i=2}^n \lfloor \lg i \rfloor = \Theta \left(\sum_{i=2}^n \lg i \right) = \Theta (\lg (2 \times 3 \times \cdots \times (n-1) \times n)) = \Theta (\lg n!) \quad (5.9)$$

От Теорема 33 знаем, че $\lg n! \asymp n \lg n$. Следователно, сложността на наивното построяване е $\Theta(n \lg n)$.

5.2.2 Бързо построяване на пирамида: алгоритъм BUILD HEAP

Алгоритъмът-предмет на тази подсекция е предложен от Robert Floyd [49]. Самият алгоритъм е показан на стр. 343. Той използва функция HEAPIFY, която показваме в два варианта: итеративен и рекурсивен.

5.2.2.1 Предварителни обяснения

Да допуснем, че е дадено попълнено двоично дърво с ключове T . Нека коренът на T е u и децата на u са v и w . За целта на това обяснение, нека ключовете са съответно $u.key$, $v.key$ и $w.key$. Нека $T[v]$ и $T[w]$ са пирамиди. Ако $u.key \geq v.key$ и $u.key \geq w.key$, то T е пирамида. Да допуснем, че $u.key < v.key$ или $u.key < w.key$. Без ограничение на общността, нека $u.key < v.key$ и $u.key < w.key$. Тогава в T може да има много пирамидални инверсии. Пирамидалните инверсии може да са $n-1$ на брой, където n е броят на върховете на T , ако всеки връх от $T[v]$ и от $T[w]$ участва в инверсия с корена u .

Ключовото наблюдение е, че с $O(\lg n)$ размени на елементи може да премахнем всички пирамидални инверсии, тоест да направим T пирамида. Интуитивно е ясно, че трябва да разменим u с този връх измежду v и w , чийто ключ е максимален[†]. Без ограничение на общността, нека $v.key < w.key$. Тогава ще разменим u с w . Със сигурност след размяната w няма да участва в инверсия с нито един връх от $T[v]$, така че след размяната ще има инверсии—ако изобщо има—само в поддървото, чийто корен сега е u . Грубо казано, броят на инверсиите ще намалее наполовина или повече.[‡]

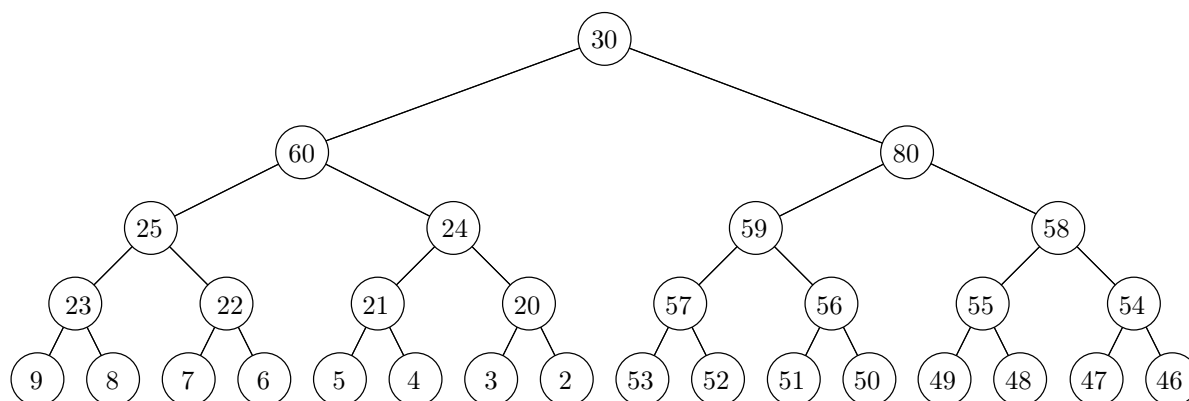
Забележете, че има примери, в които е по-изгодно да разменяме корена не с детето с по-големия ключ, а с детето с по-малкия ключ. Такова дърво е показано на Фигура 5.12. Първо да видим какво ще стане, ако върху това дърво прилагаме размяна на връх с това от двете деца, което има по-голям ключ. Фигура 5.13 показва дървото след първата размяна – елементите 30 и 80 са разменени, а с червено е очертан пътят, който ще “измине” върхът с ключ 30, движейки се надолу, докато не стане листо. На Фигура 5.14 е показано дървото след последната размяна, като е очертан пътят, който е “изминал” върхът с ключ 30. Това дърво е пирамида, получена след четири размени. За дървото от Фигура 5.12, ако разменим корена с детето с по-малък ключ, повече размени в лявото поддърво няма да има; ако разменим след това новия корен с ключ 60 с дясното дете с ключ 80, ще получим пирамидата, показана на Фигура 5.15, със само две размени.

Примерът, който видяхме току-що, се мащабира за всякакъв размер на дървото, така че наистина има случаи, в които разменянето с детето с по-малък ключ води до $\Theta(1)$ размени, а разменянето с детето с по-голям ключ води до $\Theta(\lg n)$ размени. Но ние се интересуваме от сложността в най-лошия случай. В най-лошия случай, разменянето с детето с по-малък

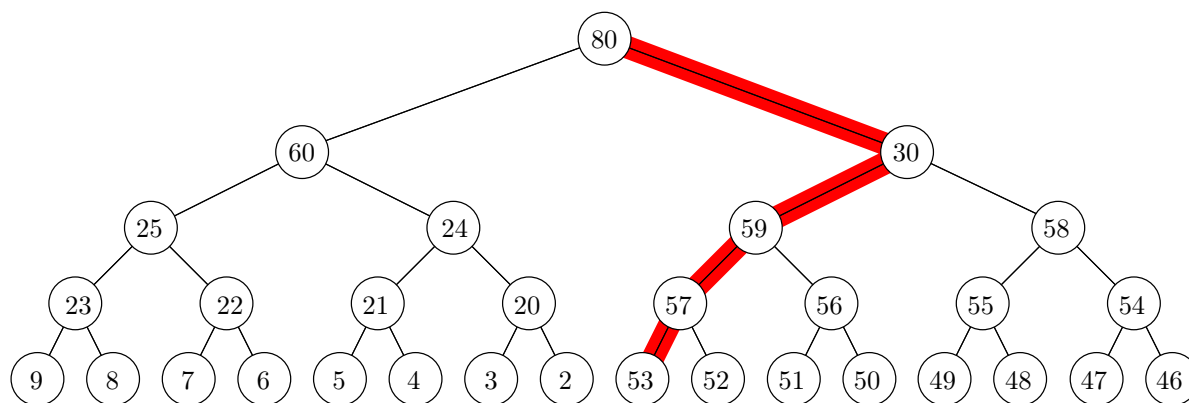
[†] Ако v и w имат еднакви ключове, ще разменим u с кой да е от тях.

[‡] Лесно е да се съобрази, че максималният брой инверсии след размяната, изразен чрез общия брой на върховете, е $\frac{2(n-2)}{3}$, където $n-2$ се дели на 3. Дори броят на инверсиите да намалява с деление на $\frac{3}{2}$ след всяка размяна, броят на размените е логаритмичен.

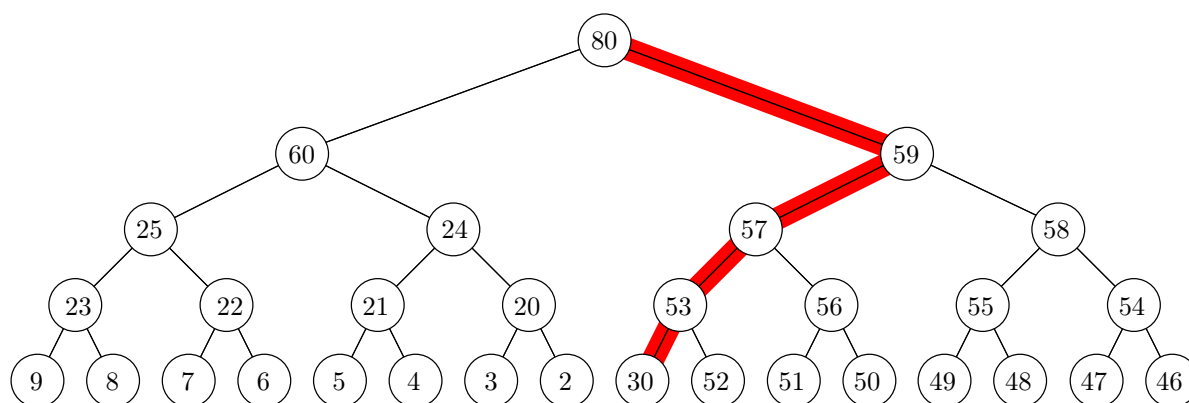
ключ може да доведе до $\Omega(n)$ размени, преди да получим пирамида, докато разменянето с детето с по-голям ключ води до $O(\lg n)$ размени **винаги**. Пример за дърво, върху което разменянето с детето с по-малък ключ води до $\Omega(n)$ размени (примерът се мащабира за безброй много n) е показан на Фигура 5.16. На Фигура 5.17 е показана пирамидата, която се получава от дървото на Фигура 5.16 след $\Omega(n)$ размени.



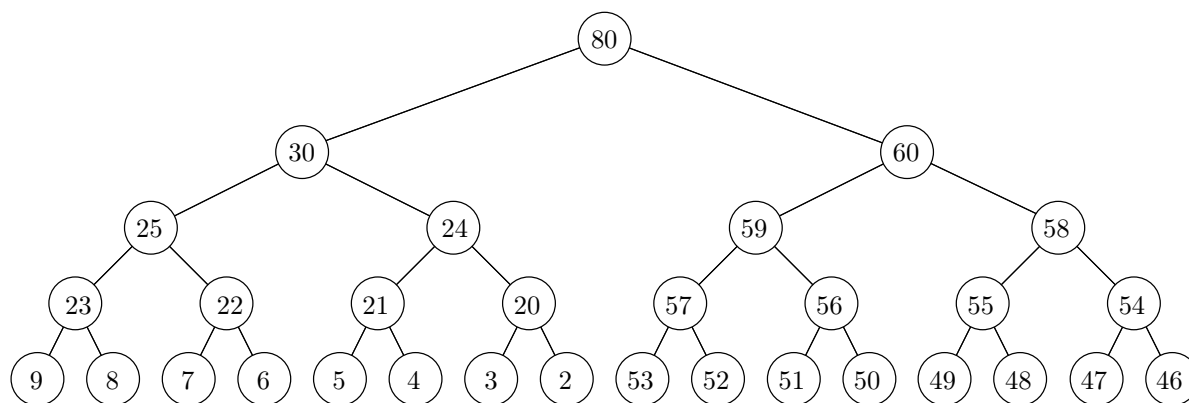
Фигура 5.12: Попълнено дърво с ключове, в което двете деца на корена са корени на поддървета-пирамиди, а ключът на корена е по-малък от ключовете на двете му деца. По-изгодно като минимален брой размени е коренът да бъде разменен с детето с по-малкия ключ.



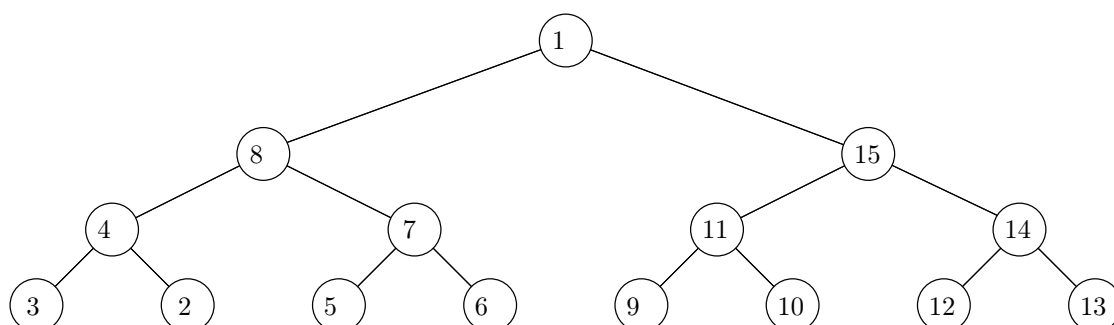
Фигура 5.13: По отношение на дървото от Фигура 5.12, ако разменим корена с неговото дете с по-голям ключ и продължим размените по същия начин—с детето с по-голям ключ—ще направим 4 размени, докато върхът с ключ 30 не стане листо. Тук е показано дървото след първата размяна и е очертан пътът, който трябва да измине върхът с ключ 30. Тази фигура не показва окончателната позиция на върха с ключ 30, а само началото на “пътуването” му.



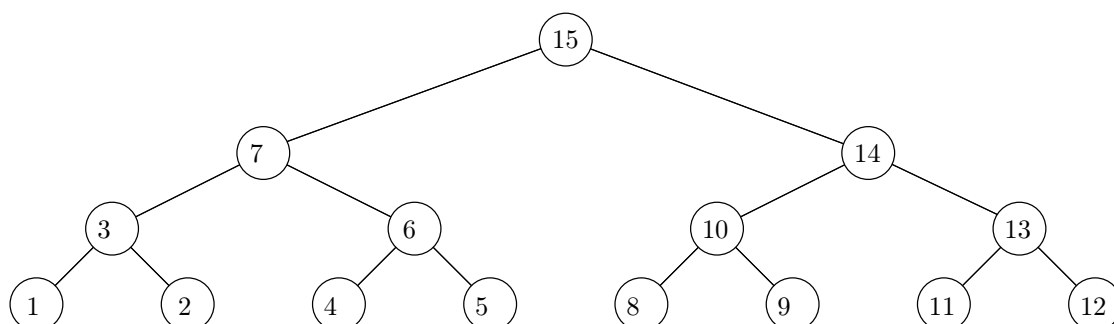
Фигура 5.14: По отношение на дървото от Фигура 5.13 е показано дървото след всички размени, където размените са от вида “разменяме корен с това дете, което има по-голям ключ”. Извършени са 4 размени и дървото е пирамида. Очертан е пътът, който е “изминал” върхът с ключ 30.



Фигура 5.15: По отношение на дървото от Фигура 5.12, ако разменяме връх с корена с детето с по-малък ключ, ще направим само 2 размени и ще получим показаната пирамида.



Фигура 5.16: Пример за дърво, в което, ако разменяме корена с детето с по-малък ключ (и после правим още една размяна на новия корен с детето с по-голям ключ), броят на размените е по-голям от броя на върховете.



Фигура 5.17: Ако в дървото, показано на Фигура 5.16, систематично разменяме първо корен с детето с по-малък ключ, и после новия корен с детето с по-голям ключ, ще получим пирамидата, показана тук. Броят на размените е по-голям от броя на върховете.

От изложението трябва да е станало ясно, че предложената идея—а именно, да разменяме с по-голямото дете—за конструиране на HEAPIFY би имала сложност $O(\lg n)$, ако се реализира грамотно: с $O(\lg n)$ размени на елементи унищожаваме всички пирамидални инверсии, а всяка размяна става в константно време.

5.2.2.2 Неформално въвеждане на HEAPIFY

В изложението досега “пирамидизирахме” цялото дърво – при допускането, че само коренът има какъв да е ключ, а лявото и дясното поддърво са пирамиди. Да разгледаме по-общата възможност: разглеждаме произволен вътрешен връх u в дървото T , такъв че и двете поддървета (или само едно, в екстремния случай, в който дясно дете няма), вкоренени в неговите деца, са пирамиди, а ключът на u е произволен, и целта е да “пирамидизираме” $T[u]$. Освен това, вече мислим за пирамидата не на високо абстрактно ниво като дърво, а като масив; тоест, разглеждаме нейната линеаризация $A[1..n]$.

И така, даден е масив A и индекс i в A . Ако $i > \lfloor \frac{n}{2} \rfloor$, то $A[i]$ отговаря на листо и няма какво да правим, понеже $A[i]$ е тривиална пирамида.

Нека $i \leq \lfloor \frac{n}{2} \rfloor$; с други думи, нека $A[i]$ отговаря на вътрешен връх. БОО, нека $2i + 1 \leq n$. Ключовото допускане е, че $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$ са пирамиди. Без това допускане, алгоритъмът HEAPIFY е **безполезен**. HEAPIFY размества елементите на $A[i]$ така, че $A[i]$ става пирамида, без да променя нищо извън $A[i]$. Това се постига с последователни размени на елемента, който в началото е на позиция i , с това от децата му, което има по-голям ключ, докато или този елемент не стане листо, или и двете му деца (или единственото му дете, ако няма дясно дете) не се окажат по-малки или равни на него.

5.2.2.3 HEAPIFY в итеративен вариант

Алгоритъм 7: Iterative Heapify

```

ITERATIVE HEAPIFY( $A[1..n]$ : масив от цели числа,  $i$ : индекс в  $A$ )
1  (*  $A[\text{LEFT}(i)]$  и  $A[\text{RIGHT}(i)]$ , ако съществуват, са пирамиди *)
2   $j \leftarrow i$ 
3  while  $j \leq \lfloor \frac{n}{2} \rfloor$  do
4       $left \leftarrow \text{LEFT}(j)$ 
5       $right \leftarrow \text{RIGHT}(j)$ 
6      if  $A[left] > A[j]$  // проверка  $left \leq n$  е безсмислена, понеже  $j \leq \lfloor \frac{n}{2} \rfloor$ 
7           $largest \leftarrow left$ 
8      else
9           $largest \leftarrow j$ 
10     if  $right \leq n$  and  $A[right] > A[largest]$ 
11          $largest \leftarrow right$ 
12     if  $largest \neq j$ 
13          $\text{swap}(A[j], A[largest])$ 
14          $j \leftarrow largest$ 
15     else
16         break

```

Допълнение 33: Друг възможен псевдокод на ITERATIVE HEAPIFY

Показаният код на Алгоритъм 7 е взет от [34]. Този код е много компактен, но многогодишните наблюдения на автора на записките са, че на пръв поглед кодът често е неубедителен. Кодът е коректен, разбира се, и Теорема 64 доказва това, но за всеки случай в това допълнение ще бъде показан друг псевдокод, който може да бъде ползван вместо кода на редове 6–16.

Този алтернативен псевдокод е доста по-дълъг и вербозен, но е по-близо до интуицията за това, какво трябва да се прави. А интуицията е тази:

- Ако поне едното от децата на $A[j]$ има по-голям ключ от $A[j]$, итерирането продължава така.
 - ◆ Ако и двете деца са с по-големи ключове, то
 - * ако $A[left] > A[right]$, разменяме $A[j]$ с $A[left]$, присвояваме $left$ на j и продължаваме да итерираме,
 - * в противен случай, разменяме $A[j]$ с $A[right]$, присвояваме $right$ на j и продължаваме да итерираме.
 - ◆ В противен случай,
 - * ако $A[left] > A[j]$, разменяме $A[j]$ с $A[left]$, присвояваме $left$ на j и продължаваме да итерираме
 - * в противен случай, разменяме $A[j]$ с $A[right]$, присвояваме $right$ на j и продължаваме да итерираме.
- В противен случай, итерирането спира.

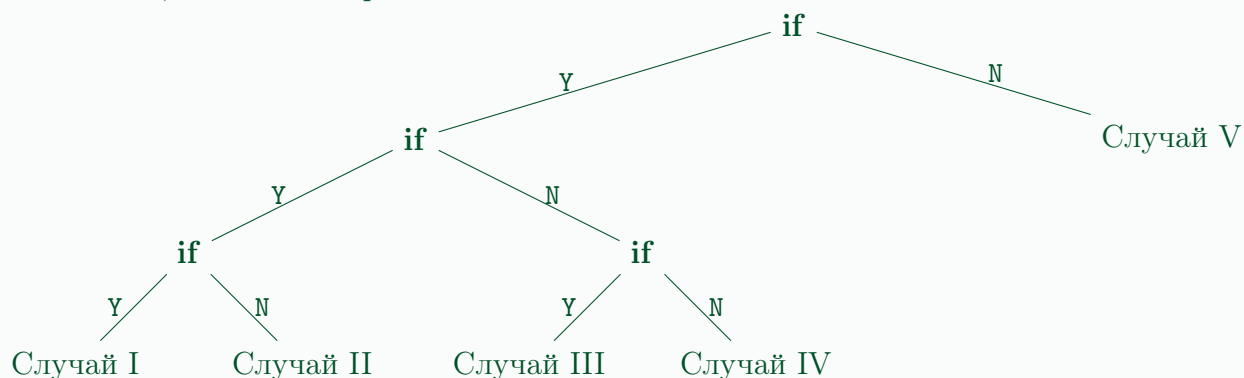
Тези съображения се транслират директно в този код.

```

if A[left] > A[j] or (right ≤ n and A[right] > A[j])
  if A[left] > A[j] and right ≤ n and A[right] > A[j]
    if A[right] > A[left]
      swap(A[right], A[j]), j ← right
    else
      swap(A[left], A[j]), j ← left
  else
    if A[right] > A[j]
      swap(A[right], A[j]), j ← right
    else
      swap(A[left], A[j]), j ← left
else
  break

```

Този код е по-обемист от кода на Cormen, Leiserson и Rivest. Читателите да кажат дали е по-убедителен на първо четене. Заслужава да се отбележи, че в HEAPIFY има точно пет различни възможни състояния и това се вижда от кода тук много по-ясно, отколкото от кода на Cormen, Leiserson и Rivest. Четирите **if**-а в този код дават дърво с пет листа, които са въпросните състояния:



Петте състояния (случая) съответстват точно на петте случая от доказателството на Инвариант 15. Това, че състоянията са точно пет, се вижда и от следната таблица. От най-общи комбинаторни съображения, има точно $2^3 = 8$ възможности неравенствата $A[left] > A[j]$, $A[right] > A[j]$ и $A[right] > A[left]$ да са изпълнени или не, но две от тези комбинации са невъзможни (маркирани с “ $\downarrow$ ”), а други две са неразличими.

A_ℓ vs A_j	A_r vs A_j	A_r vs A_ℓ	Забележка
$A[left] \leq A[j]$	$A[right] \leq A[j]$	$A[right] \leq A[left]$	едно състояние Случай V
$A[left] \leq A[j]$	$A[right] \leq A[j]$	$A[right] > A[left]$	
$A[left] \leq A[j]$	$A[right] > A[j]$	$A[right] \leq A[left]$	$A_r > A_j \geq A_\ell \geq A_r \downarrow$
$A[left] \leq A[j]$	$A[right] > A[j]$	$A[right] > A[left]$	Случай III
$A[left] > A[j]$	$A[right] \leq A[j]$	$A[right] \leq A[left]$	Случай IV
$A[left] > A[j]$	$A[right] \leq A[j]$	$A[right] > A[left]$	$A_r > A_\ell > A_j \geq A_r \downarrow$
$A[left] > A[j]$	$A[right] > A[j]$	$A[right] \leq A[left]$	Случай II
$A[left] > A[j]$	$A[right] > A[j]$	$A[right] > A[left]$	Случай I

Теорема 64: Коректността на ITERATIVE HEAPIFY

При допускането, че $A[1..n]$ и i са такива, че всеки от $A[LEFT(i)]$ и $A[RIGHT(i)]$, ако съществува, е пирамида, ефектът от Алгоритъм 7 е, че $A[i]$ е пирамида при неговото приключване.

Доказателство: Започваме с това, че ако $i > \lfloor \frac{n}{2} \rfloor$, то, от една страна, $A[i]$ е листо, така че $A[i]$ е тривиална пирамида, а от друга страна, алгоритъмът терминира за две стъпки, понеже **while**-цикълът не се изпълнява. Поради това до края на доказателството допусаме, че $i \leq \lfloor \frac{n}{2} \rfloor$. След малко ще стане ясно защо правим такова разбиване на случаи още в началото на доказателството. Сега разглеждаме първото достигане на ред 3.

Случай I $A[j] < A[LEFT(i)] < A[RIGHT(i)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 също е ИСТИНА, така че и присвояването на ред 11 се случва. Когато изпълнението е на ред 12, $largest = right$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[largest]$ биват разменени, а после на ред 14, новата стойност на j е $right$. При второто достигане на ред 3 е вярно, че

- $\langle i, LEFT(i) \rangle$ и $\langle i, RIGHT(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $PARENT(j) \neq j$ и $\forall A[k] \in A[j] : A[PARENT(j)] > A[k]$.

Случай II $A[LEFT(i)] \leq A[j] < A[RIGHT(i)]$ Условието на ред 6 е ЛЪЖА и присвояването на ред 9 се случва. Условието на ред 10 е ИСТИНА, така че и присвояването на ред 11 се случва. Когато изпълнението е на ред 12, $largest = right$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[largest]$ биват разменени, а после на ред 14, новата стойност на j е $right$. При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Случай III $A[j] < A[\text{RIGHT}(i)] < A[\text{LEFT}(i)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 е ЛЪЖА и присвояването на ред 11 не се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{left}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е left . При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Случай IV $A[\text{RIGHT}(i)] \leq A[j] < A[\text{LEFT}(i)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 е ЛЪЖА и присвояването на ред 11 не се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{left}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е left . При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

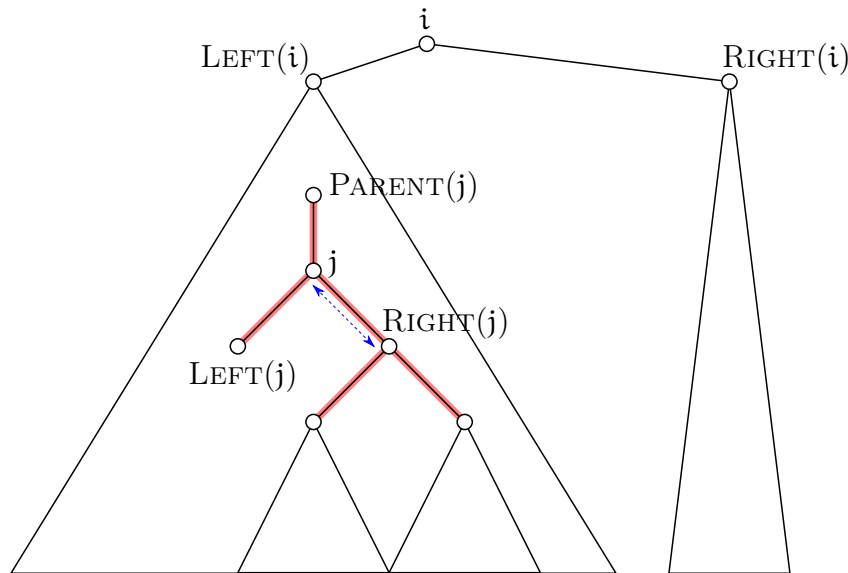
Случай V $A[\text{LEFT}(i)] \leq A[j], A[\text{RIGHT}(i)] \leq A[j]$ Условието на ред 6 е ЛЪЖА и присвояването на ред 9 се случва. Условието на ред 10 е ЛЪЖА и присвояването на ред 11 не се случва. Когато изпълнението е на ред 12, $\text{largest} = j$. Условието на ред 12 е ЛЪЖА и на ред 16 алгоритъмът терминира. При терминирането е вярно, че в $A[i]$ пирамидални инверсии няма, така че $A[i]$ е пирамида.

Заклучаваме, че ако алгоритъмът терминира през първата итерация на **while**-цикълът, $A[i]$ е пирамида, а ако има второ достигане на ред 3, в сила е

- ① единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- ② $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Оттук насетне разглеждаме случая, в който има второто достигане на ред 3.

Тази фигура илюстрира състоянието на нещата за началото на някое изпълнение на **while**-цикъла (редове 3–16), което не е първото. Щом не е първото, $j \neq i$, така че $j \neq \text{PARENT}(j)$:



БОО, нека $j \in A[\text{LEFT}(i)]$. БОО, нека $A[\text{RIGHT}(j)] > A[\text{LEFT}(j)] > A[j]$, така че алгоритъмът разменя $A[j]$ с $A[\text{RIGHT}(j)]$. Ако мислим за пирамидата като за дърво, единствените ребра, които потенциално могат да бъдат засегнати от тази размяна в смисъл, че ключа на поне един техен връх се променя, поради което неравенствата върху ключовете на краищата им **може** да не са изпълнени след размяната, са петте ребра, оцветени в червено. За всички други ребра в $A[i]$, ключовете на краищата им преди и след размяната са едни и същи, поради което неравенствата върху ключовете на краищата им преди и след размяната са едни и същи. С оглед на това, във формалното доказателство надолу показваме, че за аналогичните пет двойки ключове, ако съществуват, неравенствата са изпълнени.

Следното твърдение е инвариант за **while**-цикъла (редове 2–16).

Инвариант 15: Цикълът на ITERATIVE HEAPIFY

При всяко достигане **след първото** на ред 3:

- ❶ единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- ❷ $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

База. Разглеждаме второто достигане на ред 3. Вече изведохме ❶ и ❷, които са същите като ❶ и ❷. ✓

Поддръжка. Допускаме, че ❶ и ❷ са в сила при някое достигане на ред 3, което не е последното; това влече, че ред 16 няма да бъде изпълнен в текущата итерация. И така, $j \leq \lfloor \frac{n}{2} \rfloor$, така че $\text{LEFT}(j) \leq n$, така че $A[\text{LEFT}(j)]$ съществува. Без ограничение на общността ще допуснем, че и $A[\text{RIGHT}(j)]$ съществува, тоест, че и $\text{RIGHT}(j) \leq n$. Следните случаи са взаимно изключващи се и изчерпателни.

Случай I $A[j] < A[\text{LEFT}(i)] < A[\text{RIGHT}(i)] < A[\text{PARENT}(j)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 също е ИСТИНА, така че и присвояването на ред 11 се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{right}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е right . При следващото достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Случай II $A[\text{LEFT}(i)] \leq A[j] < A[\text{RIGHT}(i)] < A[\text{PARENT}(j)]$ Условието на ред 6 е ЛЪЖА и присвояването на ред 9 се случва. Условието на ред 10 е ИСТИНА, така че и присвояването на ред 11 се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{right}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е right . При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Случай III $A[j] < A[\text{RIGHT}(i)] < A[\text{LEFT}(i)] < A[\text{PARENT}(j)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 е ЛЪЖА и присвояването на ред 11 не се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{left}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е left . При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Случай IV $A[\text{RIGHT}(i)] \leq A[j] < A[\text{LEFT}(i)] < A[\text{PARENT}(j)]$ Условието на ред 6 е ИСТИНА и присвояването на ред 7 се случва. Условието на ред 10 е ЛЪЖА и присвояването на ред 11 не се случва. Когато изпълнението е на ред 12, $\text{largest} = \text{left}$. Условието на ред 12 е ИСТИНА и на ред 13, $A[j]$ и $A[\text{largest}]$ биват разменени, а после на ред 14, новата стойност на j е left . При второто достигане на ред 3 е вярно, че

- $\langle i, \text{LEFT}(i) \rangle$ и $\langle i, \text{RIGHT}(i) \rangle$ не са пирамидални инверсии. Единствените възможни директни пирамидални инверсии в $A[i]$ са между $A[j]$ и децата му.
- $\text{PARENT}(j) \neq j$ и $\forall A[k] \in A[j] : A[\text{PARENT}(j)] > A[k]$.

Терминация. Цикълът може да бъде напуснат по два начина: през ред 3, когато $j > \lfloor \frac{n}{2} \rfloor$, и през ред 16. Да разгледаме първата възможност. Тогава и $\text{LEFT}(j)$, и $\text{RIGHT}(j)$ са индекси извън $A[1 \dots n]$. В този случай от ❶ следва, че няма пирамидални инверсии в $A[i]$ изобщо, понеже $A[\text{LEFT}(j)]$ и $A[\text{RIGHT}(j)]$ не съществуват. Следователно, $A[i]$ е пирамида.

Да разгледаме втората възможност, а именно **while**-цикълът да бъде напуснат през ред 16. Очевидно, за да бъде достигнат ред 16, трябва да бъде вярно, че $\text{largest} = j$ на ред 12. А за да е вярно това, и $A[\text{left}] \leq A[j]$, и $A[\text{right}] \leq A[j]$ трябва да са в сила в началото на изпълнението на цикъла, понеже това е единственият начин ред 9 да бъде достигнат и ред 11 да не бъде достигнат. Но ако $A[\text{left}] \leq A[j]$ и $A[\text{right}] \leq A[j]$, съгласно ❶, пирамидални инверсии в $A[i]$ няма, така че $A[i]$ е пирамида. $\square$

Сложността по време на Алгоритъм 7 е $\Theta(\lg i)$. Това е очевидно, понеже j получава стойност i на ред 2 и за всяка следваща итерация, новата стойност на j е largest (ред 14), като

този *largest* е или $\text{LEFT}(j)$, или $\text{RIGHT}(j)$ по отношение на старата стойност на j . С други думи, всяко следващо j е горе-долу два пъти по-голямо от предишното. Забележете, че следваща итерация има само ако старото j не е равно на *largest* (ред 12); тоест, при следващата итерация j има нова стойност.

Допълнение 34: HEAPIFY в рекурсивен вариант

Ето рекурсивна реализация на HEAPIFY.

```

RECURSIVE HEAPIFY( $A[1..n]$ : масив от цели числа,  $i$ : индекс в  $A$ )
1  (*  $A[\text{LEFT}(i)]$  и  $A[\text{RIGHT}(i)]$ , ако съществуват, са пирамиди *)
2   $left \leftarrow \text{LEFT}(i)$ 
3   $right \leftarrow \text{RIGHT}(i)$ 
4  if  $left \leq n$  and  $A[left] > A[i]$ 
5       $largest \leftarrow left$ 
6  else
7       $largest \leftarrow i$ 
8  if  $right \leq n$  and  $A[right] > A[largest]$ 
9       $largest \leftarrow right$ 
10 if  $largest \neq i$ 
11     swap( $A[i], A[largest]$ )
12     RECURSIVE HEAPIFY( $A, largest$ )

```

Теорема 65: Коректността на RECURSIVE HEAPIFY

При допускането, че $A[1..n]$ и i са такива, че всеки от $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$, ако съществува, е пирамида, ефектът от RECURSIVE HEAPIFY е, че $A[i]$ е пирамида при неговото приключване.

Доказателство: По индукция по височината h на $A[i]$.

Преди да направим доказателството, ще подчертаем следното: допусканията за $A[\text{LEFT}(i)]$ и $A[\text{RIGHT}(i)]$ — а именно, че са пирамиди — са нещо напълно различно от индуктивното предположение. Индуктивното предположение е твърдението, което доказваме, формулирано чрез някаква стойност на параметъра, на която сме дали име. Индуктивното предположение не е част от условията на лемата, докато споменатите допускания са част от условията на лемата.

База. $h = 0$. Тогава $A[i]$ съдържа един единствен елемент A . С други думи, това е листо. Тогава и $\text{LEFT}(i)$, и $\text{RIGHT}(i)$ са индекси извън A . Да проследим изпълнението на RECURSIVE HEAPIFY: условието на ред 4 е ЛЪЖА, следователно присвояването на ред 7 се случва. Условието на ред 8 също е ЛЪЖА, така че ред 9 не се изпълнява и изпълнението преминава към ред 10. Условието там е ЛЪЖА и текущото рекурсивно извикване терминира. Очевидно $A[i]$ е пирамида при терминирането. ✓

Индуктивно предположение. Да допуснем, че за всяко $A[j]$ от височина $\leq h - 1$ с корен някой j , такъв че $A[j]$ е в $A[i]$, е изпълнено, че RECURSIVE HEAPIFY(A, j) превръща $A[j]$ чрез размествания в пирамида.

Индуктивна стъпка. Да разгледаме изпълнението на `RECURSIVE HEAPIFY(A, i)`. Без ограничение на общността, да допуснем, че $\text{LEFT}(i) \leq n$ и $\text{RIGHT}(i) \leq n$, така че редове 4 и 8 са съответно

`if A[left] > A[i]`

и

`if A[right] > A[largest]`

Случай I: $A[\text{left}] > A[i]$, $A[\text{right}] > A[i]$ и $A[\text{right}] > A[\text{left}]$. Условието на ред 4 е ИСТИНА и присвояването на ред 5 се случва. Условието на ред 8 също е ИСТИНА, така че и присвояването на ред 9 се случва. Когато изпълнението е на ред 10, *largest* е равно на *right*. Условието на ред 10 е ИСТИНА и на ред 11, $A[i]$ и $A[\text{largest}]$ биват разменени. Рекурсивното повикване на ред 12 се случва, като бившият $A[i]$ сега е на позиция *right*. Съгласно индуктивното предположение, това рекурсивно викане построява пирамида от $A[\text{right}]$. От друга страна, $A[\text{left}]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\text{right}]$, е:

- по-голям от текущия $A[\text{left}]$ по допускане;
- не по-малък от текущия $A[\text{right}]$, по следните причини. Първоначално, $A[\text{right}]$ е пирамида, така че бившият $A[\text{right}]$ е не по-малък от всеки друг елемент от $A[\text{right}]$ в онзи момент (в началото). В края, елементите на $A[\text{right}]$ се състоят от началните елементи без началния $A[\text{right}]$, плюс началния $A[i]$. Тъй като началният $A[\text{right}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\text{right}]$, текущият $A[i]$ не е по-малък от текущия $A[\text{right}]$.

Тогава, $A[i]$ е пирамида.

Случай II: $A[\text{left}] > A[i]$, $A[\text{right}] > A[i]$ и $A[\text{right}] \nless A[\text{left}]$. Условието на ред 4 е ИСТИНА и присвояването на ред 5 се случва. Условието на ред 8 е ЛЪЖА, така че присвояването на ред 9 не се случва и когато изпълнението е на ред 10, *largest* е равно на *left*. Условието на ред 10 е ИСТИНА и на ред 11, $A[i]$ и $A[\text{largest}]$ биват разменени. Рекурсивното извикване на ред 12 се случва, с бившия $A[i]$ сега на позиция *left*. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[\text{left}]$. От друга страна, $A[\text{right}]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\text{left}]$, е:

- не по-малък от текущия $A[\text{right}]$ по допускане
- не по-малък от текущия $A[\text{left}]$, по следните причини. Първоначално, $A[\text{left}]$ е пирамида, така че бившият $A[\text{left}]$ е не по-малък от всеки друг елемент на $A[\text{left}]$ в онзи момент (в началото). В края, елементите на $A[\text{left}]$ се състоят от началните елементи без началния $A[\text{left}]$, плюс началния $A[i]$. Тъй като началният $A[\text{left}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\text{left}]$, текущият $A[i]$ не е по-малък от текущия $A[\text{left}]$.

Тогава, $A[i]$ е пирамида.

Случай III: $A[left] \not\geq A[i]$ и $A[right] > A[i]$. Условието на ред 4 е ЛЪЖА и присвояването на ред 7 се случва. Условието на ред 8 е ИСТИНА, така че присвояването на ред 9 се случва и когато изпълнението е на ред 10, *largest* е равно на *right*. Условието на ред 10 е ИСТИНА и на ред 11, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 12 се случва, с бившия $A[i]$ сега на позиция *right*. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[right]$. От друга страна, $A[left]$ е пирамида, защото не е бил променен по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[right]$, е:

- по-голям от текущия $A[left]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[right]$ по следните причини. Първоначално, $A[right]$ е пирамида, така че бившият $A[right]$ е не по-малък от всеки друг елемент на $A[right]$ в онзи момент (в началото). В края, елементите на $A[right]$ се състоят от началните елементи без началния $A[right]$, плюс началния $A[i]$. Тъй като началният $A[right]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[right]$, текущият $A[i]$ не е по-малък от текущия $A[right]$.

Тогава, $A[i]$ е пирамида.

Случай IV: $A[left] > A[i]$ и $A[right] \not\geq A[i]$. Условието на ред 4 е ИСТИНА и присвояването на ред 5 се случва. Условието на ред 8 е ЛЪЖА, така че присвояването на ред 9 не се случва и когато изпълнението е на ред 10, *largest* е равно на *left*. Условието на ред 10 е ИСТИНА и на ред 11, $A[i]$ и $A[largest]$ биват разменени. Рекурсивното извикване на ред 12 се случва, с бившия $A[i]$ сега на позиция *left*. Съгласно индуктивното предположение, това рекурсивно извикване построява пирамида от $A[left]$. От друга страна, $A[right]$ е пирамида, защото не е бил променен по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[left]$, е:

- по-голям от текущия $A[right]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[left]$ поради следните причини. Първоначално, $A[left]$ е пирамида, така че бившият $A[left]$ е не по-малък от всеки друг елемент от $A[left]$ в онзи момент (в началото). В края, елементите на $A[left]$ се състоят от началните елементи без началния $A[left]$, плюс началния $A[i]$. Тъй като началният $A[left]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[left]$, текущият $A[i]$ не е по-малък от текущия $A[left]$.

Тогава, $A[i]$ е пирамида.

Случай V: $A[left] \not\geq A[i]$ и $A[right] \not\geq A[i]$. Условието на ред 4 е ЛЪЖА и присвояването на ред 7 се случва. Условието на ред 8 също е ЛЪЖА, така че присвояването на ред 9 не се случва и когато изпълнението достигне ред 10, *largest* е равно на *i*. Условието на ред 10 е ЛЪЖА и изпълнението терминира, оставяйки $A[i]$ непроменен. Съгласно допусканията, $A[left]$ и $A[right]$ са пирамиди и $A[left] \not\geq A[i]$ и $A[right] \not\geq A[i]$. Тогава, $A[i]$ е пирамида. $\square$

5.2.2.4 Алгоритъм BUILD HEAP

Идеята на бързия алгоритъм за построяване на пирамида е да сканира масива отдясно наляво, иначе казано, от листата към корена, започвайки от първото не-листо. Листата са тривиални пирамиди.

За всяко не-листо $A[i]$ в указания ред, при допускането, че и двете му деца (или едното му дете, ако друго дете няма) са корени на подпирамиди, ако $A[i]$ е по-голям или равен от двете си деца (или едното си дете), то $A[i]$ е подпирамида. Ако това не е вярно, ще направим $A[i]$ пирамида чрез серия от размени на елементи в нея, разменяйки корен с по-голямото дете, избутвайки малкия елемент, който образува пирамидални инверсии, в посока към листата, докато той или не стане листо, или не се окаже по-голям или равен на децата си (детето си).

Това всъщност означава, че викаме HEAPIFY върху всяко нелисто, отдясно наляво. Функцията HEAPIFY в следния алгоритъм е или Алгоритъм 7, или RECURSIVE HEAPIFY на стр. 340.

Алгоритъм 8: Build Heap

```
BUILD HEAP( $A[1..n]$ : array of integers)
1   for  $i \leftarrow \lfloor \frac{n}{2} \rfloor$  downto 1
2       HEAPIFY( $A, i$ )
```

Теорема 66: Коректността на бързото построяване на пирамида

За всеки вход A , BUILD HEAP построява пирамида от него.

Доказателство: Следното твърдение е инвариант на **for**-цикъла (редове 1–2).

Инвариант 16: Цикълът на BUILD HEAP

Всеки път, когато изпълнението е на ред 1, $A[i+1]$, $A[i+2]$, ..., $A[n]$ са пирамиди.

База. При първото достигане на ред 1, i е равно на $\lfloor \frac{n}{2} \rfloor$. Тогава $A[\lfloor \frac{n}{2} \rfloor + 1]$, $A[\lfloor \frac{n}{2} \rfloor + 2]$, ..., $A[n]$ са точно листата на попълненото дърво, чиято линеаризация е A . Очевидно всяко листо е пирамида. ✓

Поддръжка. Да разгледаме някое достигане на ред 1, което не е последното. И $A[\text{LEFT}(i)]$, и $A[\text{RIGHT}(i)]$ (ако съществува) са пирамиди съгласно индуктивното предположение, понеже $i < \text{LEFT}(i)$ и $i < \text{RIGHT}(i)$. Тогава използваме доказаната коректност на HEAPIFY и заключаваме, че $A[i]$ става пирамида. След това i намалява с единица. Спрямо новата стойност на i , инвариантът е в сила. ✓

Терминация. Когато изпълнението е на ред 2 за последен път, $i = 0$. Тогава $A[0+1]$, $A[0+2]$, ..., $A[n]$ са пирамиди. В частност, $A[1]$ е пирамида. Но $A[1]$ е A . Това доказва теоремата. ✓ □

Теорема 67: Бързото построяване на пирамида става в линейно време

BUILD HEAP работи във време $\Theta(n)$.

Доказателство: Това, че сложността по време е $\Omega(n)$, е очевидно. Ще покажем, че сложността е $O(n)$. Асимптотична горна граница за сложността на BUILD HEAP е сумата по

всички височини от 1 (BUILD HEAP прескача листата) до $\lfloor \lg n \rfloor$ (височината на дървото) от произведението от броя на върховете с дадената височина и самата височина (всеки връх може да се “придвижва” надолу към листата с най-много толкова размени, колкото е височината му поначало). Съгласно Теорема 62, броят на върховете с височина k е $\left\lfloor \frac{n}{2^k} \right\rfloor$. И така, ако $T(n)$ е функцията на сложността, то в най-лошия случай,

$$T(n) = \sum_{k=1}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^k} \right\rfloor \Theta(k) = \sum_{k=1}^{\lfloor \lg n \rfloor} \Theta\left(\frac{n}{2^k} k\right)$$

Да разгледаме $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k$. В сила е

$$\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \leq \sum_{k=1}^{\infty} \frac{n}{2^k} k = n \sum_{k=1}^{\infty} \frac{k}{2^k}$$

Тривиално се доказва, че редът $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е сходящ, примерно с критерия на d'Alembert:

$$\lim_{k \rightarrow \infty} \frac{\frac{k+1}{2^{k+1}}}{\frac{k}{2^k}} = \frac{1}{2} < 1$$

Щом $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е ограничен от константа, то $n \sum_{k=1}^{\infty} \frac{k}{2^k} \asymp n$ и очевидно $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \asymp n$. Тогава $T(n) \asymp n$. $\square$

5.3 Сортиращ алгоритъм HEAPSORT

Следният сортиращ алгоритъм е предложен от Williams [157]. Той е първият от бързите сортиращи алгоритми, които ще разгледаме. “Бърз” наричаме алгоритъм, чиято сложност по време е $O(n \lg n)$.

Има една особеност за функцията HEAPIFY. Масивът, върху който тя работи, **не** е $A[1 \dots n]$, а $A[1 \dots i-1]$. Следователно, за нея край на масива е $A[i-1]$, така че $A[i]$, $A[i+1]$, $\dots$, $A[n]$ са “извън обхвата”.

Алгоритъм 9: Heapsort

```

HEAPSORT( $A[1 \dots n]$ )
1  BUILD HEAP( $A$ )
2  for  $i \leftarrow n$  downto 2
3      swap( $A[1]$ ,  $A[i]$ )
4      HEAPIFY( $A[1 \dots i-1]$ , 1)
```

Теорема 68: Коректността на HEAPSORT

HEAPSORT е сортиращ алгоритъм.

Доказателство: Нека $A_0[1 \dots n]$ означава първоначалния масив. Следното твърдение е инвариант на цикъла за **for**-цикъл (редове 2–4).

Инвариант 17: Цикълът на HEAPSORT

Всеки път, когато изпълнението на HEAPSORT е на ред 2:

- Текущият подмасив $A[i + 1 .. n]$ се състои от $n - i$ на брой най-големи елементи на $A_0[1 .. n]$, в сортиран вид.
- Освен това, текущият $A[1 .. i]$ е пирамида.

База. При първото достигане на ред 2, $i = n$. Подмасивът $A[i + 1 .. n]$ е празен, следователно, в празния смисъл (*vacuously*), той се състои от нула на брой най-големи елементи от $A_0[1 .. n]$, в сортиран вид, така че първата част на инварианта е в сила. $A[1 .. n]$ е пирамида съгласно Теорема 66, приложена към ред 1, така че и втората част на инварианта е в сила. ✓

Поддръжка. Да допуснем, че твърдението е в сила при някакво достигане на ред 2, което не е последното. Съгласно първата част на индуктивното предположение, текущият $A[i + 1 .. n]$ се състои от $n - i$ на брой елементи от A_0 в сортиран вид. Съгласно втората част на индуктивното предположение, $A[1]$ е максимален елемент на $A[1 .. i]$. След размяната на ред 3, $A[i .. n]$ съдържа $n - i + 1$ на брой най-големи елемента на A_0 в сортиран вид. Спрямо новата стойност на i при следващото достигане на ред 2, първата част на инварианта е в сила. ✓

Ще докажем, че втората част на инварианта е в сила. Отново разглеждаме момента на същото достигане на ред 2. Съгласно втората част на инварианта, текущият $A[1 .. i]$ е пирамида. След разместването на ред 3, $A[1 .. i]$ (може да) не е пирамида. Забелязваме обаче, че $A[2]$ и $A[3]$, ако са в $A[1 .. i - 1]$, остават пирамиди, понеже разместването не ги засяга.

Изпълнението отива на ред 4. Помним, че HEAPIFY на ред 4 работи върху $A[1 .. i - 1]$. Тъй като $A[2]$ и $A[3]$, съгласно Теорема 64 или Теорема 65, в зависимост от това дали ползваме, съответно, итеративна или рекурсивна HEAPIFY, $A[1 .. i - 1]$ е пирамида след изпълнението на HEAPIFY. След това управлението отива на ред 2, където i се декрементира. Спрямо новото i , втората част на инварианта е в сила. ✓

Терминация. Да разгледаме момента, в който изпълнението е на ред 2 за последен път. Очевидно, $i = 1$. Да заместим 1 на мястото на i в инварианта. Получаваме “текущият подмасив $A[2 .. n]$ се състои от $n - 1$ на брой най-големи елементи на $A_0[1 .. n]$, в сортиран вид”. Но тогава $A[1]$ е минимален елемент на $A[1 .. n]$. Следователно, текущият подмасив $A[1 .. n]$ се състои от n на брой най-големи елементи на $A_0[1 .. n]$, в сортиран вид. □

Да разгледаме сложността по време на HEAPSORT. Сложността на BUILD HEAP е $\Theta(n)$, което доказахме в Теорема 67. Сложността на **for**-цикъла (редове 2–4) $\Theta(n \lg n)$, защото за всяко i , сложността на HEAPIFY в най-лошия случай е $\Theta(\lg i)$, а ние вече изследвахме сумата $\sum_{i=1}^n \lg i$ (вж. (5.9)) и установихме, че асимптотично ѝ нарастване е $\Theta(n \lg n)$. Следователно, сложността на HEAPSORT е $\Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$.

Сложността по памет на HEAPSORT е $\Theta(1)$, ако HEAPIFY е в итеративния вариант. Рекурсивният вариант RECURSIVE HEAPIFY на стр. 340, ако бъде реализиран директно по псевдокода, ползва в най-лошия случай $\Theta(\lg n)$ работна памет заради рекурсивните викания, които са $\Theta(\lg n)$ на брой в най-лошия случай.

HEAPSORT не е стабилен сортиращ алгоритъм. За да се убедим в това, достатъчно е да разгледаме работата му над вход от еднакви ключове. При дадения псевдокод, BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени еднаквите елементи $A[1]$ и $A[n]$, като този елемент, който бива записан в $A[n]$, ще остане там до края на алгоритъма.

Дори да сложим проверка на ред 3, такава размяна да не се изпълнява, ако $A[1]$ и $A[i]$ са равни, това няма да направи алгоритъма стабилен. Да си представим вход от k ключа 2 и k ключа 1, като двойките са преди единиците. BUILD HEAP няма да промени нищо, след което още първото изпълнение на **for**-цикъла ще размени двойката в $A[1]$ с единицата в $A[n]$, при което двойката, която е била вляво от всички други двойки, ще се окаже вдясно от тях, и ще остане така до края на алгоритъма.

Допълнение 35: HEAPSORT има сложност $\Theta(n \lg n)$ в най-добрия случай

Както отбелязахме на стр. 101, по принцип не разглеждаме сложността по време в най-добрия случай, понеже с тази мярка може лесно да се злоупотребява. Тук обаче ще разгледаме именно сложността по време в най-добрия случай на HEAPSORT, но без да добавяме изкуствени проверки от вида на “ако $A[1] \leq \dots \leq A[n]$, прекрати работата”, които свалят сложността по време в най-добрия случай на $\Theta(n)$. Разглеждаме алгоритъма така, както е описан на стр. 344.

Ще докажем, че HEAPSORT има сложност по време в най-добрия случай $\Theta(n \lg n)$, ако елементите от входа са уникални. Това, че сложността в най-добрия случай е $O(n \lg n)$, е очевидно (тривиално следва от това, че сложността в най-лошия случай е $O(n \lg n)$). Ще покажем, че в най-добрия случай сложността е $\Omega(n \lg n)$. За тази цел е достатъчно да покажем, че за $\Omega(n)$ върха е вярно, че всеки от тях участва в $\Omega(\lg n)$ размени.

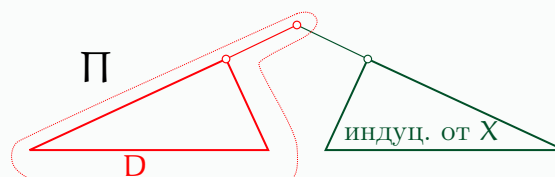
В разсъжденията до края на допълнението използваме нотация с обикновени дроби непрецизно, в смисъл че $\frac{n}{2}$ означава “ $\lfloor \frac{n}{2} \rfloor$ или $\lceil \frac{n}{2} \rceil$ ”, $\frac{n}{4}$ означава $\Theta(\frac{n}{4})$, и така нататък. Нека $A[1..n]$ е входният масив. Неко S е множеството от входните елементи, като $|S| = n$, понеже във входа няма повторения. Нека X и Y са подмножествата на S , такива че $S = X \uplus Y$, $|X| = \frac{n}{2}$, $|Y| = \frac{n}{2}$ и $\forall x \in X \forall y \in Y : x < y$. Може да кажем, че X се състои от *малките* елементи, а Y се състои от *големите* елементи.

Да разгледаме пирамидата Π , построена от входния A с процедурата BUILD HEAP. Тъй като коренът на Π е от Y и за всеки връх $y \in Y$ е вярно, че по уникалния път от y до корена всички върхове са от Y , лесно се вижда, че върховете от Y индуцират едно единствено поддърво D на пирамидата, което съдържа корена. За разлика от върховете от X , които индуцират гора.

Читателят може да се изкуши да си представи Π като дърво, в което D е “отгоре над” дърветата, индуцирани от X . Ако е така, нито едно листо на D не е листо на Π .



От друга страна, възможно е D да е поддървото, вкоренено в лявото дете на корена, плюс самия корен. Ако е така, всяко листо на D се явява и листо на Π .



Във всеки случай е вярно, че най-много $\frac{n}{4}$ елемента от Y са листа на Π , понеже листата на D са $\frac{n}{4}$ на брой. А листата на D са $\frac{n}{4}$, понеже $|Y| = \frac{n}{2}$ и броят на листата на D е $\frac{|Y|}{2}$.

Нека L_i е множеството на върховете в Π с височина i , които са от Y . Тогава L_0 е множеството от листата на Π , които са от Y . Както току-що видяхме, $0 \leq |L_0| \leq \frac{n}{4}$ и тези граници са точни.

Нека h е височината на Π . $\{L_0, \dots, L_h\}$ е разбиване^a на Y , така че, по принципа на разбиването,

$$|L_0| + |L_1| + \dots + |L_h| = \frac{n}{2}$$

Предвид това, че $|L_0| \leq \frac{n}{4}$, имаме

$$|L_1| + |L_2| + \dots + |L_h| \geq \frac{n}{4}$$

Твърдим, че $|L_1| \geq \frac{n}{8}$. За целта ще покажем, че $|L_2| + \dots + |L_h| \leq \frac{n}{8}$. Нека L'_i е множеството на върховете в Π с височина i , за $2 \leq i \leq h$. Тогава $L_i \subseteq L'_i$, за $2 \leq i \leq h$. Тогава $|L_i| \leq |L'_i|$, за $2 \leq i \leq h$. Знаем, че $|L'_2| + \dots + |L'_h| \leq \frac{n}{8}$. Тогава $|L_2| + \dots + |L_h| \leq \frac{n}{8}$.

Покажахме, че $|L_1| \geq \frac{n}{8}$. Читателят ще забележи, че конкретният множител $\frac{1}{8}$ не е важен. Важното е, че $|L_1| = \Omega(n)$.

Припомняме си, че HEAPSORT гради сортирания масив отдясно наляво. В крайния сортиран масив, елементите от X са преди тези от Y , така че HEAPSORT слага елементите от Y по местата им в дясната половина на масива **преди** да започне да слага елементите от X по местата им в лявата половина на масива.

Разглеждаме момента, в който изпълнението е за първи път на ред 2.

- Първо забелязваме, че елементите от L_0 може да се озоват на окончателните си позиции в A с малко местения. Всеки елемент $y \in L_0$ **може** да бъде сложен на окончателното си място само с две размествания.
 - ◆ първо, y “скача” във върха на пирамидата от някаква i -та позиция чрез $\text{swap}(A[1], A[i])$ на ред 3,
 - ◆ после $\text{HEAPIFY}(A, 1)$ не размества нищо, понеже y се оказва достатъчно голям,
 - ◆ после i се декрементира неявно на ред 2
 - ◆ и после $\text{swap}(A[1], A[i])$ на ред 3—спрямо новото i —слага y на окончателното му място в $A[1 \dots n]$.
- Ключовото наблюдение е, че елементите от L_1 , които са $\Omega(n)$ на брой, трябва да бъдат местени многократно преди да се окажат на окончателните си места в A . Всеки елемент от L_1 **трябва** да “пропътува” цялото разстояние до корена, преди да бъдат сложен на окончателното си място в $A[1 \dots n]$ чрез $\text{swap}(A[1], A[i])$ на ред 3.

Нека се убедим в последното. Тъй като $L_1 \subset Y$ и в окончателната наредба елементите на Y са преди елементите на X , ясно е, че елементите от L_1 трябва да бъдат сложени по окончателните си места още докато i е по-голяма или равна на $\frac{n}{2}$.

Нека y е произволен елемент от L_1 . Очевидно y бива сложен на окончателното си място чрез $\text{swap}(A[1], A[i])$ на ред 3. Но, както вече казахме, в този момент $i \geq \frac{n}{2}$. Това влече, че i не е индекс на елемент от L_1 , защото цялото L_1 е вляво от $\frac{n}{2}$ (с други думи, i в този момент е прекалено голям, за да бъде индексът на елемент от L_1). Неизбежният извод е, че y трябва преди това, някак си, да се е придвижил от началната си позиция в Π чак до корена $A[1]$. Това придвижване е станало в резултат на изпълнения на HEAPIFY, която разменя върхове от **съседни** нива. Ерго, y е бил участник в $\Omega(\lg n)$ размени.

Ако последният аргумент не е съвсем ясен, ето илюстрация. Това е състоянието на нещата след BUILD HEAP на ред 1: цялото L_1 е в лявата половина на A , така че и y е някъде там. Да кажем, на позиция $k < \frac{n}{2}$.



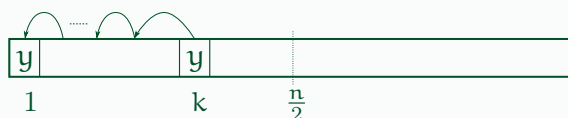
В окончателната наредба y се намира на позиция $j > \frac{n}{2}$, понеже $y \in Y$.



За да се окаже y на позиция $j > \frac{n}{2}$, трябва да е имало момент, в който $i = j$ и е било изпълнено $\text{swap}(A[1], A[i])$. Виждаме, че в този момент y е бил на позиция 1.



А как елементът y , намирайки се в началото на позиция k , се е оказал на позиция 1, при положение, че i е в другата половина на масива? Има само един начин: y е бил местен от поредица swap-ове, стартирайки от позиция k , като тези swap-ове **не са** swap-овете на HEAPSORT на ред 3, а са swap-ове, които HEAPIFY (ред 4) предизвиква, бидейки извикана върху корен, който не е y ; във всяко от тези викания, процедурата HEAPIFY, “избутвайки” въпросния корен надясно, е направила swap, който е преместил y наляво (едно ниво нагоре в масива). В този смисъл y е изминал целия път в дървото от позиция k до позиция 1, правейки $\Theta(\lg n)$ “скока нагоре”.



Щом всеки от $\Omega(n)$ върха участва в $\Omega(\lg n)$ размени, алгоритъмът работи във време $\Omega(n \lg n)$.

^a L_0 може да е празното множество, но принципът на разбиването остава в сила.

5.4 Приоритетни опашки

5.4.1 Абстрактен Тип Данни (АТД)

Приоритетна опашка (на английски, *priority queue*) е вид *абстрактен тип данни* (на английски, *abstract data type*, или ADT). Първо ще изясним понятието абстрактен тип данни (АТД). Според Sedgewick и Wayne [142, стр. 64]:

An abstract data type (ADT) is a data type whose representation is hidden from the client.

...

When using an ADT, we focus on the operations specified in the API and pay no attention to the data representation; when implementing an ADT, we focus on the data, then implement operations on that data.

To specify the behavior of an abstract data type, we use an application programming interface (API), ...

И така, АТД е множество от данни, обикновено от един и същи тип, заедно с множество функции върху тях, което множество от функции се нарича *интерфейс*. “Абстрактен” означава, че конкретната реализация остава скрита, а цялото “общуване” с данните става през интерфейса. Конкретната реализация би могла да бъде променяна, като това остава прозрачно за софтуера, който ползва АТД-то (или поне на теория софтуерът-потребител не би трябвало да “усеща” разликата, ако няма бъгове и интерфейсът на новата реализация е изграден съгласно спецификациите). Конкретната реализация остава напълно скрита зад интерфейса и можем да си представяме АТД-то като черна кутия (*black box*) плюс интерфейс. Прости примери за АТД са *стек* и *опашка*, наричани още съответно LIFO и FIFO [142, стр. 121] (вж. Фигура 5.18 за илюстрация на черна кутия с интерфейс).

Накратко, АТД казва **какво** да се реализира, тоест това е заданието, а конкретният тип данни определя **как** да се реализира това задание. Има далечна аналогия с разликата между изчислителна задача и алгоритъм.

5.4.2 Приоритетна опашка: вид АТД

Приоритетните опашки имат елементи, всеки от които има две стойности: ключ и данни, свързани с този ключ. Ключовете обикновено са цели числа, макар че всеки тип данни, който може да се използва в контекста на сортирането, може да е типът данни на ключовете. Приоритетните опашки, подобно на пирамидите, са максимални и минимални. Тук ще дискутираме само максимални приоритетни опашки, като изложението за минималния тип е напълно аналогично. Казвайки “приоритетна опашка”, имаме предвид максимална приоритетна опашка.

Интерфейсът на приоритетните опашки обикновено се ограничава до следните три функции. Ако приоритетната опашка се казва S , функциите от интерфейса са:

- MAXIMUM(S): връща елемента с максимален ключ;
- EXTRACT-MAX(S): връща елемента с максимален ключ и го премахва от S ;
- INSERT(S, x): вмъква в S елемент x , който е от подходящ за S тип.

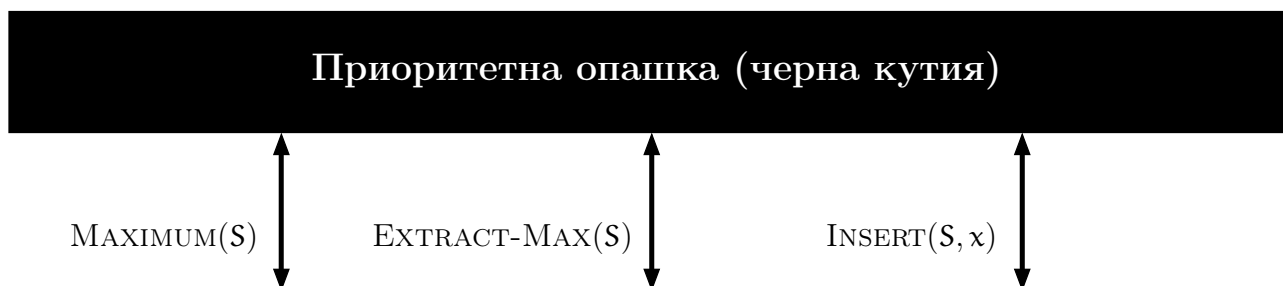
Написаното тук е според учебника на Cormen, Leiserson, и Rivest [34, стр. 162]. Авторите описват и четвърта функция от интерфейса, а именно INCREASE-KEY(S, x, k), която има

смисъл на “увеличи ключа на x на нова стойност k , при допускането, че тя е не по-малка от стария ключ на x ”. Тази функция, която ще дискутираме в Подсекция 5.4.4, не е част от стандартния интерфейс на приоритетните опашки.

Sedgewick и Wayne [142, стр. 309] дават следния изчерпателен списък на интерфейса на реални (които се ползват на практика) приоритетни опашки. Описанието е на езиковите конвенции на Java.

- `MaxPQ()`: create a priority queue
- `MaxPQ(int max)`: create a priority queue of initial capacity `max`
- `MaxPQ(Key[] a)`: create a priority queue from the keys in `a[]`
- `void insert(Key v)`: insert a key into the priority queue
- `Key max()`: return the largest key
- `Key delMax()`: return and remove the largest key
- `boolean isEmpty()`: is the priority queue empty?
- `int size()`: number of keys in the priority queue

За целите на тази лекция, интерфейсът е според учебника на Cormen, Leiserson и Rivest. Фигура 5.18 илюстрира приоритетна опашка като черна кутия плюс интерфейс.



Фигура 5.18: АДД приоритетна опашка като черна кутия. Конкретната имплементация е скрита. Отвън се “виждат” само трите функции на интерфейса.

Приложенията на приоритетните опашки са огромни. Всяка дейност, която иска да бъдат обработвани заявки с различни приоритети, пристигащи асинхронно, като в даден момент може да се обработва само една заявка, може да се моделира с приоритетна опашка. Заявките може да са какви ли не: `print jobs` в операционна система, кораби за разтоварване в пристанище, самолети, кацащи на летище, обслужване на пациенти в спешното отделение в болница, и така нататък. Както вече видяхме, с приоритетни опашки може и да се сортира: HEAPSORT е сортиращ алгоритъм с последователно избиране на максимален елемент, ползващ ефикасно реализирана приоритетна опашка.

5.4.3 Реализация на приоритетни опашки с двоични пирамиди

Както ще видим след малко, двоичните пирамиди са много подходящи за реализация на приоритетни опашки, но в никакъв случай не са единствената възможна реализация. Приоритетна опашка може да бъде реализирана и чрез следните структури данни.

- Сортиран масив. На читателя остава да измисли имплементация на трите функции от интерфейса. Тук само ще споменем, че $\text{MAXIMUM}(S)$ и $\text{EXTRACT-MAX}(S)$ при тази реализация имат сложност $\Theta(1)$, докато $\text{INSERT}(S, x)$ има сложност $\Theta(n)$.
- Несортиран масив. И тук остава на читателя да измисли имплементация на трите функции от интерфейса. При тази реализация, $\text{MAXIMUM}(S)$ и $\text{EXTRACT-MAX}(S)$ имат сложност $\Theta(n)$, докато $\text{INSERT}(S, x)$ има сложност $\Theta(1)$.
- Пирамида на Fibonacci. Това е сложна структура данни, даваща възможност за изключително ефикасна реализация на функциите от интерфейса. Нещо повече, тази структура данни позволява интерфейс от по-голямо множество функции, които включват изтриване на елемент и смесване (merge) на две пирамиди. За съжаление, реализацията на функциите от интерфейса е ефикасна само в асимптотичния смисъл. Големите скрити константи правят пирамидите на Fibonacci неефикасни на практика, освен за наистина много големи данни. За подробности, вижте [34, стр. 506].
- Двоични дървета, реализирани чрез списъци с дървовидна структура (по два указателя към децата за всеки запис). Подробна дискусия на предимствата, недостатъците и вариациите на този подход има в третия том на култовата поредица на Доналд Кнут “Изкуството на Компютърното Програмиране” (The Art of Computer Programming) [94, стр. 149]. От общи съображения е ясно, че този подход изисква значително—по-точно, линейно—повече памет от реализацията с масив. Измежду неговите предимства се откроява възможността за бързо сливане (merge) на приоритетни опашки.

Трите функции на интерфейса може да бъдат реализирани чрез двоични пирамиди по следния начин. Входът S е масив $S[1..n]$, реализиращ пирамида. Приемаме, че $S.\text{size}$ е размерът на пирамидата, тоест, само $S[1..S.\text{size}]$ е пирамида, а $S[1..n]$ е “контейнер”, в който се разполага пирамидата. Допускаме, че винаги $S.\text{size} < n$, така че няма да се притесняваме за препълване на “контейнера” в HEAP-INSERT . Ако пирамидата е празна, както е в самото начало при създаването ѝ, $S.\text{size} = 0$.

$\text{HEAP-MAXIMUM}(S[1..n])$

```
1  (* Допускаме, че  $S.\text{size} \geq 1$  *)
2  return  $S[1]$ 
```

$\text{HEAP-EXTRACT-MAX}(S[1..n])$

```
1  (* Допускаме, че  $S.\text{size} \geq 1$  *)
2   $\text{max} \leftarrow S[1]$ 
3   $S[1] \leftarrow S[S.\text{size}]$ 
4   $S.\text{size}--$ 
5   $\text{HEAPIFY}(S, 1)$ 
6  return  $\text{max}$ 
```

$\text{HEAP-INSERT}(S[1..n], x)$

```
1  (* Допускаме, че  $S.\text{size} < n$  *)
2   $S.\text{size}++$ 
3   $i \leftarrow S.\text{size}$ 
4   $S[i] \leftarrow x$ 
5  while  $i > 1$  and  $S[\text{PARENT}(i)] < S[i]$  do
```



```

6   swap(S[PARENT(i)], S[i])
7   i ← PARENT(i)

```

Коректността на тези функции е абсолютно очевидна предвид изложението в Секция 5.2.

Сложността на HEAP-MAXIMUM е $\Theta(1)$, сложността на HEAP-EXTRACT-MAX е $\Theta(\lg(S.size))$ и сложността на HEAP-INSERT е $\Theta(\lg(S.size))$.

Ако ползваме буквата “n” за големината на масива, а не на масива-контейнер, то имаме право да кажем, че HEAP-EXTRACT-MAX и HEAP-INSERT имат сложност $\Theta(\lg n)$. Имайки предвид огромната, качествена разлика между нарастванията $\Theta(n)$ и $\Theta(\lg n)$, виждаме, че реализацията с двоични пирамиди е **много по-добра** от реализацията със сортирани или несортирани масиви.

Оттук правим важен извод, който е приложим в много по-широк контекст от ефикасното реализиране на приоритетните опашки. Случва се при дизайна на нещо (реализиране на приоритетна опашка в случая), оптималният подход (пирамида в случая) да е **компромис** (на английски, *tradeoff*) между два “екстремистки” подхода. В случая, екстремистките подходи са сортиран масив (пълнен порядък, пълна ригидност, никаква гъвкавост) и несортиран масив (пълнен хаос, никакъв порядък, никаква структура). Има два различни критерия: бързо изваждане на максимален елемент и бързо вмъкване на нов елемент. Единият екстремистки подход дава оптимален резултат $\Theta(1)$ по първия критерий, но лош резултат $\Theta(n)$ по втория критерий. Другият подход дава оптимален резултат по втория критерий, но лош резултат по първия критерий. Компромисният подход на пирамидата е, в някакъв смисъл, междинен вариант и отказ от екстремизма. Пирамидата не е напълно хаотична, защото някакви неравенства върху ключовете трябва да са в сила, но не е напълно ригидна, защото има много пермутации на едни и същи ключове, реализиращи пирамида. Използвайки пирамида, ние постигаме на задоволително време за всички функции. Отказът от екстремизма, направен интелигентно, се оказва печеливш.

5.4.4 Функцията INCREASE-KEY

В приоритетна опашка, чийто интерфейс се състои само от функциите MAXIMUM, EXTRACT-MAX и INSERT, приоритетите са статични: с какъвто приоритет влезе някакъв елемент, с такъв приоритет си остава, докато не излезе от опашката. Това ограничение е неадекватно за някои житейски ситуации. Има задачи, в които приоритетите са динамични, така че трябва да имаме способността да променяме приоритета на елемент, докато той се намира в опашката.

Следната функция, наречена INCREASE-KEY в [34], увеличава приоритета на елемент, който се намира в опашката, ако опашката е реализирана с двоична пирамида. Смисълът на аргументите е ясен: S е пирамидата, i е позицията на елемента на пирамидата, чийто приоритет ще увеличим, а k е стойността на новия приоритет.

```

HEAP-INCREASE-KEY( $S[1..n]$ ,  $i$ ,  $k$ )
1  if  $k < S[i]$ 
2      error “опит да бъде намален ключът”
3   $S[i] \leftarrow k$ 
4  while  $i > 1$  and  $S[PARENT(i)] < S[i]$  do
5      swap( $S[PARENT(i)]$ ,  $S[i]$ )
6       $i \leftarrow PARENT(i)$ 

```

Очевидно, тази имплементация не може да е част от интерфейса на АТД, защото в нея е заложено, че имплементацията е именно чрез масив. В случая параметърът i е индексът на елемента, чийто ключ ще нараства. Ако имплементацията на приоритетната опашка е чрез някакви свързани списъци, този индекс няма смисъл.

Правилното викане на функцията би трябвало да е $\text{INCREASE-KEY}(S, x, k)$, където x е елементът от опашката, чийто приоритет искаме да увеличим. Но да си представим какви биха били последиците от това да има аргумент x , който е стойността на елемент от опашката, а не позицията му. С други думи, x да е старата стойност на ключа, която трябва да стане k . Тогава този елемент първо трябва да бъде намерен в опашката. А намирането на елемент по ключ е операция с линейна сложност в пирамидите—примерно, най-малкият ключ може да е във всяко листо—дори ако допуснем уникални ключове.

Ние искаме INCREASE-KEY да има логаритмична сложност по време, също както останалите функции от интерфейса. Поради това се налага да “счупим” изискването за абстрактен тип данни и да заложим, че приоритетната опашка е реализирана с пирамида и че знаем индекса в пирамидата на елемента, чийто ключ искаме да увеличим.

5.4.5 Задачи, които се решават ефикасно с приоритетни опашки

Решенията на Задача 50 и Задача 51 ползват ∞ за “пазач” в следния смисъл.

Определение 61: Пазач (sentinel) ∞

Ако ∞ се използва с роля като тази на ред 3 в алгоритъма НА k ОТ МЕСТАТА СИ, казваме, че ∞ е *пазач* (на английски, *sentinel*, по терминологията на [34]). Използването му не е задължително – всеки код, ползващ пазач, може да се препише в еквивалентен код без пазач, но с пазачи кодът може да е по-лесен за четене и верификация.

Задача 50 е Problem 6.5-9 от [34, стр. 166].

Задача 50: Бързо сливане на k масива

Дадени сортирани масиви $A[1 \dots n_1], \dots, A[1 \dots n_k]$. Предложете алгоритъм със сложност $O(n \lg k)$, който слива масивите в един сортиран масив, където $n = n_1 + \dots + n_k$.

Решение: Наивното решение е алгоритъм, който създава изходния масив $B[1 \dots n]$ и създава $k + 1$ на брой индекси $i_1, \dots, i_k, j$, които първоначално са единици, след което в n итерации прави това: намира минимума $A[i_\ell]$ на $A_1[i_1], \dots, A_k[i_k]$, записва този елемент в $B[j]$, след което инкрементира i_ℓ и j , и продължава с итерациите.

Ако k е константа, този алгоритъм е оптимален като сложност по време: намирането на минимума на $A_1[i_1], \dots, A_k[i_k]$ става за време $\Theta(1)$, така че сложността по време е $\Theta(n)$. Прочее, алгоритъмът MERGE (Алгоритъм 12) от Секция 6.3 прави точно това: слива два сортирани масива в един сортиран масив, в линейно време.

Ако обаче k не е константа, описаният подход води до алгоритъм със сложност $\Theta(nk)$, което не е допустимо по условие. Има по-бърз начин да бъдат слети входните масиви, ако k не е константа.

СЛИВАНЕ НА СОРТИРАНИ МАСИВИ($A_1[1 \dots n_1], \dots, A_k[1 \dots n_k]$: сортирани числени масиви)

- 1 $n \leftarrow n_1 + \dots + n_k$
- 2 Увеличаваме всеки от $A_1[1 \dots n_1], \dots, A_k[1 \dots n_k]$ с по една клетка
- 3 Във всяка от тези нови клетки записваме ∞

```

4  Построяваме масив  $B[1..n]$ 
5  Построяваме индекси  $i[1], \dots, i[k]$ 
6  for  $j \leftarrow 1$  to  $k$ 
7       $i[j] \leftarrow 1$ 
8  Построяваме  $\min$  двоична пирамида  $S$  от  $\langle A_1[1], 1 \rangle, \dots, \langle A_k[1], k \rangle$ 
9  (* Ключовете в  $S$  са първите елементи на наредените двойки *)
10 for  $j \leftarrow 1$  to  $n$ 
11      $x \leftarrow \text{HEAP-EXTRACT-MIN}(S)$ 
12      $B[j] \leftarrow x.\text{key}$ 
13      $\text{theindex} \leftarrow x.\text{index}$ 
14      $i[\text{theindex}]++$ 
15      $\text{INSERT}(S, A_{\text{theindex}}[i[\text{theindex}]])$ 
16 return  $B$ 

```

Добавените в краищата на $A_1, \dots, A_k$ елементи ∞ играят ролята на пазачи – алгоритъмът е по-прост с тях, а асимптотичната сложност не се влошава.

// да се добави коректност и сложност!

□

Задача 51: Бързо сортиране, когато числата са близо до крайните си места

Даден е масив $A[1..n]$ от числа, които са две по две различни. Известно е, че за някакво k , всеки елемент на A е на разстояние не по-голямо от k от позицията, на която би бил, ако A бъде сортиран.

Предложете колкото е възможно по-ефикасен като сложност по време, в асимптотичния смисъл, алгоритъм, който сортира A . Докажете формално и прецизно коректността на предложения от Вас алгоритъм. Изследвайте сложността му по време.

Пояснение: за всеки два елемента на масива, *разстоянието* между тях е $|i - j|$, където i и j са техните позиции в масива.

Решение: Следният алгоритъм решава задачата.

На k ОТ МЕСТАТА СИ($A[1..n]$: числа; k : цяло положително число)

```

1  (* Всеки елемент на  $A$  е на разстояние  $\leq k$  от позицията си в сортирания масив *)
2  Увеличаваме  $A$  с  $k + 1$  нови клетки  $A[n + 1], \dots, A[n + k + 1]$ 
3  На всяка нова клетка на  $A$  присвояваме  $\infty$ 
4  Построяваме от  $A[1..k + 1]$  приор. опашка  $S$  чрез пирамида  $\min$  тип  $S[1..k + 1]$ 
5  Построяваме масив  $C[1..n]$ 
6  for  $i \leftarrow 1$  to  $n$ 
7       $C[i] \leftarrow \text{EXTRACT-MIN}(S)$ 
8       $\text{INSERT}(S, A[k + i + 1])$ 
9  return  $C$ 

```

Добавените към A елементи ∞ играят ролята на пазачи – алгоритъмът е по-прост с тях, а асимптотичната сложност не се влошава.

Очевидно $0 \leq k \leq n - 1$.

Инвариант за цикъла е следното съждение.

Инвариант 18: Цикълът на НА k ОТ МЕСТАТА СИ

При всяко достигане на ред 6,

- ❶ $C[1 \dots i - 1]$ съдържа $i - 1$ на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите на входа, които са били в S и са били извадени от S ,
- ❷ S е пирамида от тип $\min$,
- ❸ i -ият по големина елемент на входа е минималният елемент в S ,
- ❹ елементите $A[1], A[2], \dots, A[k + i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S .

База. Базата е $i = 1$. ❶ е вярно в празния смисъл, понеже $i - 1$ е нула. ❷ е вярно заради действието, което се върши на ред 4. ❸ е вярно, понеже най-малкият елемент на входа се намира във входния $A[1 \dots k + 1]$, а ние построяваме пирамидата на ред 4 именно от елементите на входния $A[1 \dots k + 1]$. ❹ е вярно, понеже става дума за $A[1], A[2], \dots, A[k + 1]$ от входа: на ред 4 точно тези елементи влизат в S , а в този момент извадени елементи от пирамидата няма.

Да допуснем, че е инвариантът е в сила за някое достигане на ред 6, което не е последното. Нека това е момента t от изпълнението, а t' е моментът на следващото достигане на ред 6. Ще докажем, че инвариантът е в сила в момента t' . Инвариантът е конюнкция от четири съждения и ще ги докажем поотделно.

1. Ще докажем ❶. По допускане, в момента t е вярно, че $C[1 \dots i - 1]$ съдържа $i - 1$ на брой минимални елементи на входния A , в сортиран вид и това са точно извадените от S елементи до момента, а i -ият най-малък елемент на входа е в S . На ред 7, функцията EXTRACT-MIN премахва същия този елемент от пирамидата и го слага в C на позиция i . Вече е вярно, че $C[1 \dots i]$ съдържа i на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите, извадени от S .

Функцията INSERT на ред 8 не променя C .

При следващото достигане на ред 6, i се инкрементира. Спрямо новото i отново е вярно, че $C[1 \dots i - 1]$ съдържа $i - 1$ на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите, извадени от S .

2. Ще докажем ❷. Ако S е пирамида в момента t , тя остава пирамида след изпълнението на EXTRACT-MIN(S) на ред 7 и INSERT($S, A[k + i + 1]$) на ред 8, защото EXTRACT-MAX и INSERT запазват “пирамидалността”, както знаем от лекции.
3. Ще докажем ❸. Очевидно е, че $(i + 1)$ -ият по големина елемент на входния масив не може да се намира на позиция, по-голяма от $k + i + 1$, във входа, заради ограничението всеки елемент във входа да е на разстояние $\leq k$ от мястото си в сортирания масив. Но тогава $(i + 1)$ -вият по големина елемент е или един от елементите $A[1], A[2], \dots, A[k + i]$, за които говори ❹, или е $A[k + i + 1]$. Съгласно ❹, $A[1], A[2], \dots, A[k + i]$ от входа са точно елементите, които или са били в S и извадени от S преди момента t , или са в S в момента t . Съгласно ❶, най-малките $i - 1$ на брой елементи вече не се намират в S в момента t . Заключаваме, че $(i + 1)$ -ият по големина елемент или се намира в S , или е $A[k + i + 1]$. След изпълнението на INSERT($S, A[k + i + 1]$) на ред 8, $(i + 1)$ -ият по

големина елемент се намира в S . След инкрементирането на i на следващото достигане на ред 6, спрямо новата стойност на i е вярно, че i -ият оп големина елемент се намира в S ; това е в момента t' .

Вече видяхме, че в момента t' , $i - 1$ на брой минимални елементи на входния масив са били в S , но извадени от S . Заключаваме, че i -ият по големина елемент не просто е в S , а е най-малкият елемент в S .

4. Ще докажем 4. Това е елементарно: по допускане, елементите $A[1], A[2], \dots, A[k + i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S , в момента t . След изпълнението на $\text{INSERT}(S, A[k + i + 1])$ на ред 8, $(i + 1)$ -ият по големина елемент се намира в S . Имаме право да кажем, че вече елементите $A[1], A[2], \dots, A[k + i], A[k + i + 1]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S .

След инкрементирането на i при следващото достигане на ред 6, спрямо новата стойност на i е вярно, че елементите $A[1], A[2], \dots, A[k + i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S ; това е в момента t' .

Доказахме инварианта. При последното достигане на ред 6, $i = n + 1$. От част 1 на инварианта имаме “ $C[1..n]$ съдържа n на брой минимални елементи на входния A , в сортиран вид”. На ред 9 алгоритъмът връща C , така че той всъщност връща входа в сортиран вид.

Сега да изследваме сложността по време. Действието на ред 2, което се състои в увеличаване на A с $k + 1$ елемента и присвояване на стойности на тези нови елементи, се извършва във време $O(n + k + 1)$. Построяването на приоритетната опашка на ред 4 става във време $\Theta(k + 1)$, ако използваме бързия алгоритъм BUILDHEAP . Построяването на C на ред 5 става във време $O(n)$. Цикълът се “извърта” n пъти, а всяко негово изпълнение става във време $\Theta(\lg(k + 1)) + \Theta(\lg(k + 1))$ в най-лошия случай, понеже, както знаем, и EXTRACT-MIN , и INSERT работят във време, пропорционално на логаритъм от големината на пирамидата, в най-лошия случай. Тогава сложността по време, в асимптотичния смисъл, е

$$O(n + k + 1) + \Theta(k + 1) + O(n) + n(\Theta(\lg(k + 1)) + \Theta(\lg(k + 1)))$$

Предвид факта, че $k \leq n$, можем да оппростим този израз до $\Theta(n \lg(k + 1))$. Причината да не ползваме просто “ $\lg k$ ” е, че k може да бъде нула. $\square$

Допълнение 36: Таблица на Young: аналог на двоична пирамида

Определение 62: Таблица на Young

Таблица на Young^a е правоъгълна матрица, чиито елементи са от $\mathbb{R} \cup \{\infty\}$ и която е сортирана по редове и колони. Таблица на Young е

- *непълна*, ако съдържа поне едно число и поне една ∞ ,
- *пълна*, ако съдържа само числа,
- *празна*, ако съдържа само ∞ .

^aНа английски е *Young tableau*, в множествено число *Young tableaux*.

Ако таблицата на Young е $A[1..m, 1..n]$, да бъде сортирана по редове и колони означава следното:

$$\forall i \in \{1, \dots, m\} \forall j \in \{1, \dots, n-1\} : A[i, j] \leq A[i, j+1]$$

$$\forall i \in \{1, \dots, m-1\} \forall j \in \{1, \dots, n\} : A[i, j] \leq A[i+1, j]$$

Очевидно е, че ако в някои ред има елементи ∞ , то те са накрая, и ако в някоя колона има елементи ∞ , то те са най-долу.

Ето пример за непълна таблица на Young 4×5 :

1	20	25	25	26
1	22	25	30	40
15	30	40	45	∞
16	40	50	∞	∞

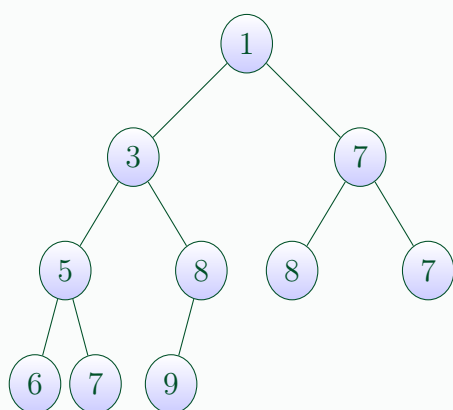
Мултимножеството от числата на таблица на Young е мултимножеството от тези негови елементи, които не са ∞ ; елементът ∞ не е число. В показания пример, мултимножеството от числата е

$$\{1, 1, 15, 16, 20, 22, 25, 25, 25, 26, 30, 30, 40, 40, 40, 45, 50\}_M$$

Често срещаната дефиниция на “Young tableau” е привидно различна: според нея, Young tableau се състои от редове с **не непременно еднакви дължини** от клетки, подредени вляво, като дължините на редовете са в ненарастващ ред отгоре надолу. Всяка клетка съдържа число, като тези числа в клетките са сортирани по редове отляво надясно и по колони отгоре надолу. Двете дефиниции са практически еквивалентни. Ако в тази дефиниция допълним редовете с ∞ , така че да станат с еднакви дължини, ще получим таблица на Young съгласно [Определение 62](#), а ако третираме елементите ∞ в [Определение 62](#) като празни клетки, ще получим тази дефиниция.

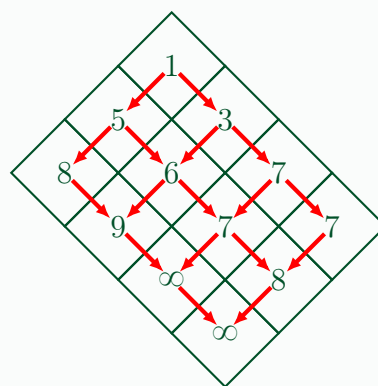
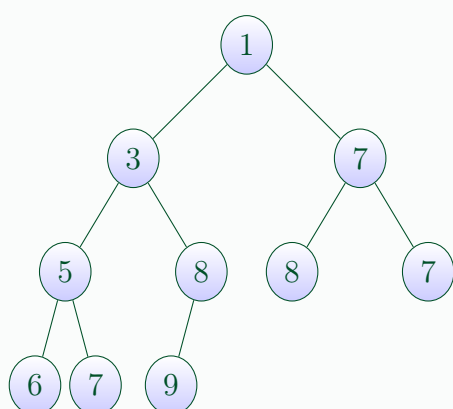
Таблиците на Young може да се използват за реализиране на сравнително ефикасен АТД приоритетна опашка и за конструиране на сравнително ефикасен сортирац алгоритъм, приличащ на HEAPSORT; тоест, сортиране по схемата на SELECTION SORT. Ефикасността е далече по-лоша от тази на реализациите с двоична пирамида, поради което този материал е изложен в допълнение, но все пак може да представлява концептуален интерес.

Ето каква е аналогията между двоичните пирамиди от тип min и таблиците на Young. Отново: сега разглеждаме **min** пирамиди, а не max пирамиди като в изложението за HEAPSORT и приоритетните опашки. Нека мултимножеството от ключовете е $\{1, 3, 5, 6, 7, 7, 7, 8, 8, 9\}_M$. Нека разгледаме следните min пирамида и таблица на Young върху тези ключове. Стрелките в таблицата отговарят на сортирането.



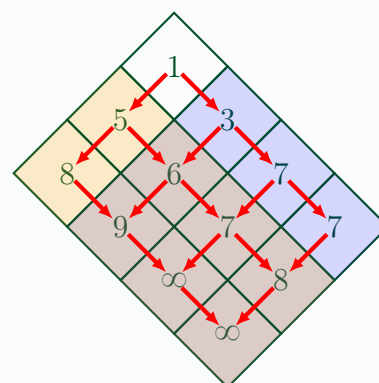
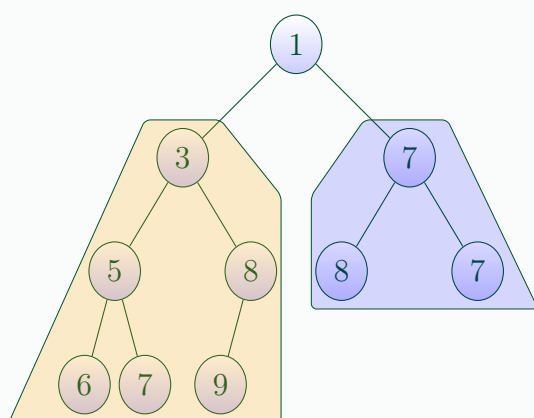
1	→	3	→	7	→	7
↓		↓		↓		↓
5	→	6	→	7	→	8
↓		↓		↓		↓
8	→	9	→	∞	→	∞

Да си представим отново пирамидата и таблицата, но сега таблицата е завъртяна на 45° . По този начин се вижда аналогията с пирамидата.



И така, може да мислим за таблица на Young $A[m, n]$ като за даг с един източник $A[1, 1]$ и един сифон $A[m, n]$, в който върховете имат ключове, които са числа или ∞ . Минималният ключ е в $A[1, 1]$ и по всеки ориентиран път от източника към сифона, ключовете не намаляват.

В някакъв смисъл, източникът $A[1, 1]$ в таблицата е “корен”, а двете му деца се явяват корени на подтаблицы. Но има една важна разлика между поддърветата, вкоренени в децата на корена на пирамидата, и въпросните подтаблицы: в пирамидата, тези поддървета нямат общи върхове, докато в таблицата, тези подтаблицы имат общи елементи.



Въпреки посочената фундаментална разлика, има и известна прилика между таблица

на Young $m \times n$ и двоична пирамида тип min с mn елемента. Представяме таблицата—също като пирамидата—с **едномерен** масив $B[1, \dots, mn]$. В масива, представящ таблицата на Young, се иска

1. подмасивите $B[1 .. n]$, $B[n + 1 .. 2n]$, $\dots$, $B[mn - n + 1 .. mn]$ да са сортирани и
2. тези редици:

$$\begin{aligned} &\langle B[1], B[n + 1], \dots, B[mn - n + 1] \rangle \\ &\langle B[2], B[n + 2], \dots, B[mn - n + 2] \rangle \\ &\dots \\ &\langle B[n], B[2n], \dots, B[mn] \rangle \end{aligned}$$

да са сортирани.

А в масива, представящ min пирамидата, се иска сортиране в ненамаляващ ред на редиците от елементи, отговарящи на пътищата от корена (първия елемент) до листата. Следните четири задачи са булети от Problem 6.3 от [34, стр. 167–168].

Задача 52

Предложете ефикасен алгоритъм, който връща минимума на $m \times n$ таблица на Young и го премахва от таблицата. Това ще рече, че минимумът се премахва от мултимножеството на числата, но получената таблица също е $m \times n$ таблица на Young. Очевидно получената таблица има един елемент ∞ в повече. Анализирайте алгоритъма.

Решение: Нека входната таблица е A . За удобство допускаме, че A всъщност е с $m + 1$ реда и $n + 1$ колони, но елементите в най-долния ред и в най-дясната колона са само ∞ ; тоест, sentinel values. Причината е, че ако най-долният ред и най-дясната колона съдържат sentinel values, кодът на алгоритъма е по-кратък, тъй като винаги има елемент долу и елемент вдясно спрямо текущата позиция.

Очевидно е, че минимумът е в $A[1, 1]$. Съхраняваме $A[1, 1]$ в някаква променлива, записваме ∞ в $A[1, 1]$ и възстановяваме сортирането по редове и колони по начин, далечно подобен на MIN-HEAPIFY при двоичните пирамиди. Въвеждаме *текущата позиция*, която в началото е $[1, 1]$ и след това итерираме така: сравняваме елемента на текущата позиция с неговите “деца”, тоест, елементите долу и вдясно. Ако и двата се окажат по-големи от елемента на текущата позиция, итерирането спира, в противен случай разменяме елемента на текущата позиция с не по-големия от тези “деца” и съответно обновяване стойността на текущата позиция.

За удобство приемаме, че таблицата не е празна.

YOUNG-TABLEAU-EXTRACT-MIN($A[1 .. m + 1, 1 .. n + 1]$: не-празна таблица на Young)

- 1 (* последният ред и най-дясната колона съдържат само ∞ *)
- 2 $tmp \leftarrow A[1, 1]$
- 3 $A[1, 1] \leftarrow \infty$
- 4 $\langle j, k \rangle \leftarrow \langle 1, 1 \rangle$
- 5 **while** $A[j, k] > A[j, k + 1]$ **or** $A[j, k] > A[j + 1, k]$ **do**


```

6      if A[j, k] > A[j, k + 1] and A[j, k] > A[j + 1, k]
7          if A[j, k + 1] < A[j + 1, k]
8              swap(A[j, k], A[j, k + 1])
9              k++
10         else
11             swap(A[j, k], A[j + 1, k])
12             j++
13     else
14         if A[j, k] > A[j + 1, k]
15             swap(A[j, k], A[j + 1, k])
16             j++
17     else
18         swap(A[j, k], A[j, k + 1])
19         k++
20 return tmp

```

Удобно е да дефинираме понятие, аналогично на “min пирамидална инверсия” (Определение 59). В контекста на таблици на Young дефинираме *Young инверсия*. Нека $A[1 \dots p, 1 \dots q]$ е масив с елементи числа или ∞ . “Young инверсия” в A е наредена двойка от наредени двойки $\langle j, k \rangle, \langle j', k' \rangle$, такава че $1 \leq j \leq j' \leq p$ и $1 \leq k \leq k' \leq q$ и $\langle j, k \rangle \neq \langle j', k' \rangle$ и $A[j, k] > A[j', k']$. A е таблица на Young тстк в A няма Young инверсии.

Инвариант за **while** цикъла на ред 4 е: при всяко достигане на ред 4, всички Young инверсии в $A[1 \dots m + 1, 1 \dots n + 1]$ се намират в подмасива $A[j \dots m + 1, k \dots n + 1]$.

Сложността на алгоритъма е $\Theta(m + n)$ в най-лошия случай, понеже **while**-цикълът се изпълнява най-много $\Theta(m + n)$ пъти, понеже индексите j и k само могат да нарастват и наредената двойка $\langle j, k \rangle$ може да приема по-малко от $m + n$ различни стойности. $\square$

Задача 53

Предложете алгоритъм, който добавя число ℓ към $m \times n$ таблица на Young, която не е пълна. Да са добави ново число към таблица на Young, която има поне един елемент ∞ и чието мултимножество от числа е $\{s_1, \dots, s_q\}_M$, означава таблицата да се преобразува в $m \times n$ таблица на Young, чието мултимножество от числа е $\{s_1, \dots, s_q\}_M \cup \{\ell\}_M$ (което влече, че има един елемент ∞ по-малко). Анализирайте алгоритъма.

Решение: И тук решението е алгоритъм, доста подобен на аналогичния алгоритъм за добавяне на елемент в двоична min пирамида. Намираме първия ред, в който има елемент ∞ (щом масивът е непълен, такъв ред има), намираме позицията на най-левия ∞ в този ред, записваме новия елемент ℓ на тази позиция, след което разменяме ℓ с елемента измежду елементите горе и вляво, който е не по-малък от другия, докато е необходимо. Все пак има съществена разлика с пирамидите – там има само един елемент, с който да разменяме новия и това е родителят, докато при таблиците на Young има два предшественика в общия случай – елементът вляво и елементът отгоре.

YOUNG-TABLEAU-INSERT($A[1 \dots m, 1 \dots n]$: празна или непълна табл. на Young, число ℓ)

```

1  j ← минималният номер на ред, в който има  $\infty$ 

```

```

2  k ← минималният номер на колона, такава че A[j, k] = ∞
3  A[j, k] ← ℓ
4  while (j > 1 and A[j - 1, k] > A[j, k]) or (k > 1 and A[j, k - 1] > A[j, k]) do
5      if (j > 1 and A[j - 1, k] > A[j, k]) and (k > 1 and A[j, k - 1] > A[j, k])
6          if A[j - 1, k] > A[j, k - 1]
7              swap(A[j, k], A[j - 1, k])
8              j--
9          else
10             swap(A[j, k], A[j, k - 1])
11             k--
12     else
13         if j > 1 and A[j - 1, k] > A[j, k]
14             swap(A[j, k], A[j - 1, k])
15             j--
16         else
17             swap(A[j, k], A[j, k - 1])
18             k--

```

Инвариант за **while** цикъла на ред 4 е: при всяко достигане на ред 4, всички Young инверсии в $A[1 \dots m, 1 \dots n]$ се намират в подмасива $A[1 \dots j, 1 \dots k]$.

Сложността по време в най-лошия случай е $\Theta(m + n)$, което показваме по начин, напълно аналогичен на анализа на сложността по време от Задача 52. $\square$

Задача 54

Без да използвате каквито и да е сортиращи алгоритми като процедури, покажете как да генерираме сортираната редица от числата на таблица на Young с размери $n \times n$ във време $O(n^3)$, използвайки само алгоритми от Задача 52 и Задача 53. Обосновете коректността на идеята и асимптотичната горна граница върху сложността.

Решение: Алгоритъмът е нещо като HEAPSORT, само че не върху двоична пирамида, а върху таблица на Young. Няма изискване за константна сложност по памет, така че може да си позволим да ползваме допълнителен масив $D[1, \dots, n^2]$, в който да разположим сортираните елементи-числа на входния $A[1 \dots n, 1 \dots n]$.

За удобство допускаме, че таблицата на Young е пълна.

YOUNG-TABLEAU-SORT($A[1 \dots n, 1 \dots n]$: пълна таблица на Young)

```

1  създай масив D[1 .. n2]
2  for i ← 1 to n2
3      D[i] ← YOUNG-TABLEAU-EXTRACT-MIN(A)
4  return D

```

Инвариант за **for** цикъла е: при всяко достигане на ред 3, подмасивът $D[1, \dots, i - 1]$ се състои от $i - 1$ на брой най-малки елементи A в сортиран вид. Освен това, A е таблица на Young.

Сложността по време е $O(n^3)$. Цикълът се изпълнява $\Theta(n^2)$ пъти, а всяко викане на YOUNG-TABLEAU-EXTRACT-MIN има сложност по време $O(n)$, откъдето имаме асимптотична горна граница $O(n^3)$ за целия алгоритъм. $\square$

Задача 55

Предложете алгоритъм със сложност $O(m+n)$, който по дадена $m \times n$ таблица на Young A и число key връща или 1, ако числото key се среща в таблицата, или 0, в противен случай. Анализирайте накратко алгоритъма.

Решение: Тук ще опишем алгоритъма съвсем неформално за правоъгълни матрици; дали са пълни или не, няма значение. Имаме текуща клетка, която в началото е $A[m, 1]$. Сравняваме съдържанието на текущата клетка с key : ако тя е равна на key , връщаме 1, ако тя е по-малка от key , правим клетката вдясно текуща, а ако тя е по-голяма от key , правим клетката отгоре текуща. Ако има опит за излизане извън границите на масива, връщаме 0. Коректността е очевидна, а сложността по време е $O(m+n)$, защото най-много $m+n-1$ клетки може да бъдат текущата клетка и за всяка текуща клетка, алгоритъмът върше само $O(1)$ работа.

Същият алгоритъм, но за квадратни пълни таблици на Young, е описан подробно в Подсекция 13.4.9. Псевдокодът е на стр. 820. $\square$

Наблюдение 42

Решения на Задача 52 и Задача 53 ни дават право да твърдим, че АДТ приоритетна опашка може да се реализира с таблици на Young. Ако таблицата има N елемента, сложността по време е такава:

1. EXTRACT-MIN е със сложност $\Theta(\sqrt{N})$,
2. INSERT е със сложност $\Theta(\sqrt{N})$.

Освен това, от решението на Задача 55 следва, че можем да добавим към интерфейса нова функция $FIND(A, key)$, която връща или 1, ако key се намира в опашката, или 0, в противен случай. Сложността е $\Theta(\sqrt{N})$. Когато разглеждахме приоритетни опашки, реализирани с двоични пирамиди, дори не помислихме за функция, търсеща ключ, понеже сложността ѝ би била линейна в размера на пирамидата.

Освен това, от решението на Задача 54 следва, че можем да конструираме сортиращ алгоритъм, далечно приличащ на HEAPSORT, със сложност $\Theta(N\sqrt{N})$.

Според автора на записките, идеята на Problem 6.3 от [34, стр. 167–168] е да стане ясно, че има и други начини за подредба на данни-ключове—освен двоични пирамиди—които позволяват горе-долу ефикасни имплементации на функциите от интерфейса на АДТ приоритетна опашка. За съжаление, таблиците на Young имат един недостатък, който не се посочва в [34]. По принцип, таблиците на Young са правоъгълни, но функциите EXTRACT-MIN, INSERT и FIND работят ефисасно, ако таблиците са квадратни или близки до квадратни. Ако таблицата на Young има само един ред, което не е невъзможно, тя се изражда в сортиран масив, а да се имплементира приоритетна опашка със сортиран масив е лоша идея, както се убедихме в Подсекция 5.4.3. Нека читателят

се убеди сам/сама, че ако построим таблица на Young, използвайки YOUNG-TABLEAU-INSERT на стр. 361, започвайки с празна таблица и след това слагайки в нея ключовете един след друг чрез викания на YOUNG-TABLEAU-INSERT, таблицата ще има само един ред.

Лекция 6

Алгоритмична схема Разделяй-и-Владей: същност и примери за не-сортиращи алгоритми по нея. Примери за сортиращи алгоритми по нея: MERGESORT и QUICKSORT.

Резюме: Въвеждаме понятието алгоритмична схема и в частност алгоритмичната схема Разделяй-и-Владей. Разглеждаме два примера за бързи алгоритми, конструирани по схемата **Разделяй-и-Владей**: за намиране на най-близка двойка числа в двумерен вариант и за намиране на k -то по големина число в несортиран масив. Разглеждаме два бързи сортиращи алгоритъма по схемата Разделяй-и-Владей: MERGESORT и QUICKSORT. Изследваме тяхната коректност, сложност по време и сложност по памет, както и средната сложност по време на QUICKSORT.

6.1 Разделяй-и-Владей

Алгоритмична схема е шаблон, по който са конструирани много алгоритми за много задачи. В този курс ще разгледаме три алгоритмични схеми: **Разделяй-и-Владей**, **Алчна схема** и **Динамично Програмиране**. Известни са и други схеми, например **Пълно Изчерпване** (на английски **Exhaustive Search**) и **Търсене с Връщане** (на английски **Backtracking**), които в тези записки се споменават бегло на стр. 192 и в Допълнение 76.

В тази лекция ще разгледаме схемата **Разделяй-и-Владей**. При нея алгоритъмът се състои от три *фази*:

- **Разделяй** – ако входът е достатъчно голям, то той се разделя[†] на части и се преминава към следващата стъпка **Владей**. В противен случай, тоест, ако входът е с размер, по-малък от някаква предварително зададена големина, задачата върху този вход решава другояче и се преминава към стъпка **Комбинирай**.
- **Владей** – пускаме алгоритъма върху всяка от по-малките части и получаваме резултатите от всяко пускане.

[†]Терминът “разбива” не е подходящ, защото общоприетата дефиниция на “разбива” означава “на части, които не се припокриват”. При алгоритмите, изградени по схемата **Разделяй-и-Владей**, не е задължително частите на входа да не се припокриват, просто всяка трябва да е по-малка от целия вход.

- **Комбинирай** – използваме тези резултати, за да генерираме желания изход за първоначалния вход.

Да повторим: първата фаза (на разделянето) става само докато размерът на входа не стане достатъчно малък, защото няма как да разделяме дискретен вход на по-малки части неограничено.

Допълнение 37: За произхода на термина “Разделяй-и-Владей”.

Терминът “Разделяй-и-Владей” (на английски, *Divide and Conquer*) идва от латинската фраза *Divide et impera*. Според наложилия се фолклор, тази максима е била ръководен принцип на древния Рим: и кралството, и републиката, и империята. Рим е побеждавал силните си противници като първо е създавал вътрешни разделения в тях, фрагментирайки ги по този начин, и след това е побеждавал всеки от получените фрагменти последователно. Легендарната битка между братята Хорации и братята Куриации е спечелена именно с прилагане на разделяй-и-владей.

Както предстои да видим, за доста изчислителни задачи подходът разделяй-и-владей работи чудесно, давайки алгоритми, които са по-бързи от наивните алгоритми (вижте неформалната дефиниция на стр. 92). Трябва да се подчертае обаче, че разделяй-и-владей съвсем не е универсален подход! По правило, за практически всички трудни изчислителни задачи, схемата **Разделяй-и-Владей** не е приложима[†]. Примерно, не е известен ефикасен алгоритъм за намиране на Хамилтонов цикъл (Задача 2), който декомпозира графа на подграфи, търси Хамилтонови цикли в тях и от резултатите някак си “сглобява” Хамилтонов цикъл за целия граф. Прочее, по отношение на този пример, задачата върху хипотетичните подграфи би била друга: да се намерят не Хамилтонови цикли (в подграфите), а някакви колекции от независими пътища, обединението на които съдържа всеки връх точно веднъж, от които пътища да бъде “сглобен” Хамилтонов цикъл за целия граф.

6.2 Примери за ефикасни Разделяй-и-Владей алгоритми

Ще разгледаме два примера за задачи, които се решават ефикасно с алгоритми, конструирани по схемата Разделяй-и-Владей.

6.2.1 НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D

Задачата е обобщение на НАЙ-БЛИЗКИ ЕЛЕМЕНТИ на стр. 281. Сега са дадени не числа, а двойки от числа, и се търси най-близка двойка. Може да мислим за задачата като за геометрична задача, където двойките са точки в Евклидовата равнина, но може да мислим и за по-общ контекст, в който точките са елементи на някакво фазово пространство.

Има качествена разлика между 1D и 2D вариантите на задачата: за 1D вариантът има ефикасен алгоритъм със сортиране и пребитане, но този подход не се обобщава за повече измерения.

Оригиналната статия за тази задача е от 1976 година и съдържа ефикасен алгоритъм както за двумерния случай, така и за k -мерния случай, ако k е константа [17].

[†]Както ще видим в Лекция 12, алгоритмичната схема **Динамично Програмиране** също е свързана с рекурсия, но при онези алгоритми акцентът е върху ефикасното пресмятане, което означава умело използване на вече получени резултати на общи подподзадачи.

Разстоянието между две двойки числа (a, b) и (c, d) най-често се дефинира като Евклидовото разстояние, което още се нарича L_2 разстоянието[†]

$$\sqrt{(a - c)^2 + (b - d)^2}$$

но може да бъде Манхатънското разстояние, което още се нарича L_1 разстоянието

$$|a - c| + |b - d|$$

или L_∞ разстоянието

$$\max\{|a - c|, |b - d|\}$$

Как точно се дефинира разстоянието не е особено важно, въпреки че някои детайли от алгоритъма, който ще разгледаме, зависят от това. Най-важното е разстоянието да може да се пресмята във време $\Theta(1)$.

Изч. Задача 23: Най-близки елементи 2D, олекотеният вариант

екземпляр: Множество от точки $P = \{p_1, p_2, \dots, p_n\}$, където $p_i = (a_i, b_i)$, където $a_i, b_i \in \mathbb{Q}$, за $1 \leq i \leq n$; $n \geq 2$

решение: $\min\{\text{dist}(p_i, p_j) : 1 \leq i < j \leq n\}$

Изч. Задача 24: Най-близки елементи 2D, тежкият вариант

екземпляр: Множество от точки $P = \{p_1, p_2, \dots, p_n\}$, където $p_i = (a_i, b_i)$, където $a_i, b_i \in \mathbb{Q}$, за $1 \leq i \leq n$; $n \geq 2$

решение: Двойка точки (p_i, p_j) , такива че $i \neq j$ и $\text{dist}(p_i, p_j)$ е минимална.

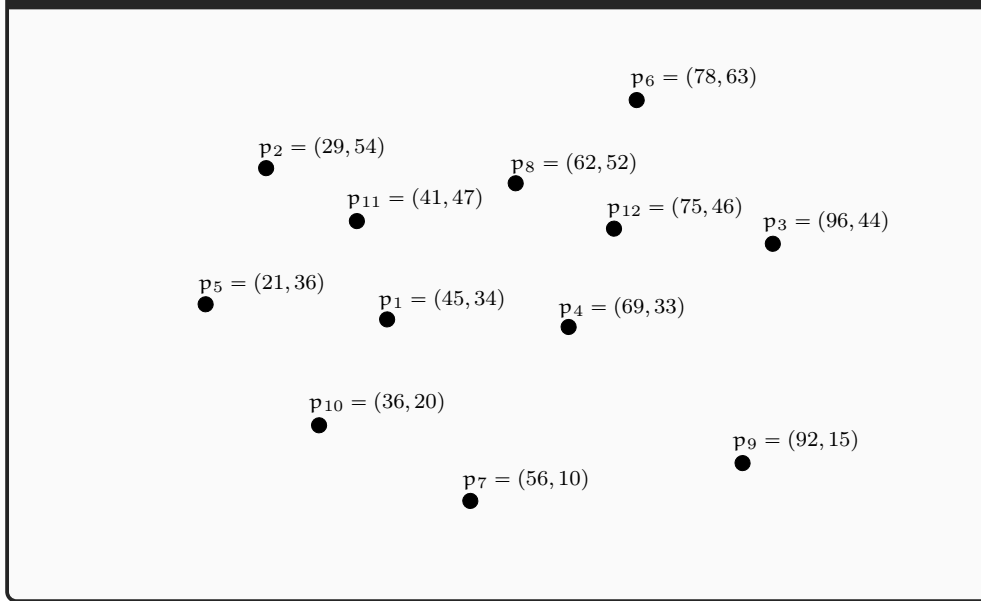
Числата a_i и b_i са произволни рационални числа. Може две точки да имат една и съща първа координата или една и съща втора координата. Може и двете им координати да съвпадат, което означава, че това е една и съща точка в равнината. Може дори всички двойки-точки да съвпадат. Това не е непременно безсмислица. За целите на това изложение смятаме, че всяка точка се идентифицира с координатите си, но в някакъв практически контекст “точките” може да са много по-сложни данни и двете координати да са само два от многото атрибути на една точка.

Наивният алгоритъм разглежда всички ненаредени двойки точки и оттам намира най-близка двойка. Сложността при този подход е $\Theta(n^2)$, което е прекалено бавно. Ние ще построим алгоритъм със сложност $\Theta(n \lg n)$ за олекотения вариант на задачата, изграден по схемата Разделяй-и-Владей. Изложението тук повтаря в основни линии изложението в [34, стр. 1039].

Ще направим пример с точките, показани на Фигура 6.1.

[†] L_m разстоянието между (a, b) и (c, d) е $(|a - c|^m + |b - d|^m)^{\frac{1}{m}}$.

Фигура 6.1 : Точките, които ще ползваме за пример.



За удобство ще въведем следната нотация. Нека е дадена точка $p \in P$. Тогава “ $p.x$ ” означава нейната първа координата, а “ $p.y$ ” означава нейната втора координата. Например, на Фигура 6.1, $p_{11}.x = 41$, $p_8.y = 52$ и така нататък.

Предварителна обработка (preprocessing). Сортираме P веднъж по първата координата и записваме резултата в масив $A[1..n]$, и после още веднъж по втората координата и записваме резултата в масив $B[1..n]$. Подчертаваме две неща.

1. A и B са масиви от точки, а не масиви от числа. В нашия пример:

$$A = [p_5, p_2, p_{10}, p_{11}, p_1, p_7, p_8, p_4, p_{12}, p_6, p_9, p_3]$$

$$B = [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6]$$

2. Сортирането **не** е част от Разделяй-и-Владей! Ако беше част от него, щяхме да сортираме на всяко ниво на рекурсията. Ние сортираме **само веднъж**: в предварителната обработка. По време на същинското изпълнение на рекурсивния алгоритъм не сортираме, а ползваме началното сортиране.

Това е съществено за ефикасността на алгоритъма. Ако сортираме на всяко ниво на рекурсията (в която делим входа на две равни части и правим по едно рекурсивно викане върху всяка от тях), сложността по време ще се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n \lg n)$$

Причината да има нехомогенна част $\Omega(n \lg n)$ е, че сортирането изисква време поне $n \lg n$, в асимптотичния смисъл (вижте Подсекция 13.2.2). Но рекурентното уравнение $S(n) = 2S\left(\frac{n}{2}\right) + \Theta(n \lg n)$ има решение $S(n) = \Theta(n \lg^2 n)$ (вижте Теорема 42 в Допълнение 24). Тогава уравнението $T(n) = 2T\left(\frac{n}{2}\right) + \Omega(n \lg n)$ има решение $T(n) = \Omega(n \lg^2 n)$. Ерго, ако сортираме на всяко ниво на рекурсията, няма как да “слезем” под $\Theta(n \lg^2 n)$, така че $\Theta(n \lg n)$ би била непостижима сложност за алгоритъма.

Продължаваме изложението за предварителната обработка. Заради ефикасността на алгоритъма **има смисъл всеки p_i от B да съдържа и индекса на p_i в A** . Заради това,

елементите на B всъщност са наредени двойки от вида (p_i, j) , където $p_i \in P$, а j е позицията на p_i в A . Да си припомним, че в нашия пример според досегашните дефиниции имаме:

$$\begin{aligned} A &= [p_5, p_2, p_{10}, p_{11}, p_1, p_7, p_8, p_4, p_{12}, p_6, p_9, p_3] \\ B &= [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6] \end{aligned}$$

Сега казваме, че всъщност елементите на B са наредени двойки от точка и индекс. В нашия пример, искаме след предварителната обработка B да изглежда така:

$$\begin{aligned} B &= [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), \\ &\quad (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)] \end{aligned} \quad (6.1)$$

Смисълът от това ще стане ясен нататък.

Допълнение 38: Как да изчисляваме индексите на елементите на B

Индексите на елементите на B трябва да бъдат изчислени ефикасно. Не е ефикасна следната идея: след създаването на A и B като сортирани масиви от точки, за всеки елемент на B да търсим къде се намира той (същата точка) в A . Това би означавало предварителната обработка да има квадратична сложност, което би означавало на свой ред целият алгоритъм да има сложност $\Omega(n^2)$.

Ефикасен е следният начин за пресмятане на тези индекси. Поначало всеки от тези индекси е индексът на дадената точка в P , ако мислим за P като за масив^a. В нашия пример, да кажем, че P е

$$P = [p_4, p_2, p_3, p_{12}, p_1, p_8, p_7, p_5, p_{10}, p_{11}, p_6, p_9]$$

Едно уточнение. “Истинските” елементи на P са наредени двойки от координати. Тези означения: “ p_4 ”, “ p_2 ” и така нататък са съкратени записи за наше удобство. “Истинският” P е (Фигура 6.1):

$$\begin{aligned} P &= [(69, 33), (29, 54), (96, 44), (75, 46), (45, 34), (62, 52), \\ &\quad (56, 10), (21, 36), (36, 20), (41, 47), (78, 63), (92, 15)] \end{aligned}$$

Тогава B непосредствено след сортирането си е

$$B = [(p_7, 7), (p_9, 12), (p_{10}, 9), (p_4, 1), (p_1, 5), (p_5, 8), (p_3, 3), (p_{12}, 4), (p_{11}, 10), (p_8, 6), (p_2, 2), (p_6, 11)]$$

като всъщност B е

$$\begin{aligned} B &= [((56, 10), 7), ((92, 15), 12), ((36, 20), 9), ((69, 33), 1), ((45, 34), 5), ((21, 36), 8), \\ &\quad ((96, 44), 3), ((75, 46), 4), ((41, 47), 10), ((62, 52), 6), ((29, 54), 2), ((78, 63), 11)] \end{aligned}$$

Нека елементите на P всъщност са наредени двойки (p_i, k) , като p_i е точка, а k е индексът на точката в сортирания масив A . Поначало тези индекси са недефинирани; тоест, P поначало е

$$\begin{aligned} P &= [((69, 33), ?), ((29, 54), ?), ((96, 44), ?), ((75, 46), ?), ((45, 34), ?), ((62, 52), ?), \\ &\quad ((56, 10), ?), ((21, 36), ?), ((36, 20), ?), ((41, 47), ?), ((78, 63), ?), ((92, 15), ?)] \end{aligned}$$

Нека елементите на A всъщност са наредени двойки (p_i, s) , като p_i е точка, а s е индексът на точката p_i в P . След сортирането си A става:

$$A = [(p_5, 8), (p_2, 2), (p_{10}, 9), (p_{11}, 10), (p_1, 5), (p_7, 7), \\ (p_8, 6), (p_4, 1), (p_{12}, 4), (p_6, 11), (p_9, 12), (p_3, 3)]$$

което всъщност е

$$A = [((21, 36), 8), ((29, 54), 2), ((36, 20), 9), ((41, 47), 10), ((45, 34), 5), ((56, 10), 7), \\ ((62, 52), 6), ((69, 33), 1), ((75, 46), 4), ((78, 63), 11), ((92, 15), 12), ((96, 44), 3)]$$

С едно сканиране на този A отляво надясно запълваме липсващите индекси в P :

$$P = [((69, 33), 8), ((29, 54), 2), ((96, 44), 12), ((75, 46), 9), ((45, 34), 5), ((62, 52), 7), \\ ((56, 10), 6), ((21, 36), 1), ((36, 20), 3), ((41, 47), 4), ((78, 63), 10), ((92, 15), 11)]$$

Да си припомним, че в момента B е

$$B = [((56, 10), 7), ((92, 15), 12), ((36, 20), 9), ((69, 33), 1), ((45, 34), 5), ((21, 36), 8), \\ ((96, 44), 3), ((75, 46), 4), ((41, 47), 10), ((62, 52), 6), ((29, 54), 2), ((78, 63), 11)]$$

С едно сканиране на B отляво надясно променяме индексите на елементите му по следния начин. Нека $B[i]$ съдържа индекс j . Тогава новият индекс на $B[i]$ е индексът, който се съдържа в $P[j]$. Например, $B[1]$ има индекс 7. Тоест, $j = 7$. $P[7]$ съдържа индекс 6. $B[1]$ става $((56, 10), 6)$. Аналогично, $B[2]$ става $((92, 15), 11)$. И така нататък. Наистина, B става

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), \\ (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)]$$

точно както искахме според (6.1). Очевидно е, че генерирането на тези индекси по описания начин увеличава само с $\Theta(n)$ сложността по време на предварителната обработка. Следователно, сложността по време на предварителната обработка остава $\Theta(n \lg n)$.

^aДори множеството P да не е реализирано чрез масив, всеки негов елемент-точка има адрес в паметта, който адрес е известен.

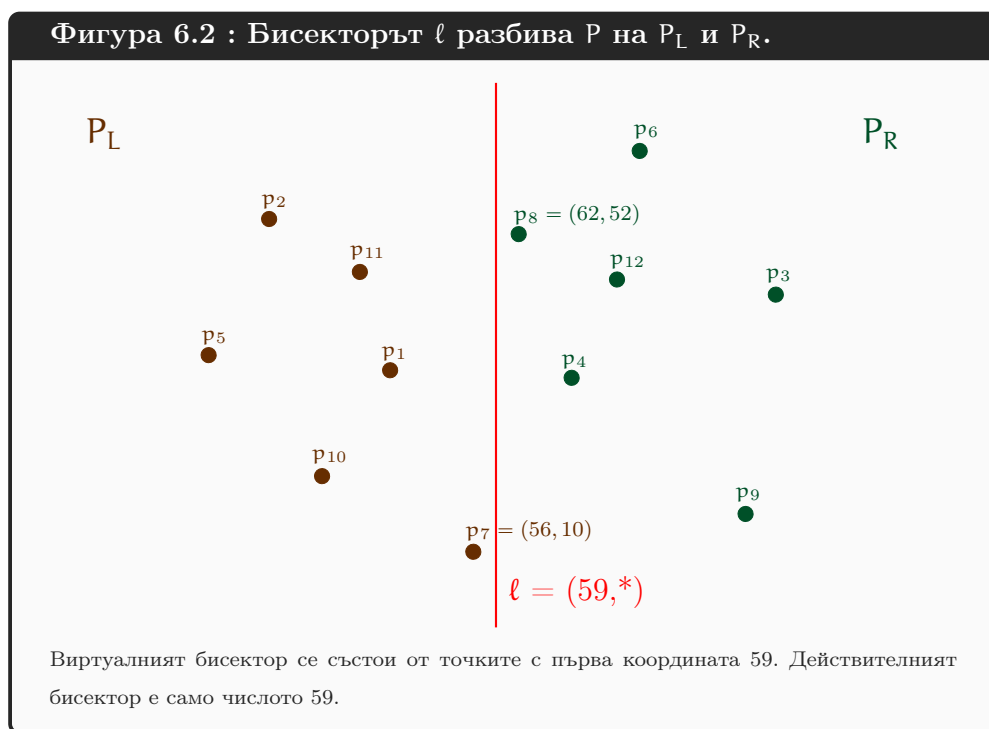
След извършването на предварителната обработка започва същинският рекурсивен алгоритъм. Базата е $n \in \{2, 3\}$. Ако $n \in \{2, 3\}$, то решаваме задачата с брутална сила, опитвайки всички $\binom{n}{2}$ ненаредени двойки от числа. Заслужава си да отбележим, че при $n = 1$ задачата е безсмислена (поради което в дефиницията на задачата има изискване $n \geq 2$), така че базата не трябва да е $n = 1$.

Да допуснем, че $n > 3$.

Фаза 1: разделяй. Разбиваме P на две подмножества с максимално близки мощности. Това действие ще наричаме *бисекция*. Разбиването става спрямо някаква права ℓ , като едното подмножество се състои от точките, които са в едната полуравнина спрямо ℓ , а другото подмножество, от точките в другата полуравнина. Правата ℓ се казва *бисектор*. Удачно е

ℓ да е успоредна на някоя от координатните оси. Избираме да е успоредна на ординатата[†]. Подмножеството вляво от ℓ наричаме P_L , а това вдясно от ℓ наричаме P_R . Алгоритмично, бисекцията е елементарна. Ние вече имаме масива $A[1..n]$ от точките, сортирани по първа координата. Нека подмасивът $A[1.. \lfloor \frac{n}{2} \rfloor]$ се казва A_L , а подмасивът $A[\lfloor \frac{n}{2} \rfloor + 1..n]$ да е A_R . Тогава P_L се състои точно от елементите на A_L , а P_R се състои точно от елементите на A_R . Бисекторът е “виртуална права” – мисловна конструкция за по-лесно обяснение на алгоритъма. В действителност бисекторът е число: щом сме избрали бисекторът да е вертикална права, то всички нейни точки имат една и съща първа координата и това число е бисекторът. Нека това число бъде средното аритметично от първите координати на средните точки (при сортиране по първа координата), а именно $\frac{1}{2} (A[\lfloor \frac{n}{2} \rfloor].x + A[\lfloor \frac{n}{2} \rfloor + 1].x)$. Не е съществено важно бисекторът да е именно средното аритметично! Ако $A[\lfloor \frac{n}{2} \rfloor].x$ и $A[\lfloor \frac{n}{2} \rfloor + 1].x$ са различни, всяко число от отворения интервал $(A[\lfloor \frac{n}{2} \rfloor].x, A[\lfloor \frac{n}{2} \rfloor + 1].x)$ би “свършило работа” като бисектор. По отношение на разбиването на P няма значение кое число от този отворен интервал ще вземем. Но за фазата **комбинирай** точната стойност на бисектора има значение, както ще видим надолу.

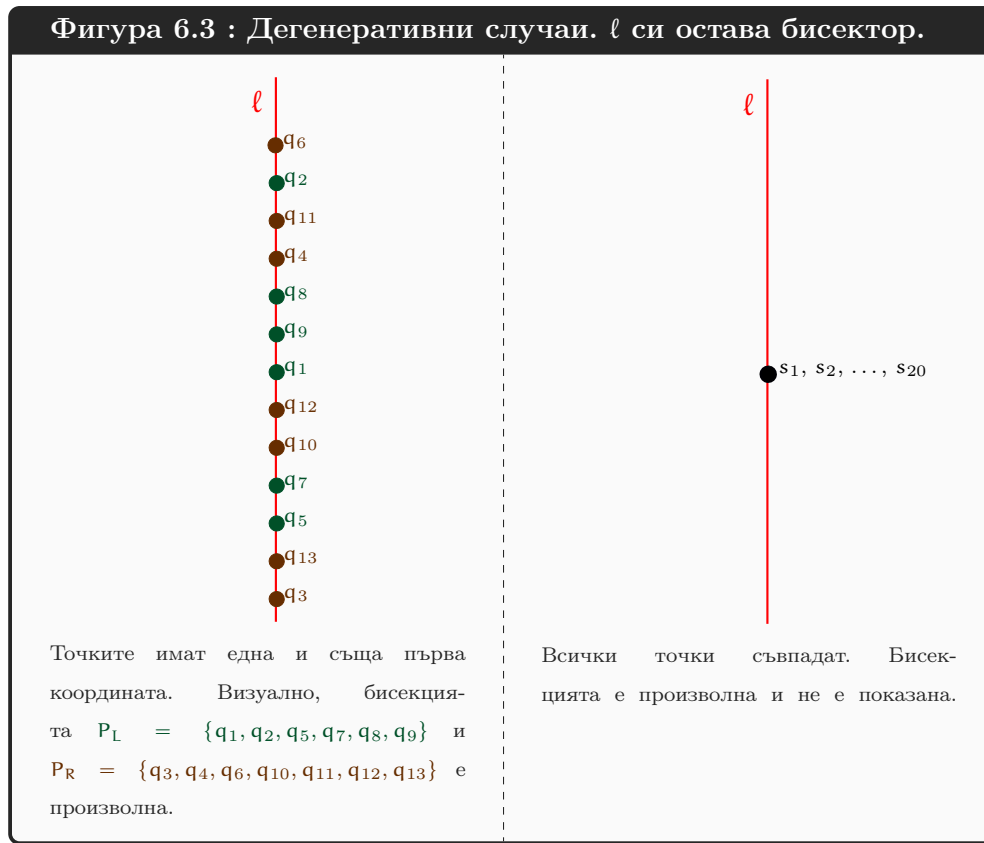
Като пример вижте Фигура 6.2. $n = 12$, така че $\lfloor \frac{n}{2} \rfloor = 6$ и $\lfloor \frac{n}{2} \rfloor + 1 = 7$, така че $A[\lfloor \frac{n}{2} \rfloor].x = 56$ и $A[\lfloor \frac{n}{2} \rfloor + 1].x = 62$, така че $\frac{1}{2} (A[\lfloor \frac{n}{2} \rfloor].x + A[\lfloor \frac{n}{2} \rfloor + 1].x) = \frac{56+62}{2} = 59$, което е и стойността на бисектора.



Едно уточнение. Дефиницията на задачата позволява различни точки да имат една и съща първа координата. Тъй като виртуалният бисектор е вертикална права, при съвпадения на първите координати може бисекторът да минава през много точки, стоящи една над друга или дори съвпадащи. Може дори това да са всички точки. Ако бисекторът минава през повече от една точки, това не е истински бисектор от геометрична гледна точка, защото това коя от тези точки в кое от двете подмножества отива е произволно. Но това е само от геометрична гледна точка. От алгоритмична гледна точка, бисекторът е число и разбиването

[†]Бисекторът да е вертикална права не е съществено за алгоритъма. Алгоритъм с хоризонтални бисектори има същата сложност и като описание, и като алгоритмична сложност.

на сортирания масив A спрямо това число на A_L и A_R (с максимално близки мощности) е безпроблемно. Такива дегенеративни случаи са показани на Фигура 6.3.



Както вече казахме, алгоритмичната бисекция става много лесно:

$$A_L = A \left[1 .. \left\lceil \frac{n}{2} \right\rceil \right]$$

$$A_R = A \left[\left\lceil \frac{n}{2} \right\rceil + 1 .. n \right]$$

P_L се състои точно от елементите на A_L , а P_R се състои точно от елементите на A_R . По отношение на примера, показан на Фигура 6.1 и Фигура 6.2:

$$A_L = [p_5, p_2, p_{10}, p_{11}, p_1, p_7]$$

$$A_R = [p_8, p_4, p_{12}, p_6, p_9, p_3]$$

Налага се обаче да разбием и B на B_L и B_R , където B_L се състои от същите точки като A_L , но сортирани по втората координата, а B_R се състои от същите точки като A_R , но сортирани по втората координата. Да видим кои множества са B_L и B_R в примера на Фигура 6.1 и Фигура 6.2, като първо си припомним B :

$$B = [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6]$$

$$B_L = [p_7, p_{10}, p_1, p_5, p_{11}, p_2]$$

$$B_R = [p_9, p_4, p_3, p_{12}, p_8, p_6]$$

Очевидно B_L и B_R не се получават като първата и втората половина на масива B (за разлика от A_L и A_R , които са именно първата и втората половина на масива A). Това си заслужава

да бъде подчертано отново.

Наблюдение 43

В текущия контекст, в общия случай не е вярно, че $B_L = B[1 \dots \lfloor \frac{n}{2} \rfloor]$ и не е вярно, че $B_R = B[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$. Тази разлика в конструирането на B_L и B_R спрямо A_L и A_R идва оттам, че избрахме вертикален, а не хоризонтален бисектор ℓ , с което в някакъв смисъл направихме хоризонталното направление предпочитано пред вертикалното. Естествено, можехме да изберем хоризонтален бисектор и тогава щяхме да предпочитаме вертикалното направление. Тогава B_L и B_R щяха да са съответно първата и втората половина на масива B , а A_L и A_R щяха да се конструират по-трудно.

Забележете, че не трябва да конструираме B_L като сортираме A_L по втора координата, нито да конструираме B_R като сортираме A_R по втора координата. Това би било ефективно, но неефикасно. Вече казахме, че ако искаме сложност по време $O(n \lg n)$ на алгоритъма, не трябва да сортираме на всяко ниво на рекурсията. Ще конструираме B_L и B_R от B във време $O(n)$. Да си припомним, че в предварителната обработка конструирахме масива B така, че всяка точка съдържа и индекса, който има тя самата в масива A . В нашия пример, B е

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), \\ (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)]$$

Слагаме в B_L точно тези елементи на B , чиито индекси в A са в множеството $\{1, 2, \dots, \lfloor \frac{n}{2} \rfloor\}$, а в B_R слагаме точно тези елементи на B , чиито индекси в A са в множеството $\{\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \dots, n\}$. В нашия пример, в B_L слагаме елементите на B , чиято втора компонента е в множеството $\{1, 2, \dots, 6\}$, а в B_R слагаме тези, чиято втора компонента е в множеството $\{7, 8, \dots, 12\}$. Забележете, че елементите на B_L са разположени взаимно по същия начин, по който са разположени в B , и елементите на B_R са разположени взаимно по същия начин, по който са разположени в B . Ерго, генерирането на B_L и B_R можем да постигнем с едно сканиране на B отляво надясно, като слагаме съответния $B[i]$ в B_L или B_R според индекса в A . В нашия пример, $B[1]$ отива в B_L , $B[2]$ отива в B_R , $B[3]$ отива в B_L , и така нататък, като наистина получаваме

$$B_L = [p_7, p_{10}, p_1, p_5, p_{11}, p_2] \\ B_R = [p_9, p_4, p_3, p_{12}, p_8, p_6]$$

Очевидно тази фаза се изпълнява в линейно време.

Фаза 2: владей. Рекурсивно изпълняваме алгоритъма върху P_L и P_R , като от тези две викания получаваме стойности d_L и d_R . d_L е минимално разстояние между точки в P_L , а d_R е минимално разстояние между точки в P_R .

Фаза 3: комбинирай. Нека t е отсечка с минимална дължина, чиито краища са две различни точки от P . Очевидно е, че точно едно от следните е истина:

- Краищата на t са в P_L . В този случай $|t| = d_L$, тъй като допускаме, че рекурсивното викане върху P_L работи коректно.
- Краищата на t са в P_R . В този случай $|t| = d_R$, тъй като допускаме, че рекурсивното викане върху P_R работи коректно.

- Единият край на t е в P_L , а другият е в P_R . В този случай казваме, че t е *прекосяваща отсечка*. Изпълнено е $|t| = d_M$, където

$$d_M = \min \{ \text{dist}(x, y) \mid x \in P_L, y \in P_R \} \quad (6.2)$$

В началото на фазата **комбинирай** ние не разполагаме с d_M и трябва тепърва да го изчислим.

От тези съображения става ясно, че задачата се свежда до намирането на d_M , след което алгоритъмът връща

$$\min \{ d_L, d_R, d_M \}$$

Изразът “ $\min \{ \text{dist}(x, y) \mid x \in P_L, y \in P_R \}$ ” може да бъде превърнат директно в алгоритъм, който изчислява d_M . Този подход обаче е неефекасен. Той опитва всяка точка от P_L с всяка точка от P_R , тоест опитва всяка прекосяваща отсечка, което води до квадратична сложност по време. Естествено, ако фазата **комбинирай** работи в $\Theta(n^2)$, няма как алгоритъмът да работи в $O(n \lg n)$. Налага се да направим нещо по-умно, за да изчислим d_M . Правим две подобрения.

Първо подобрение. Нека $d = \min \{ d_L, d_R \}$. Да дефинираме, че *късите прекосяващи отсечки* са тези с дължина, по-малка от d . Такива може и да няма. Ако няма къси прекосяващи отсечки, решението е d и нашият алгоритъм следва да върне d . Ако има къси прекосяващи отсечки, решението е дължината на най-къса прекосяваща отсечка и нашият алгоритъм следва да върне това число.

Наблюдение 44

Ако никакви прекосяващи отсечки не са къси, то няма смисъл да изчисляваме дължините им при изчисляването на d_M .

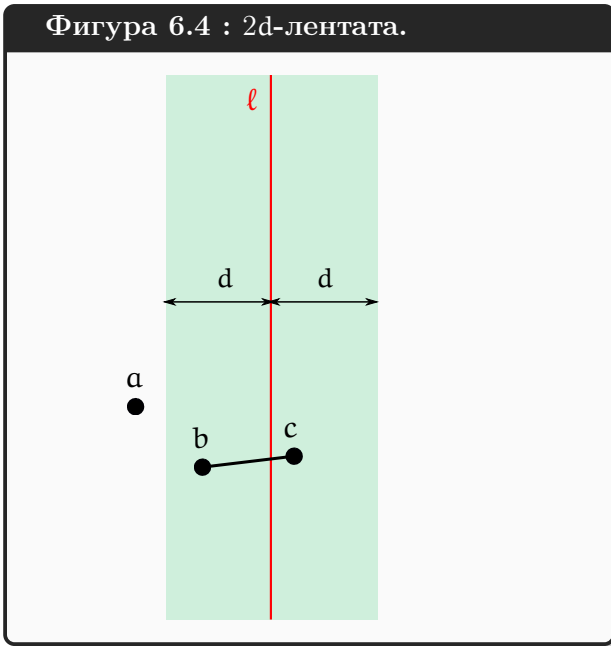
Да си представим точките от P заедно с правата-бисектор ℓ . Да дефинираме *2d-лентата* като областта от равнината, чиито точки са на разстояние не по-голямо от d от двете страни на ℓ . Иначе казано, ако поне единият край на някоя прекосяваща отсечка е извън 2d-лентата, то нейната дължина е по-голяма от d и не ни интересува.

Наблюдение 45

Ако някоя точка лежи извън 2d-лентата, то тя не е край на къса прекосяваща отсечка.

Това е илюстрирано на Фигура 6.4, на която 2d-лентата е показана в светло синьо. Очевидно точка a не може да е край на къса прекосяваща отсечка, защото другият край на тази отсечка трябва да е от другата страна на ℓ . Обаче точка b се намира в 2d-лентата и наистина тя е край на къса прекосяваща отсечка (другият ѝ край е c).

Фигура 6.4 : 2d-лентата.



И така, първото подобрение е следното. Съгласно Наблюдение 45 и Наблюдение 44, за изчисляването на d_M игнорираме всички точки извън 2d-лентата. Ако поне две точки от P попадат в 2d-лентата, да изчислим d_M само от точките на P , които са в 2d-лентата; а ако в 2d-лентата има нула или една точки от P , то d_M да бъде безкрайност, което е същото като да игнорираме d_M и алгоритъмът да върне d . Ако трябва да сме прецизни, разглеждайки само точките в 2d-лентата, ние не изчисляваме непременно d_M , а нещо друго, което ще наречем d'_M . Причината е, че d_M и d'_M не са непременно равни:

$$d_M = \min \{ \text{dist}(x, y) \mid x \in P_L, y \in P_R \} \quad (6.3)$$

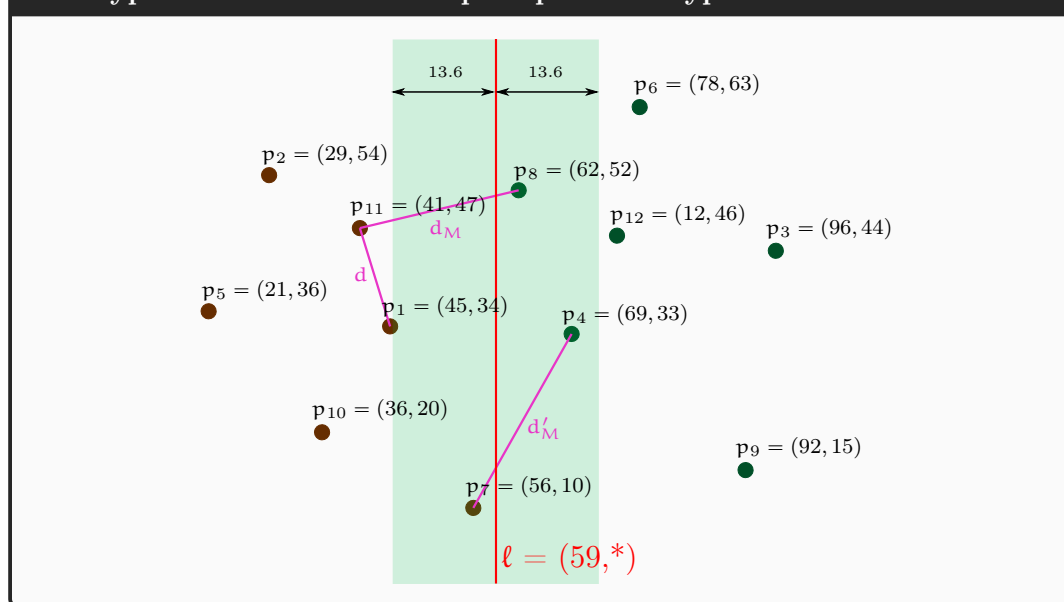
$$d'_M = \min \{ \text{dist}(x, y) \mid x \in P_L \text{ и } x \text{ е в } 2d\text{-лентата}, y \in P_R \text{ и } y \text{ е в } 2d\text{-лентата} \} \quad (6.4)$$

Забележете, че възможно $d'_M > d_M$, а в първото подобрение става дума именно за изчисляването на d'_M . Това обаче не е проблем, защото алгоритъмът връща $\min \{d, d'_M\}$, а ако $d'_M > d_M$, то задължително $d_M > d^\dagger$; ерго, ако $d'_M > d_M$, то решението е d и алгоритъмът ще върне именно d . С други думи, невъзможно е $d'_M > d > d_M$; ако това би било възможно, то би било възможно и да “изтървем” оптимално решение d_M .

Фигура 6.5 показва 2d-лентата в примера от Фигура 6.1. Минималното разстояние между точки от P_L е между $p_1 = (45, 34)$ и $p_{11} = (41, 47)$, а именно $d = \sqrt{(45 - 41)^2 + (34 - 47)^2} \approx 13.60147051$. Тогава $d_L \approx 13.6$. Минималното разстояние между точки от P_R е между p_4 и p_{12} , както и между p_8 и p_{12} , а именно $\sqrt{(75 - 69)^2 + (46 - 33)^2} = \sqrt{(62 - 75)^2 + (52 - 46)^2} \approx 14.31782106$. Тогава $d_R \approx 14.3$ и $d = d_L$. На Фигура 6.5 виждаме P_L и P_R с бисектора $\ell = (59, *)$ между тях и 2d-лентата, простираща се на около 13.6 отляво и отдясно на бисектора.

[†]Това е така, защото, ако $d'_M > d_M$, то d_M е дължина на отсечка, поне единият край на която е извън 2d-лентата и която отсечка, съгласно Наблюдение 45, не е къса, от което следва, че $d_M > d$.

Фигура 6.5 : 2d-лентата в примера от Фигура 6.1.



Само три точки от P попадат в 2d-лентата: p_7 , p_4 и p_8 (p_1 е извън 2d-лентата). Тогава $d'_M = \min \{ \text{dist}(x, y) \mid x \in \{p_7\}, y \in \{p_4, p_8\} \} = \text{dist}(p_4, p_7) = \sqrt{(56 - 69)^2 + (10 - 33)^2} \approx 26.41968963$. Виждаме, че в нашия пример е изпълнено $d_M < d'_M$, понеже d_M се реализира между p_{11} и p_8 , като $\text{dist}(p_{11}, p_8) = \sqrt{(41 - 62)^2 + (47 - 52)^2} \approx 21.58703314$. Но, както вече отбелязахме, ако $d'_M > d_M$, какъвто е този случай, то $d_M > d$, така че за крайното решение няма значение дали сравняваме 13.6 с 26.4 или с 21.5 – решението си остава 13.6.

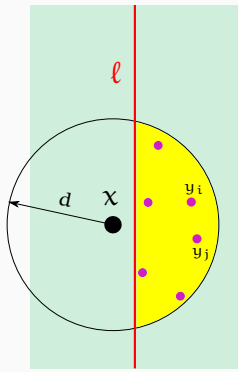
Но по отношение на най-лошия случай само първото подобрене изобщо не е подобрене. Ако всички точки от P са в 2d-лентата и търсим d_M изчерпателно, то сложността пак би била квадратична.

Второ подобрене. Забелязваме, че за всяка точка x в 2d-лентата има твърде ограничен брой други точки, такива че има смисъл да изчисляваме разстоянието между x и всяка от тях. Например, нека $x \in P_L$. Може ли да има сто точки $y_1, y_2, \dots, y_{100}$ в P_R , такива че във фазата **комбинирай** всяка от тях да е на разстояние, по-малко от d , от точка x ? Отговорът е, че не може. За да е изпълнено това, би трябвало $y_1, \dots, y_{100}$ да са в кръга с център x и радиус d ; по-точно казано, в сечението на този кръг и дясната половина на 2d-лентата. Но тогава биха съществували различни y_i и y_j , такива че $\text{dist}(y_i, y_j) < d$, което е невъзможно, защото $d = \min \{d_L, d_R\}$ по дефиниция.

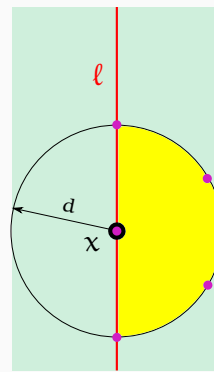
Това е илюстрирано на Фигура 6.6. Сечението на кръга с център x и радиус d с дясната половина на 2d-лентата е оцветено в жълто. Вляво разглеждаме възможността да има шест точки (в лилаво), всяка от които е “кандидат” за друг край на къса прекосяваща отсечка (единият край е x). Очевидно е, че както и да бъдат разположени шестте лилави точки в жълтия сегмент, няма как всеки две от тях да са на разстояние поне d една от друга. Например, на фигурата y_i и y_j са толкова близко една до друга, че разстоянието d_R не може да е повече от $\text{dist}(y_i, y_j)$, следователно, би трябвало $d \leq \text{dist}(y_i, y_j)$, в противоречие с показаното на фигурата.

Вдясно илюстрираме факта, че пет е точната горна граница за броя на въпросните кандидати. Едната от десните (лилавите) точки съвпада с x , което не е невъзможно. x лежи точно върху бисектора, така че сечението на кръга с диаметър d и център x с дясната страна на 2d-лентата е полукръг. Ако d_R е минималното разстояние между точки измежду тези пет (това е възможно), и $d = d_R$ (и това е възможно), и петте лилави точки попадат в жълтия полукръг.

Фигура 6.6 : Възможните други краища на къса прек. отсечка са малко.



Очевидно $\text{dist}(y_i, y_j) < d$. Очевидно няма как да разположим шест точки в жълтия сегмент, така че всеки две от тях да са на разстояние поне d една от друга.

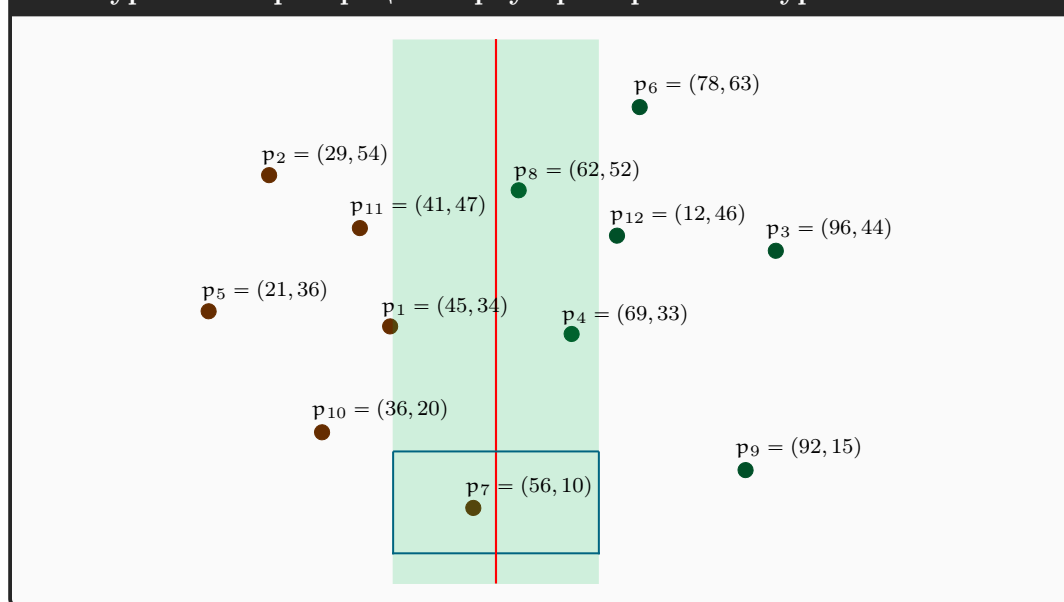


Ако жълтият сегмент е полукръг, може да разположим пет точки в него, така че всеки две да са на поне d една от друга. Едната от тези точки съвпада с x . Повече от пет точки не може.

След като се убедихме, че съществува неголяма константа c , такава че за всяка точка от $2d$ -лентата трябва да разглеждаме най-много c други точки в изчисляването на d'_M , да видим как ще реализираме тази идея ефикасно. Досегашните разсъждения навеждат на мисълта да направим следното: за всяка точка x от лявата страна на $2d$ -лентата да намерим всички точки от дясната страна на $2d$ -лентата, които се намират в кръга с център x и радиус d , ако има такива, и да видим коя от тях е най-близо до x . Но това би довело до усложнения, които е по-добре да си спестим. Ще разгледаме подхода на [34, стр. 1040–1043], който е значително по-прост и е достатъчно ефикасен. А досегашните разсъждения за ограничения брой на точките “от другата страна”, които има смисъл да разглеждаме, са само за придобиване на по-добра интуиция.

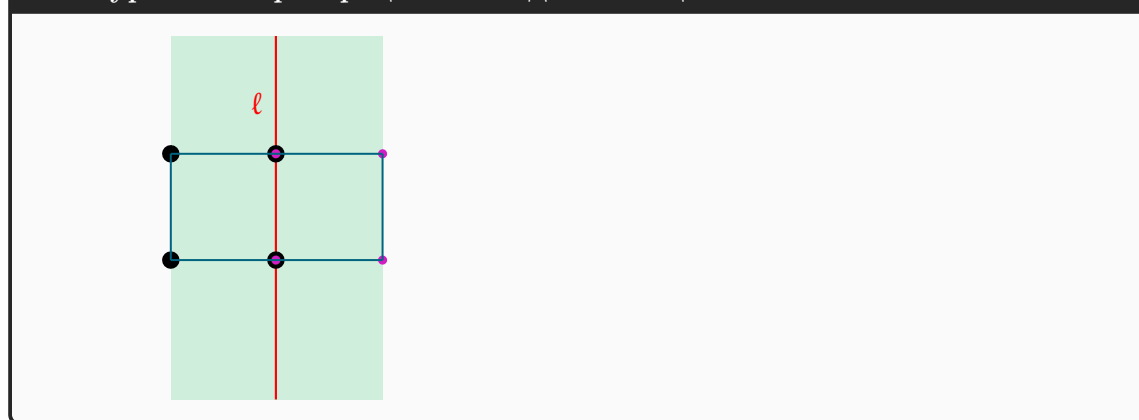
Представяме си *прозорец* – правоъгълник с размери $2d \times d$, със страни, успоредни на координатните оси, като голямата страна е успоредна на абсцисата, а центърът на прозореца лежи върху бисектора. Идеята е да плъзнем прозореца от най-долу (най-малката втора координата) до най-горе (най-голямата втора координата), като центърът му се плъзга по бисектора, а страните му остават успоредни на координатните оси. Фигура 6.7 показва прозореца, нарисован върху примера от Фигура 6.1.

Фигура 6.7 : Прозорецът върху примера от Фигура 6.1.



На Фигура 6.7 прозорецът обхваща само една точка, а именно p_7 . Колко точки от P —това означава и от P_L , и от P_R —най-много може да обхваща прозореца? Този брой е ограничен, тъй като размерите на прозореца не са произволни. Лесно се забелязва, че най-много осем точки от P може да са в прозореца. Това е демонстрирано на Фигура 6.8. Четирите черни точки са от P_L , а четирите лилави са от P_R . Черните точки са върху върховете на квадрата-лява половина на прозореца, а лилавите са върху върховете на квадрата-дясна половина на прозореца. Две от черните точки съвпадат с две от лилавите.

Фигура 6.8 : Прозорецът може да обхваща най-много осем точки.



Ето и второто подобрение: да плъзнем прозореца отдолу нагоре в $2d$ -лентата, да изчислим минималното разстояние между две (различни) точки, които се намират в него в който да е момент, едната от P_L , другата от P_R , и полученото да присвоим на d'_M .

Разбира се, прозорецът е метафора. Ние не правим компютърна геометрия и не плъзгаме правоъгълници, центрирани върху вертикални прави. В действителност правим следното и това е фазата **комбинирай**.

- Първо създаваме масив C от всички точки, намиращи се в $2d$ -лентата, сортирани по втора координата. За тази цел използваме масивите A и B . Да си припомним, че истинският бисектор е число, да го наречем β , а вертикалната права е виртуален бисектор.

Ерго, създаваме C като подмасивът на A , състоящ се от точно тези точки p_i , за които $|p_i.x - \beta| \leq d$.

Ето как изглежда това в нашия пример. Да си припомним, че

$$A = [(21, 36), (29, 54), (36, 20), (41, 47), (45, 34), (56, 10), \\ (62, 52), (69, 33), (75, 46), (78, 63), (92, 15), (96, 44)]$$

$\beta = 59$ и $d \approx 13.6$. $59 - 13.6 = 45.4$ и $59 + 13.6 = 72.6$, така че точно три точки влизат в C , а именно $(56, 10)$, $(62, 52)$ и $(69, 33)$; нещо, което видяхме още на Фигура 6.5.

И така, използвахме A , който е сортиран по първа координата, за да отделим точките, попадащи в $2d$ -лентата.

- От масива B отделяме сортираната по втора координата последователност от тези същите точки, слагайки ги в някакъв масив D в същия ред, в който са в B (тоест, сортирани по втора координата). Това можем да направим ефикасно, защото в предварителната обработка сме добавили към елементите на B техните местоположения в A , така че сега можем във време $\Theta(1)$ да определим за всеки елемент от B дали той е в C или не; с други думи, в константно време можем да определим за произволен елемент от B дали тази точка попада в $2d$ -лентата или не. Ерго, с едно сканиране на B можем да копираме в D тези елементи, които попадат в $2d$ -лентата, като ги копираме в същия ред, в който са в B .
- В нашия пример D е много малък:

$$D = [(56, 10), (69, 33), (62, 52)]$$

но да допуснем, че D е достатъчно голям. Да кажем, че D има k елемента за някакво $k \geq 8$. Нека $m \leftarrow \infty$. Инициализираме $i \leftarrow 1$, $j \leftarrow 8$ и изпълняваме следното $k - j + 1$ пъти:

- ♦ С брутална сила намираме двойка различни елементи r, s от $D[i..j]$, които са на минимално разстояние един от друг, като единият от r, s е от P_L , а другият е от P_R . Ако има такива, $m = \min\{m, \text{dist}(r, s)\}$. Това става в константно време, защото броят на ненаредените двойки от $D[i..j]$ е $\binom{8}{2} = 28$.
- ♦ $i++$, $j++$.

След края на това, полученото m е точно търсеното d'_M .

Забележете, че масивът $D[i..j]$ не реализира точно прозореца $2d \times d$, за който стана дума във второто подобрение. Ако D е достатъчно голям, нашата алгоритмична реализация работи с $D[i..j]$, такъв че $j - i = 7$, тоест, винаги в нашия прозорец има осем точки. Както се вижда на Фигура 6.7, ако прозорецът е наистина $2d \times d$, той може да обхваща само една точка или дори нула точки (лесно се вижда, че има такива негови положения). С други думи, осемте точки, които разглеждаме на всяка итерация, може да бъдат прекалено далече една от друга, за да се намират в рамките на $2d \times d$ прозореца.

*Това, че $D[i..j]$ обхваща винаги осем точки при достатъчно голямо D , не води до проблеми за нашия алгоритъм. Важното е, че **гледайки не повече от осем последователни по вертикал точки** в даден момент, не*

можем да изпуснем оптимална стойност на d'_M . Иначе казано, няма смисъл да търсим d'_M , разглеждайки двойка точки, които не са измежду осем последователни точки в D . Нашата имплементация може и да опита да търси d'_M върху точки, които са прекалено далече една от друга, но няма да изпусне оптимално решение и е с линейна сложност.

И така, алгоритъмът връща

$$\min \{d_L, d_R, d'_M\}$$

Коректност. Коректността на алгоритъма е очевидна предвид разсъжденията, които направихме.

Сложност. Очевидно е, че фазата **Комбинирай**, реализирана по гореописания начин, работи във време $\Theta(n)$. Фазата **Разделяй** също работи в линейно време. Тогава сложността на алгоритъма след предварителната обработка се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

което има решение $T(n) \asymp n \lg n$. Предварителната обработка има сложност $\Theta(n \lg n)$, доминирана от сортиранията; останалата работа в предварителната обработка е $\Theta(n)$. Като цяло, сложността на алгоритъма е $\Theta(n \lg n)$.

6.2.2 ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА

Този нетривиален алгоритъм е открит от Blum, Floyd, Pratt, Rivest и Tarjan през 70-те [23], [24]. Изложението тук е по учебника на Cormen, Leiserson, Rivest и Stein [34, стр. 220–222].

Изч. Задача 25: ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА

екземпляр: крайно непразно множество $S \subset \mathbb{Z}$; число $k \in \mathbb{N}$, такова че $0 \leq k \leq |S| - 1$
решение: Числото $x \in S$, такова че $|\{a \in S \mid a < x\}| = k$.

Една забележка: очевидно в тази изчислителна задача става дума не за k -ия, а за $(k+1)$ -вия по големина елемент, като k е броят на по-малките от него елементи. Така че формулировката на задачата чрез k , а не чрез $k+1$, може да е леко подвеждаща, но в [34] задачата е дефинирана точно така.

Тази задача може да се реши тривиално във време $\Theta(n \lg n)$ чрез бързо сортиране на елементите на S , последвано от връщане на $(k+1)$ -вия елемент. Тук ще разгледаме по-бързо решение: линеен алгоритъм, наречен PICK в [24] и SELECT в [34]. SELECT вика процедура PARTITION-SELECT, която е практически същата като PARTITION-LOMUTO (Алгоритъм 14), която се вика от QUICKSORT (Алгоритъм 15), с две незначителни разлики:

1. в PARTITION-SELECT, елементът *pivot*, спрямо който се разместват елементите на масива, не е непременно най-десният елемент поначало, а е произволен елемент на входния масив; в началото той се открива и се разменя с най-десния, след което се изпълнява PARTITION-LOMUTO;
2. PARTITION-SELECT връща $pp - 1$, а не pp , защото за целите на SELECT ни трябва не позицията на *pivot*, а броят на елементите вляво от него, в пренаредения масив.

Следва псевдокод на PARTITION. За да се различава от PARTITION в Секция 6.4, ще наречем тукашната версия PARTITION–SELECT. Функцията find-index връща индекса на елемента (първия аргумент) в масива (втория аргумент), при допускането, че този елемент се среща точно веднъж в този масив.

Алгоритъм 10: Partition–Select

```

PARTITION–SELECT( $A[l \dots h]$ : int;  $pivot$ : element from  $A[l \dots h]$ )
1   $k \leftarrow \text{find-index}(pivot, A[l \dots h])$ 
2   $\text{swap}(A[k], A[h])$ 
3   $pp \leftarrow l$ 
4  for  $i \leftarrow l$  to  $h - 1$ 
5      if  $A[i] < pivot$ 
6           $\text{swap}(A[i], A[pp])$ 
7           $pp++$ 
8   $\text{swap}(A[pp], A[h])$ 
9  return  $pp - 1$ 

```

Няма да правим формално доказателство на коректността на PARTITION–SELECT. Доказателството за коректността на PARTITION–LOMUTO (Теорема 74) върши работа и за PARTITION–SELECT. Само ще споменем ефекта от работата на PARTITION: той размества елементите на входния масив A по такъв начин, че

- $A[l \dots pp - 1]$ се състои точно от елементите на $A[l \dots h]$, по-малки от $pivot$;
- $A[pp]$ съдържа $pivot$;
- $A[pp + 1 \dots h]$ се състои точно от елементите на $A[l \dots h]$, по-големи от $pivot$.

Да си припомним, че според текущите допускания входният масив няма повтарящи се елементи, докато при PARTITION–LOMUTO (Алгоритъм 14) се допускат повторения във входа.

Внимание! Алгоритъмът PARTITION **не сортира**. В края на изпълнението му, нито “малките елементи” в $A[l \dots pp - 1]$ са непременно сортирани, нито “големите елементи” в $A[pp + 1 \dots h]$ са непременно сортирани.

Следва псевдокод на SELECT. Първият аргумент A е масивът, който реализира множеството S от Задача 25, а вторият аргумент k е числото от Задача 25. Описанието на алгоритъма, взето от [34], е на по-високо ниво от обичайния псевдокод.

Алгоритъм 11: Select

```

SELECT( $A[1 \dots n]$ : числен масив без повторения;  $k \in \{0, 1, \dots, n - 1\}$ )
1  if  $n = 1$ 
2      return  $A[1]$ 
3   $q \leftarrow \lceil \frac{n}{5} \rceil$ ,  $r \leftarrow n \bmod 5$ 
4  if  $r = 0$ 
5      разбий  $A$  на подмасиви  $B_1, \dots, B_q$ , всеки с 5 елемента
6  else
7      разбий  $A$  на подмасиви  $B_1, \dots, B_{q-1}$ , всеки с 5 елемента, и  $B_q$  с  $r$  елемента
8  за  $i \in \{1, 2, \dots, q\}$ , сортирай  $B_i$  и намери съответната медиана  $m_i$ 
9  направи масив  $[m_1 \dots m_q]$  от медианите и го наречи  $C$ 

```

```

10  m ← SELECT(C, ⌊ $\frac{q-1}{2}$ ⌋)
11  j ← PARTITION-SELECT(A, m)
12  if k = j
13      return m
14  else
15      if k < j
16          return SELECT(A[1 .. j], k)
17      else
18          return SELECT(A[j + 2 .. n], k - j - 1)

```

Да разгледаме малък пример. Нека $n = 25$ и елементите на A са $1, 2, \dots, 25$, като

$$A = [17, 6, 8, 22, 11, 1, 7, 4, 19, 21, 3, 20, 16, 24, 10, 25, 12, 18, 15, 14, 13, 2, 23, 9, 5]$$

Нека $k = 17$. Това е входът.

Очевидно $q = 5$. Нека A бъде разбит (ред 5) на 5 подмасива по най-естествения начин:

$$B_1 = [17, 6, 8, 22, 11]$$

$$B_2 = [1, 7, 4, 19, 21]$$

$$B_3 = [3, 20, 16, 24, 10]$$

$$B_4 = [25, 12, 18, 15, 14]$$

$$B_5 = [13, 2, 23, 9, 5]$$

След сортирането на всеки от тях (ред 8), те стават:

$$B_1 = [6, 8, 11, 17, 22]$$

$$B_2 = [1, 4, 7, 19, 21]$$

$$B_3 = [3, 10, 16, 20, 24]$$

$$B_4 = [12, 14, 15, 18, 25]$$

$$B_5 = [2, 5, 9, 13, 23]$$

Тогава $m_1 = 11$, $m_2 = 7$, $m_3 = 16$, $m_4 = 15$ и $m_5 = 9$, така че $C = [11, 7, 16, 15, 9]$ (ред 9). На ред 10, $\lfloor \frac{5-1}{2} \rfloor = 2$, така че викаме SELECT с втори аргумент 2. Това викане връща 3-ия по големина елемент на C , който е 11, така че на ред 10, m получава стойност 11.

На ред 11 викаме PARTITION-SELECT върху A с втори аргумент 11. Това викане връща $j = 10$, което е напълно очаквано, ако елементите на A са $1, \dots, 25$ и *pivot* е 11. Самият A изглежда така след разместванията, направени от PARTITION-SELECT:

$$A = [6, 8, 5, 1, 7, 4, 3, 10, 2, 9, 11, 20, 16, 24, 22, 25, 12, 18, 15, 14, 13, 19, 23, 21, 17]$$

Тъй като $k = 17$ и $j = 10$, условието на ред 12 е лъжа, условието на ред 15 също е лъжа и изпълнението отива на ред 18. $A[j + 2 .. n]$ е $A[12 .. 25]$. В случая:

$$A[12 .. 25] = [20, 16, 24, 22, 25, 12, 18, 15, 14, 13, 19, 23, 21, 17] \quad (6.5)$$

$k - j - 1$ е 6, така че викането на ред 18 е SELECT($A[12 .. 25]$, 6). Очевидно в този пример това е коректно, понеже осемнадесетия ($17 + 1 = 18$) по големина елемент на входния A , а именно 18, е същият като седмия ($6 + 1 = 7$) по големина елемент на $A[12 .. 25]$ от (6.5), а именно 18.

В този пример седмият по големина елемент в $A[12..25]$, а именно 18, се намира на седма позиция, но това е чиста случайност. Ако винаги беше така, нямаше да има нужда от рекурсивното викане на ред 18.

Разглеждането на примера приключи. Сега ще докажем коректността на SELECT формално.

Теорема 69: Коректността на алгоритъма SELECT

Ако $A[1..n]$ е масив от числа без повторения на елементи и $k \in \{0, 1, \dots, n-1\}$, то $\text{SELECT}(A, k)$ връща $(k+1)$ -ия по големина елемент на входа.

Доказателство: Ще използваме силна индукция по n .

Базовият случай е $n = 1$. Ако $n = 1$, то $\{0, 1, \dots, n-1\} = \{0\}$, така че единствената допустима стойност за k е 0.

От друга страна, ако $n = 1$, то условието на ред 1 е истина и изпълнението отива на ред 2, където алгоритъмът връща $A[1]$. И наистина е $(k+1)$ -вия по големина елемент на масива $A[1..1]$ при $k = 0$. ✓

Индуктивното допускане е, че за масив M , такъв че $1 \leq |M| \leq n-1$, $\text{SELECT}(M, \bullet)$ работи коректно.

В индуктивната стъпка разглеждаме $\text{SELECT}(A, \bullet)$. Разглеждаме направо ред 11. За коректността няма значение точно каква е стойността на m . От значение е само това, че m е елемент от масива и $\text{PARTITION-SELECT}(A, m)$ на ред 11 размества A по такъв начин и връща такава стойност j , че: всички елементи, по-малки от m , са в $A[1..j]$, $A[j+1]$ съдържа m , а всички елементи, по-големи от m , са в $A[j+2..n]$.

Следните три случая са взаимно изключващи се и изчерпателни.

Случай 1. Ако $k = j$, то m е $(k+1)$ -вият по големина елемент. От друга страна, условието на ред 12 е истина и алгоритъмът коректно връща m на ред 13. ✓

Случай 2. Ако $k < j$ (ред 15), то $(k+1)$ -вият по големина елемент на входния A е същият като $(k+1)$ -вият по големина елемент на $A[1..j]$. За да се убедим в това, да съобразим, че всички елементи, по-малки от m , са в подмасива $A[1..j]$.

От друга страна, условието на ред 15 е истина. На ред 16 се вика $\text{SELECT}(A[1..j], k)$ и това викане връща $(k+1)$ -вия по големина елемент на $A[1..j]$ съгласно индуктивното предположение. Но, както видяхме, това е точно $(k+1)$ -вият по големина елемент на входа. ✓

Случай 3. Ако $k > j$, то $(k+1)$ -вият по големина елемент на входния A е същият като $(k-j)$ -ият по големина елемент на $A[j+2..n]$. За да се убедим в това, да съобразим, че преди $(k+1)$ -вия в A има точно k елемента, което означава, че преди него в $A[j+2..n]$ има $k - (j+1) = k - j - 1$ елемента, защото сега не броим $A[1]$, $A[2]$, ..., $A[j]$ и $A[j+1]$.

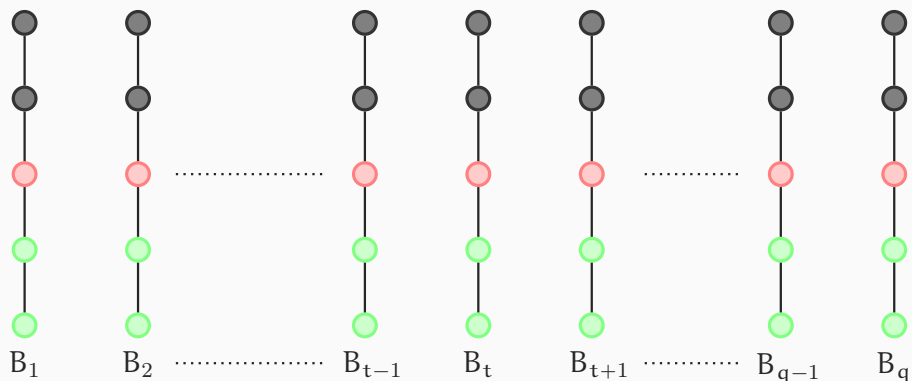
От друга страна, изпълнението достига ред 18. Там се вика $\text{SELECT}(A[j+2..n], k-j-1)$ и това викане връща $(k-j)$ -ия елемент на $A[j+2..n]$ съгласно индуктивното предположение. Но, както видяхме, това е точно $(k+1)$ -вия по големина елемент на входа. ✓ □

Теорема 70: SELECT е линеен алгоритъм

Ако $A[1..n]$ е масив от числа без повторения на елементи и $k \in \{0, 1, \dots, n-1\}$, то $\text{SELECT}(A, k)$ работи във време $\Theta(n)$.

Доказателство: БОО, нека n е кратно на 5, така че всички масиви B_i са с по точно 5 елемента. Тогава за всяко $i \in \{1, 2, \dots, q\}$, медианата m_i е гарантирано по-голяма от точно два елемента и е гарантирано по-малка от точно два елемента на B_i . Фигура 6.9 показва диаграма на Hasse на множеството след ред 8 на SELECT. Всеки B_i е линейно наредено, петелементно множество, а за $i \neq j$, всеки елемент от B_i е несравним с всеки елемент от B_j . Във всеки B_i , двата елемента, по-малки от медианата, са изобразени в зелено, а двата елемента, по-големи от медианата, в тъмно сиво.

Фигура 6.9 : Диаграма на Hasse на данните след ред 8 на SELECT.



Малките елементи са в зелено, големите са в тъмно сиво, а медианите са в червено.

На ред 10 викаме SELECT върху масива от медианите $[m_1 \dots m_q]$ с втори аргумент $\lfloor \frac{q-1}{2} \rfloor$. Съгласно индуктивното предположение (да си припомним, че правим доказателството със силна индукция), на ред 10, SELECT връща този елемент на масива $[m_1 \dots m_q]$, който по големина е номер $\lfloor \frac{q-1}{2} \rfloor + 1$. Но $\lfloor \frac{q-1}{2} \rfloor + 1 = \lfloor \frac{q-1+2}{2} \rfloor = \lfloor \frac{q+1}{2} \rfloor$, а по определение елементът, който по големина е номер $\lfloor \frac{q+1}{2} \rfloor$, е медианата. Следователно, на ред 10, m получава стойността на медианата на медианите.

Медианата на медианите m може да не съвпада с медианата на A . В примера, който видяхме, m беше 11, а медианата на A беше 13. Допълнение 41 показва, че медианата на медианите може да е най-малко 9, но не по-малко.

Сега ще покажем, че m да е медианата на A е оптимално с оглед на бързодействието в най-лошия случай. Ако m е медианата на A , PARTITION-SELECT(A, m) на ред 11 слага m на позиция $\lfloor \frac{q+1}{2} \rfloor$ в A и връща j , равно на $\lfloor \frac{q+1}{2} \rfloor$. В най-лошия случай k не е равно на j (проверката на ред 12), защото тогава алгоритъмът би терминиран с **return m** на ред 13. В най-лошия случай се стига до рекурсивно викане или на ред 16, или на ред 18. Кое точно от тези рекурсивни викания се случва, зависи от k при фиксирано j . Същественото за “лошотията” на най-лошия случай е размерът на масива, върху който става рекурсивното викане. Колкото по-голям е той, толкова по-зле за бързодействието. И тук правим ключовото наблюдение, че ако m на ред 10 получи стойността на медианата на A , масивите $A[1 \dots j]$ (ред 16) и $A[j+2 \dots n]$ (ред 18) са максимално близки по големина; те са или с равна големина, ако n е нечетно, или големините им се различават на единица, ако n е четно. Ерго, m да е медианата на A минимизира максимума от дължините им, и с това минимизира сложността по време в най-лошия случай.

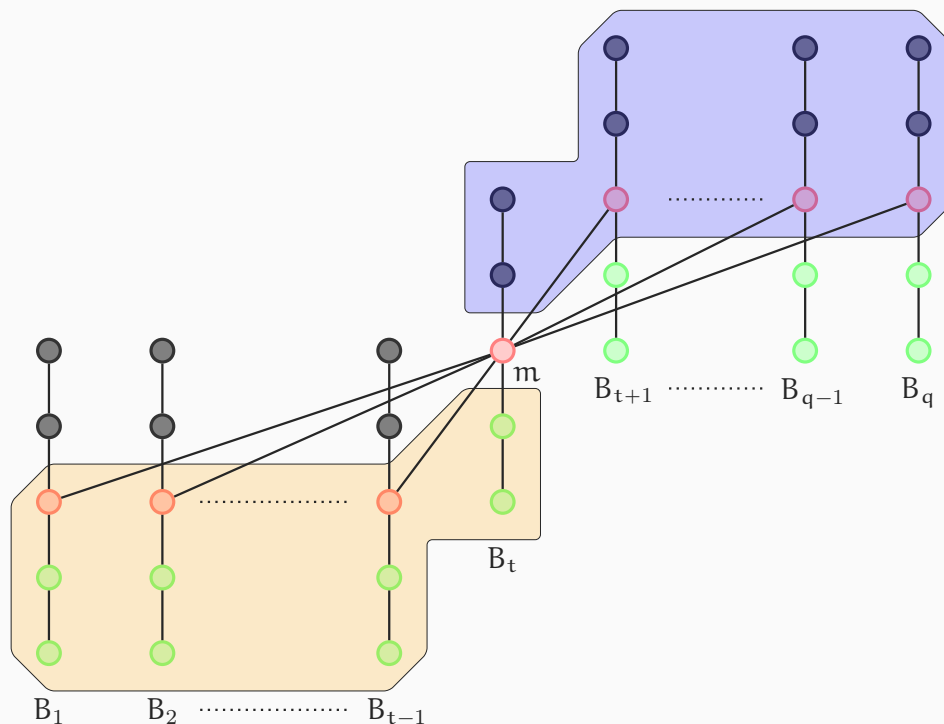
Както видяхме обаче, m може да не е медианата на A . Колкото по-отдалечена по стойност е m от медианата на A , толкова повече $A[1 \dots j]$ и $A[j+2 \dots n]$ се различават по големина, а в най-лошия случай рекурсивното викане (ред 16 или ред 18) ще се случи върху по-големия

от тях. БОО, нека

- $B_{\lfloor \frac{q+1}{2} \rfloor}$ е масивът, от който идва m ; нека $t = \lfloor \frac{q+1}{2} \rfloor$,
- медианите на $B_1, \dots, B_{t-1}$ са по-малки от m ,
- а медианите на $B_{t+1}, \dots, B_q$ са по-големи от m .

При това положение, Фигура 6.10 показва диаграмата на Hasse за елементите на A след изпълнението на ред 10.

Фигура 6.10 : Диаграма на Hasse на данните след ред 10 на SELECT.



Медианата на медианите m е медианата на B_t . Елементите, гарантирано по-малки от m , са показани върху оранжев полупрозрачен фон. Елементите, гарантирано по-големи от m , са показани върху син полупрозрачен фон. Ако q е нечетно, то оранжевата и синята област съдържат еднакъв брой елементи. Ако q е четно, то тези от синята област са с три повече, понеже вземаме за медиана долния кандидат.

Нека S е множеството от елементите на входния масив $A[1..n]$. Нека

$$L = \{x \in S : x < m\}$$

$$G = \{x \in S : x > m\}$$

Да намерим долна граница за $|L|$. На Фигура 6.10 всички елементи, показани върху оранжев полупрозрачен фон, са в L . Това са трите най-малки елементи от всеки B_i , за $1 \leq i \leq t-1$, и двата най-малки елементи от B_t . L може да има още елементи: някои от зелените елементи на $B_{t+1}, \dots, B_q$ може да са елементи на L , но това не е сигурно. Само за тези от оранжевата

област е **гарантирано**, че са в L . Броят на елементите от оранжевата област е

$$3 \left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 3 \left\lfloor \frac{\frac{n}{5}-1}{2} \right\rfloor + 2 = 3 \left\lfloor \frac{n}{10} - \frac{1}{2} \right\rfloor + 2 \approx \frac{3n}{10}$$

Апроксимирайки с $\frac{3n}{10}$, не променяме асимптотиката, но правим сметките по-лесни.

И така, изведохме, че $\frac{3n}{10} \leq |L|$. Напълно аналогично, $\frac{3n}{10} \leq |G|$.

Ключовото наблюдение е, че максималният размер на масива, върху който се прави рекурсивно викане (ред 16 или ред 18), е приблизително n минус получената долна граница $\frac{3n}{10}$; тоест, $n - \frac{3n}{10} = \frac{7n}{10}$. Това е минималният брой елементи, които **гарантирано няма** да бъдат част от множеството, върху което се извършва рекурсивно викане. Следователно, на ред 16 или ред 18 рекурсивното викане ще бъде върху вход с размер **най-много** $\frac{7n}{10}$.

Вече можем да построим израз за сложността по време. Имайки предвид, че рекурсивното викане на ред 11 е върху вход с големина приблизително $\frac{n}{5}$, а извън рекурсивните викания, SELECT извършва линейна по време работа, изразът е:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \quad (6.6)$$

Но Теорема 39 е приложима към (6.6) с $k = 2$, $b_1 = 5$ и $b_2 = \frac{10}{7}$. Наистина, $b_1, b_2 > 1$ и $\frac{1}{b_1} + \frac{1}{b_2} < 1$, понеже $\frac{1}{5} + \frac{7}{10} = \frac{9}{10}$. Съгласно Теорема 39, (6.6) има решение $T(n) \asymp n$. $\square$

Допълнение 39: Решение на $T(n) = T(n/5) + T(7n/10) + n$ чрез Akra-Bazzi

Уравнение (6.6) може да се реши лесно и чрез теоремата на Akra-Bazzi (Теорема 43). В термините на Теорема 43, $a_1 = a_2 = 1$, $b_1 = \frac{1}{5}$, $b_2 = \frac{7}{10}$ и $g(n) = n$. Числото p е уникалното число, такова че

$$a_1 \cdot b_1^p + a_2 \cdot b_2^p = 1 \leftrightarrow \left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1$$

С Maple(TM) можем да получим апроксимация на p :

```
> evalf(solve((1/5)^p + (7/10)^p = 1, p));
memory used=3.2MB, alloc=40.3MB, time=0.11
```

0.8397803045

И така, $p \approx 0.83978$. Конкретната стойност няма значение – важното е, че $0 < p < 1$, така че $1 - p > 0$. Съгласно Теорема 43,

$$\begin{aligned} T(n) &\asymp n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}} du \right) = n^p \left(1 + \int_1^n \frac{du}{u^p} \right) \\ &= n^p \left(1 + \frac{u^{1-p}}{(1-p)} \Big|_1^n \right) = n^p \left(1 + \frac{n^{1-p}}{(1-p)} - \underbrace{\frac{1^{1-p}}{(1-p)}}_{\text{константа}} \right) \\ &\asymp n^p \left(1 + \frac{n^{1-p}}{(1-p)} \right) \asymp n^p + n^1 \asymp n^1 \end{aligned}$$

Допълнение 40: Когато константата в алгоритъма SELECT не е 5

Читателят може да се пита, какво е значението на избора на 5 като числото, спрямо което е конструиран алгоритъмът SELECT. Това, че 5 е нечетно число, прави еднозначен изборът на медиана във всеки от масивите $B_1, \dots, B_{q-1}$ (ред 8). Но какво ще стане, ако константата е 3 или 7? А ако е четно число? Тук ще изследваме две възможности за нея—да е 4 или 7—които ще покажат убедително, че 5 е граничната стойност за нея, при която SELECT остава линеен алгоритъм. Ако тази константа е по-малка от 5, SELECT не е линеен, а ако е по-голяма от 5, дори да е четна, SELECT е линеен. Приемаме за очевидно, че коректността на SELECT не зависи от стойността на тази константа; тя се отразява само върху сложността по време.

Нека константата е 4. Да наречем получения алгоритъм SELECT4. Той започва така:

```

SELECT4( $A[1 \dots n]$ : int;  $k \in \{0, \dots, n-1\}$ )
1  (*  $A[x] \neq A[y]$  за  $1 \leq x < y \leq n$  *)
2  if  $n = 1$ 
3      return  $A[1]$ 
4   $q \leftarrow \lfloor \frac{n}{4} \rfloor$ ,  $r \leftarrow n \bmod 4$ 
5  if  $r = 0$ 
6      разбий  $A$  на подмасиви  $B_1, \dots, B_q$ , всеки с 4 елемента
7  else
8      разбий  $A$  на подмасиви  $B_1, \dots, B_{q-1}$ , всеки с 4 елемента, и  $B_q$  с  $r$  елемента

```

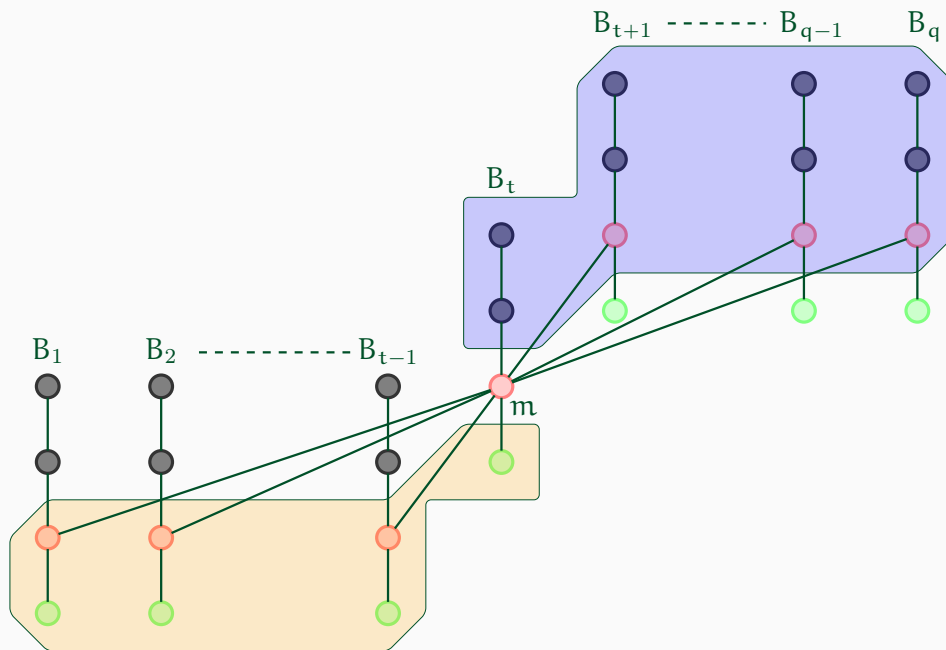
Нататък алгоритъмът е същият. Да анализираме сложността по време. БОО, нека n ератно на 4, така че всички масиви B_i са с по точно 4 елемента. При това положение не можем да дефинираме медианата на B_i като средния елемент, понеже такъв няма. Не е добра идея да дефинираме медианата като средното аритметично на втория и третия елемент. Медианата трябва да е **елемент, който се среща**. БОО, дефинираме, че $B_i[2]$ е медианата на B_i . Тогава за всяко $i \in \{1, 2, \dots, q\}$, медианата m_i е гарантирано по-голяма от точно един елемент и е гарантирано по-малка от точно два елемента на B_i . Виждаме известна асиметрия между броя на по-малките и по-големите от медианите елементи, дължаща се на асиметричното разположение на медианите в B_i .

И в SELECT4 применливата m получава стойността на медианата на медианите на ред 10, а най-лошият случай се получава, когато $k \neq j$ и правим едното от рекурсивните викания SELECT($A[1 \dots j]$, k) или SELECT($[j+2 \dots n]$, $k-j-1$). Най-лошият случай и в този алгоритъм се получава, когато масивът, върху който се прави рекурсивното викане, е с максимален размер.

БОО, нека

- $B_{\lfloor \frac{q+1}{2} \rfloor}$ е масивът, от който идва m ; нека $t = \lfloor \frac{q+1}{2} \rfloor$,
- медианите на $B_1, \dots, B_{t-1}$ са по-малки от m ,
- а медианите на $B_{t+1}, \dots, B_q$ са по-големи от нея.

При това положение, Фигура 6.11 показва диаграмата на Hasse за елементите на A след като m стане медианата на медианите.

Фигура 6.11 : Диагр. на Hasse след като m стане медианата на медианите.

Медианата на медианите m е медианата на B_t . Елементите, гарантирано по-малки от m , са показани върху оранжев полупрозрачен фон. Елементите, гарантирано по-големи от m , са показани върху син полупрозрачен фон.

Ще намерим долна граница за броя на елементите, които са по-малки от m . На Фигура 6.10 това са елементите, показани върху оранжев полупрозрачен фон. Това са точно два елемента от всеки от масивите $B_1, \dots, B_{t-1}$, плюс още един елемент от B_t . Може да има и други елементи, по-малки от m —някои от зелените елементи на $B_{t+1}, \dots, B_q$ —но само за тези от оранжевата област е гарантирано, че са по-малки от m . Броят на елементите от оранжевата област е точно

$$2 \left\lfloor \frac{q-1}{2} \right\rfloor + 1 = 2 \left\lfloor \frac{\frac{n}{4}-1}{2} \right\rfloor + 1 \approx \frac{n}{4}$$

Апроксимираме сложния израз с $\frac{n}{4}$, игнорирайки адитивната константа $+1$, нотацията $\lfloor \cdot \rfloor$ и -1 в числителя. Лесно се вижда, че това не променя асимптотиката на алгоритъма, който конструираме.

За разлика от алгоритъма SELECT, тук броят на елементите, гарантирано по-големи от m , е различен. Това очевидно се дължи на несиметричното разположение на медианите в B_i . Броят на елементите от синята област е точно

$$3 \left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 3 \left\lfloor \frac{\frac{n}{4}-1}{2} \right\rfloor + 2 \approx \frac{3n}{8}$$

Апроксимираме сложния израз с $\frac{3n}{8}$, игнорирайки адитивната константа $+1$, нотацията $\lfloor \cdot \rfloor$ и -1 в числителя.

За нашия анализ вземаме по-малкото от $\frac{n}{4}$ и $\frac{3n}{8}$, защото анализираме най-лошия случай. Така че вземаме $\frac{n}{4}$ и това е минималният брой елементи, които **гарантирано няма** да бъдат част от множеството, върху което се извършва рекурсивно викане. С други думи, рекурсивното викане ще бъде върху вход с размер **най-много** $\frac{3n}{4}$.

И така, в най-лошия случай се налага да викаме рекурсивно върху масив от $\frac{3n}{4}$ елемента. Тогава

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n$$

Това рекурентно уравнение има решение $T(n) \asymp n \lg n$, което може да покажем по индукция, а може и чрез теоремата на Акра-Bazzi. Имаме $a_1 = a_2 = 1$, $b_1 = \frac{1}{4}$, $b_2 = \frac{3}{4}$ и $g(n) = n$. Числото p е уникалното число, такова че

$$a_1 \cdot b_1^p + a_2 \cdot b_2^p = 1 \leftrightarrow \left(\frac{1}{4}\right)^p + \left(\frac{3}{4}\right)^p = 1$$

Не ни трябва Maple, за да сметнем, че $p = 1$. Оттук

$$\begin{aligned} T(n) &\asymp n \left(1 + \int_1^n \frac{g(u)}{u^2} du\right) = n \left(1 + \int_1^n \frac{du}{u}\right) \\ &= n(1 + \ln n) \asymp n \lg n \end{aligned}$$

И така, сложността по време на SELECT4 е $\Theta(n \lg n)$.

Нека константата е 7. Сега ще прескочим въведението в алгоритъма и направо ще сметнем сложността по време в най-лошия случай. Точната долна граница за броя на елементите, които гарантирано няма да бъдат част от множеството, върху което се извършва рекурсивно викане, е

$$4 \left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 4 \left\lfloor \frac{\frac{n}{7}-1}{2} \right\rfloor + 2 \approx \frac{2n}{7}$$

Тогава рекурсивното викане ще бъде върху вход с размер най-много $\frac{5n}{7}$.

Сложността в най-лошия случай се описва от рекурентното уравнение

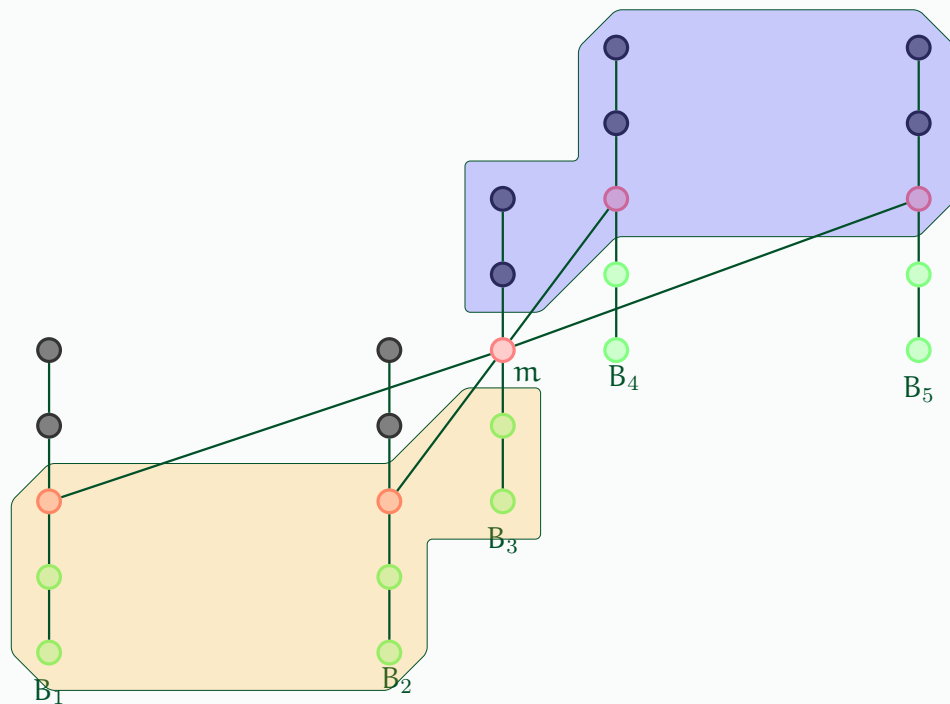
$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + n$$

Теорема 39 е приложима, понеже $\frac{1}{7} + \frac{5}{7} < 1$. Съгласно Теорема 39, това уравнение има решение $T(n) \asymp n$. И така, SELECT остава линеен алгоритъм, ако константата, спрямо която е конструиран, е 7.

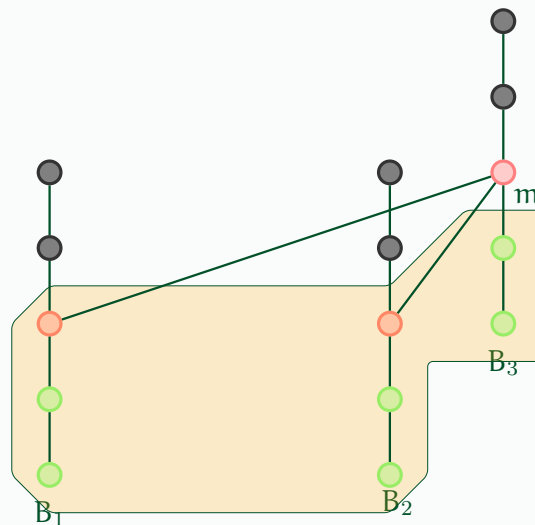
Допълнение 41: Най-лош случай за първото викане в SELECT

Разглеждаме примера за вход на SELECT на стр. 381. В него, медианата на медианите се оказа 11, а медианата е 13. Колко най-далече може да е медианата на медианите m , получена на ред 10, от 13? Отговорът е 9. Ето защо.

Да си припомним диаграмата на Hasse на данните след m . Сега $n = 25$, $q = 5$ и диаграмата изглежда така:



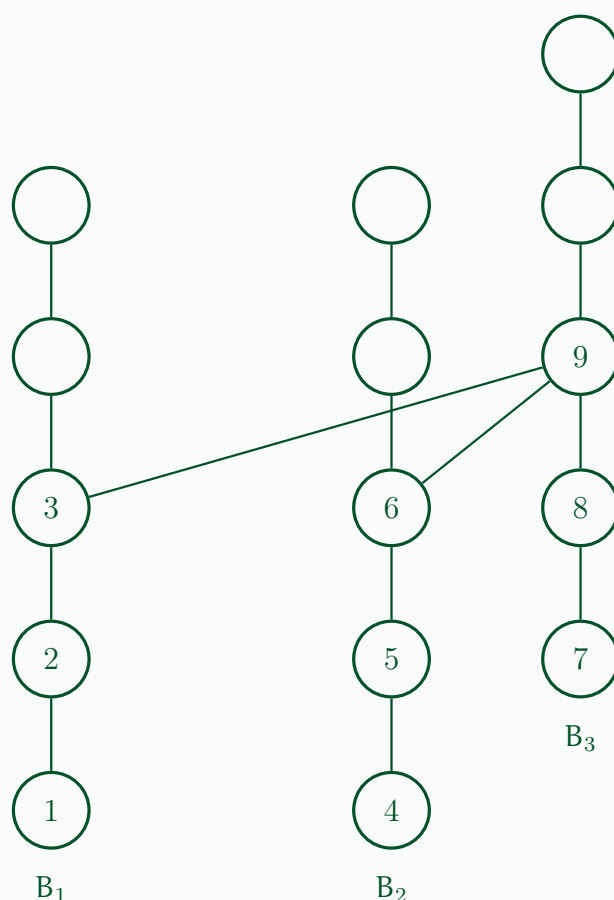
БОО, да се фокусираме върху елементите, по-малки от m :



m е задължително по-голямо число от трите най-малки елемента на B_1 , от трите най-малки елемента на B_2 , и от двата по-малки от него елемента на B_3 , но само толкова. Възможно е всеки друг елемент да е по-голям от m . Така че точна долна граница за броят на елементите, по-малки от m , е $3 + 3 + 2 = 8$. Това може да се изведе и чрез формулата от лекционните записки

$$3 \left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 3 \left\lfloor \frac{5-1}{2} \right\rfloor + 2 = 3 \cdot 2 + 2 = 8$$

Тази ситуация е постижима, ако $m = 9$. Въпросните осем елемента са числата $1, \dots, 8$. Ето едно тяхно възможно разположение:



За да се окажат по този начин числата $1, \dots, 9$ след сортирането на $B_1, \dots, B_5$ и намирането на медианата от медианите, достатъчно е

- 1, 2 и 3 да са в B_1 поначало,
- 4, 5 и 6 да са в B_2 поначало,
- 7, 8 и 9 да са в B_3 поначало.

Всички други числа са по-големи от тези и наистина няма значение кое от тях (по-големите от 9) в кой масив B_i се намира.

За да се получи желаното “разпределение” на $1, \dots, 9$ в на B_1, B_2 и B_3 преди сортирането на масивите B_i , достатъчно е във входния A , числата 1, 2, 3 да са в първата петица, 4, 5, 6 да са във втората петица, а 7, 8, 9 да са в третата петица. С други думи, A от входа да е

$$A = [1, 2, 3, *, *, 4, 5, 6, *, *, 7, 8, 9, *, *, *, *, *, *, *, *, *, *, *]$$

“*” означава “кое да е число, по-голямо от 9”.

6.3 MERGESORT

Сортиращият алгоритъм MERGESORT е открит от John von Neumann през 1945 г. [94]. Той е типичен пример за алгоритъм, изграден по схемата **Разделяй-и-Владей**.

- Ако размерът на входа n е по-голямо от някаква прагова стойност n_0 , във фазата **Разделяй**, MERGESORT дели входа на две равни части, всяка с размер $\frac{n}{2}$ [†], без да променя наредбата на елементите. Във фазата **Владей** прави по едно рекурсивно викане върху всяка от двете части. Във фазата **Комбинирай** всяка от двете части вече е сортирана, така че алгоритъмът слива (откъдето идва и името, *сливам* в този смисъл е *to merge* на английски) двете сортирани части в една окончателна сортирана редица. Акцентът е върху третата фаза, където се извършва истинската работа по сортирането.
- Ако размерът на входа не надхвърля n_0 , то се задейства спирачката на рекурсията и входът бива сортиран по някакъв елементарен начин, примерно с INSERTION SORT (Алгоритъм 5).

Първо ще видим пример за сортиране с MERGESORT. Примерът нарочно използва вход с големина, която е точна степен на двойката, но кодът, който разглеждаме нататък, работи коректно за всяка големина. И така, нека входът е:

5 2 3 1 4 8 7 6

С жълт фон означаваме тази част от масива, която е вход на текущо активното рекурсивно извикване. В самото начало тази част съвпада с целия масив.

Тъй като големината на входа е повече от единица, делим входа на две равни части и правим по едно рекурсивно извикване върху всяка от тях. Тези две рекурсивни извиквания не се случват едновременно, защото нашият изчислителен модел не позволява паралелизъм. Да речем, че първо извикваме рекурсивно алгоритъма върху масива вляво.

5 2 3 1 4 8 7 6

Отново, с жълт фон означаваме само тази част от началния масив, която е вход на текущото рекурсивно викане. Понеже големината на този вход е четири, което е по-голямо от едно, пак делим входа на две и правим две рекурсивни викания, първото върху частта вляво:

5 2 3 1 4 8 7 6

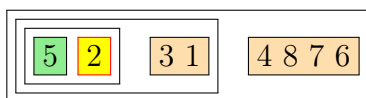
Тъй като големината на входа на текущото рекурсивно викане е две, което е по-голямо от едно, пак делим входа на две и правим две рекурсивни викания, първото върху частта вляво:

5 2 3 1 4 8 7 6

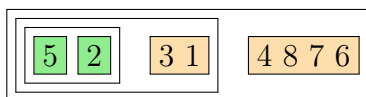
Сега вече големината на входа на текущото рекурсивно викане е едно, и алгоритъмът не прави нищо в това извикване и връща директно управлението на предното извикване, което

[†]В действителност разделянето е на една част с размер $\lfloor \frac{n}{2} \rfloor$ и друга част с размер $\lceil \frac{n}{2} \rceil$.

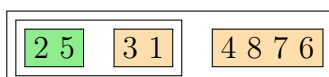
на свой ред вика алгоритъма върху другия подмасив с големина едно:



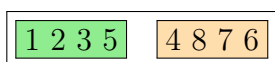
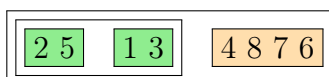
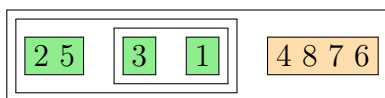
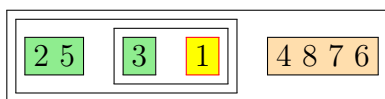
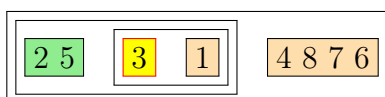
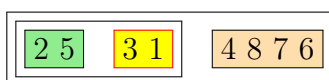
Светлозеленият фон означава, че върху този подмасив вече е достигнато дъното на рекурсията. Отново големината на текущия вход (жълт фон) е едно и алгоритъмът директно връща управлението на горното ниво, бидейки привършил и с двете викания



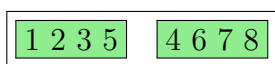
Сега алгоритъмът слива двата сортирани подмасива с големини единици в един сортиран подмасив с големина две:



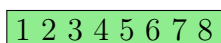
Изчислението продължава така:



Прескачаме няколко стъпки, които съответстват на второто рекурсивно викане върху 4 8 7 6, и разглеждаме масива след приключване на това второ рекурсивно викане:



Сливането на двата сортирани подмасива води до



Сега ще дадем псевдокод на функцията, която осъществява сливането. Нейната коректност е условна: тя работи коректно при условие, че двете половини на входния ѝ масив са сортирани. Функцията използва ∞ като пазач (Определение 61).

Алгоритъм 12: Merge

```

MERGE( $A[1..n]$ : int;  $l, mid, h$ : цели числа, такива че  $1 \leq l \leq mid < h \leq n$ )
1  (* подмасивите  $A[l..mid]$  и  $A[mid+1..h]$  са сортирани *)
2   $n_1 \leftarrow mid - l + 1$ 
3   $n_2 \leftarrow h - mid$ 
4  създай  $L[1..n_1 + 1]$  и  $R[1..n_2 + 1]$ 
5   $L[1..n_1] \leftarrow A[l..mid]$ 
6   $R[1..n_2] \leftarrow A[mid+1..h]$ 
7   $L[n_1 + 1] \leftarrow \infty$ 
8   $R[n_2 + 1] \leftarrow \infty$ 
9   $i \leftarrow 1$ 
10  $j \leftarrow 1$ 
11 for  $k \leftarrow l$  to  $h$ 
12   if  $L[i] \leq R[j]$ 
13      $A[k] \leftarrow L[i]$ 
14      $i++$ 
15   else
16      $A[k] \leftarrow R[j]$ 
17      $j++$ 

```

Теорема 71: Коректността на MERGE

Да допуснем, че подмасивите $A[l..mid]$ и $A[mid+1..h]$ във входа на MERGE са сортирани. След термилирането на MERGE, масивът $A[l..h]$ се състои точно от елементите на входните $A[l..mid]$ и $A[mid+1..h]$, но в сортиран вид.

Доказателство: Приемаме за очевидно, че при първото достигане на ред 11, масивите L и R са точни копия на съответно $A[l..mid]$ и $A[mid+1..h]$, плюс един sentinel накрая.

Инвариант 19: MERGE

част i Всеки път, когато изпълнението на MERGE е на ред 11, $A[l..k-1]$ съдържа $k-l$ най-малки елемента на L и R , в сортиран вид.

част ii Освен това, $L[i]$ и $R[j]$ са най-малките елементи съответно в L и R , които още не са копирани в A .

База. Когато изпълнението е на ред 11 за пръв път е вярно, че $k = l$. Тогава подмасивът $A[l..k-1]$ всъщност е $A[l..l-1]$, тоест е празен. Празният масив е сортиран и съдържа $k-l = 0$ най-малки елемента на L и R , в сортиран вид. Освен това, $L[1]$ и $R[1]$ са най-малките елементи съответно в L и R , които все още не са копирани в A .

Поддръжка. Нека твърдението е в сила при някое достигане на ред 11, което не е последното. Има два алтернативни начина, по които изпълнението може да премине през тялото на цикъла. Ще ги разгледаме и двата. Преди да направим това обаче, ще докажем едно важно помощно твърдение, Лема 30. Забележете, че и L , и R съдържат стойност ∞ , така че трябва да сме сигурни, че двете стойности ∞ не биват сравнявани, понеже сравнение на две ∞ стойности не е дефинирано.

Лема 30

В сравнението на ред 12, не може едновременно $L[i]$ и $R[j]$ да са ∞ .

Доказателство: Да допуснем противното. Съгласно индуктивното предположение, $k - l$ най-малки елемента на L и R вече са копирани в A . Щом изпълнението е на ред 12, то $k \leq h$. Ерго, броят на копираните елементи е $\leq h - l$.

Очевидно е, че щом на ред 12 сравняваме ∞ с ∞ , всички останали елементи на L и R вече са копирани. В L и R има точно $n_1 + n_2$ елемента, които не са ∞ . Следователно, броят на копираните елементи е поне $n_1 + n_2 = mid - l + 1 + h - mid = h - l + 1 > h - l$. $\downarrow$

Да разгледаме сравнението на ред 12. Тъй като няма как и $L[i]$, и $R[j]$ да са ∞ , резултатът от сравнението е дефиниран. Първо да попуснем, че $L[i] \leq R[j]$. Очевидно, $L[i] < \infty$. Съгласно **част ii** от индуктивното предположение и допускането, че $L[i] \leq R[j]$, заключаваме, че $L[i]$ е най-малкият елемент както в L , така и в R , който все още не е копиран. Съгласно **част i** на индуктивното предположение, $L[i]$ не е по-малък от никой елемент на $A[l \dots k-1]$. Изпълнението отива на ред 13. Забелязваме, че $A[k]$ не е по-малък от нито един елемент на $A[l \dots k-1]$ и заключаваме, че $A[l \dots k]$ е сортиран и съдържа $k - l + 1 = (k + 1) - l$ на брой най-малки елемента на L и R . Но k бива инкрементирано при следващото достигане на ред 11. Спрямо новата стойност на k е вярно, че $A[l \dots k-1]$ съдържа $k - l$ най-малки елемента на L and R , в сортиран вид. И така, **част i** на инварианта остава в сила.

Сега ще докажем и **част ii** на инварианта. По допускане, L и R са сортирани. Преди присвояването на ред 13, $L[i]$ беше най-малък елемент от L , който все още не е копиран в A . След това присвояване, $L[i + 1]$ е най-малък елемент от L , който все още не е копиран в A . Но променливата i бива инкрементирана при следващото достигане на ред 14. По отношение на новата стойност на i , $L[i]$ е най-малък елемент от L , който все още не е копиран в A .

Сега да допуснем, че изпълнението все още е на ред 12 и $L[i] \not\leq R[j]$, тоест $L[i] > R[j]$. Доказателството е напълно аналогично на току-що направеното.

Терминация. Променливата k съдържа $h + 1$ при последното достигане на ред 11. Заместваме тази стойност в инварианта и получаваме “подмасивът $A[l \dots h]$ съдържа $h - l + 1$ най-малки елемента на L и R , в сортиран вид”. Тъй като L и R съдържат елементите на входния $A[l \dots h]$ и двата елемента ∞ , заключаваме, че $h - l + 1$ най-малки елемента на L и R са точно елементите на входния $A[l \dots h]$. Тогава текущият $A[l \dots h]$ съдържа точно елементите на входния $A[l \dots h]$, но в сортиран вид. $\square$

Алгоритъм 13: Mergesort

```

MERGESORT( $A[1 \dots n]$ : int;  $l, h$ : индекси в  $A$ )
1  if  $l < h$ 
2       $mid \leftarrow \lfloor \frac{l+h}{2} \rfloor$ 
3      MERGESORT( $A, l, mid$ )
4      MERGESORT( $A, mid + 1, h$ )
5      MERGE( $A, l, mid, h$ )

```

Теорема 72: Коректността на MERGESORT

Алгоритъм MERGESORT е коректен сортиращ алгоритъм, ако началното извикване е MERGESORT($A, 1, n$).

Доказателство: По индукция по разликата $h - l$ [†]. Смятаме за очевидно, че $h - l$ може да стане най-малко нула, но не и по-малко, и че базата на нашето доказателство е $h - l = 0$.

База. Нека $h - l = 0$, тоест $h = l$. Масивът $A[l..h]$ е едноелементен. От една страна, едноелементният масив $A[l]$ е тривиално сортиран. От друга страна, MERGESORT не прави нищо, когато $h = l$. Така че масивът е сортиран в края на алгоритъма.

Поддръжка. Допускаме, че MERGESORT сортира коректно подмасивите $A[l..mid]$ и $A[mid + 1..h]$ (редове 3 и 4) при всички рекурсивни викания, такива че $h > l$. От Теорема 71 следва, че в края на работата на текущото рекурсивно извикване, целият $A[l..h]$ е сортиран.

Терминация. Когато доказваме коректност на рекурсивни алгоритми по индукция, стъпката **Терминация** се отнася до приключването на изпълнението на началното викане. В този случай, началното викане е MERGESORT($A, 1, n$). С MERGESORT, тази стъпка е тривиална: просто забелязваме, че когато алгоритъмът приключи работата си, $A[1..n]$ е сортиран. $\square$

Сложността по време на MERGESORT е решението на рекурентното уравнение:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad (6.7)$$

Решението, съгласно втория случай на Мастър теоремата, е $T(n) \asymp n \lg n$.

Сложността по памет MERGESORT е $\Theta(n)$, но само ако се имплементира грамотно. Ако имплементацията на MERGE е съгласно Алгоритъм 12, то MERGESORT би имал сложност по памет $\Theta(n \lg n)$, защото на всяко ниво на рекурсията би се алокирала общо $\Theta(n)$ допълнителна памет от всички изпълнения на ред 4 на MERGE в това ниво на рекурсията, а нивата на рекурсията на MERGESORT са $\Theta(\lg n)$ на брой.

MERGESORT е стабилен сортиращ алгоритъм, защото на ред 12 във функцията MERGE има нестрого неравенство. Поради това, ако биват сравнявани два елемента (от L и от R) с равни ключове, елементът от L ще се окаже вляво от този от R в окончателната подредба.

Допълнение 42: Бързо намиране на броя на инверсиите в масив.

Определение 63: инверсия в масив

Нека $A[1..n]$ е масив от цели числа. *Инверсия* в A е всяка двойка индекси $\langle i, j \rangle$, такива че $1 \leq i < j \leq n$ и $A[i] > A[j]$.

Като прост пример да разгледаме $A = [5, 1, 3, 2, 4]$. Инверсиите са $\langle 1, 2 \rangle$, $\langle 1, 3 \rangle$, $\langle 1, 4 \rangle$, $\langle 1, 5 \rangle$ и $\langle 3, 4 \rangle$.

Очевидно максимален брой инверсии се достига тогава и само тогава, когато елементите на масива са два по два различни и масивът е сортиран обратно. В такъв случай има точно $\frac{n(n-1)}{2}$ инверсии. Минимален брой инверсии се достига при сортиран масив. В такъв случай инверсиите са точно 0.

Забележете приликата между Определение 63 и Определение 59. Посоката на неравенството в Определение 59 е обратната: там се иска $A[i] < A[j]$, но това е, защото Определение 59 е в контекста на максимални пирамиди (вж. Определение 56). Ако ставаше дума за минимални пирамиди, неравенството в Определение 59 щеше да е $A[i] > A[j]$,

[†]Забележете, че това не е доказателство с инвариант на цикъла, понеже MERGESORT не е итеративен, а е рекурсивен, алгоритъм.

също както в Определение 63. Изключвайки несъществената подробност за посоката на неравенството, забелязваме, че в някакъв смисъл Определение 59 е частен случай на Определение 63, тъй като в Определение 59 се иска $A[i]$ да е предшественик на $A[j]$, което е частен случай на изискването $i < j$ в Определение 63. И така, Наблюдение 46 е аналог на Наблюдение 39.

Наблюдение 46

Масивът от цели числа A е сортиран тогава и само тогава, когато няма нито една инверсия. □

Разглеждаме задачата, за даден масив $A[1..n]$ да се изчисли във време $O(n \lg n)$ броят на инверсиите. Наивния алгоритъм не се признава, понеже е квадратичен.

COUNT-INVERSIONS-NAIVE($A[1..n]$: int)

```

1  c ← 0
2  for i ← 1 to n - 1
3      for j ← i + 1 to n
4          if A[i] > A[j]
5              c ++
6  return c
```

Ще решим задачата с алгоритъм, изграден по схемата Разделяй-и-Владей. Ако разбием A на два подмасива L и R , то броят на инверсиите е сумата от

- броят на инверсиите в L ,
- броят на инверсиите в R и
- броят на инверсиите $\langle i, j \rangle$, такива че i е индекс в L и j е индекс в R .

Заради ефикасността искаме L и R да имат колкото се може по-близки големина. Това обяснява защо алгоритъмът INVERSIONCOUNT, който предлагаме, е малка модификация на MERGESORT. За разлика от MERGESORT той връща стойност.

INVERSIONCOUNT($A[1..n]$: int; l, h : индекси в A)

```

1  if l ≥ h
2      return 0
3  else
4      mid ← ⌊ $\frac{l+h}{2}$ ⌋
5      x ← INVERSIONCOUNT( $A, l, mid$ )
6      y ← INVERSIONCOUNT( $A, mid + 1, h$ )
7      z ← MERGE-MODIFIED( $A, l, mid, h$ )
8      return x + y + z
```

Имайки предвид горните съображения за броя на инверсиите, коректността на алгоритъма е очевидна, ако MERGE-MODIFIED връща броя на инверсиите $\langle i, j \rangle$, такива че $i \in \{l, \dots, mid\}$ и $j \in \{mid + 1, \dots, h\}$. MERGE-MODIFIED е малка модификация на MERGE на стр. 393.

```

MERGE-MODIFIED( $A[1 \dots n]$ : int;  $l, mid, h$ :  $1 \leq l < mid < h \leq n$ )
1  (* подмасивите  $A[l \dots mid]$  и  $A[mid + 1 \dots h]$  са сортирани *)
2   $n_1 \leftarrow mid - l + 1$ 
3   $n_2 \leftarrow h - mid$ 
4  създай  $L[1 \dots n_1 + 1]$  и  $R[1 \dots n_2 + 1]$ 
5   $L[1 \dots n_1] \leftarrow A[l \dots mid]$ 
6   $R[1 \dots n_2] \leftarrow A[mid + 1 \dots h]$ 
7   $L[n_1 + 1] \leftarrow \infty$ 
8   $R[n_2 + 1] \leftarrow \infty$ 
9   $i \leftarrow 1$ 
10  $j \leftarrow 1$ 
11  $c \leftarrow 0$ 
12 for  $k \leftarrow l$  to  $h$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] \leftarrow L[i]$ 
15      $i++$ 
16   else
17      $A[k] \leftarrow R[j]$ 
18      $c \leftarrow c + n_1 - i + 1$ 
19      $j++$ 
20 return  $c$ 

```

Обосновката на коректността на MERGE-MODIFIED е обосновката на MERGE с добавена аргументация, свързана с брояча c . Преди да направим по-формална обосновка ще дадем интуитивно обяснение. Винаги при проверката на ред 13:

- Ако $L[i] \leq R[j]$, то $\langle L[i], R[j] \rangle$ не е инверсия. Нещо повече, тъй като R е сортиран, $\langle L[i], R[j + 1] \rangle$, $\langle L[i], R[j + 2] \rangle$, $\dots$, $\langle L[i], R[n_2] \rangle$ също не са инверсии. Тогава броят на инверсиите в $L[i \dots n_1] \parallel R[j \dots n_2]$ е равен на броя на инверсиите в $L[i + 1 \dots n_1] \parallel R[j \dots n_2]$, където “ $\parallel$ ” е буква за операцията конкатенация на масиви.

Тогава това, че инкрементиране i на ред 15, без да променяме c , е правилно.

- Ако $L[i] > R[j]$, $\langle L[i], R[j] \rangle$ е инверсия. Нещо повече, тъй като L е сортиран, $\langle L[i + 1], R[j] \rangle$, $\langle L[i + 2], R[j] \rangle$, $\dots$, $\langle L[n_1], R[j] \rangle$ също са инверсии. Тогава броят на инверсиите в $L[i \dots n_1] \parallel R[j \dots n_2]$ е равен на броя на инверсиите в $L[i \dots n_1] \parallel R[j + 1 \dots n_2]$ плюс $n_1 - i + 1$. Тук $n_1 - i + 1$ е броят на елементите на масива $L[i \dots n_1]$.

Тогава това, че инкрементиране j на ред 19 и добавяме $n_1 - i + 1$ към c на ред 18, е правилно.

6.4 QUICKSORT

QUICKSORT е открит от Sir Charles Antony Richard Hoare [75] в началото на 1960-те (вж. [73], [74], [75]). В много случаи, той е най-бързият сортиращ алгоритъм за истински изчисления—а не в асимптотичния смисъл—и оттам идва и името му. Реалните имплементации са по-сложни от псевдокода, който разглеждаме тук.

QUICKSORT е типичен за схемата Разделяй-и-Владей алгоритъм, макар да е много раз-

личен от MERGESORT. Най-общо казано, идеята е да бъде избрана някаква стойност, която се нарича *pivot*, и спрямо нея масивът да бъде пренареден така, че в лявата част (може и да е празна, това зависи от *pivot*) да са само елементи, по-малки или равни на *pivot*, а вдясно от тях, само елементи, по-големи от *pivot*. Каква точно е наредбата в лявата и дясната част няма значение – не се иска при това пренареждане да се постигне сортиран масив, това би било прекалено силно изискване. Сортиране ще се получи чак след края на рекурсивните викания. На този етап искаме просто вляво да са “малките” елементи, а вдясно от тях, “големите”. Това пренареждане е фазата **Разделяй** на алгоритъма. След това алгоритъмът бива викан рекурсивно върху всяка от тези части, ако тя е достатъчно голяма. Това е фазата **Владей**. Фазата **Комбинирай** е празна: след приключването на извикванията, масивът е сортиран. Виждаме една основна разлика между QUICKSORT и MERGESORT. При MERGESORT фазата **Разделяй** е тривиална и същината на алгоритъма е във фазата **Комбинирай**. При QUICKSORT фазата **Разделяй** е същината на алгоритъма, докато фазата **Комбинирай** е празна.

Изборът на *pivot*, както ще видим, е от огромно значение за сложността по време. Не е задължително *pivot* да е елемент от масива. Простите имплементации избират за *pivot* елемент от масива, и то по прост начин, например най-левия или най-десния елемент.

Допълнение 43: За избора на *pivot* в QUICKSORT.

Говорейки не съвсем строго, в идеалния случай стойността на *pivot* е такава, че половината елементи са по-малки от него, а другата половина, по-големи от него. С други думи, ако *pivot* е елемент от масива, то идеалният *pivot* е медианата.

Нещо повече. Както ще видим след малко, много неудачен избор на *pivot*, по-точно серия от много неудачни избори на *pivot*, може да доведе до това, че сложността по време на QUICKSORT да дегенерира до $\Theta(n^2)$, което го прави драстично по-бавен от HEAPSORT и MERGESORT в най-лошия случай. Както също ще видим след малко, такава крайно неудачна серия от избори на *pivot* е малко вероятна и средната сложност по време на QUICKSORT е $\Theta(n \lg n)$.

Възниква въпросът – не може ли да изчисляваме медианата и да избираме нея за *pivot*, с което сложността по време на QUICKSORT да стане $\Theta(n \lg n)$ в най-лошия случай? Вече видяхме в Подсекция 6.2.2, че медианата може да се намери в линейно време. Говорейки **чисто теоретично** и мислейки за сложността по време **само в асимптотичния смисъл**, изчисляването на *pivot* в линейно време няма да увеличи сложността нито в най-лошия, нито в средния случай. Защо тогава не изчисляваме идеалния *pivot* – медианата?

Отговорът е, че QUICKSORT е алгоритъм от голям практически интерес. Той “бие” останалите сортиращи алгоритми, дори бързите HEAPSORT и MERGESORT. Поради това ние се интересуваме не само от асимптотичните оценки на бързодействието му, а и от реалното бързодействие на неговите софтуерни имплементации.

На практика, забавянето, до което би довело намирането на медианата, би направило QUICKSORT напълно безсмислен. HEAPSORT и MERGESORT са бързи сортиращи алгоритми. Фолклорът казва, че QUICKSORT не е повече от, горе-долу, два пъти по-бърз от кой да е от тях (вижте *тази статия на D.Abhyankar, M.Ingle* или *този проект на Rashmi Raj*). Ако променим QUICKSORT по такъв начин, че първо да намираме медианата и после *pivot* да е медианата, той ще стане значително по-бавен от HEAPSORT или MERGESORT и ще престане да бъде от практически интерес, въпреки че сложността му в най-лошия случай, в асимптотичния смисъл, ще се подобри от $\Theta(n^2)$ на $\Theta(n \lg n)$.

6.4.1 Имплементация на фазата Разделяй чрез PARTITION–НОАРЕ

Фазата **Разделяй** на QUICKSORT се реализира чрез функция, която е широко известна под името PARTITION. Ние вече видяхме една имплементация на тази функция (Алгоритъм 10), но за пълнота на тукашното изложение се правим, че виждаме функцията за първи път сега.

Очевидно PARTITION може да се имплементира в линейно време, като това е и долна граница (защото всеки елемент от масива трябва да бъде “прегледан” при това пренареждане). Ключовото наблюдение е, че освен това, тя може да бъде реализирана само с **константна допълнителна памет**, тоест in-place. Известни са няколко имплементации на оригиналната PARTITION, която е открита от Ноаре. Ето как изглежда изглежда функцията PARTITION в [35, стр. 154]. Там тя е описана с **repeat–until** цикъл, който се отличава от **do–while** цикъл само по това, че **repeat–until** “върти”, докато постусловието е лъжа, докато **do–while** “върти”, докато постусловието е истина; оттук постусловието “ $A[j] \leq pivot$ ” от [35, стр. 154] е транслирано в “ $A[j] > pivot$ ” тук.

PARTITION–НОАРЕ($A[1 \dots n]$: цели числа; l, h : индекси в A , такива че $l < h$)

```

1  pivot ←  $A[l]$ 
2   $i \leftarrow l - 1$ 
3   $j \leftarrow h + 1$ 
4  while TRUE do
5      do
6           $j--$ 
7          while  $A[j] > pivot$ 
8      do
9           $i++$ 
10     while  $A[i] < pivot$ 
11     if  $i < j$ 
12         swap( $A[i], A[j]$ )
13     else
14         return  $j$ 
```

Грубо казано, i и j са индекси, които “вървят” един срещу друг, съответно отляво надясно и отдясно наляво, докато не “открият” елементи от масива X и Y , които са неправилно разположени спрямо избора на $pivot$ в смисъл, че $X \geq pivot$ и $Y \leq pivot$, но X е вляво от Y . Тези елементи се разменят и i и j продължават да “вървят” един срещу друг, докато не се разминат, след което желаното пренареждане е постигнато.

Съгласно предложението псевдокод, ходовете на j надолу и на i нагоре **спират**, ако достигнат елемент, равен на $pivot$. От общи съображения човек може да помисли, че ходовете надолу или нагоре трябва да **продължават** през елементи, равни на $pivot$. На пръв поглед това ще спести излишни размени на еднакви елементи. Тази “оптимизация” обаче не е добра идея, както е показано в [тази онлайн лекция в Университета Princeton](#), защото върху масив от еднакви елементи тя води автоматично до квадратичен алгоритъм, а даденият псевдокод работи почти винаги във време $\Theta(n \lg n)$.

Ето илюстрация на работата на Partition–Ноаре. В някакъв смисъл, алгоритъмът разбива масив $A[l \dots h]$ на три подмасива–зони, някои от които, но не всички, може да са празни:

- **Зелената зона:** $A[l \dots i - 1]$. Там са елементи, по-малки или равни на $pivot$.
- **Сивата зона:** $A[i \dots j]$. Това е неизследваната част от масива.

- **Жълтата зона:** $A[j + 1 .. h]$. Там са елементи, по-големи или равни на *pivot*.

Да кажем, че входът е този:

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

След изпълнението на редове 1, 2 и 3, нещата изглеждат така:

pivot = 5

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

i

j

Започва изпълнението на външния **while** цикъл (редове 4–14). Започва изпълнението на първия **do-while** цикъл (редове 5–7) и то завършва по следния начин. Червеният фон означава елемент, който подлежи на swap-ване, така че няма да остане там.

pivot = 5

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

i

j

Започва изпълнението на втория **do-while** цикъл (редове 8–10) и то завършва по следния начин. И тук червеният фон означава елемент, който подлежи на swap-ване, така че няма да остане там. Забележете, че първото изпълнение на втория **do-while** цикъл (редове 8–10) винаги изпълнява тялото на цикъла само веднъж, понеже за *pivot* е избран $A[l]$.

pivot = 5

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

i

j

Тъй като $i < j$, условието на ред 11 е истина и на ред 12, $A[i]$ и $A[j]$ се разменят.

pivot = 5

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

i

j

Отново се изпълнява тялото на външния **while** цикъл (редове 4–14). Изпълняват се първият **do-while** цикъл (редове 5–7) и вторият **do-while** цикъл (редове 8–10) и при следващото достигане на ред 11, нещата изглеждат така:

pivot = 5

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6

i

j

Тъй като $i < j$, условието на ред 11 е истина и на ред 12, $A[i]$ и $A[j]$ се разменят.

pivot = 5

5	0	4	9	2	8	0	1	9	0	2	3	5	7	5	6

i

j

При следващото достигане на ред 11, нещата изглеждат така:

$pivot = 5$

5	0	4	9	2	8	0	1	9	0	2	3	5	7	5	6
i											j				

Тъй като $i < j$, условието на ред 11 е истина и на ред 12, $A[i]$ и $A[j]$ се разменят.

$pivot = 5$

5	0	4	3	2	8	0	1	9	0	2	9	5	7	5	6
i											j				

При следващото достигане на ред 11, нещата изглеждат така:

$pivot = 5$

5	0	4	3	2	8	0	1	9	0	2	9	5	7	5	6
i											j				

Тъй като $i < j$, условието на ред 11 е истина и на ред 12, $A[i]$ и $A[j]$ се разменят.

$pivot = 5$

5	0	4	3	2	2	0	1	9	0	8	9	5	7	5	6
i											j				

При следващото достигане на ред 11, нещата изглеждат така:

$pivot = 5$

5	0	4	3	2	2	0	1	9	0	8	9	5	7	5	6
i								j							

Тъй като $i < j$, условието на ред 11 е истина и на ред 12, $A[i]$ и $A[j]$ се разменят.

$pivot = 5$

5	0	4	3	2	2	0	1	0	9	8	9	5	7	5	6
i									j						

Отново се изпълнява тялото на външния **while** цикъл (редове 4–14). Изпълнява се първият **do-while** цикъл (редове 5–7), но тялото му се изпълнява само веднъж, понеже след ред 6, $A[j]$ става 0 и условието на ред 7 се оказва лъжа. Нещата изглеждат така:

$pivot = 5$

5	0	4	3	2	2	0	1	0	9	8	9	5	7	5	6
i									j						

Изпълнява се вторият вторият **do-while** цикъл (редове 8–10), но тялото му се изпълнява само веднъж, понеже след ред 9, $A[i]$ става 9 и условието на ред 10 се оказва лъжа. Нещата изглеждат така:

$pivot = 5$

5	0	4	3	2	2	0	1	0	9	8	9	5	7	5	6
l								j	i						h

Изпълнението отива на ред 11. Условието на **if**-а е лъжа и изпълнението отива на ред 14, където PARTITION-НОАРЕ връща j и терминира.

В този пример се оказва вярно, че в края на алгоритъма, $A[l \dots i-1]$ се състои от елементи, по-малки или равни на $pivot$, а $A[j+1 \dots h]$ се състои от елементи, по-големи или равни на $pivot$. Индексът j , който алгоритъмът връща, е достатъчен, за да опише разбиването, понеже сивата зона в края на изпълнението е празна (няма неизследвана част от масива). Забележете, че елементът $pivot$ се намира на позиция, която няма нищо общо с j : петицата, която стана $pivot$ в началото, сега е на предпоследна позиция.

Забележете още нещо. На страница 399 ние дефинирахме зелената, сивата и жълтата зона, но това всъщност са твърдения, което трябва да се докажат, тъй като дефинициите обвързват конкретни подмасиви с конкретни възможни стойности за елементите в тях по отношение на $pivot$. Тези твърдения са верни не в произволни моменти от изпълнението, а при изпълненията на ред 11; тоест, когато двата вложени **do-while** са приключили току-що.

Да разгледаме само достиганията на ред 11 от примера, който видяхме преди малко. Тези достигания са точно шест на брой. Ето ги:

 $pivot = 5$

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
i														j	

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
i												j			

5	0	4	9	2	8	0	1	9	0	2	3	5	7	5	6
i											j				

5	0	4	3	2	8	0	1	9	0	2	9	5	7	5	6
				i					j						

5	0	4	3	2	2	0	1	9	0	8	9	5	7	5	6
								i	j						

5	0	4	3	2	2	0	1	0	9	8	9	5	7	5	6
l								j	i						h

Формалното доказателство за коректността на PARTITION-НОАРЕ е доста тежко и затова се намира в Допълнение 44.

Допълнение 44: Коректността на PARTITION-НОАРЕ

Няма нужда да се доказва, че мултимножеството от елементите на A се запазва по време на работата на алгоритъма, защото единственото място в алгоритъма, на което се променя масивът, е ред 12, а там променянето става чрез размяна на елементи.

Първо разглеждаме едно единствено, произволно изпълнение на **while** цикъла (редове 4–14) и формулираме и доказваме Лема 31 и Лема 32 по отношение на него. В рамките на това изпълнение на **while** цикъла (редове 4–14), нека t_1 е моментът на първото достигане на ред 5, t_2 е моментът на последното достигане на ред 7, t_3 е моментът на първото достигане на ред 8 и t_4 е моментът на последното достигане на ред 10. Очевидно $t_1 < t_2 < t_3 < t_4$.

За всеки момент t от изпълнението, което разглеждаме, дефинираме множествата $\mathcal{L}^t$ и $\mathcal{R}^t$ спрямо състоянието на масива A в момента t и стойностите на i и j в момента t :

$$\mathcal{L}^t = \{q \in \{l, \dots, j-1\} \mid A[q] \leq pivot\} \quad (6.8)$$

$$\mathcal{R}^t = \{q \in \{i+1, \dots, h\} \mid A[q] \geq pivot\} \quad (6.9)$$

Следното твърдение е очевидно и е сложено заради пълнотата на доказателството.

Лема 31: Ефектът от първия do-while

Нека $\mathcal{L}^{t_1} \neq \emptyset$ и $z = \max \mathcal{L}^{t_1}$ в момента t_1 . Тогава, в момента t_2 е вярно, че $j = z$.

Доказателство: Следното твърдение е инвариант за първия **do-while** цикъл. Приемаме за очевидно, че масивът не се променя по време на работата на този цикъл.

Инвариант 20

При всяко достигане на ред 7, $j \geq z$.

База: Разглеждаме момента t_1 : първото достигане на ред 5. По конструкция, $z \leq j-1$ спрямо текущото j (6.8). След това j се декрементира на ред 6. Спрямо текущото j е вярно, че $z \leq j$. Инвариантът е в сила. ✓

Поддръжка: Да допуснем, че инвариантът е в сила при някое достигане на ред 7, което не е последното. Щом не е последното, в този момент е изпълнено $A[j] > pivot$. Но $A[z] \leq pivot$ по конструкция (6.8). От това следва, че $j > z$. Щом $A[j] > pivot$, тялото на цикъла се изпълнява, j се декрементира на ред 6 и при следващото достигане на ред 7, спрямо новото j отново е вярно, че $j \geq z$.

Терминация: Разглеждаме момента t_2 . Очевидно е, че той съществува, понеже $\mathcal{L}^{t_1}$ от момента t_1 е непразно. Щом изпълнението е на ред 7 за последен път (в текущото изпълнение на външния **while** цикъл!), трябва да е изпълнено $A[j] \not> pivot$, тоест $A[j] \leq pivot$. От това следва и вече доказания факт, че $j \geq z$, следва, че $j = z$. С което доказателството на Лема 31 приключи. □

Следствие 24: на Лема 31

Ако в момента t_1 е вярно, че $i < z$, то в момента t_2 е вярно, че $i < j$.

Доказателство: Щом в t_1 имаме $i < z$, а t_2 имаме $j = z$ и i не е променяна между t_1 и t_2 , в сила е $i < j$ в t_2 . □

Следното твърдение е очевидно и е сложено заради пълнотата на доказателството.

Лема 32: Ефектът от втория **do-while**

Нека $\mathcal{R}^{t_3} \neq \emptyset$ и $\alpha = \min \mathcal{R}^{t_3}$ в момента t_3 . Тогава, в момента t_4 е вярно, че $i = \alpha$.

Доказателство: Следното твърдение е инвариант за втория **do-while** цикъл. Приемаме за очевидно, че масивът не се променя по време на работата на този цикъл.

Инвариант 21

При всяко достигане на ред 10, $i \leq \alpha$.

База: Разглеждаме момента t_3 : първото достигане на ред 8. По конструкция, $\alpha \geq i+1$ спрямо текущото i (6.9). След това i се инкрементира на ред 9. Спрямо текущото i е вярно, че $\alpha \geq i$. Инвариантът е в сила. ✓

Поддръжка: Да допуснем, че инвариантът е в сила при някое достигане на ред 10, което не е последното. Щом не е последното, в този момент е изпълнено $A[i] < pivot$. Но $A[\alpha] \geq pivot$ по конструкция (6.9). От това следва, че $i < \alpha$. Щом $A[i] < pivot$, тялото на цикъла се изпълнява, i се инкрементира на ред 9 и при следващото достигане на ред 10, спрямо новото i отново е вярно, че $i \leq \alpha$.

Терминация: Разглеждаме момента t_3 . Очевидно е, че той съществува, понеже $\mathcal{R}^{t_3}$ от момента t_3 е непразно. Щом изпълнението е на ред 7 за последен път (в текущото изпълнение на външния **while** цикъл!), трябва да е изпълнено $A[i] \nless pivot$, тоест $A[j] \geq pivot$. От това следва и вече доказания факт, че $i \leq \alpha$, следва, че $i = \alpha$. С което доказателството на Лема 32 приключи. □

Следствие 25: на Лема 31

Ако в момента t_3 е вярно, че $\alpha \leq j$, то в момента t_4 е вярно, че $i \leq j$.

Доказателство: Щом в t_3 имаме $\alpha \leq j$, а t_4 имаме $i = \alpha$ и j не е променяна между t_3 и t_4 , в сила е $i \leq j$ в t_4 . □

Теорема 73: Коректността на PARTITION-HOARE

Нека $A[1..n]$, където $n \geq 1$, е произволен входен целочислен масив. Изпълнението на PARTITION-HOARE(A, l, h), където $1 \leq l < h \leq n$, достига до ред 14 и тогава е в сила

$$\forall x \in A[l..j] \forall y \in A[j+1..h] : x \leq y$$

Доказателство: Следното твърдение е инвариант за **while** цикъла на редове 4–14.

Инвариант 22: Външният цикъл на PARTITION–HOARE

При всяко достигане на ред 11 на PARTITION–HOARE:

- ① $\forall x \in A[l \dots i - 1] : x \leq pivot.$
- ② $\forall x \in A[j + 1 \dots h] : x \geq pivot.$
- ③ $i \leq j + 1.$
- ④ $i = j \rightarrow A[i] = A[j] = pivot.$
- ⑤ $i < j \rightarrow \exists x \in A[l \dots j - 1] : x \leq pivot \wedge \exists x \in A[i + 1 \dots h] : x \geq pivot.$

База:

Поддръжка:

Терминация:



Известни са няколко вариации на PARTITION–HOARE, една от които е следната. Тя е описана в [155, стр. 16]. Тази функция ползва *sentinel*, който е $-\infty$ и се намира в $A[0]$. Числата, които ще се сортират, са в $A[1 \dots n]$. В оригинала вътрешните цикли са **repeat ... until**. За удобство при четене, тук те са сменени с **do ... while** цикли.

PARTITION–HOARE–ANOTHER($A[0 \dots n]$: масив; l, h : индекси в A , такива че $l < h$)

```

1  (* Данните за сортиране са  $A[1 \dots n]$ .  $A[0]$  е  $-\infty$ . *)
2   $pivot \leftarrow A[h]$ 
3   $i \leftarrow l - 1$ 
4   $j \leftarrow h$ 
5  while TRUE do
6    do
7       $j--$ 
8      while  $A[j] > pivot$ 
9        do
10        $i++$ 
11       while  $A[i] < pivot$ 
12       if  $i < j$ 
13          $swap(A[i], A[j])$ 
14       else
15         break
16    $swap(A[i], A[h])$ 
17   return  $i$ 
```

Разлика между PARTITION–HOARE и PARTITION–HOARE–ANOTHER е, че втората връща позицията на *pivot* и сме сигурни, че този елемент си е на мястото.

Пак swap-ваме елемент със себе си.

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp, i														h

След което се случва това:

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp, i														h

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp, i														h

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp, i														h

Това е първото достигане на ред 4, при което не е вярно, че $A[i] < pivot$. Поради това редове 5 и 6 не се изпълняват, така че swap не се извършва и pp “изостава” от i при следващото достигане на ред 4.

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp	i													h

Условието на ред 4 отново е истина, но сега $pp \neq i$, така на ред 5 се прави истински swap на различни елементи.

5	0	5	9	2	8	0	1	9	0	2	3	4	7	5	6
l	pp	i													h

5	0	5	2	9	8	0	1	9	0	2	3	4	7	5	6
l	pp	i													h

pp се инкрементира на ред 6 и после i се инкрементира неявно при изпълнението на ред 3. При следващото достигане на ред 4 имаме:

5	0	5	2	9	8	0	1	9	0	2	3	4	7	5	6
l	pp	i													h

Но $A[i] \nless pivot$, така че условието на ред 4 е лъжа и при следващото достигане на ред 4:

5	0	5	2	9	8	0	1	9	0	2	3	4	7	5	6
l	pp	i													h

Сега $A[i] < pivot$ и предстои swap:

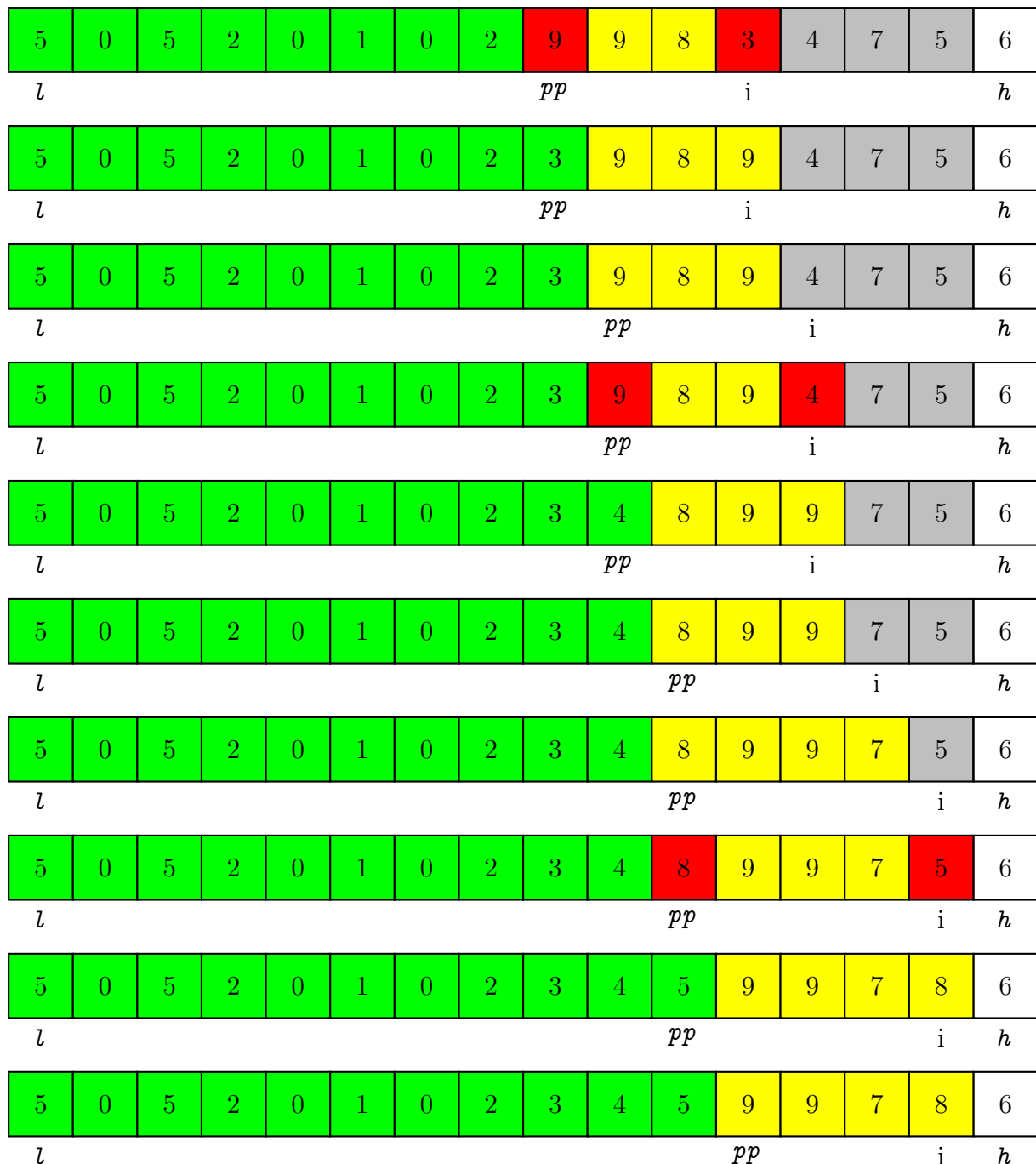
5	0	5	2	9	8	0	1	9	0	2	3	4	7	5	6
l				pp		i									h
5	0	5	2	0	8	9	1	9	0	2	3	4	7	5	6
l				pp		i									h

pp се инкрементира и при следващото достигане на ред 4:

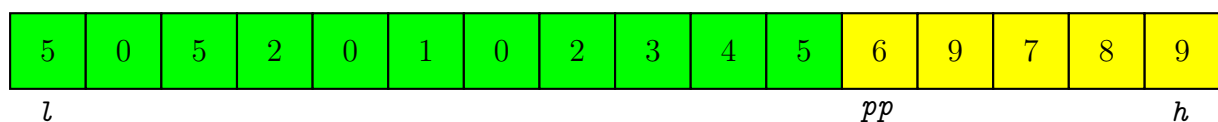
5	0	5	2	0	8	9	1	9	0	2	3	4	7	5	6
l					pp		i								h

Следва

5	0	5	2	0	8	9	1	9	0	2	3	4	7	5	6
l					pp		i								h
5	0	5	2	0	1	9	8	9	0	2	3	4	7	5	6
l					pp		i								h
5	0	5	2	0	1	9	8	9	0	2	3	4	7	5	6
l						pp		i							h
5	0	5	2	0	1	9	8	9	0	2	3	4	7	5	6
l						pp		i							h
5	0	5	2	0	1	9	8	9	0	2	3	4	7	5	6
l						pp		i							h
5	0	5	2	0	1	0	8	9	9	2	3	4	7	5	6
l						pp		i							h
5	0	5	2	0	1	0	8	9	9	2	3	4	7	5	6
l							pp		i						h
5	0	5	2	0	1	0	8	9	9	2	3	4	7	5	6
l							pp		i						h
5	0	5	2	0	1	0	2	9	9	8	3	4	7	5	6
l							pp		i						h
5	0	5	2	0	1	0	2	9	9	8	3	4	7	5	6
l							pp		i						h



for-цикълът терминара и изпълнението отива на ред 7, където се разменят $A[pp]$ и $A[h]$:



След което алгоритъмът терминара, връщайки pp на ред 8.

Ето формалното доказателство за коректността на PARTITION-LOMUTO.

Теорема 74: Коректността на PARTITION-LOMUTO

Ефектът от работата на PARTITION-LOMUTO е следният. Елементът $A[h]$ във входния масив е *pivot*. Алгоритъмът пренарежда елементите на входа така, че *pivot* се оказва в клетка $A[pp]$ и освен това:

$$\forall x \in A[l \dots pp - 1] : x < A[pp]$$

$$\forall x \in A[pp + 1 \dots h] : x \geq A[pp]$$

Освен това, алгоритъмът връща именно *pp*.

Доказателство: Следното твърдение е инвариант на **for**-цикъла (редове 3-6).

Инвариант 23: PARTITION-LOMUTO

При всяко достигане на ред 3 на PARTITION-LOMUTO:

$$\forall x \in A[l \dots pp - 1] : x < pivot$$

$$\forall x \in A[pp \dots i - 1] : x \geq pivot$$

База. Когато изпълнението е на ред 3 за първи път, $i = l$ и $pp = l$. Инвариантът е:

$$\forall x \in A[l \dots l - 1] : x < pivot$$

$$\forall x \in A[l \dots l - 1] : x \geq pivot$$

И двете твърдения са верни, в празния смисъл. ✓

Поддръжка. Да допуснем, че инвариантът е верен при някое достигане на ред 3, което не е последното.

Случай I $A[i] < pivot$. Ползваме индуктивното предположение и заключаваме, че:

$$\forall x \in A[l \dots pp - 1] : x < pivot \wedge A[i] < pivot \quad (6.10)$$

Условието на ред 4 е истина и изпълнението отива на ред 5. Там се извършва размяна на $A[i]$ и $A[pp]$. Имайки предвид тази размяна, (6.10) става:

$$\forall x \in A[l \dots pp] : x < pivot \quad (6.11)$$

След инкрементирането на *pp* на 6, (6.11) става:

$$\forall x \in A[l \dots pp-1] : x < pivot \quad (6.12)$$

Виждаме, че първата част на инварианта се запазва. За да се убедим, че и втората част е вярна, да “върнем лентата” назад до момента, в който изпълнението беше на ред 4. От индуктивното предположение знаем, че

$$\forall x \in A[pp \dots i - 1] : x \geq pivot \quad (6.13)$$

След размяната на $A[i]$ и $A[pp]$, която става на ред 5, в сила е:

$$\forall x \in A[pp + 1 \dots i] : x \geq pivot \quad (6.14)$$

След инкрементиранията на pp на 6 и неявното инкрементиране на i при следващото достигане на ред 3, спрямо новите стойности на тези две променливи, (6.14) става:

$$\forall x \in A[pp \dots i - 1] : x \geq pivot \quad (6.15)$$

Случай II $A[i] \geq pivot$. От индуктивното предположение знаем, че:

$$\forall x \in A[l \dots pp - 1] : x < pivot$$

$$\forall x \in A[pp \dots i - 1] : x \geq pivot$$

Тогава:

$$\forall x \in A[l \dots pp - 1] : x < pivot$$

$$\forall x \in A[pp \dots i] : x \geq pivot$$

Но условието на ред 4 е лъжа и изпълнението отива на ред 3, като i се инкрементира неявно. Спрямо новата стойност на i е вярно, че:

$$\forall x \in A[l \dots pp - 1] : x < pivot$$

$$\forall x \in A[pp \dots i - 1] : x \geq pivot$$

Терминация. При последното достигане на ред 3 е вярно, че $i = h$. Тогава

$$\forall x \in A[l \dots pp - 1] : x < pivot$$

$$\forall x \in A[pp \dots h - 1] : x \geq pivot$$

Доказахме инварианта и видяхме ефекта от работата на цикъла. Но доказателството за коректността все още не е готово. Изпълнението отива на ред 7, където се разменят $A[pp]$ и $A[h]$, който всъщност е $pivot$. Сега вече е вярно, че $pivot$ се намира в клетка $A[pp]$ и освен това е вярно, че

$$\forall x \in A[l \dots pp - 1] : x < A[pp]$$

$$\forall x \in A[pp + 1 \dots h] : x \geq A[pp]$$

После алгоритъмът връща pp . □

6.4.3 Самият QUICKSORT

Алгоритъм 15 ползва PARTITION, който поставя $pivot$ на окончателната му позиция, така че този елемент не е в масивите в рекурсивните викания. Такъв PARTITION е PARTITION-LOMUTO (Алгоритъм 14) или PARTITION-HOARE-ANOTHER на стр. 405.

Алгоритъм 15: Quicksort

QUICKSORT($A[1 \dots n]$: цели числа; l, h : индекси в A)

1 if $l < h$

2 $mid \leftarrow \text{PARTITION}(A, l, h)$

3 QUICKSORT($A, l, mid - 1$)

4 QUICKSORT($A, mid + 1, h$)

Ако обаче искаме да ползваме PARTITION–HOARE на стр. 399, трябва да направим рекурсивните викания върху подмасиви, които представляват разбиване на оригиналния масив; с други думи, всеки от оригиналните елементи е елемент в някой от тях.

QUICKSORT–ANOTHER($A[1 \dots n]$: цели числа; l, h : индекси в A)

```

1  (* На стр. 154 в [35] *)
2  if  $l < h$ 
3       $mid \leftarrow \text{PARTITION}(A, l, h)$ 
4      QUICKSORT( $A, l, mid$ )
5      QUICKSORT( $A, mid + 1, h$ )

```

Доказателството за коректност е напълно аналогично на доказателството за коректност на MERGESORT.

QUICKSORT не е стабилен сортиращ алгоритъм. Лесно е да се намери пример, в който PARTITION разменя взаимната наредба на два еднакви елемента. Примерът за PARTITION–LOMUTO може да се различава от примера за PARTITION–HOARE, но и за двата варианта има такива примери.

Забелязваме още една разлика между QUICKSORT и MERGESORT. Фазата **Разделяй** на QUICKSORT връща стойност, а именно индексът, който определя разделянето на масива на две части за рекурсивните викания. Това се налага съвсем естествено: за разлика от MERGESORT, сега не можем да изчислим индекса, определящ разделянето, без да сме прегледали масива.

6.4.4 Сложност по време на QUICKSORT

Сложността на QUICKSORT е изключително чувствителна към изборите на pivot. Казваме “изборите” в множествено число, защото става дума за изборите във всички рекурсивни викания. Ако във всяко отделно викане се оказва, че pivot дели съответния входен (под)масив на две равни части, сложността се описва от рекурентното уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

което, както вече видяхме при MERGESORT, има решение $T(n) \asymp n \lg n$. Ако обаче всеки избор на pivot е такъв, че този елемент дели масива на подмасив с един елемент (самият той) и друг подмасив с $n - 1$ елемента, сложността се описва от рекурентното уравнение

$$T(n) = T(n - 1) + n$$

С метода с характеристичното уравнение е лесно да се покаже, че това рекурентно уравнение има решение $T(n) \asymp n^2$.

Следователно, в най-лошия случай QUICKSORT е квадратичен сортиращ алгоритъм, с линейна сложност по памет. Ако се задоволяваме само с анализ на сложността в най-лошия случай, това е всичко, което имаме да кажем за сложността на QUICKSORT. На практика обаче QUICKSORT е по-бърз от всеки друг известен сортиращ алгоритъм, говорейки осреднено. Причината е, че лошите избори на pivot са твърде малко вероятни, ако pivot е случаен елемент от масива и големините на елементите от масива са униформно разпределени. Затова, за пръв и последен път в този курс, ще направим анализ на средната сложност на QUICKSORT.

Иначе би трябвало да класифицираме най-бързия на практика сортиращ алгоритъм като бавен.

И така, анализът на средната сложност се прави с рекурентно уравнение, което обаче отразява именно средния случай. Както се убедихме, това на какви части бива разделен входът се определя от `pivot` елемента. Ако `pivot` е случаен елемент от масива, то всяко разделяне на входа на две части е равновероятно. Отчитаме само **броя на сравненията**. Неговата асимптотика е асимптотиката на самия алгоритъм. Следното рекурентно уравнение моделира казаното дотук:

$$T(n) = \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + T(n-k) \right) + (n-1) \quad (6.16)$$

Променливата k е мястото **по големина**, а не по позиция, на `pivot` във входа. Казано по друг начин, k е “правилно място” за елемента `pivot` след разместването на елементите, направено от функцията `PARTITION`. И така, `pivot` може да се окаже или най-малък елемент ($k = 1$), или втори по големина елемент ($k = 2$), или ..., или най-голям елемент ($k = n$). Тогава рекурентното уравнение се чете така: работата на `QUICKSORT` е сумата от

- сумата по всички възможности за `pivot` ($k \in \{1, \dots, n\}$) от сумата от работата върху получени спрямо него два подмасива. Единият подмасив е с големина $k-1$, а другият е с големина $n-k$, оттам е и изразът $T(k-1) + T(n-k)$. Сумата е умножена с $\frac{1}{n}$, като този множител отразява факта, че всички n възможности за относителната големина на `pivot` спрямо останалите елементи са равновероятни.
- и $n-1$, което е броят на сравненията за определяне на мястото на `pivot`. Очевидно `pivot` трябва да бъде сравнен с всеки друг елемент и това е достатъчно.

Сега ще решим (6.16).

$$\begin{aligned} T(n) &= \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + T(n-k) \right) + (n-1) \\ &= \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + \sum_{k=1}^n T(n-k) \right) + (n-1) \\ &= \frac{1}{n} \left(\underbrace{T(0) + T(1) + \dots + T(n-1)}_{\sum_{k=1}^n T(k-1)} + \underbrace{T(n-1) + T(n-2) + \dots + T(0)}_{\sum_{k=1}^n T(n-k)} \right) + (n-1) \end{aligned} \quad (6.17)$$

Очевидно двете подчертани суми в (6.17) са равни, така че

$$T(n) = \frac{2}{n} (T(0) + T(1) + \dots + T(n-1)) + (n-1) \quad (6.18)$$

Умножаваме двете страни на (6.18) по n и получаваме

$$nT(n) = 2(T(0) + T(1) + \dots + T(n-1)) + n(n-1) \quad (6.19)$$

Но тогава, ако n е достатъчно голямо,

$$(n-1)T(n-1) = 2(T(0) + T(1) + \dots + T(n-2)) + (n-1)(n-2) \quad (6.20)$$

Изваждаме (6.20) от (6.19) и получаваме

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\ &= 2T(n-1) + 2(n-1) \quad \leftrightarrow \\ nT(n) &= (n+1)T(n-1) + 2(n-1) \end{aligned} \quad (6.21)$$

Делим двете страни на (6.21) на $n(n+1)$ и получаваме

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \quad (6.22)$$

Удобно е да направим следното полагане, за да се освободим от нотация със знаменатели: $\frac{T(n)}{n+1}$ ще бъде наричано $S(n)$. Тогава очевидно $\frac{T(n-1)}{n} = S(n-1)$ и замествайки в (6.22), получаваме следното рекурентно уравнение:

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)} \quad (6.23)$$

То не може да бъде решено чрез метода с характеристичното уравнение, защото нехомогенната част не е от вида, полином на n по константа на n -та степен, така че ще използваме универсалното средство – развиване.

$$\begin{aligned} S(n) &= S(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= S(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(n-3) + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\dots \\ &= S(1) + \frac{2 \cdot 1}{2 \cdot 3} + \frac{2 \cdot 2}{3 \cdot 4} + \frac{2 \cdot 3}{4 \cdot 5} + \dots + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(1) + 2 \sum_{k=2}^n \frac{k-1}{k(k+1)} \end{aligned} \quad (6.24)$$

Да разгледаме (6.24). $S(1)$ е константа, това е някакво начално условие, и асимптотиката на $S(n)$ се определя от $\sum_{k=2}^n \frac{k-1}{k(k+1)}$. Но

$$\sum_{k=2}^n \frac{k-1}{k(k+1)} = \sum_{k=2}^n \frac{k}{k(k+1)} - \sum_{k=2}^n \frac{1}{k(k+1)} = \sum_{k=2}^n \frac{1}{k+1} - \sum_{k=2}^n \frac{1}{k(k+1)}$$

Както знаем от Теорема 36, $\sum_{k=1}^n \frac{1}{k} \asymp \lg n$, откъдето веднага следва, че $\sum_{k=2}^n \frac{1}{k+1} \asymp \lg n$. От друга страна, $\sum_{k=2}^n \frac{1}{k(k+1)}$ е ограничена от константа, защото редът $\sum_{k=1}^{\infty} \frac{1}{k^2}$ е сходящ със сума $\frac{\pi^2}{6}$. Следва, че $\sum_{k=2}^n \frac{k-1}{k(k+1)} \asymp \lg n$. Тоест, $S(n) \asymp \lg n$. Връщаме се към нотацията $T(n)$ и получаваме, че

$$T(n) \asymp n \lg n$$

Доказахме, че QUICKSORT има средна сложност по време $\Theta(n \lg n)$. Оттук и средната слож-

ност по памет е логаритмична (а не линейна като в най-лошия случай), защото средната дълбочина на стека на рекурсията е $\Theta(\lg n)$, а на всяко ниво на рекурсията се ползва само константна допълнителна памет от функцията PARTITION.

Лекция 7

Сортиране в линейно време. COUNTING SORT. RADIX SORT.

Резюме: Демонстрираме сортиране в линейно време върху вход, чиито елементи не са произволни, а се подчиняват на определени ограничения. Въвеждаме сортиращите алгоритми COUNTING SORT и RADIX SORT

7.1 Въведение

Както ще видим в Лекция 13, сортирането на произволни елементи, до чиито стойности нямаме достъп, а само можем да сравняваме елементи един с друг, не може да става асимптотично по-бързо от $n \lg n$. Но тази долна граница не е в сила, ако има някакви ограничения за възможните стойности на елементите. Съвсем прост пример е сортирането на масив от записи, чиито ключове са нули или единици – очевидно е достатъчно да преброим колко са елементите от всеки вид, да създадем нов масив с големина, равна на големината на входния, и да прехвърлим елементите от входа в новия масив по такъв начин, че ключовете нула са преди ключовете единица, като сортирането е стабилно, и всичко това става във време $\Theta(n)$.

7.2 COUNTING SORT

Този алгоритъм третира елементите от входа не просто като числа, а като записи, чиито ключове са числа. Тъй като записите освен ключовете имат и някаква сателитна информация, сортирането става с местене на елементи, а не с просто презаписване (на числа). Дори когато казваме “входът е масив от числа”, имаме предвид, че входът е масив от записи с ключове-числа.

Входът е масив от числа $A[1 \dots n]$. Известно е, че за някакво $k \in \mathbb{N}^+$, за всяко $A[i]$ е изпълнено

$$A[i] \in \{1, 2, \dots, k\}$$

Алгоритъмът ползва работен масив $C[0 \dots k]$. Изходът е масив $B[1 \dots n]$.

Алгоритъм 16: Counting SortCOUNTING SORT($A[1..n]$): цели положителни числа; k : цяло положително число)

```

1  (*  $\forall i \in \{1, \dots, n\} : A[i] \in \{1, \dots, k\}$  *)
2  създай  $B[1..n]$  и  $C[0..k]$ 
3  for  $i \leftarrow 0$  to  $k$ 
4       $C[i] \leftarrow 0$ 
5  for  $i \leftarrow 1$  to  $n$ 
6       $C[A[i]]++$ 
7  for  $i \leftarrow 1$  to  $k$ 
8       $C[i] \leftarrow C[i] + C[i-1]$ 
9  for  $i \leftarrow n$  downto 1
10      $B[C[A[i]]] \leftarrow A[i]$ 
11      $C[A[i]]--$ 
12 return  $B$ 

```

Първо ще илюстрираме работата на алгоритъма с пример и после ще дадем формално доказателство за коректност. Нека $n = 8$, $k = 6$ и масивът A е:

A	3	6	4	1	3	4	1	4
	1	2	3	4	5	6	7	8

Очевидно първият **for**-цикъл (редове 3–4) просто нулира масива C . Вторият **for**-цикъл (редове 5–6) преброява колко елемента от всяко $i \in \{1, \dots, k\}$ има в A и записва това в C :

C	0	2	0	2	3	0	1
	0	1	2	3	4	5	6

Очевидно, $\sum_{i=0}^k C[i] = n$. Третият **for**-цикъл (редове 7–8) присвоява на всяко $C[i]$ броя на всички елементи на A , по-малки или равни на i :

C	0	2	2	4	7	7	8
	0	1	2	3	4	5	6

Очевидно, сега $C[k] = n$. Същината на алгоритъма е в четвъртия **for**-цикъл (редове 9–11). В началото $i = 8$, $A[8] = 4$, $C[4] = 7$. $B[7]$ става 4:

B						4	
	1	2	3	4	5	6	7

а след изпълнението на ред 11, C става:

C	0	2	2	4	6	7	8
	0	1	2	3	4	5	6

В червено е намаленият елемент на C.

После $i = 7$, $A[7] = 1$, $C[1] = 2$. $B[2]$ става 1:

В		1					4	
	1	2	3	4	5	6	7	8

а C става:

С	0	1	2	4	6	7	8
	0	1	2	3	4	5	6

После $i = 6$, $A[6] = 4$, $C[4] = 6$. $B[6]$ става 4:

В		1				4	4	
	1	2	3	4	5	6	7	8

а C става:

С	0	1	2	4	5	7	8
	0	1	2	3	4	5	6

И така нататък. В края на алгоритъма В е:

В	1	1	3	3	4	4	4	6
	1	2	3	4	5	6	7	8

а C е:

С	0	0	2	2	4	7	7
	0	1	2	3	4	5	6

В Определение 42 въведохме нотацията “ $\#(a, A)$ ” за броя на появите на a в мултимножеството A . Тук ще използваме нотацията “ $\#(x, Z[1..m])$ ”, за да означаваме броя на появите на x в масива $Z[1..m]$.

Сега ще докажем коректността на COUNTING SORT. Приемаме за очевидно, че масивът A не се мени от алгоритъма.

Лема 33: Ефектът от втория цикъл на COUNTING SORT

След приключването на втория **for**-цикъл (редове 5–6):

$$\forall j \in \{0, \dots, k\} : C[j] = \#(j, A[1..n])$$

Доказателство: Следното твърдение е инвариант за **for**-цикъл на редове 5–6.

Инвариант 24: Вторият for-цикъл на COUNTING SORT

За всяко достигане на ред 5, $\forall j \in \{0, \dots, k\} : C[j] = \#(j, A[1 .. i - 1])$.

База. Разглеждаме произволно $j \in \{0, \dots, k\}$. При първото достигане на ред 5, всички елементи на C са нули заради първия **for**-цикъл (редове 3–4), така че $C[j] = 0$. От друга страна, $i = 1$, така че масивът $A[1 .. i - 1]$ е празен, така че j се среща 0 пъти в $A[1 .. i - 1]$. Твърдението е вярно в празния смисъл. ✓

Поддръжка. Да допуснем, че твърдението е вярно при някое достигане на ред 5, което не е последното. Разглеждаме произволно $j \in \{0, \dots, k\}$. Разглеждаме поотделно тези две възможности за j : $j = A[i]$ и $j \neq A[i]$.

- Нека $j = A[i]$. Съгласно допускането, $C[A[i]] = \#(A[i], A[1 .. i - 1])$. На ред 6, $C[A[i]]$ се инкрементира, така че вече е в сила $C[A[i]] = \#(A[i], A[1 .. i - 1]) + 1$.
Очевидно е, че $\#(A[i], A[1 .. i]) = \#(A[i], A[1 .. i - 1]) + 1$. Тогава $C[A[i]] = \#(A[i], A[1 .. i])$.
- Нека $j \in \{0, \dots, k\} \setminus \{A[i]\}$. Съгласно допускането, $C[j] = \#(j, A[1 .. i - 1])$.
Очевидно е, че $\#(j, A[1 .. i - 1]) = \#(j, A[1 .. i])$. Тогава $C[j] = \#(j, A[1 .. i])$. Изпълнението на ред 6 не променя $C[j]$, така че $C[j] = \#(j, A[1 .. i])$ остава вярно.

Накратко, $\forall j \in \{0, \dots, k\} : C[j] = \#(j, A[1 .. i])$ след изпълнението на ред 6.

Изпълнението отива отново на ред 5, където i се инкрементира. По отношение на новото i е вярно, че $\forall j \in \{0, \dots, k\} : C[j] = \#(j, A[1 .. i])$. ✓

Терминация. Да разгледаме момента, в който изпълнението е на ред 5 за последен път. Очевидно i е $n + 1$. Заместваме i с $n + 1$ в инварианта и получаваме

$$\forall j \in \{0, \dots, k\} : C[j] = \#(j, A[1 .. n])$$

□

Следствие 26

След приключването на втория **for**-цикъл (редове 5–6), $C[0] = 0$.

Лема 34: Ефектът от третия цикъл на COUNTING SORT

След приключването на третия **for**-цикъл (редове 7–8), за всяко $j \in \{0, \dots, k\}$, $C[j]$ съдържа броя на елементите на $A[1 .. n]$, които са по-малки или равни на j .

Доказателство: Следното твърдение е инвариант за **for**-цикъл на редове 7–8.

Инвариант 25: Третият for-цикъл на COUNTING SORT

За всяко достигане на ред 7, $\forall j \in \{0, \dots, i - 1\} : C[j] = \sum_{t=0}^j \#(t, A[1 .. n])$.

База. При първото достигане на ред 7, i е 1, така че $i - 1$ е 0, така че твърдението става

$$C[0] = \#(0, A[1 .. n])$$

От Следствие 26 знаем, че $C[0] = 0$. От друга страна, $\#(0, A[1..n]) = 0$. ✓

Поддръжка. Да допуснем, че при някое достигане на ред 7, което не е последното:

$$\begin{aligned} C[0] &= \#(0, A[1..n]) \\ C[1] &= \#(0, A[1..n]) + \#(1, A[1..n]) \\ C[2] &= \#(0, A[1..n]) + \#(1, A[1..n]) + \#(2, A[1..n]) \\ &\dots \\ C[i-1] &= \#(0, A[1..n]) + \#(1, A[1..n]) + \#(2, A[1..n]) + \dots + \#(i-1, A[1..n]) \end{aligned}$$

Забелязваме, че елементът $C[i]$ не е променян засега от изпълнението на третия **for**-цикъл, така че, съгласно Лема 33, $C[i] = \#(i, A[1..n])$. След изпълнението на ред 8 е в сила

$$\underbrace{C[i]}_{\text{новото } C[i]} = \underbrace{\sum_{t=0}^{i-1} \#(t, A[1..n])}_{C[i-1]} + \underbrace{\#(i, A[1..n])}_{\text{старото } C[i]}$$

тоест,

$$C[i] = \#(0, A[1..n]) + \#(1, A[1..n]) + \dots + \#(i-1, A[1..n]) + \#(i, A[1..n])$$

Заклучаваме, че $\forall j \in \{0, \dots, i\} : C[j] = \sum_{t=0}^j \#(t, A[1..n])$.

След инкрементирането на i на ред 7, по отношение на новото i е вярно, че

$$\forall j \in \{0, \dots, i-1\} : C[j] = \sum_{t=0}^j \#(t, A[1..n])$$

Терминация. Разглеждаме момента, в който изпълнението е на ред 7 за последен път. Очевидно $i = k + 1$. Заместваме i с $k + 1$ в инварианта и получаваме

$$\forall j \in \{0, \dots, k\} : C[j] = \sum_{t=0}^j \#(t, A[1..n])$$

Имаме право да кажем, че за всяко $j \in \{1, \dots, k\}$, $C[j]$ съдържа броя на елементите на $A[1..n]$, които са по-малки или равни на j . □

Наблюдение 47

Измежду всички числа от $\{1, \dots, k\}$, които потенциално може да са елементи на A , от съществено значение са точно тези, които се срещат като елементи на входния $A[1..n]$. Те и само те се използват за индексирание в C .

Съгласно Лема 33, в края на втория **for**-цикъл, индексите $j \in \{1, \dots, k\}$, които са съществени в смисъла на Наблюдение 47, са точно тези, за които $C[j] > 0$. В примера, който видяхме на стр. 417, съществените индекси са 1, 3, 4 и 6.

Определение 64

За всеки x от $A[1..n]$, *правилното място на x* е броят на елементите от A , които са по-малки от x , плюс броя на елементите равни на x , които са вляво от x , плюс едно.

С други думи, правилното място на x е мястото, на което би го сложил стабилен сортиращ алгоритъм. Забележете, че в Определение 64, x не е само стойността на елемент от A , а и позицията на този елемент в A .

Теорема 75: COUNTING SORT е стабилен сортиращ алгоритъм

COUNTING SORT е стабилен сортиращ алгоритъм.

Доказателство: Следното твърдение е инвариант за четвъртия **for**-цикъл.

Инвариант 26: Четвъртият for-цикъл (редове 9–11)

Всеки път, когато изпълнението е на ред 9:

- ❶ за всяко $j \in \{0, \dots, k\}$, ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i]$, който има стойност j ;
- ❷ всички елементи на $A[i + 1 \dots n]$ са на своите правилни места в B .

База. Разглеждаме първото достигане на ред 9. $i = n$, а ❶ става

“за всяко $j \in \{0, \dots, k\}$, ако $j \in A[1 \dots n]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots n]$, който има стойност j ”.

Да разгледаме произволна стойност $j \in \{0, \dots, k\}$, която се среща в $A[1 \dots n]$. Нека x е най-десният елемент на $A[1 \dots n]$ със стойност j . Очевидно е, че правилното място на x е броят на елементите на $A[1 \dots n]$ със стойност, по-малка или равна на j . Но масивът C все още не е променян от четвъртия **for**-цикъл, така че Лема 34 е в сила. Според Лема 34, за всяко $j \in \{0, \dots, k\}$, $C[j]$ съдържа броя на елементите на $A[1 \dots n]$, които са по-малки или равни на j . Заключаваме, че наистина $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots n]$, който има стойност j . ✓

Да разгледаме ❷. Подмасивът $A[i + 1 \dots n] = A[n + 1 \dots n]$ е празен, така че твърдението е вярно в празния смисъл. ✓

Поддръжка. Да допуснем, че инвариантът е в сила при някое достигане на ред 9, което не е последното. Нека това достигане става в момента t' . Нека следващото следващото достигане на ред 9 е в момента t'' . Ще докажем, че инвариантът е в сила в момента t'' .

Първо ще докажем ❶. Разглеждаме момента t' . От индуктивното предположение имаме

“за всяко $j \in \{0, \dots, k\}$, ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i]$, който има стойност j ”

Разглеждаме произволно j от $\{0, \dots, k\}$, което се среща в $A[1 \dots i]$. Следните случаи са изчерпателни и взаимно изключващи се.

- $j = A[i]$. На ред 10 алгоритъмът извършва присвояването $B[C[A[i]]] \leftarrow A[i]$ (това не се използва за доказателството на ❶) и след това на ред 11 декрементира $C[A[i]]$.
 - ♦ Да допуснем, че $j \in A[1 \dots i - 1]$. Нека $\ell = \max\{1, \dots, i - 1\}$, такава че $A[\ell] = j$. Очевидно е, че новата стойност на $C[A[i]]$ е правилното място на елемента $A[\ell]$. Имаме право да кажем

“ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i - 1]$, който има стойност j ”

Но след това i се декрементира и по отношение на новото i в момента t'' пак можем да кажем

“ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i]$, който има стойност j ” ✓

- Да допуснем, че $j \notin A[1 \dots i - 1]$. Имаме право да кажем

“ако $j \in A[1 \dots i - 1]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i - 1]$, който има стойност j ”

понеже, антецедентът $j \in A[1 \dots i - 1]$ е лъжа, което прави импликацията истина, независимо от истинността на консеквента. Но след това i се декрементира и по отношение на новото i в момента t'' пак можем да кажем

“ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i]$, който има стойност j ” ✓

- $j \neq A[i]$. По отношение на това j , декрементирането на $C[A[i]]$ на ред 11 няма значение за $C[j]$, така че имаме право да кажем, че

“ако $j \in A[1 \dots i - 1]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i - 1]$, който има стойност j ”

Но след това i се декрементира и по отношение на новото i в момента t'' пак можем да кажем

“ако $j \in A[1 \dots i]$, то $C[j]$ съдържа правилното място на най-десния елемент на $A[1 \dots i]$, който има стойност j ” ✓

Сега ще докажем ②. От индуктивното предположение ② знаем, че в момента t' е вярно, че всички елементи на $A[i + 1 \dots n]$ са на своите правилни места в B .

От индуктивното предположение ① знаем, че за всеки елемент j на $\{0, \dots, k\}$, $C[j]$ съдържа правилното място в B на най-дясната му поява в $A[1 \dots i]$. Но $A[i]$ е най-дясната поява в $A[1 \dots i]$ на елемента с тази стойност. Тогава $C[A[i]]$ съдържа правилното място на $A[i]$ в B . На ред 10 се извършва присвояването $B[C[A[i]]] \leftarrow A[i]$. Имаме право да кажем, че след това присвояване, всички елементи на $A[i \dots n]$ са на своите правилни места в B .

След декрементирането на i , в момента t'' , по отношение на новата стойност на i е вярно, че всички елементи на $A[i + 1 \dots n]$ са на своите правилни места в B . □

Терминация. При завършването на цикъла, $i = 0$. Заместваме i с 0 в ② и получаваме “всички елементи от $A[1 \dots n]$ са на своите правилни места в B .” □

Анализът на сложността е тривиален: COUNTING SORT работи във време $\Theta(n + k)$. Ако $k = O(n)$, както често се случва при използването на този алгоритъм, сложността по време е $\Theta(n)$. Това е значително подобрение спрямо $\Theta(n \lg n)$ на бързите сортиращи алгоритми, базирани на сравнения. Сложността по памет е същата: $\Theta(n + k)$, а ако $k = O(n)$, тя е $\Theta(n)$. Следователно, алгоритъмът не е *in-place*. COUNTING SORT е стабилен сортиращ алгоритъм.

Следната задача е по същество като Problem 8.2-4 от [34, стр. 197]. Тя се решава елементарно, използвайки идеи от COUNTING SORT.

Задача 56

Представете си игра между агенти X и Y , в която Y разполага с n цели положителни числа $a_1, \dots, a_n$. Известно е, че съществуват $\ell, m \in \mathbb{N}^+$, такива че $\ell < m$ и освен това $\forall i \in \{1, \dots, n\} : a_i \in \{\ell, \dots, m\}$. X знае ℓ и m , но не знае числата $a_1, \dots, a_n$. Играта се състои в това, X да задава въпроси на Y от вида “Колко числа измежду $a_1, \dots, a_n$ са по-големи или равни на a и по-малки или равни на b ?”. Може да мислите, че въпросите на X се състоят само от наредени двойки (a, b) , където $\ell \leq a < b \leq m$. Y трябва да отговори коректно на всяко запитване на X в константно време.

Опишете предварителната обработка, която Y извършва.

Опишете алгоритъма, по който Y отговаря на X в константно време.

Решение: Y прави предварителна обработка, подобна на тази от COUNTING SORT. Тъй като възможните стойности за a_i не започват от единица, а от някакво ℓ , има смисъл работният масив C да е с граници $C[\ell - 1 .. m]$. Y прави следната предварителна обработка.

ALG PREPROCESS($A[1 .. n]$: цели положителни числа; ℓ, m : цели положителни числа)

```

1  (*  $\ell$  и  $m$  са такива, че  $\ell < m$  и  $\ell \leq A[i] \leq m$  за всички  $i$  *)
2  for  $i \leftarrow \ell - 1$  to  $m$ 
3       $C[i] \leftarrow 0$ 
4  for  $i \leftarrow 1$  to  $n$ 
5       $C[A[i]] \leftarrow C[A[i]] + 1$ 
6  for  $i \leftarrow \ell$  to  $m$ 
7       $C[i] \leftarrow C[i] + C[i - 1]$ 

```

Това, че алгоритъмът работи във време $O(n + (m - \ell))$ се показва по същия начин, по който се показва, че сложността по време на COUNTING SORT е $O(n + k)$. В случая $m - \ell$ играе ролята на k .

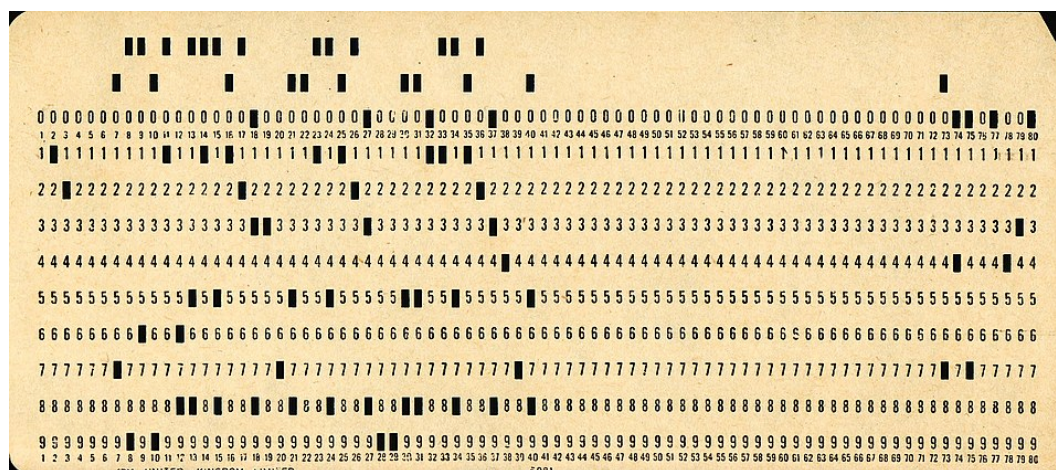
От лекции знаем, че след третия цикъл на COUNTING SORT, за всяко j , такова че $\ell \leq j \leq m$, $C[j]$ съдържа броят на елементите в масива A , които са по-малки или равни на j . Тогава $C[b] - C[a - 1]$ е броят на елементите в $w \in A$, за които

$$w \leq b \wedge \neg(w \leq a - 1) \leftrightarrow a - 1 < w \leq b \leftrightarrow a \leq w \leq b$$

Така че при запитване $\boxed{(a, b)}$, Y връща $C[b] - C[a - 1]$. Това се изчислява във време $\Theta(1)$, понеже Y разполага с масива C . $\square$

7.3 RADIX SORT

RADIX SORT е алгоритъмът, използван за сортиране на перфокарти. Перфокарта, на английски *punched card* или *Hollerith card*, е цифров носител на информация, използван преди появата на “истински” програмируеми компютри, и използван до 70те и дори 80те години на 20 век. Работещ компютър, в който информацията се въвежда с перфокарти, в днешно време едва ли може да бъде намерен извън музея, но снимки на перфокарти се намират лесно по Интернет. Ето една такава снимка, взета от [wikipedia](https://en.wikipedia.org/wiki/File:IBM_800_card_deck.jpg) с лиценз CC2.0:



Карта като тази е във формата на IBM с 12 реда и 80 колони. Във всяка колона може да бъде пробит една или повече правоъгълни дупки. Всяка колона с дупка или дупки представлява символ. Детайлно описание на IBM формата има в [тази статия](#).

Задачата е, при дадено множество карти, върху които са записани числа, картите да се сортират от електро-механично устройство, така че записаните числа да са в намаляващ ред. Електро-механичното устройство може да чете в даден момент точно един от десетичните разряди, с други думи, една от колоните, като допира вертикално разположени игли до картата и отчита коя игла потъва (през вече направената дупка), по този начин отчитайки коя е цифрата от съответния разряд.

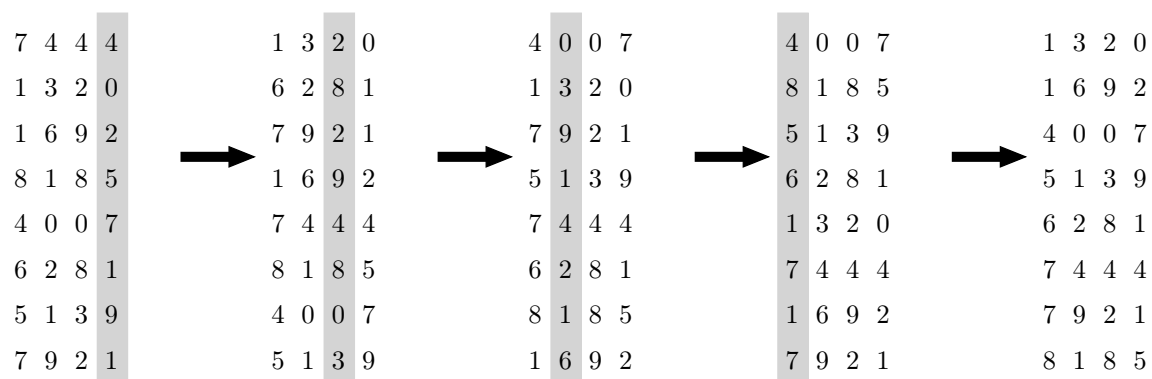
Очевидната идея за сортиращ алгоритъм, който да управлява това електро-механично устройство, е да сортираме картите по най-старшата цифра, после по следващата по старшинство, и така нататък, до най-младшата цифра. Недостатъкът на този подход е, че след сортирането по най-старша цифра, трябва да държим картите в няколко купчини[†] до края. След сортирането по втора по старшинство цифра, купчините ще станат повече[‡], и така нататък. Това не е практично – ако продължим по този начин, най-вероятно в един момент ще получим купчини от по една карта.

Практичният подход е цифрите (колони) да се тълкуват като ключове—най-старшата цифра е първичният ключ, втората по старшинство е вторичният ключ, и така нататък, до най-младшата цифра—и сортирането да се прави в посока от последния по значимост ключ към първичния ключ. Както знаем от лекцията по сортиране (Подсекция 4.3), ако сортираме цялата купчина по този начин, то ще получим правилно сортирани карти, **при условие, че сортираме със стабилен сортиращ алгоритъм**. Стабилността гарантира, че когато сортираме по колона i , ако има няколко карти с една и съща цифра в колона i , техният взаимен порядък няма да се наруши; с други думи, вече извършените сортирания по колони $i + 1$, $i + 2$ и така нататък са определили правилното им взаимно разположение.

Да видим пример за работата на RADIX SORT. Ще сортираме числата 7444, 1320, 1692, 8185, 4007, 6281, 5139 и 7921. Представаме си, че са написани едно над друго в този ред, и започваме да сортираме от колоната на най-младшите разряди към колоната на най-старшите разряди. Колоната със сив фон отбелязва по кои разряди сортираме в момента.

[†]Но не повече от десет. Купчините биха били точно десет, ако всяка от десетте цифри се появява като най-старша цифра на някоя карта.

[‡]Но не повече от сто.



Псевдокодът на RADIX SORT е съвсем прост (съгласно [34]):

RADIX SORT($A[1 \dots n]$): положителни числа, записани с едн. брой цифри; d : броят на цифрите)

1 (* най-младшата цифра е номер 1, най-старшата цифра е номер d *)

2 **for** $i \leftarrow 1$ **to** d

3 сортирай A по цифра i със стабилен сортиращ алгоритъм

Очевиден кандидат за стабилен сортиращ алгоритъм е COUNTING SORT. Коректността на RADIX SORT е очевидна, имайки предвид това, което знаем за стабилните сортирания. Ако използваме COUNTING SORT като стабилен сортиращ алгоритъм, сложността на RADIX SORT е $\Theta(d(n + k))$, където k е броят на възможните цифри. Ако k е константа, то сложността по време е $\Theta(dn)$.

Използвайки RADIX SORT, можем да решим следната задача:

Задача 57

Да се сортира в линейно време масив от n числа, всяко от което принадлежи на множеството $\{1, 2, \dots, n^2\}$.

Решение: Ако да решим задачата директно с COUNTING SORT, алгоритъмът ще работи в $\Theta(n^2)$, защото работният масив на COUNTING SORT ще бъде с размер n^2 . Но ако разгледаме записите на тези числа, примерно в двоична позиционна бройна система, ще видим, че всяко от тях се записва с $\lceil \lg n^2 \rceil + 1 = \lfloor 2 \lg n^2 \rfloor + 1 \approx 2 \lg n$ бита. Ако разделим всеки бинарен запис на две части, всяка с по $\approx \lg n$ бита, може да смятаме дясната част за младши ключ, а лявата, за старши ключ. Прилагайки RADIX SORT (който на свой ред ползва COUNTING SORT) първо по младшите ключове, а после по старшите, конструираме алгоритъм, който работи във време $\Theta(2n) = \Theta(n)$. Това, което ни помогна да направим линеен алгоритъм е, че прилагаме COUNTING SORT два пъти, но при всяко прилагане, работният масив е с линеен размер, защото и младшите, и старшите ключове са с големина $\approx \lg n$, което означава, че могат да имат най-много $\Theta(n)$ различни стойности. $\square$

Част III

Алгоритми върху графи

Лекция 8

Въведение в алгоритмите върху графи. Обхождания на графи.

Резюме: Правим кратка рекапитулация на графите. Правим сравнителен анализ на най-популярните представяния на графи. Разглеждаме основната алгоритмична задача върху графи: обхождане. Разглеждаме двата най-популярни алгоритъма за обхождане на графи: BFS и DFS.

8.1 Рекапитулация на графите

Определение 65: Граф

Граф е наредена двойка (V, E) , където V е непразно множество от *върховете*, а E е множество от *ребрата*. V е опорното множество (ground set), а E е подмножество на $\{\{x, y\} \mid \{x, y\} \subseteq V\}$.

Графите, дефинирани в Определение 65 се наричат *неориентирани*, тъй като техните ребра са множества—двуелементни множества от върхове, по-точно—а множествата нямат наредба. Ако кажем само “граф”, ще разбираме неориентиран граф.

Определение 66: Ориентиран граф

Ориентиран граф е наредена двойка (V, E) , където V е непразно множество от върховете, а E е множество от ребрата. V е опорното множество, а E е подмножество на $(V \times V) \setminus \{(v, v) \in V \times V\}$.

Определение 65 и Определение 66 не допускат *примки*; примка е ребро, на което двата края са един и същи връх. На читателя остава да прецени как да ги модифицира по такъв начин, че да се допускат примки.

Определение 67: Ориентиран мултиграф

Ориентиран мултиграф е наредена тройка (V, E, f_G) , където V е непразно множество от върхове, E е множество от ребра, а f_G е *свързващата функция*. V и E са опорни множества, а $f_G : E \rightarrow V \times V$.

Оттук насетне ще описваме мултиграфите без да споменаваме свързващата функция; примерно, $G = (V, E)$.

На читателя остава да дефинира “неориентиран мултиграф”. Когато кажем “мултиграф” без уточнения, ще разбираме ориентиран мултиграф. Под *обикновен граф* разбираме граф, който не е мултиграф.

Нотация 5: $N(u)$ и $N(U)$: съседите на u и U

Нека $G = (V, E)$ е обикновен граф, $u \in V$ и $U \subseteq V$.

$$N(u) = \{v \in V \mid u \text{ и } v \text{ са съседни}\}$$

$$N(U) = \bigcup_{x \in U} N(x) \setminus U$$

Буквата ‘N’ идва от **n**eighbourhood. В алгоритмите върху графи обаче пишем “**for** $u \in \text{adj}[u]$ ”, а не “**for** $u \in N(u)$ ”. Причината е, че нотацията “ $N(u)$ ” се ползва в контекста на теорията на графите, а “ $\text{adj}[u]$ ” има по-практически смисъл и означава списъка на съседство на връх u .

Всички определения върху графи от курса по Дискретни Структури са в сила. Тук добавяме няколко определения и нотации, които не са изучавани по Дискретни Структури.

Нотация 6: $u \rightsquigarrow v$, $u \xrightarrow{P} v$, $\text{Paths}(u, v)$

Нека $u, v \in V$. “ $u \rightsquigarrow v$ ” е кратък запис за “ориентиран път от u до v ”. “ $u \xrightarrow{P} v$ ” е кратък запис за “ P е ориентиран път от u до v ”. “ $\text{Paths}(u, v)$ ” означава $\{P \mid u \xrightarrow{P} v\}$.

Нотация 7: $\mathcal{SC}(G)$ и $\mathcal{SCC}(G)$: релацията на силна свързаност и нейните класове на екв.

Нека G е ориентиран граф. “ $\mathcal{SC}(G)$ ” означава релацията на силна свързаност над $V(G)$, а “ $\mathcal{SCC}(G)$ ” означава множеството от нейните класове на еквивалентност.

Определение 68: Даг

Ако G е ориентиран ацикличен граф, ще казваме накратко, че G е *даг*.

Името идва от английския акроним “dag” (**d**irected **a**cyclic **g**raph). Авторът на тези лекционни записки нито знае общоприет български термин за “ориентиран граф без цикли/ориентиран ацикличен граф”, нито харесва “огбц/оаг”, така че ще ползва “даг”.

Определение 69: Арборесценция и антиарборесценция

Арборесценция, на английски *arborescence* или *out-tree*, се нарича ориентиран граф, който се получава от кореново дърво—което е неориентиран граф—с корен r , като на ребрата се даде ориентация навън от корена. Алтернативна дефиниция е, ориентиран граф G , в който съществува връх r , такъв че за всеки връх $u \in V(G)$ е вярно, че има един единствен ориентиран път от r до u . Върхът r е коренът на арборесценцията. Аналогично, *антиарборесценция*, на английски *anti-arborescence* или *in-tree*, е ориентирано кореново дърво, в което ориентацията е от листата към корена.

Думата “arborescence” произхожда от латинското *arbor*, което означава дърво. Авторът на тези лекционни записки не знае общоприет български термин за “ориентирано дърво с ориентация навън от корена”, така че по неволя ще използва “арборесценция”. По същата причина

ще ползва и “антиарборесценция” Забележете, че всяка арборесценция или антиарборесценция е даг, докато обратното не е вярно.

Конвенция 10: Буквите m и n в контекста на графите

Когато разглеждаме графи, неизменно буквата n ще има смисъл на броя на върховете, а m , на броя на ребрата, освен ако изрично не е казано нещо друго.

Наблюдение 48: Ограничения за m при дадено n

Ако разглеждаме обикновен неориентиран граф без примки, то $0 \leq m \leq \binom{n}{2}$, като границите са точни. Ако разглеждаме обикновен ориентиран граф без примки, то $0 \leq m \leq n(n-1)$, като границите са точни. Ако разглеждаме обикновен ориентиран граф, който може да има примки, $0 \leq m \leq n^2$, като границите са точни. Ако разглеждаме мултиграф, няма горна граница за m , така че $0 \leq m$ е всичко, което знаем за m , ако не е дадено някакво ограничение в условието.

Основните представяния на графи, които не са мултиграфи, са следните две. Нека графът е $G = (V, E)$, $V = \{u_1, \dots, u_n\}$ и $E = \{e_1, \dots, e_m\}$.

Списъци на съседство: За всеки връх е даден списък от неговите съседи, ако става дума за неориентиран граф, или списък от неговите деца, ако става дума за ориентиран граф. Списъците са точно n . Списъците **не са подредени по никакъв начин**, както един спрямо друг, така и всеки от тях вътрешно.

Матрица на съседство: Това е булева матрица $n \times n$, където клетка (i, j) съдържа 1 тогава и само тогава, когато

- има ребро **между** u_i и u_j , ако графът е неориентиран; в този случай матрицата е симетрична.
- има ребро **от** u_i **до** u_j , ако графът е ориентиран; в този случай матрицата не е непременно симетрична.

Мултиграфите се представят със списъци на съседство по гореописания начин. Има разлика в матричното представяне между обикновени графи и мултиграфи: ако искаме да представяме мултиграф с матрица, тя трябва да е не булева, а от естествени числа, като клетка (i, j) съдържа броя на ребрата от u_i до u_j .

Наблюдение 49

Ако неориентиран граф, мултиграф или обикновен граф, е представен чрез списъци, всяко ребро (u_i, u_j) , което не е примка, се появява **два пъти**: веднъж в списъка на u_i и веднъж в списъка на u_j .

Размерът на граф—независимо от това, дали е мултиграф или обикновен граф—е

- $m + n$, ако е представен чрез списъци,
- n^2 , ако е представен чрез матрица.

Конвенция 11

Тъй като използваме списъчното представяне по-често, по подразбиране размерът на граф е $m + n$. Следователно, по подразбиране, линеен алгоритъм върху графи е такъв, който работи във време $\Theta(m + n)$.

Нотация 8: “adj[x]” означава списъка на съседство на връх x

Ако G е граф и x е връх в него, “adj[x]” означава списъка на съседство на x в списъчното представяне на G .

Действия върху графи. Графите, които разглеждаме, понякога са статични, но понякога искаме да добавяме или да изтриваме техни елементи, в който случай ще казваме, че са *динамични*[†]. Естествено, в най-общия случай бихме искали да добавяме или изтриваме както върхове, така и ребра. На практика се оказва, че значително по-лесно е при **фиксирано** множество от върхове да се добавят или премахват ребра. Такива графи—с фиксирано множество от върхове, спрямо което добавяме или изтриваме ребра—се наричат *полудинамични*. Ние ще се ограничим до действията добавяне на ребра и изтриване на ребра, което значи, че ако графите, които разглеждаме, не са статични, те са полудинамични.

Таблица 8.1 показва сложността в **най-лошия случай** на основни алгоритмични примитиви и алгоритми върху графи при матрично и при списъчно представяне. Допускаме, че е даден обикновен граф $G = (V, E)$, ориентиран или не, като $V = \{u_1, \dots, u_n\}$.

Действие	Представяне	
	чрез матрица	чрез списъци
тестване дали $(u_i, u_j) \in E$	$\Theta(1)$	$\Theta(n)$
намиране на $d(u_i)$	$\Theta(n)$	$\Theta(n)$
опит за добавяне на ребро (u_i, u_j)	$\Theta(1)$	$\Theta(n)$
опит за изтриване на ребро (u_i, u_j)	$\Theta(1)$	$\Theta(n)$
преброяване на ребрата	$\Theta(n^2)$	$\Theta(m + n)$
обхождане на графа	$\Theta(n^2)$	$\Theta(m + n)$

Таблица 8.1: Сложностите на действията върху графи при двете представяния.

Забележете, че ако графът е мултиграф, опит за добавяне на ребро отнема само $\Theta(1)$ време и при списъчното представяне, защото просто вмъкваме новото ребро в началото на списъка; в такъв случай има смисъл да говорим само за “добавяне на ребро”, а не за “опит за добавяне”, защото няма причина да не може да добавим ново ребро между/от-до съществуващи върхове. При обикновените графи такъв опит може да не е успешен, защото такова ребро може вече да има, оттам идва и сложността $\Theta(n)$ при списъчното представяне – налага се да минем през целия списък на u_i или u_j , за да се убедим, че няма ребро (u_i, u_j) .

Определение 70: Разредени графи

Нека $\mathfrak{G}$ е безкрайно множество от графи. Казваме, че $\mathfrak{G}$ са *разредени графи*, на английски *sparse graphs*, ако за всеки $G \in \mathfrak{G}$ имаме $|E(G)| = O(|V(G)|)$. По-подробно казано, съществува положителна константа c , такава че за всеки $G \in \mathfrak{G}$ имаме $|E(G)| \leq c|V(G)|$.

[†]Това няма **нищо общо** с динамичното програмиране, което ще разгледаме в Лекция 12

Забележете, че въпросът дали даден граф е разреден граф е **безсмислен въпрос**, говорейки формално. Съгласно Определение 70, въпросът е смислен само за безкрайни множества от графи, или *класове от графи*. Ето някои важни класове от графи, които са разредени.

Дърветата. Знаем, че за всяко дърво $m = n - 1$.

Планарните графи. Знаем, че за всеки планарен граф $m \leq 3n - 6$.

Графите, в които максималната степен на връх е ограничена от константа. Да си припомним, че “ $\Delta(G)$ ” означава максималната степен на връх в графа G . От курса Дискретни Структури знаем, че $\sum_{u \in V} d(u) = 2m$, което влече $\Delta(G) \cdot n \geq 2m$, а оттук $m \leq \frac{\Delta(G)}{2}n$. Ако съществува константа c , такава че $\Delta(G) \leq c$ за всеки граф от $G \in \mathfrak{G}$, то $m = O(n)$.

Наблюдение 50: Размер на граф от клас от разредени графи

Ако $\mathfrak{G}$ е клас от разредени графи, за всеки $G \in \mathfrak{G}$ е вярно, че размерът на G е $\Theta(n)$, ако използваме списъчно представяне. Ако използваме матрично представяне, размерът е $\Theta(n^2)$. Ерго, списъчното представяне е особено удачно за разредени графи.

Дали неориентираните графи са всъщност ориентирани? Забележете, че за всеки неориентиран граф, представен чрез списъци, можем да мислим като за ориентиран граф, чиито ребра са “сдвоени” в смисъл, че всяко неориентирано ребро (u, v) **всъщност** е двойка ориентирани ребра (u, v) и (v, u) . Ако възприемем тази гледна точка, можем да заключим, че **всъщност** неориентираните графи няма – всички графи са само ориентирани, а понякога на тези със “сдвоените” ребра казваме “неориентираните графи”. От тази гледна точка, понятието “неориентиран граф” е само метафора, или начин за удобно изразяване, докато на ниско ниво графите са само ориентирани. Ако гледаме само представянето със списъци, това е точно така.

Ние обаче **няма** да възприемаме тази гледна точка. Това, че представяме неориентираните графи като ориентирани със сдвоени ребра, съвсем не означава, че всъщност неориентираните графи няма. Ето някои от проблемите, до които би довело решението да отречем съществуването на неориентираните графи.

- Веднага ще следва, че няма нетривиални ациклически неориентираните графи – ако има поне едно неориентирано ребро, то всъщност е двойка ребра, които образуват ориентиран цикъл. Тогава понятието “дърво” се обезсмисля. Но интуитивно е ясно, че има гигантска разлика между “истински” неориентиран цикъл и “ерзац” цикъл от сдвоени ориентирани ребра. И че “дърво” е смислено и важно понятие.
- Алгоритми като DFS, които категоризират ребрата на графа по някакъв начин, ще категоризират двете ориентирани ребра (u, v) и (v, u) в различни класове. Но—ако става дума за неориентиран граф—бихме искали това да е само едно ребро, което да бъде категоризирано в един клас.
- Има изчислителни задачи, които по принцип са върху ориентирани графи, но може в частност да бъдат и върху неориентираните графи. Примерно, ОБХОЖДАНЕ НА ГРАФ или НАЙ-КЪС ПЪТ В ГРАФ[†]. Но задачите върху неориентираните графи като МИНИМАЛНО ПОКРИВАЩО ДЪРВО или ВЪРХОВО ПОКРИВАНЕ или ДОМИНИРАЩО МНОЖЕСТВО

[†]Както ще видим в Лекция 11, НАЙ-КЪС ПЪТ В ГРАФ върху неориентираните графи е безсмислена при наличие на отрицателни тегла, докато върху ориентирани графи има смисъл дори при отрицателни тегла, стига да няма отрицателни цикли.

са задачи именно върху неориентирани графи. Те или не може да се обобщят естествено върху ориентирани графи, или обобщенията им върху ориентирани графи имат решения със съвсем друга сложност.

Като пример, обобщението на МИНИМАЛНО ПОКРИВАЩО ДЪРВО върху ориентирани графи е **NP**-трудната задача **MINIMUM EQUIVALENT DIGRAPH**[†] [56, стр. 65].

8.2 Обхождания на графи

Обхождането е базовата изчислителна задача върху графи. На английски терминът е *graph traversal*. Понякога се използва и *graph search*, но това звучи прекалено близо до *graph searching*, което означава съвсем друга задача. Много неформално казано, обхождането на граф се извършва от някакво същество, което живее в графа; тръгвайки от **даден начален връх**, то иска да се разходи из графа по такъв начин, че да обиколи максимална част (подграф) на графа.

Съществото обхожда както върхове, така и ребра. Формулировката на задачата, която казва, че биват обхождани върхове, без да се споменават ребра, е **некоректна**. Прочее, ако няма изолирани върхове, достатъчно е да кажем, че се обхождат ребрата на графа; при липса на изолирани върхове обхождането на ребрата влече и обхождане на върховете.

Задачата се формулира за ориентирани мултиграфи с възможни примки. Това, че ще я разглеждаме и отделно като задача за неориентирани мултиграфи, не променя факта, че общата ѝ формулировка е за ориентирани мултиграфи. Паралелните ребра и примките имат значение: те трябва да бъдат обходени, всяко/всяка поотделно.

Удачно е да мислим за графа като за *лабиринт*, състоящ се от *стаи* и *коридори*, свързващи стаи; даден коридор може да свързва стая със себе си и такива коридори отговарят на примките в графа. Коридорите са еднопосочни, ако графът е ориентиран, или двупосочни, ако е неориентиран. Съществото иска да обиколи максимална част от лабиринта, минавайки през всеки коридор и посещавайки всяка стая, без да пропуска коридори/стаи и без да “зацикля”.

За да се избегне зацикляне се използва *маркировка*. Противно на житейската интуиция, според която съществото би маркирало коридорите, за да знае къде вече е било и да не зацикля, в алгоритмите-решения на тази задача се маркират **върховете**.

- В простия вариант всеки връх има две състояния, иначе казано, бива маркиран по точно един от следните два начина: **посетен** и **непосетен**. В началото всички върхове са непосетени, с изключение на стартовия връх; в неформалното обяснение с лабиринта, всички стаи в началото са непосетени с изключение на стартовата стая.

Можем да мислим, че се използват *цветове*, като непосетените стаи/върхове са бели, а посетените, черни. В началото всички стаи без стартовата са *бели*, а стартовата е *черна*. След това, когато съществото посети бяла стая, то разбира, че не е било в нея досега, и я маркира, боядисвайки я в черно, за да знае, че вече е било в нея при повторно попадане.

- В по-изтънчените решения се ползат три цвята, тоест, всяка стая има три състояния. Белите, както и в другия вариант, са непосетените. Тези, които са посетени, но с тях не е приключено, са *сиви*. Тези, с които е приключено, са черни. Да не бъде приключено с посетена стая означава, че от нея излизат коридори, по които (може би) не сме

[†]MED е дефинирана за ориентирани графи, които не са тегловни. Ясно е, че ако добавим и тегла, задачата няма да стане по-лесна.

минавали. Със стая приключваме, когато сме убедени, че от нея не излизат коридори, по които не сме минавали.

Защо маркираме върховете, а не ребрата? За да пестим памет. За да маркираме върховете ни е необходима и достатъчна $\Theta(n)$ памет, тъй като за всеки връх има $\Theta(1)$ различни възможности (бял, сив и черен, или само бял и черен в по-простия вариант). За да маркираме ребрата ни е необходима и достатъчна $\Theta(m)$ памет, дори да ползваме само един бит за ребро. Както вече видяхме, m може да е квадратично по-голямо от n при графи, които не са мултиграфи, а върху мултиграфи m може да е произволно по-голямо от n .

Аналогията с лабиринта може да е подвеждаща. За реално обхождане на физически лабиринт, реализирано по BFS или DFS идеята, ще е необходимо да се измине много по-голямо разстояние (заради повтаряне на коридори), отколкото налагат нашите BFS и DFS. За пълна аналогия с обхождане на лабиринт трябва да допуснем, че съществото може да се телепортира от стая в друга стая, но само при условие, че вече е било в другата стая. Такова обхождане много точно съответства на обхождането, което следва от BFS и DFS.

В неориентираните (мулти)графи има една особеност: всяко ребро се обхожда по точно два пъти, дори да е примка. Причината е, че маркираме не ребрата, а върховете. В аналогията с лабиринта, ходейки по някакъв коридор, ние няма как да знаем, че сме били в него. Това, че минаваме за втори път по даден коридор става ясно едва когато излезем от него и установим, че сме попаднали в сива или черна стая. При ориентираните (мулти)графи ребрата се обхождат точно по веднъж.

Какво обхождаме в неориентиран граф, стартирайки от даден връх. Ако графът, мултиграф или не, е неориентиран, при обхождане, започващо във връх u , обхождаме точно свързаната компонента, на която принадлежи u . Както знаем, всеки връх в неориентиран граф се намира в точно една свързана компонента, а обхождането няма как да “излезе” от нея. За да обходим целия граф се налага да рестартираме обхождането върху всяка свързана компонента.

Какво обхождаме в ориентиран граф, стартирайки от даден връх. Ако обаче графът е ориентиран, въпросът какво точно ще обходим, стартирайки от връх u , е по-триков. Ако е силно свързан, задължително ще го обходим целия. Ако всяка от слабо свързаните му компоненти е силно свързана, задължително ще обходим цялата слабо свързана компонента, на която принадлежи u . Ако обаче слабо свързаната компонента, на която принадлежи u , не е силно свързана, може:

- в един екстремален случай, да я обходим цялата,
- в друг екстремален случай, да обходим само u , без да може да излезем от него,
- и в общия случай, да обходим само част от нея. Естествено, не е възможно да се прехвърляме от слабо свързана компонента в друга слабо свързана компонента, понеже няма ребро от никой връх на едната от тях до никой връх на другата, но каква част от слабо свързаната компонента, на която принадлежи u , е силно зависещо от конкретиката на графа.

С извинение за тавтологичния изказ: стартирайки от u , ще обходим точно този подграф, който е достижим от u . Следното определение прецизира това.

Определение 71: Достижим подграф

Нека G е ориентиран мултиграф и u е връх в него. *Подграфът на G , достижим от u* , е подграфът на G , индуциран от $\{v \in V(G) \mid \text{съществува път от } u \text{ до } v\}$.

8.3 Обща схема за обхождане

Ще опишем общата схема, която следват алгоритмите за обхождане, които ще разгледаме. За ограничените цели на тази секция, върховете имат само две състояния: необходим и обходен. Белите върхове са необходимите, черните са обходените. Схемата е една и съща за всички видове графи, които разглеждаме, така че допускаме, че е даден ориентиран мултиграф с възможни примки $G = (V, E)$. Даден е и стартов връх u .

Идея за обхождане. В началото всички върхове са бели с изключение на u . Поддържаме множество от върхове S . В началото $S = \{u\}$. Обхождането е итеративно, като на всяка итерация изваждаме един връх от S и го слагаме в променлива (тип връх) x . След това разглеждаме всички ребра, излизащи от x , което означава—при списъчно представяне на графа—че минаваме през списъка на x . За всяко ребро, да го наречем (x, y) :

- ако връх y е бял, правим y черен и го слагаме в S ,
- в противен случай прескачаме y .

Това приключва, когато S стане празно.

Самото обхождане на ребрата става при тези минавания през списъците. Ако беше даден лабиринт и трябваше да намерим нещо в коридорите му—изхода, някакво съкровище или *Минотавъра*—щяхме да го сторим по този начин.

Доказателство за коректност на идеята за обхождане. Ще покажем, не особено прецизно, че тази идея (защото това е прекалено общо описание, за да бъде алгоритъм) работи. Обхождането може да сбърка по два начина: да зацikli или да пропусне. Нашето обхождане не зацikliя, защото в S влизат само бели върхове, които веднага с влизането стават черни; веднъж станали черни, те остават черни. Ще покажем, че не може и да пропусне. Без ограничение на общността, нека (мулти)графът е силно свързан[†]. Да допуснем, че поне един връх v остава бял, тоест, непосетен, след края на обхождането. Но в графа има ориентиран път p от u до v . В края на алгоритъма u е черен, защото бива направен черен поначало, а v е бял по допускане. Всеки връх от p е или бял, или черен. Щом началото на p е черен връх, а краят му е бял връх, то в p има върхове a и b , такива че a е родител на b и a е черен и b е бял. Очевидно, в даден момент от обхождането, a е бил изваден от S (щом $S = \emptyset$ накрая) и всички деца на a са били направени черни и сложени в S . Тогава няма как b да е бил пропуснат и да е останал бял. ⚡

Ако множеството S бъде реализирано чрез АТД опашка[‡], тази схема за обхождане се превръща в алгоритъма BFS, който ще разгледаме първо. Ако S бъде реализирано чрез стек, получаваме алгоритъм, близък до DFS. DFS ще разгледаме след BFS.

[†]Ако не е, аргументът се отнася само за подграфа, достижим от u .

[‡]Да си припомним Подсекция 5.4.1. Опашка е именно АТД, защото се дефинира чрез интерфейса си от `Enqueue` и `Dequeue`, а имплементацията остава скрита зад този интерфейс. Аналогично, стек е АТД с интерфейс `push` и `pop`.

8.4 BFS

Името на този алгоритъм идва от Breadth-First Search, където “search” е синоним на “traversal”. При него, границата между посетените и непосетените върхове—образно и непрецизно казано—расте равномерно във всички посоки спрямо стартовия връх.

Отново допускаме, че е даден ориентиран мултиграф с възможни примки, а някои особености на алгоритъма върху неориентирани (мулти)графи ще разгледаме отделно.

Винаги е даден е стартов връх, който ще наричаме s . Но ако не е вярно, че целият граф е достижим от s , има две възможности.

1. BFS спира след като обходи подграфа, който е достижим от s .
2. След като обходи подграфа, който е достижим от s , BFS бива рестартиран с нов начален връх измежду непосетените досега върхове, и така нататък, докато целият граф бъде обходен.

Приемаме, че върховете са числата $1, 2, \dots, n$.

Цветове на върховете. Всеки връх на графа във всеки момент от работата на BFS има едно от следните три състояния: *непосетен*, *посетен*, *но неприключен*, и *приключен*, със съответните цветове бял, сив и черен. Тези цветове се реализират от масив $\text{color}[1..n]$, където $\text{color}[i]$ е текущият цвят на връх i ; $\text{color}[i] \in \{\text{white}, \text{gray}, \text{black}\}$.

Дърво или гора на обхождането. Ако целият граф е достижим от s или BFS спира изчерпване на достижими от s непосетени върхове (възможност 1), BFS строи *дърво на обхождането* T . Това T е арборесценция (вижте Определение 69) с корен s , чиито върхове са точно върховете, достижими от s . В T се намират точно тези ребра на G , с които BFS открива бели върхове.

Ако BFS се рестартира, докато не обходи целия граф (възможност 2), BFS строи колекция от арборесценции $T_1, \dots, T_k$, по една за всяко рестартиране, като корените им са стартовите върхове на всяко от пусканията на BFS и всеки връх на графа е в точно една от $T_1, \dots, T_k$. Казваме, че колекцията $\{T_1, \dots, T_k\}$ е *гората на обхождането*.

В алгоритъма ни дървото или гората на обхождането се реализира с масив на предшествията $\pi[1..n]$, в който за всеки връх i , който не е корен, съответният елемент $\pi[i]$ е родителят на i ; ако i е корен, то $\pi[i] = \text{Nil}$.

Разстояния спрямо стартовия връх. Във възможност 1, BFS може да изчисли разстоянията от s до всеки друг връх. Забележете изказа с “от-до”. Става дума за разстояния в ориентирания смисъл.

Определение 72: Разстояние в ориентирания смисъл

Нека G е ориентиран граф. Дали има или няма паралелни ребра или примки е без значение. За всеки два върха $u, v \in V(G)$, *разстоянието в ориентирания смисъл от u до v* , което означаваме с $\delta(u, v)$, е дължината на най-къс ориентиран път от u до v , ако такъв има, или ∞ , ако такъв няма. Ако от контекста е ясно, че разстоянието е в ориентирания смисъл, казваме кратко *разстоянието от u до v* .

Ако е даден и някакъв подграф H на G , то с $\delta_H(u, v)$ означаваме разстоянието от u до v в H . Това е дължината на най-къс ориентиран път от u до v , състоящ се само от ребра от $E(H)$. Ако такъв няма, то $\delta_H(u, v) = \infty$.

Очевидно е, че $\delta(u, v) \leq \delta_H(u, v)$.

Съществената разлика между разстоянието в ориентирания смисъл и разстоянието в неориентирания смисъл (в неориентирани графи) е, че разстоянието в ориентирания смисъл не е непременно симетрично, поради което не е метрика. Тривиално е да се измислят примери, в които $\delta(u, v) \neq \delta(v, u)$, като дори може едното да е число, а другото да е ∞ . Поради това Определение 72 ползва изказа “от-до”, а не “между”, който ползваме при симетричните разстояния в неориентирания смисъл.

Нататък ще покажем, че BFS наистина изчислява разстоянията в ориентирания смисъл. Тук само отбелязваме, че BFS записва тези разстояния в масив $d[1..n]$, където $d[i]$ след термилирането е равно на $\delta(s, i)$, за всеки връх i .

Опашката от сивите върхове. Ще разгледаме итеративен BFS, в който опашка Q съдържа текущите сиви върхове, а на всяка итерация се изважда един връх x от Q с `dequeue(Q)` — стига Q да не е празна преди това — и се обхождат всички излизащи от x ребра, като при това всеки открит бял връх y , дете на x , бива направен сив и вкаран в Q с `enqueue(Q, y)`. Алгоритъмът спира, когато Q стане празна. Очевидно интерфейсет на абстрактния тип данни опашка, за целите на нашия алгоритъм, се състои от трите функции `isempty(Q)`, `enqueue(Q, y)` и `dequeue(Q)`, като имената са индикативни за работата им.

8.4.1 BFS от един стартов връх

Алгоритъм 17: BFS

```

BFS( $G = (V, E)$ ): ориентиран мултиграф с възможни примки, като  $V = \{1, \dots, n\}$ ;  $s \in V$ 
1  for  $i \leftarrow 1$  to  $n$ 
2       $color[i] \leftarrow white$ 
3       $d[i] \leftarrow \infty$ 
4       $\pi[i] \leftarrow Nil$ 
5   $color[s] \leftarrow gray$ 
6   $d[s] \leftarrow 0$ 
7  създай опашка  $Q$ 
8   $enqueue(Q, s)$ 
9  while not  $isempty(Q)$  do
10      $x \leftarrow dequeue(Q)$ 
11     foreach  $y \in adj[x]$ 
12         if  $color[y] = white$ 
13              $color[y] \leftarrow gray$ 
14              $d[y] \leftarrow d[x] + 1$ 
15              $\pi[y] \leftarrow x$ 
16              $enqueue(Q, y)$ 
17      $color[x] \leftarrow black$ 

```

Коректността на BFS. Ще покажем, че BFS термилира върху всеки вход. Наистина, нови върхове влизат в опашката само ако са бели. Веднъж влезли в опашката, те вече не са бели, а не-бял връх не може да стане бял. Тъй като на всяка итерация се вади връх от

опашката и графът е краен, рано или късно опашката ще стане празна и BFS ще спре[†].

Ще покажем, че BFS обхожда точно подграфа на входа, който е достижим от s (Определение 71). Това, че нито един връх, достижим от s , не остава необходен, показахме в доказателството на коректността на схемата на обхождането на стр. 435, допускайки противното. Ще покажем, че нито един връх, който не е достижим от s , не бива обходен от BFS. Забелязваме, че дървото на обхождането е обединение на колекция от пътища от s до върхове на графа. Ерго, няма как връх, недостижим от s , да в дървото на обхождането: това би значело в графа-вход да няма път от s до него, а в дървото, което е подграф на входа, да има.

Последното ни дава право да допуснем без ограничение на общността, че входният граф е достижим (Определение 71) от началния връх. И така, в Теорема 76 и помощните лемми, $G = (V, E)$ е произволен мултиграф и s е произволен връх в него и G е достижим от s .

Лема 35: BFS апроксимира разстоянията отгоре

Всеки път, когато изпълнението е на ред 9, за всеки $v \in V$ е вярно, че $d[v] \geq \delta(s, v)$.

Доказателство: Ще докажем лемата по индукция по броя на достиганията на ред 9. Няма смисъл да формулираме инвариант, понеже не се интересуваме от крайното състояние на нещата при терминирането, а разглеждаме произволен момент от изпълнението, така че формулировката на лемата е инвариантът.

База. При първото достигане на ред 9 разглеждаме s отделно от останалите върхове.

- Да разгледаме s . Тъй като $d[s] = 0$ (ред 6) и $\delta(s, s) = 0$ (от теорията на графите), неравенството $d[s] \geq \delta(s, s)$ е в сила. ✓
- Да разгледаме $V \setminus \{s\}$. За всеки $v \in V \setminus \{s\}$ е вярно, че $d[v] = \infty$ заради присвояването на ред 3. Тогава $d[v] \geq \delta(s, v)$ независимо от това дали v е достижим от s или не; с други думи, дали $\delta(s, v)$ е различна от или равна на ∞ .

Поддръжка. Да допуснем, че твърдението е в сила в даден момент, в който изпълнението е на ред 9 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че Q не е празна.

На ред 10 изваждаме връх x от непразната опашка Q . Ефектът от работата на редове 11–17 очевидно е следният:

- всяко бяло дете y на x бива “обработено” на редове 13–16. $d[y]$ получава стойност $d[x] + 1$ на ред 14. По допускане, в този момент е вярно, че

$$d[x] \geq \delta(s, x)$$

Оттук

$$d[x] + 1 \geq \delta(s, x) + 1$$

което е същото като

$$d[y] \geq \delta(s, x) + 1$$

[†]В доказателството на коректността на схемата на обхождането на стр. 435 ние показахме, че обхождането не може да зацikli, което е по същество същото.

Но $\delta(s, x) + 1 \geq \delta(s, y)$, понеже, щом има път p от s до x , има и път от s до y с дължина $|p| + 1$ [†]. Тогава

$$d[y] \geq \delta(s, y)$$

Доказахме, че за всеки връх y , който е бил “обработен” при изпълнение на текущата итерация, $d[y] \geq \delta(s, y)$.

- Сивите и черните деца на x не биват “обработвани” и техните d -стойности не биват променяни в тялото на цикъла (редове 13–16), така че за тях неравенството остава в сила. $\square$

Лема 36 казва, че във всеки момент от изпълнението на BFS, в опашката има върхове или с една и съща d -стойност, или с точно две различни, но съседни по големина, d -стойности, като по-малките предшестват по-големите в опашката. С други думи, опашката “обработва” върховете по d -стойности. Което, както ще видим в Теорема 76, е същото като да “обработва” върховете по ориентираните разстояния от s .

Това е Lemma 22.3 в [34, стр. 599].

Лема 36: d -стойностите на върховете в опашката

Нека в произволен момент t от работата на $\text{BFS}(G, s)$, опашката Q съдържа следната редица от върхове: $\langle u_1, u_2, \dots, u_k \rangle$, като u_1 е най-отдавна сложеният връх, а u_k е най-скоро сложеният връх. Тогава

$$d[u_1] \leq d[u_2] \leq \dots \leq d[u_k] \leq d[u_1] + 1$$

Доказателство: Както в Лема 35, и тук няма смисъл да формулираме инвариант, понеже не се интересуваме от крайното състояние на нещата при термилирането, а разглеждаме произволен момент от изпълнението, така че формулировката на лемата е инвариантът.

Базата на индукцията е първото достигане на ред 9. В този момент опашката съдържа само s и твърдението със сигурност е вярно. ✓

Разглеждаме произволно достигане на ред 9, което не е последното. Нека това е в момент t от изпълнението на BFS. Нека t' е моментът на следващото достигане на ред 9. Очевидно опашката Q не е празна в момент t . Да кажем, че в момент t опашката Q съдържа редицата на върхове $\langle u_1, u_2, \dots, u_k \rangle$. Индуктивното допускане е, че в този момент е изпълнено

$$d[u_1] \leq d[u_2] \leq \dots \leq d[u_k] \leq d[u_1] + 1$$

На ред 10 връх u_1 бива изваден от Q и сложен в променливата x . Вече опашката съдържа редицата $\langle u_2, u_3, \dots, u_k \rangle$, така че неравенствата по отношение на върховете в опашката стават

$$d[u_2] \leq \dots \leq d[u_k] \leq d[u_1] + 1$$

Имайки предвид това, че $d[u_1] \leq d[u_2]$, имаме $d[u_1] + 1 \leq d[u_2] + 1$, откъдето следва, че

$$d[u_2] \leq \dots \leq d[u_k] \leq d[u_2] + 1$$

Ерго, твърдението остава в сила веднага след изваждането на връх от опашката.

[†]В [34] това очевидно твърдение е Lemma 22.1 на стр. 598.

Да се убедим, че твърдението е в сила и в момента t' . Ясно е, че всички изпълнения на вътрешния цикъл (редове 11–16) вкарват нула или повече върхове в опашката Q . Това са точно върховете от $\text{adj}[u_1]$, които са бели в момента t . Да кажем, че в опашката влизат върхове $z_1, \dots, z_b$, и то в този ред, за някакво $b \geq 0$. Тогава в момента t' , опашката съдържа следната редица от върхове

$$\langle u_2, u_3, \dots, u_k, z_1, \dots, z_b \rangle$$

Очевидно е, че $d[z_i] = d[u_1] + 1$ за $1 \leq i \leq b$, понеже при всяко достигане на ред 14, за което говорим тук, x съдържа u_1 , а y съдържа някой от $z_1, \dots, z_b$. Имаме право да кажем, че

$$d[u_2] \leq \dots \leq d[u_k] \leq d[z_1] = d[z_2] = \dots = d[z_b] = d[u_1] + 1$$

Това може да напишем и така

$$d[u_2] \leq \dots \leq d[u_k] \leq d[z_1] \leq d[z_2] \leq \dots \leq d[z_b] \leq d[u_1] + 1$$

Както вече отбелязахме, $d[u_1] + 1 \leq d[u_2] + 1$. В крайна сметка,

$$d[u_2] \leq \dots \leq d[u_k] \leq d[z_1] \leq d[z_2] \leq \dots \leq d[z_b] \leq d[u_2] + 1$$

□

Следствие 27

За всеки два различни върха u и v , ако по време на работата на BFS u влиза в Q преди v , то $d[u] \leq d[v]$.

Доказателство: Следва веднага от Лема 36 и очевидния факт, че никой връх не влиза в опашката повече от веднъж. □

Теорема 76 и Теорема 77 са Theorem 22.5 в [34, стр. 599].

Теорема 76: Коректността на BFS за разстоянията

След терминирането на $\text{BFS}(G, s)$ е вярно, че $\forall v \in V : d[v] = \delta(s, v)$.

Доказателство: Да допуснем, че съществува връх $v \in V$, такъв че $d[v] \neq \delta(s, v)$ след терминирането на $\text{BFS}(G, s)$. БОО, нека v е такъв връх с минимална $\delta(s, v)$ стойност; с други думи, за всеки $u \in V$, такъв че $\delta(s, u) < \delta(s, v)$ е вярно, че $d[u] = \delta(s, u)$.

Очевидно $v \neq s$. Съгласно Лема 35, $d[v] \geq \delta(s, v)$. От това и допускането, че $d[v] \neq \delta(s, v)$ следва, че $d[v] > \delta(s, v)$.

Щом $v \in V$ и $v \neq s$, съществува път p от s до v с дължина поне единица. БОО, нека p е път от s до v с минимална дължина. Тогава $|p| = \delta(s, v)$. Нека w е родителят на v в p . Нека подпътят на p от s до w се казва q . Забележете, че $|p| = |q| + 1$.

Известно е, че най-къс път се състои от най-къси подпътища (Теорема 96). Тогава q е най-къс път от s до w , така че $|q| = \delta(s, w)$. От това следва, че $\delta(s, v) = \delta(s, w) + 1$.

Но $\delta(s, v) > \delta(s, w)$ и v е избран като връх-нарушител с минимална δ -стойност. Тогава w не е “нарушител”; тоест, $d[w] = \delta(s, w)$. Изведените дотук факти изглеждат така на един ред:

$$d[v] > \delta(s, v) = \delta(s, w) + 1 = d[w] + 1 \tag{8.1}$$

Нека t е моментът от работата на BFS, в който w бива изваден от опашката (ред 10). С други думи, променливата x получава съдържание w в момент t . Следните възможности за цвета на v в момент t са изчерпателни.

color[v] = white Тогава при изпълнението на **foreach**-цикъла (редове 11–16) в даден момент $\hat{t} > t$, променливата y ще получи съдържание v . При присвояването на ред 14 с $y = v$ е изпълнено $d[v] = d[w] + 1$. Но това е в противоречие с (8.1). ⚡

color[v] = gray В такъв случай v е бил оцветен в сиво на някоя предишна итерация на **while**-цикъла, когато върхът в променливата x е бил някой $a \in V$, като v е бил y и е получил d -стойността си като $d[v] \leftarrow d[a] + 1$. Но този a влиза в опашката преди w , от което следва, съгласно Следствие 27, че $d[a] \leq d[w]$, което е същото като $d[a] + 1 \leq d[w] + 1$. Тогава $d[v] \leq d[w] + 1$, в противоречие с (8.1). ⚡

color[v] = black Щом v е черен, той вече е бил в Q и е бил изваден оттам. Тогава v бива изваден от Q преди w , което влече, че v влиза в Q преди w . Съгласно Следствие 27, $d[v] \leq d[w]$, в противоречие с (8.1). ⚡ □

Теорема 77: Коректността на BFS за дървото

След термилирането на $\text{BFS}(G, s)$ е вярно, че $\pi[1..n]$ реализира дърво на най-къси пътища за G с корен s .

Доказателство: За да докажем това, достатъчно е да докажем, че при термилирането на алгоритъма, за всеки връх $v \neq s$ съществува най-къс път от s до v , в който $\pi[v]$ е предпоследният връх. Предвид очевидния факт, че за всеки връх $v \neq s$, стойността се $\pi[v]$ се присвоява на ред 15 и, веднъж присвоена, тя не се променя повече, достатъчно е да докажем, че за всеки връх $v \neq s$, в момента, в който y съдържа v и ред 15 току-що е бил изпълнен, $\pi[v]$ е предпоследният връх в някой най-къс път от s до v .

Това ще докажем по индукция по разстоянията от s . БОО, нека $|V| > 1$. Нека $h_{\max} = \max \{\delta(s, v) \mid v \in V\}$. Нека $L_h = \{v \in V \mid \delta(s, v) = h\}$. Ще докажем, че за всяко $h \in \{1, \dots, h_{\max}\}$, за всеки връх $v \in L_h$, в итерацията на **foreach**-цикъла, в която y съдържа v , след изпълнението на ред 15 е вярно, че $\pi[v]$ е предпоследният връх по някой най-къс път от s до v .

Базата е $h = 1$. Веднага виждаме, че L_1 се състои точно от децата на s в G ; с други думи, $L_1 = \text{adj}[s]$. Наистина, при първото изпълнение на **while**-цикъла (редове 9–17), променливата x съдържа s , всички върхове от $\text{adj}[s]$ са бели, така че биват “обработени” от **foreach**-цикъла (редове 11–16), като за всеки $v \in L_1$, $\pi[v]$ получава стойност s . Но реброто (s, v) е най-къс път от s до v , така че наистина s е предшественик на v по някой най-къс път от s до v . ✓

Индуктивното предположение е, че твърдението е вярно за някое $h \in \{1, \dots, h_{\max} - 1\}$.

В индуктивната стъпка ще докажем твърдението за стойност на аргумента $h + 1$. Разглеждаме произволен $v \in L_{h+1}$. Такъв съществува, защото $h + 1 \leq h_{\max}$. Щом $\delta(s, v) = h + 1$, съгласно Теорема 76, в края на алгоритъма $d[v] = h + 1$. Тогава очевидно стойността, която се присвоява на $d[v]$ на ред 14 по време на итерацията на **foreach**-цикъла, в която y съдържа v , е $h + 1$.

Разглеждаме въпросната итерация на **foreach**-цикъла. Променливата x съдържа някакъв връх u . Очевидно $d[u] = h$, щом $d[u] + 1$ на ред 14 е $h + 1$. От Теорема 76 и факта, че d -стойностите не се менят след присвояването им на ред 14 следва, че $\delta(s, u) = h$, така че

$u \in L_h$. Прилагаме индуктивното предположение и заключаваме, че по време на итерацията на **foreach**-цикъла, в която y съдържа v , $\pi[u]$ съдържа предпоследния връх в най-къс път p от s до u ; очевидно $|p| = h$. От това и факта, че $y \in L_{h+1}$, заключаваме, че пътят $p' = p, v$, чиято дължина е $h + 1$, е най-къс от s до v ; нещо повече, връх u е предпоследен връх по него. Но по време на въпросната итерация на **foreach**-цикъла, на ред 15, на $\pi[v]$ се присвоява именно u . □

Допълнение 45: Друго доказателство за коректността на BFS

Коректността на BFS може да се докаже и с инвариант на цикъла, понеже BFS е итеративен алгоритъм. Но доказателството за коректността на BFS с инвариант, ако се направи подробно и прецизно, е твърде дълго и тежаво, поради което в тези лекционни записки е показано гореизложеното доказателство с Теорема 76 и Теорема 77, основано на [34].

Доказателството с инвариант обаче има едно предимство: то дава яснота за същността на работата на BFS, каквато яснота не се получава от доказателството с допускане на противното в Теорема 76. Тук, в Допълнение 45, ще видим смислен инвариант, но без доказателство. Въз основа на инварианта ще представим илюстрация, която е много полезна за осмисляне на същността на BFS.

Разглеждаме ориентиран граф $G = (V, E)$ и стартов връх $s \in V$. БОО, нека всички върхове на графа са достижими от s . Словесите на G спрямо s са множествата L_k , дефинирани така

$$L_k = \{v \in V \mid \delta(s, v) = k\}$$

Очевидно слоевете са само краен брой: $L_0, L_1, \dots, L_{d_{\max}}$, където $d_{\max} = \max\{k \mid \exists v \in V : k = \delta(s, v)\}$. Също така е очевидно, че в индексите на слоевете няма “дупки”: всички цели числа от 0 до $d_{\max}$ са индекси на слоеве. И е очевидно, че слоевете са разбиване на V , понеже слоевете са непразни и всеки връх е в точно един слой.

За всяко $k \in \{0, \dots, d_{\max}\}$, нека t_k е първият момент от работата на BFS, при който всички върхове от L_k са получили своята окончателна d стойност и изпълнението е на ред 9. Ясно е, че $t_0, t_1, \dots, t_{d_{\max}}$ са добре дефинирани. Интуитивно е ясно, че $t_0 < t_1 < \dots < t_{d_{\max}}$, но това трябва да се докаже. Нека $\mathcal{T} = \{t_0, t_1, \dots, t_{d_{\max}}\}$. Нека $V(Q)$ е множеството от върховете в опашката Q в кой да е момент от работата на BFS.

Инвариантът на цикъла на BFS не е формулиран чрез типичното “При всяко достигане на ред 9, ...”, а е формулиран чрез моментите $t_0, \dots, t_{d_{\max}}$. Тоест, разглеждаме само **някои** достигания на ред 9; а именно тези, при които BFS току-що е приключил напълно с върховете на разстояние k от s .

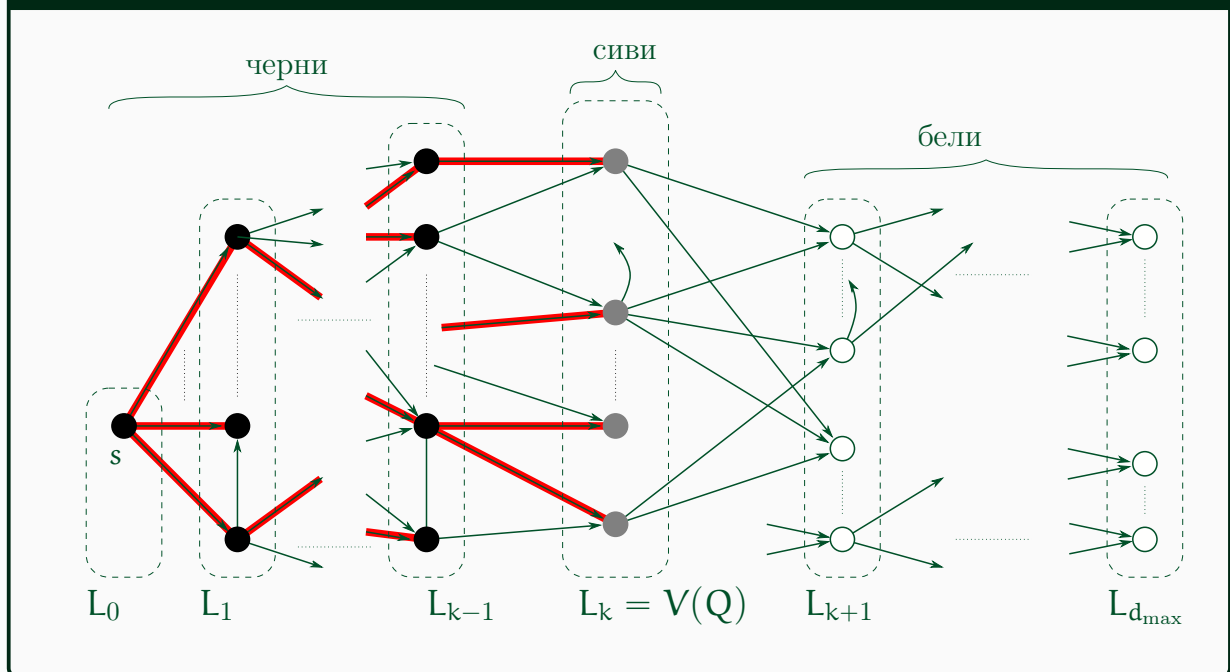
Лема 37: За коректността на BFS

За всяко $k \in \{0, \dots, d_{\max}\}$, в момента t_k :

1. $\bigcup_{0 \leq i \leq k-1} L_i$ съдържа точно черните върхове, L_k съдържа точно сивите върхове и $L_k = V(Q)$, а $\bigcup_{k+1 \leq i \leq d_{\max}} L_i$ съдържа точно белите върхове.
2. $\forall i \in L_0 \cup \dots \cup L_k : d[i] = \delta(1, i)$.
3. Моментите от $\mathcal{T}$, които са се случили, са $t_0, t_1, \dots, t_k$, и то в този ред.

Фигура 8.1 илюстрира момент t_k от работата на BFS за някое t_k , такова че $1 < k < d_{\max}$.

Фигура 8.1 : BFS в момента t_k .



Съгласно Лема 37, в този момент черните върхове са в слоевете $L_0, \dots, L_{k-1}$, слой L_k съдържа сивите върхове и това са точно върховете в опашката, а белите върхове са в слоевете $L_{k+1}, \dots, L_{d_{\max}}$. За “обработените” върхове, а именно $L_0, \dots, L_{k-1}, L_k$, разстоянията от s са записани в d -стойностите и това е основното в доказателството за коректност, що се отнася до разстоянията: от него следва, че при термилирането на BFS е в сила $\forall v \in V : d[v] = \delta(s, v)$.

В момента t_k , дървото на обхождането е изградено само частично. Частта от него, която е изградена (чрез подходящи стойности в масива π), е показана с червено. Неформално казано, в момента t_k дървото стига само до L_k ; тоест, поддървото, изградено до момента, е такова, че листата са му в L_k . Забележете, че за всеки връх от $L_1, \dots, L_k$ е вярно, че има точно един родител в досега изграденото дърво.

Сложност по време. Реализацията на BFS, която разгледахме, има сложност по време $\Theta(n + m)$, ако мултиграфът е реализиран чрез списъци на съседство. Както в много други алгоритми върху графи, които ще разгледаме, сложността по време **НЕ СЕ ОПРЕДЕЛЯ**

така:

while-цикълът се изпълнява n пъти, по веднъж за всеки връх, а всяко изпълнение отнема време, в най-лошия случай, $\Theta(m)$, така че сложността е $\Theta(nm)$.

Това разсъждение в общия случай е **ГРЕШНО**. От “**while** се изпълнява n пъти, а всяко изпълнение в най-лошия случай е $\Theta(m)$ ” следва само асимптотична горна граница $O(nm)$ и нищо повече. В общия случай, тази граница не е точна. Да, има безкрайни класове графи, за които това разсъждение е приложимо – примерно, клас графи, за всеки граф в който е изпълнено $m = \Theta(1)$. За графите от този клас, наистина сложността на BFS е $\Theta(nm)$. Но в общия случай не е така.

Прецизното разсъждение, което ни дава точна асимптотична оценка винаги, е да игнорираме **while**-цикъла и да се фокусираме върху въртредния цикъл, тоест, върху **foreach**-цикъла. Но не върху всички изпълнения на **foreach**-цикъла в рамките на едно изпълнение на **while**-цикъла, а върху всички изпълнения на **foreach**-цикъла по време на цялата работа на BFS. Тогава виждаме, че променливата y минава систематично през всички елементи на всички списъци на съседство. Размерът на списъците на съседство е $\Theta(n + m)$, а алгоритъмът върши само $\Theta(1)$ работа за всеки списъчен елемент. Оттук веднага получаваме точна асимптотична оценка $\Theta(n + m)$ за сложността по време на BFS.

Ако G не е мултиграф и е реализиран чрез матрица на съседство, може да разсъждаваме така: **while**-цикълът се изпълнява n пъти, а всяко изпълнение става във време $\Theta(n)$, така че сложността на BFS е $\Theta(n^2)$. Но $\Theta(n + m)$ остава по-прецизна оценка, тъй като функцията в израза е на две променливи и това е по-информативно в случай, че графът е реализиран чрез списъци на съседство.

В заключение, сложността по време на BFS е $\Theta(n + m)$.

8.4.2 BFS, обхождащ целия граф

Следва псевдокод на BFS, който обхожда целия граф. Отново графът на входа е ориентиран мултиграф с възможни примки. Този път стартов връх няма.

Ако мултиграфът е неориентиран, променливата `count` се инкрементира точно веднъж за всяка свързана компонента, така че алгоритъмът връща броя на свързаните компоненти. Тривиално е да се добави код, така че алгоритъмът да връща самите свързани компоненти, а не само броя им.

BFS-MOD($G = (V, E)$, като $V = \{1, \dots, n\}$)

```

1  for i ← 1 to n
2      color[i] ← white
3      d[i] ← ∞
4      π[i] ← Nil
5  count ← 0
6  for i ← 1 to n
7      if color[i] = white
8          count++
9          BFS-VISIT(G, i)
10 return count
```

BFS-VISIT($G = (V, E)$; $s \in V$)

```

1  color[s] ← gray
2  d[s] ← 0
3  създай опашка Q
4  enqueue(Q, s)
5  while not isempty(Q) do
6      x ← dequeue(Q)
7      foreach y ∈ adj[x]
8          if color[y] = white
9              color[y] ← gray
10             d[y] ← d[x] + 1
11             π[y] ← x
12             enqueue(Q, y)
13  color[x] ← black
```

Аргументация за коректност няма да правим. Очевидно сложността е $\Theta(n + m)$.

8.4.3 Приложения на BFS

Изчисляване на двуделност на неориентиран граф. Нека $G = (V, E)$ е обикновен граф. БОО, нека G е свързан. Пита се дали G е двуделен; алтернативно, дали е 2-оцветим, което е същото или почти същото нещо, в зависимост от формалните дефиниции[†]. Знаем необходимо и достатъчно условие за това: да няма нечетни цикли. Това условие обаче не се превежда директно в ефикасен алгоритъм. Циклите на графа може да са суперекспоненциално много в n , така че, дори да имаме алгоритъм, намиращ всеки цикъл във време $\Theta(1)$ (в обикновения или амортизиран смисъл), той би бил неприемливо бавен за определяне на двуделността.

Оптимално ефикасен, в асимптотичния смисъл, е алгоритъм, базиран на BFS. Ако изобщо G е 2-оцветим[‡], да кажем в зелено и червено, то има точно две възможности:

- s е червен, неговите съседни са зелени, техните съседни без s са червени, и така нататък,
- и обратно, s е зелен, неговите съседни са червени, техните съседни без s са зелени, и така нататък.

Виждаме, че разбиването на V е едно и също и при двете схеми за оцветяване – те само разменят цветовете на двата дяла. Същественото е, че двата дяла може да дефинират по отношение на разстоянието (сега говорим за обикновено симетрично разстояние, понеже G е неориентиран) между s и всеки друг връх – върховете на четно разстояние са единият дял, примерно червените, а тези на нечетно разстояние са другия дял, в случая зелените.

Това веднага дава идея за опит за оцветяване на произволен свързан граф. Започваме BFS обхождане от произволен връх s и за всяко ребро (u, v) следим дали двата края са на различно разстояние от s , или не. Забележете, че е невъзможно разстоянието между u и s да се различават на повече от единица от разстоянието между v и s . Ерго, тези разстояния или се различават на единица, което означава, че са с противоположна четност, което означава, че u и v са в различни цветове, или са равни, при което u и v са в един и същи цвят. Ако при обхождането се окаже, че съществува ребро (u, v) , такова че $d[u] = d[v]$, алгоритъмът връща НЕ. В противен случай връща ДА.

Тривиално лесно е да се модифицира кода на BFS съгласно тази идея, така че сложността по време да остане $\Theta(n + m)$.

8.5 DFS

8.5.1 Неформално въведение и сравнение с BFS

DFS реализира, в някакъв смисъл, обратната идея на BFS обхождането. BFS обхождането е “предпазливо”. Ако си представим обхождане на лабиринт, BFS идеята е тази: има множество от стаи R , в които сме били, но с които не сме приключили. Първо посещаваме всички стаи, които са достижими с един коридор от някоя стая от R , преди да продължим нататък.

DFS обхождането е “смело”. От началната стая излизаме през първия възможен коридор, от следващата също, и така нататък, като надлежно маркираме стаите, в които вече сме били, за да не зациклим. Рано или късно ще се окажем в стая, в която сме били. Тогава се

[†] Някои формални дефиниции на “двуделен граф” предполагат, че графът има поне два върха.

[‡] Тези цветове нямат **нищо общо** с цветовете бял, сив и черен, които BFS ползва по време на работата си!

връщаме колкото е възможно по-малко назад до стая, в която сме били, и опитваме същото нещо, само че излизайки от нея през врата, която не сме използвали. На английски тази идея се нарича *backtracking*[†].

Допълнение 46: Разликата между BFS и DFS, илюстрирана с шах

Ще илюстрираме същностната разлика между BFS и DFS с шаха. Нека е дадена задача да открием мат в три хода в дадена позиция за белите^a. Един *ход* е местене на фигура от играча с белите, последвано от местене на фигура играчът с черните. Местене на фигура от само единия играч е *полуход*. Да подходим алгоритмично.

BFS идеята. BFS подходът е да генерираме всички възможни, съгласно правилата на шаха, позиции с един полуход на белите. Нека множеството от тези позиции е P_1 . За всяка от тези позиции генерираме всички позиции с един полуход на черните, получавайки множество Q_1 : това са всички позиции на един ход. Използвайки Q_1 и всички възможни полуходове на белите, генерираме множеството P_2 , а от него с всички възможни полуходове на черните генерираме Q_2 : множеството от всички позиции на два хода от началната. От Q_2 с всички възможни полуходове на белите генерираме P_3 . Ако задачата е смислена, в една или повече от позициите на P_3 има мат на черния цар, но в P_1 и P_2 няма мат на черния цар. Нещо повече. Трябва:

- да **съществува** полуход на белите в началната позиция, такъв че
- **за всеки** полуход на черните след него
- да **съществува** полуход на белите, такъв че
- **за всеки** полуход на черните след него
- да **съществува** полуход на белите, такъв че черният цар е мат.

В тази редица от полуходове се редуват квантори $\exists$ и $\forall$, като $\exists$ са на белите полуходове, а $\forall$, на черните. Тъй като възможните полуходове от дадена позиция може да са десетки, дори при малко фигури, говорим за дърво, чиято разклоненост е поне от порядъка на десетки. Височината му в полуходове е шест. Това е дървото на BFS-а, който току-що си представихме. При BFS идеята генерираме нивата на дървото систематично от корена към листата.

DFS идеята. DFS подходът е, да опитаме един възможен полуход на белите от началната позиция, после един полуход на черните, и така нататък, докато направим трети полуход за белите. Отчитаме дали черният цар е мат или не, и се връщаме в предната позиция, която е получена при втори полуход за черните. От нея правим следващия възможен трети полуход за белите—ако в предишния трети полуход за белите не е имало мат на черния цар—и така нататък.

След като “влезе” в позиция p , DFS генерира **една след друга** позициите, достижими от нея, като за всяка от тях, да я наречем q , “влиза” в q и се връща обратно в p чак след като е изчерпила всички възможности “до долу”, за да опита следващата след q ,

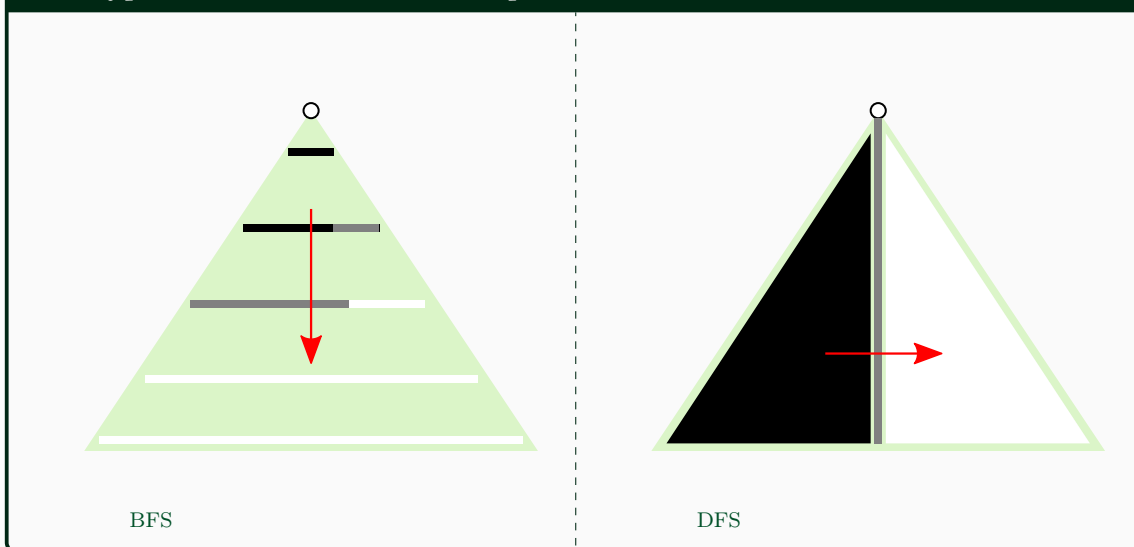
[†]Backtracking е алгоритмична схема, също както разделяй-и-владей е алгоритмична схема. В този курс няма да разглеждаме алгоритми, използващи backtracking, освен DFS.

ако има такава. Ако p не е началната позиция, трябва да се “излезе” от нея, което става чак когато са изчерпани всички възможности за следваща позиция.

Както и при BFS, така и при DFS решението на задачата е полуход на белите, такъв че за всеки полуход на черните има полуход на белите, такъв че за всеки полуход на черните има полуход на белите, в който черния цар е мат. Съществената разлика с BFS е, че BFS генерира цялото P_1 , от него цялото Q_1 , и така нататък, докато DFS не генерира пълните нива. DFS генерира елементите на нивата, но последователно, а не всички наведнъж.

Ако си представим търсенето на мат като дърво с корен началната позиция и коренът е “горе”, то BFS генерира едно след друго нивата на дървото, вървейки “вертикално” отгоре надолу, и в края на обхождането, цялото дърво се оказва генерирано в явен вид. Докато DFS поддържа само един “вертикален” път от корена до листата, като този път се движи “хоризонтално”. Във всеки момент DFS съхранява в явен вид само позициите от този път, а следващите позиции се пресмятат от текущите. Фигура 8.2 илюстрира разликата в двата подхода. На нея цветовете бял, сив и черен имат смисъла на цветовете на върховете от BFS и DFS и нямат нищо общо с цветовете на шахматните фигури.

Фигура 8.2 : Обхождане на дърво от BFS и от DFS.



В най-лошия случай, и BFS, и DFS генерират всеки връх от дървото и “преглеждат” всички листа. Огромната разлика е в това, че BFS в края на обхождането е генерирала **цялото** дърво. Това значи, че BFS използва памет, пропорционална на размера на дървото. Докато DFS в никой момент не е генерирала цялото дърво, а само част от него – а именно, път от корена надолу, най-много до някое листо, плюс допълнителна памет за ефикасното извършване на обхождането, която също е пропорционална на височината. И така, ако се имплементира правилно, систематичното търсене на мат с DFS ползва само логаритмична, в размера на дървото, памет, докато BFS ползва линейна в размера на дървото памет.

^aЕто сайт с такива [шахматни задачи](#).

Ако заменим в кода на BFS опашката със стек (съответно и функциите от интерфейса с push

и рор), ще получим **нещо като DFS**, но не точно DFS. Нека читателят пробва да замени опашката със стек в BFS и да се убеди върху малък пример, че полученото обхождане е, в някакъв смисъл, междинен вариант: нито истинско BFS, нито истинско DFS обхождане.

8.5.2 Псевдокод на DFS

Следният псевдокод е буквално същият като псевдокода в [34, стр. 604]. Смисълът на цветовете е същият като при BFS. Разстояния сега няма – DFS не пресмята разстояния в графа. В някои приложения на DFS въвеждаме нива, които се пресмятат като d -стойностите в BFS, но при DFS нивата нямат смисъл на разстояния.

Използваме глобална променлива $time$, чрез която пресмятаме времето на откриване на всеки връх и времето на финализиране на всеки връх. Два масива $d[1..n]$ и $f[1..n]$ съхраняват тази информация за върховете.

Разглеждаме вариант на DFS, който задължително обхожда целия граф, подобно на модификацията на BFS на стр. 444. Тривиално е да направим от него вариант на DFS, който получава стартов връх s като аргумент и обхожда само подграфа, който е достижим от s .

Алгоритъм 18: DFS

```
DFS( $G = (\{1, \dots, n\}, E)$ )
1   $time$  е глобална
2  for  $i \leftarrow 1$  to  $n$ 
3       $color[i] \leftarrow white$ 
4       $\pi[i] \leftarrow Nil$ 
5   $time \leftarrow 0$ 
6  for  $i \leftarrow 1$  to  $n$ 
7      if  $color[i] = white$ 
8          DFS-VISIT( $G, i$ )
```

Алгоритъм 19: DFS-VISIT

```
DFS-VISIT( $G = (V, E); x \in V$ )
1   $time++$ 
2   $d[x] \leftarrow time$ 
3   $color[x] \leftarrow gray$ 
4  foreach  $y \in adj[x]$ 
5      if  $color[y] = white$ 
6           $\pi[y] \leftarrow x$ 
7          DFS-VISIT( $G, y$ )
8   $color[x] \leftarrow black$ 
9   $time++$ 
10  $f[x] \leftarrow time$ 
```

Това, че DFS обхожда целия граф, следва от доказателството за коректността на общата схема на стр. 435.

Сложността по време на DFS е $\Theta(n + m)$, което се извежда със същите разсъждения, с които се извежда линейната сложност по време на BFS на стр. 443.

8.5.3 Свойства на DFS

8.5.3.1 Скобова структура на интервалите, генерирани от DFS

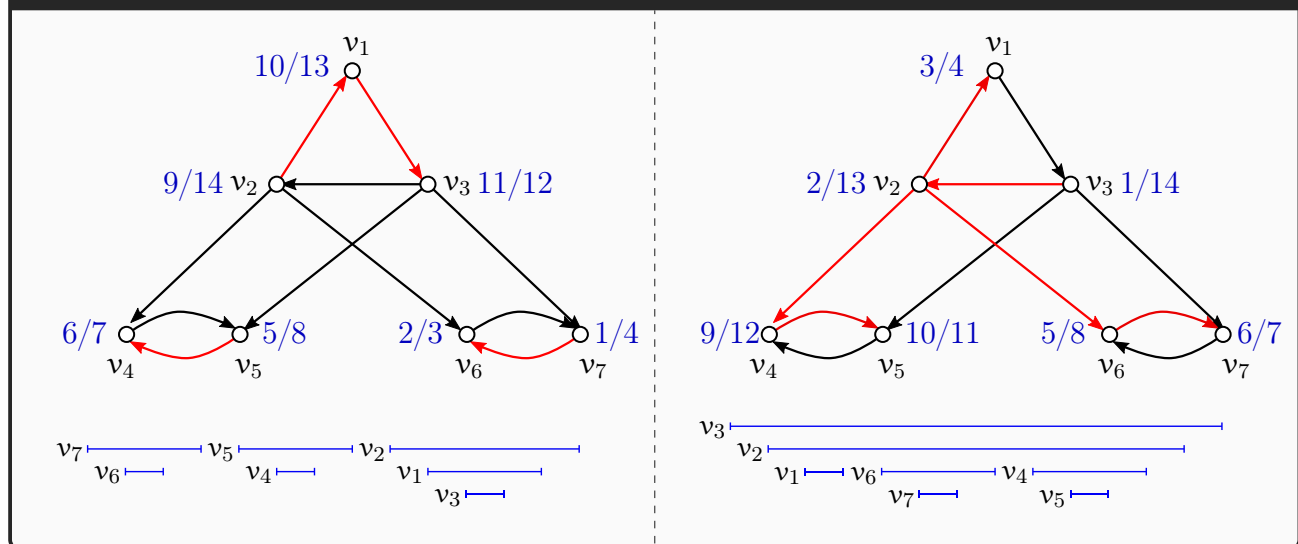
Нека $G = (V, E)$ е мултиграф с възможни примки. За по-голяма яснота и прецизност, в тази подсекция множеството от върховете е $\{v_1, \dots, v_n\}$, а **не** $\{1, \dots, n\}$. Разглеждаме работата на DFS(G). Алгоритъмът изгражда гора на обхождането, която се реализира от масива на родителите π .

Наблюдение 51

Променливата $time$ се инициализира с 0 на ред 5 и приема последователно стойностите $1, 2, \dots, 2n$, тъй като бива инкрементирана от DFS-VISIT два пъти за всеки връх. Ерго, числата $d[v_1], \dots, d[v_n], f[v_1], \dots, f[v_n]$ са две по две различни. Нещо повече, те са точно елементите на $\{1, 2, \dots, 2n\}$. Освен това, $\forall k \in \{1, \dots, n\} : d[v_k] < f[v_k]$.

Групираме d -стойностите и f -стойностите по двойки за всички върхове и получаваме n интервала $[d[v_1], f[v_1]], [d[v_2], f[v_2]], \dots, [d[v_n], f[v_n]]$. Казваме, че това са *интервалите, генерирани от DFS(G)*. Краищата на интервалите са, две по две, различни точки, така че всеки два интервала или имат празно сечение, или имат общ подинтервал с дължина поне единица. Фигура 8.3 илюстрира интервалите, които се получават при две различни изпълнения на DFS върху един и същи граф. Гората на обхождането е маркирана с червено и за двете изпълнения. За всеки връх v_i , на фигурата с “ $d[v_i]/f[v_i]$ ” до него са записани неговите времена на откриване и финализиране. Този запис е много удобен за недвусмислено описание на работата на DFS.

Фигура 8.3 : Интервалите, генерирани от различни изпълнения на DFS.

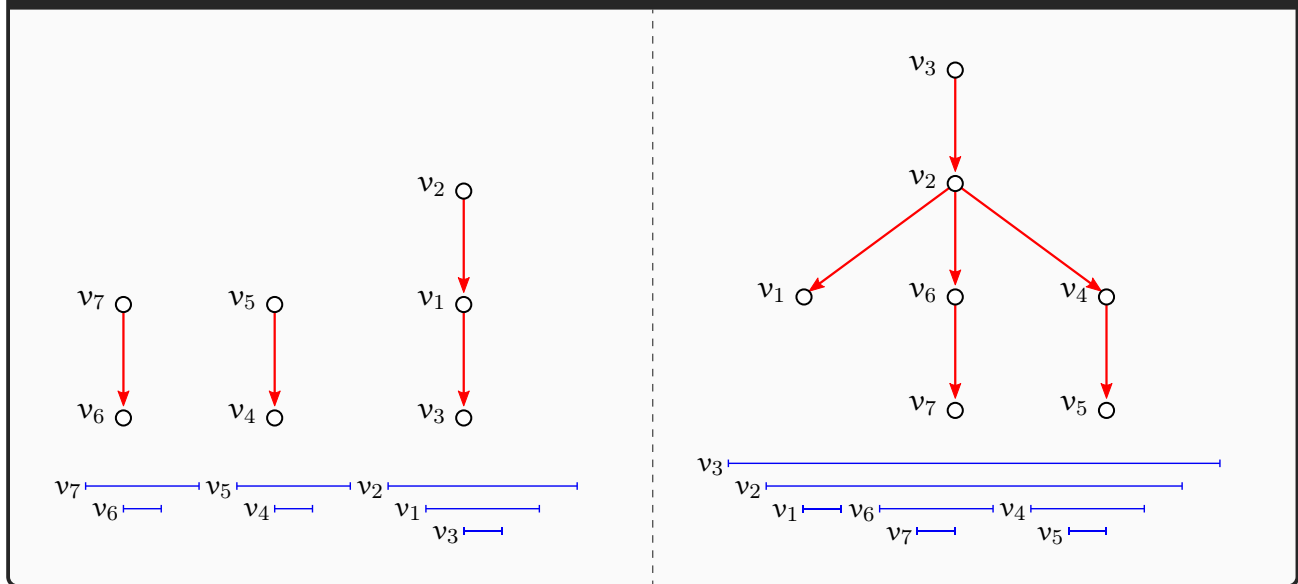


Интервалите на Фигура 8.3 не изглеждат произволно разположени един спрямо друг. Те имат скобова структура: ако за всеки интервал сложим по един чифт скоби “(” и “)”, ще получим добре скобуван израз. Примерно, на интервалите вляво отговаря $((()))((()))$, а на интервалите вдясно отговаря $((()))((()))$. Нещо повече, вложеността на интервалите определя напълно гората на обхождането и обратното: има ребро (v_i, v_j) тстк

$$[d[v_j], f[v_j]] \subset [d[v_i], f[v_i]] \wedge \neg \exists k \in \{1, \dots, n\} : [d[v_j], f[v_j]] \subset [d[v_k], f[v_k]] \subset [d[v_i], f[v_i]]$$

Това се вижда ясно на Фигура 8.4, която показва двете гори на обхождането, съответни на двете изпълнения на DFS от Фигура 8.3, заедно със съответните интервали.

Фигура 8.4 : Интервалите и гората на обхождането на DFS.



Сега ще докажем всичко това формално.

Наблюдение 52

По отношение на работата DFS и DFS-VISIT, разглеждаме произволни $u, v \in V(G)$, такива че $u \neq v$. Нека R_u означава рекурсивното викане, в което аргументът x на DFS-VISIT е u . Нека R_v означава рекурсивното викане, в което аргументът x на DFS-VISIT е v . Връх u е предшественик на връх v в гората на обхождането при термирирането на DFS тстк R_v се случва по време на R_u ; с други думи, измежду R_u и R_v първо започва R_u и преди то да приключи, се случва R_v . В такъв случай $d[u] < d[v] < f[v] < f[u]$.

Теорема 78 е Theorem 22.7 от [34, стр. 606].

Теорема 78: Теоремата за скобовата структура на интервалите на DFS

Нека $i, j \in \{1, \dots, n\}$ и $i \neq j$. Точно едно от следните е в сила.

- ❶ $[d[v_i], f[v_i]] \cap [d[v_j], f[v_j]] = \emptyset$ и в гората на обхождането, нито единият от v_i и v_j не е предшественик на другия.
- ❷ $[d[v_i], f[v_i]] \subset [d[v_j], f[v_j]]$ и в гората на обхождането, v_j е предшественик на v_i .
- ❸ $[d[v_i], f[v_i]] \supset [d[v_j], f[v_j]]$ и в гората на обхождането, v_i е предшественик на v_j .

Доказателство: Доказателството е чрез разглеждане на случаи и подслучай.

Случай I: $d[v_i] < d[v_j]$.

Случай I.a: $d[v_j] < f[v_i]$. От транзитивността на релацията $<$ имаме $d[v_i] < d[v_j] < f[v_i]$. Очевидно е, че v_i е сив точно когато $\text{time} \in [d[v_i], f[v_i]]$. Ерго, в този подслучай, v_j бива открит, докато v_i е сив. Съгласно Наблюдение 52, v_i е предшественик на v_j в гората на обхождането и $d[v_i] < d[v_j] < f[v_j] < f[v_i]$. Последното влече $[d[v_i], f[v_i]] \supset [d[v_j], f[v_j]]$.

И така, в този подслучай хем $[d[v_i], f[v_i]] \supset [d[v_j], f[v_j]]$, хем в гората на обхождането v_i е предшественик на v_j . Тоест, ❸ е в сила.

Случай I.b: $f[v_i] < d[v_j]$. Но $d[v_i] < f[v_i]$ и $d[v_j] < f[v_j]$ съгласно Наблюдение 51, така че $d[v_i] < f[v_i] < d[v_j] < f[v_j]$. В този подслучай, v_i е черен, когато v_j бива открит. Кое то означава, че рекурсивното викане, в което $x = v_i$ е приключило преди момента $d[v_i]$. Съгласно Наблюдение 52, нито единият от v_i и v_j не е предшественик на другия в гората на обхождането. От друга страна, очевидно е, че $[d[v_i], f[v_i]] \cap [d[v_j], f[v_j]] = \emptyset$.

И така, в този подслучай хем $[d[v_i], f[v_i]] \cap [d[v_j], f[v_j]] = \emptyset$, хем нито единият от v_i и v_j не е предшественик на другия в гората на обхождането. Тоест, ❶ е в сила.

Случай II: $d[v_j] < d[v_i]$. Доказателството е аналогично на доказателството на **Случай I** с разменени v_i и v_j . Тук заключението е, че или $[d[v_i], f[v_i]] \subset [d[v_j], f[v_j]]$ и в гората на обхождането v_j е предшественик на v_i , тоест, ❷ е в сила, или $[d[v_i], f[v_i]] \cap [d[v_j], f[v_j]] = \emptyset$ и нито единият от v_i и v_j не е предшественик на другия в гората на обхождането, тоест, ❶ е в сила. $\square$

Следствие 28

За всеки два интервала: или единият се съдържа строго в другия, или те не се пресичат.

Дефинираме, че “бял път” е път, чиито върхове са бели. Тази дефиниция е по отношение на даден момент от изпълнението на DFS.

Теорема 79: Теоремата за белите пътища (Theorem 22.9 в [34, стр. 608])

Връх v_i е предшественик на връх v_j в гората на обхождането тстк в момента $d[v_i]$ съществува бял път от v_i до v_j .

Доказателство: ➔ В едната посока, допускаме, че v_i е предшественик на v_j в гората на обхождането. Ще докажем, че в момента $d[v_i]$ съществува бял път от v_i до v_j . Тук няма допускане, че $i \neq j$, така че потенциално v_i и v_j може да са един и същи връх. Ако $v_i = v_j$, то такъв път има: това е тривиалният път v_i с дължина нула. Наистина, неговият единствен връх v_i е бял в момента $d[v_i]$.

Нека $v_i \neq v_j$. Тогава v_i е същински предшественик на v_j . Нека p е уникалният път от v_i до v_j в гората на обхождането. Нека $p = v_i, v_{\ell_1}, \dots, v_{\ell_t}, v_j$. Очевидно v_i е същински предшественик на всеки от върховете $v_{\ell_1}, \dots, v_{\ell_t}, v_j$.

От Теорема 78 знаем, че v_i е същински предшественик на u тстк $[d[v_i], f[v_i]] \supset [d[u], f[u]]$. Заклучаваме, че $d[v_i] < d[v_{\ell_1}], \dots, d[v_i] < d[v_{\ell_t}], d[v_i] < d[v_j]$, така че всеки от върховете $v_{\ell_1}, \dots, v_{\ell_t}, v_j$ е бял в момента $d[v_i]$. Предвид факта, че и v_i е бял в момента $d[v_i]$, заключаваме, че p бял път в момента $d[v_i]$. ✓

⬅ В другата посока, допускаме, че в момента $d[v_i]$ в G съществува бял път p от v_i до v_j . Ще докажем, че v_i е предшественик на v_j в гората на обхождането. Допускаме противното: v_i не е предшественик на v_j в гората на обхождането. Тогава $v_i \neq v_j$, понеже предшествението е рефлексивно. БОО, нека v_j е първият връх в p , който не е наследник на v_i . Нека v_k е върхът преди v_j в p :

$$p = \underbrace{v_i, \dots, v_k}_{\text{наследници на } v_i}, \underbrace{v_j}_{\text{не е наследник на } v_i}$$

Може $v_k = v_i$, а може и $v_k \neq v_i$: това няма значение за доказателството. Ключовите наблюдения са тези.

- Първо, $f[v_k] \leq f[v_i]$. Ето защо. Ако $v_k \neq v_i$, то, съгласно Теорема 78, трябва да е вярно, че $[d[v_i], f[v_i]] \supset [d[v_k], f[v_k]]$, така че $f[v_k] < f[v_i]$. Ако пък $v_k = v_i$, очевидно $f[v_k] = f[v_i]$. Във всеки случай, $f[v_k] \leq f[v_i]$.
- Второ, $d[v_i] < d[v_j]$. Тъй като сме допуснали, че v_i не е предшественик на v_j , не можем да ползваме Теорема 78 директно. Разглеждаме момента $d[v_i]$. По допускане, в този момент всички върхове от p са бели, така че v_j е бял, така че v_j не е открит в момента $d[v_i]$. От кода на Алгоритъм 19 ясно се вижда, че $d[v_i]$ се случва, когато $x = v_i$. Заклучаваме, че моментът $d[v_j]$ е по-късен от $d[v_i]$, понеже той настъпва по време на викане на Алгоритъм 19, в което $x = v_j$. Щом $d[v_j]$ е по-късен от $d[v_i]$, вярно е, че $d[v_i] < d[v_j]$.
- Трето, $f[v_k] < d[v_j]$. Ето защо. Според Теорема 78, щом v_j не е наследник на v_i в гората на обхождането и $d[v_i] < d[v_j]$, трябва да е изпълнено $d[v_i] < f[v_i] < d[v_j] < f[v_j]$. Но щом $f[v_k] \leq f[v_i]$, наистина $f[v_k] < d[v_j]$.

От друга страна, $f[v_k] < d[v_j]$ е невъзможно. Ако $f[v_k] < d[v_j]$, съгласно Наблюдение 51 е изпълнено $d[v_k] < f[v_k] < d[v_j]$. Това означава, че измежду v_k и v_j , първият открит връх е v_k , следователно v_j е бял в момента $d[v_k]$. Щом v_k непосредствено предшества v_j в пътя p , трябва в G да има ребро (v_k, v_j) . От кода на Алгоритъм 19 ясно се вижда, че няма как връх v_k да бъде финализиран преди DFS-VISIT да бъде извикана върху всички върхове от неговия списък на съседство, включително и върху v_j —който е бял в момента $d[v_k]$, както вече видяхме—и тези извиквания да приключат. ⚡

Полученото противоречие опровергава допускането, че v_i не е предшественик на v_j в гората на обхождането. Заклучаваме, че ако в момента $d[v_i]$ съществува бял път от v_i до v_j , то v_i е предшественик на v_j в гората на обхождането. $\square$

Следствие 29: Първият открит връх е последният финализиран връх

Нека U_i е множеството от върховете, достижими от v_i чрез бели пътища в момента $d[v_i]$. Връх v_i е първият връх от U_i , който DFS открива, и е последният връх от U_i , който DFS финализира.

8.5.3.2 Категоризация на ребрата на ориентиран граф от DFS

Нека $G = (V, E)$ е ориентиран мултиграф с възможни примки. Нека A е гората на обхождането, реализирана чрез масива π . DFS(G) категоризира ребрата от E по следния начин.

Дървесни ребра. Това са точно ребрата на A . С други думи, (u, v) е дървесно ребро, ако u и v са върхове от едно и също дърво $T \in A$, като u е родителят на v в T .

Ребра назад. (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като v е предшественик на u в T . Тоест, в T има път от v до u . Примките в G задължително са в тази категория.

Ребра напред. (u, v) е ребро напред, ако u и v са върхове от едно и също дърво $T \in A$, като u е предшественик на v в T , но (u, v) не е дървесно ребро. Тоест, (u, v) не е ребро в T .

Ребра настрани. (u, v) е ребро настрани, ако u и v не са в отношение на предшествие в A . Това означава, че

- или u и v са върхове от едно и също дърво от A , но нито единият не е предшественик на другия,
- или са върхове от две различни дървета от A .

В [34, стр. 609] съответните термини са *tree edges*, *back edges*, *forward edges* и *cross edges*. Примери за граф и DFS върху него, след който се срещат и четирите вида ребра, има на Фигура 22.4 в [34, стр. 605] и на Фигура 22.5 в [34, стр. 607], като на Фигура 22.5.с е показан и графът, прерисуван като дърветата на обхождането плюс останалите ребра.

Как да разберем по време на работата на DFS дадено ребро от кой вид е? Ребрата на G биват обхождани на ред 4 от DFS-VISIT. Нека реброто е (x, y) . Първо да разгледаме възможностите за цвета на y .

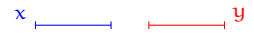
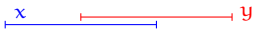
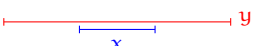
- Ако y е бял, то реброто (x, y) е дървесно, понеже условието на ред 5 е TRUE и изпълнението отива на ред 6, където x става родителят на y в дървото; това означава, че (x, y) става ребро от дървото.
- Ако y е сив, то y е предшественик на x в дървото. Това следва от Наблюдение 52.
- Ако y е черен, то или y е наследник на x и с него вече сме приключили, или y не е в отношение на предшествие с x и с него сме приключили. Тоест, (x, y) е ребро напред или ребро настрани. Само от черния цвят на y не можем да категоризираме точно (x, y) .

Теорема 80: Възможностите за $d[x]$, $d[y]$, $f[x]$ и $f[y]$ на $(x, y) \in E$ след DFS(G)

Това са всички възможности след DFS(G) за взаимното разположение на $d[x]$, $d[y]$, $f[x]$ и $f[y]$, където $(x, y) \in E$:

1. $d[x] < d[y] < f[y] < f[x]$ тстк (x, y) е дървесно ребро или ребро напред,
2. $d[y] < d[x] < f[x] < f[y]$ тстк (x, y) е ребро назад,
3. и $d[y] < f[y] < d[x] < f[x]$ тстк (x, y) е ребро настрани.

Доказателство: Да разгледаме възможностите за взаимните разположения на $d[x]$, $d[y]$, $f[x]$ и $f[y]$. Предвид това, че тези са две по две различни и че $d[x] < f[x]$ и $d[y] < f[y]$ (Наблюдение 51), има точно $3 + 2 + 1 = 6$ възможности от чисто комбинаторни съображения. С по-подробен анализ, възможностите стават само три.

1.  Това е невъзможно. Щом (x, y) е ребро, y е елемент на $\text{adj}[x]$ и няма как x да бъде финализиран на ред 10 преди бялото му дете y да бъде открито на ред 5.
2.  Това е невъзможно съгласно Следствие 28.
3.  y е предшественик на x . В такъв случай y е сив и (x, y) е ребро назад. Всички примки са такива.
4.  x е родителят на y . В такъв случай или y е бял и (x, y) е дървесно ребро, или y е черен и (x, y) е ребро напред.
5.  Това е невъзможно съгласно Следствие 28.

6.  Това е възможно. Щом u е финализиран преди откриването на x , нито x е предшественик на u , нито u е предшественик на x . В този случай u е черен и (x, u) е ребро настрани.

Сега е ясно как да различаваме дали (x, u) , при черен връх u , е ребро напред или ребро настрани. Ако $d[x] < f[u]$, то (x, u) е ребро напред, ако $f[u] < d[x]$, то (x, u) е ребро настрани. $\square$

8.5.3.3 Категоризация на ребрата на неориентиран граф от DFS

Нека $G = (V, E)$ е неориентиран мултиграф с възможни примки. Нека A е гората на обхождането, реализирана чрез масива π . DFS(G) категоризира ребрата от E по следния начин.

Дървесни ребра. Това са точно ребрата на A .

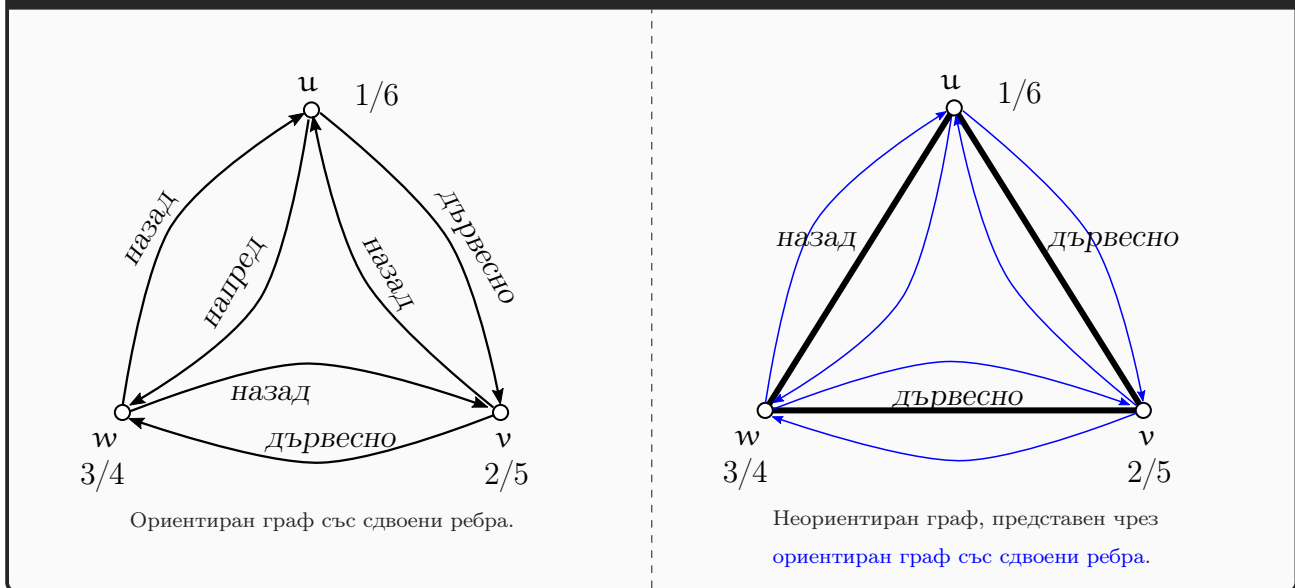
Ребра назад. (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като v е предшественик на u в T . Примките в G задължително са в тази категория.

С други думи, в неориентираните графи няма ребра напред или ребра настрани. Общото за ребрата напред и ребрата настрани е, че чрез тях откриваме черни върхове. Дали е невъзможно DFS върху неориентирани графи да открива черни върхове? Естествено, че е възможно. Но винаги, когато DFS върху неориентиран граф открие черен връх—тоест, връх u на ред 5 е черен—е вярно, че неориентираното ребро (x, u) **вече** е било обходено от DFS и **вече** е било категоризирано като дървесно ребро или ребро напред. Същността на тази аргументация е в това, че

1. ако гледаме на даден граф със сдвоени (вж. дискусията на стр. 432) ориентирани ребра като на ориентиран граф, DFS може да открие ребра напред или ребра настрани, като всяко такова ребро е от двойка, другото ребро от която е дървесно ребро или ребро назад;
2. но ако гледаме на същия граф като на неориентиран граф, който само е представен като ориентиран—всяко неориентирано ребро (u, v) е представено чрез двойка ориентирани ребра (u, v) и (v, u) —то няма как да категоризираме едно и също неориентирано ребро по два начина, съответно за двете ориентирани ребра, които го представят. И категоризирането на реброто никога не става като ребро напред или настрани.

Разгледайте Фигура 8.5. Тя сравнява резултатите от работата на DFS върху ориентиран и неориентиран граф. Двата графа имат едни и същи списъци на съседства, но графът вдясно се смята за неориентиран и всяка двойка сдвоени ориентирани ребра в представянето (в синьо) отговаря на едно неориентирано ребро на графа (в черно), докато при графа вляво всяко ориентирано ребро в представянето отговаря на точно едно ориентирано ребро на графа. И при ориентирания, и при неориентирания граф има точно две дървесни ребра след приключването на DFS. Забележете, че реброто (u, v) в ориентирания граф е дървесно, а реброто (v, u) е ребро назад; тоест, ребрата от двойката биват категоризирани по различен начин. От друга страна, в неориентирания граф на тях отговаря само едно ребро и то бива категоризирано като дървесно, защото първият път, в който прекосяваме едното от двете отговарящи му ориентирани ребра от представянето (в синьо), откриваме бял връх (v) .

Фигура 8.5 : Ориентиран граф със сдвоени ребра и неориентиран граф.



Фигура 8.5 е добра илюстрация на факта, че в неориентиран граф не може да има ребра напред. Сравнете ориентираното ребро напред (u, w) в ориентирания граф с неориентираното ребро (u, w) в неориентирания граф – очевидно е, че при сдвоени ориентирани ребра, ребро може да бъде категоризирано като ребро напред само ако другото ребро от двойката вече е било прекосено и категоризирано като ребро назад. Ако графът е неориентиран и това са само ориентираните ребра от преставянето, очевидно е, че реброто като един обект ще бъде категоризирано съгласно първото прекосяване, а именно като ребро назад. С аналогични разсъждения може да изведем, че в неориентирани графи няма ребра настрани.

Теорема 81 доказва формално, че при неориентираните графи ребрата са само дървесни и ребра назад.

Теорема 81: Theorem 22.10 в [34, стр. 610]

При всяка работа на DFS върху неориентиран граф G , всяко обходено ребро е или дървесно, или ребро назад.

Доказателство: Разглеждаме произволно ребро $(u, v) \in E(G)$. БОО, нека $d[u] < d[v]$. Тогава $f[v] < f[u]$, понеже $v \in \text{adj}[u]$ и съгласно Наблюдение 52. Очевидно u е сив в момента $f[v]$. Да разгледаме двете възможности за обхождане на реброто (u, v) :

1. Ако то бъде обходено първо в посока от u към v , то чрез него ще открием бял връх (v) и ще го категоризираме като дървесно.
2. Ако то бъде обходено първо в посока от v към u , то чрез него ще открием сив връх (u) и ще го категоризираме като ребро назад. $\square$

8.5.3.4 Изследване на цикличност на графи чрез DFS

И за ориентирани, и за неориентирани графи може да отговорим ефикасно на въпроса дали графът е цикличен, използвайки DFS.

Теорема 82: Граф е цикличен тстк DFS открива ребро назад

Нека G е ориентиран или неориентиран граф. G е цикличен тогава и само тогава, когато произволно изпълнение на $\text{DFS}(G)$ открива поне едно ребро назад.

Доказателство: ➡ В едната посока, нека има цикъл c . Очевидно DFS открива върховете на цикъла последователно, но не в непременно непрекъсната последователност. Нека $v \in V(c)$ е този връх на цикъла, който бива открит последен от DFS. Но v има поне един съсед или дете в c .

1. Ако G е неориентиран, то v има поне два съседа u и w , които са от цикъла. Ако G е мултиграф, може $u = w$, ако c е примка, може дори $v = u = w$: това е без значение за доказателството.
2. Ако G е ориентиран, v има поне едно дете u , който е от цикъла. Възможно е $u = v$, ако c е примка: това е без значение за доказателството.

Когато DFS-VISIT минава през списъка на съседство на v , открива u като сив връх и реброто (v, u) бива категоризирано като ребро назад.

⬅ В другата посока, да допуснем, че $e = (u, v)$ е ребро назад. Ако G е неориентиран, нека обхождането става в посока от u към v . Тогава v е предшественик на u в дървото на обхождането. Тогава в дървото има път от v до u (или между v и u , в неориентирания случай), който път заедно с реброто e дава цикъл. Този аргумент е валиден дори когато e е примка; тогава въпросният път се състои от един единствен връх, а именно $u = v$. $\square$

Наблюдение 53

При пускане на DFS върху G от един връх s , броят на дървесните ребра, които DFS открива, зависи единствено от G , а не от представянето му чрез списъци. Нека G' е подграфът, достижим от s . Очевидно G' не зависи от представянето. Тогава броят на дървесните ребра, които DFS открива, е $|V(G')| - 1$.

Броят на ребрата от всички останали видове е $|E(G')| - (|V(G')| - 1) = |E(G')| - |V(G')| + 1$ и очевидно също не зависи от представянето на G . Ако G е неориентиран, останалите ребра са само ребра назад. Но ако G е ориентиран, останалите ребра са от (най-много) три вида и това, по колко има от всеки вид, може да зависи от представянето на G .

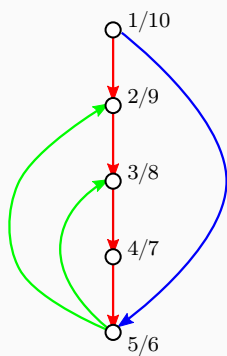
Илюстрация: Фигура 8.6 показва две различни изпълнения на DFS върху един и същи граф, от един и същи начален връх, но с различни представяния на графа (различни списъци на съседство). Броят на дървесните ребра е един и същи—винаги четири, защото върховете са пет—но останалите ребра се категоризират като ребра назад и ребра напред по различни начини и с различен брой ребра в тези категории. $\square$

Фигура 8.6 : Различен брой ребра назад.

→
дървесно

→
назад

→
напред

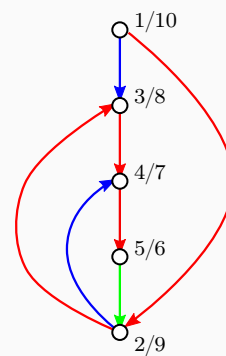


Две ребра назад.

→
дървесно

→
назад

→
напред



Едно ребро назад.

Лекция 9

Топологическо сортиране. Намиране на силно свързани компоненти. Намиране на срязващи върхове и мостове.

Резюме: Разглеждаме два линейни алгоритъма за топологическо сортиране на дагове: алгоритъмът на Kahn и алгоритъмът на Tarjan. Разглеждаме линейен алгоритъм за намиране на силно свързани компоненти на ориентиран граф. Разглеждаме линейни алгоритми за намиране на срязващите върхове и мостовете на неориентиран граф.

9.1 Фундамент

Навсякъде в тази лекция, нека $G = (V, E)$ е ориентиран граф.

Определение 73: Топологическо сортиране

Топологическо сортиране на G , или накратко *топо-сортиране* на G , е всяка биекция $h : V \rightarrow \{1, \dots, n\}$, такава че за всяко ребро $(u, v) \in E$ е вярно, че $h(u) < h(v)$.

Забележка 2: Трите смисъла на “топологическо сортиране”

Фразата “топологическо сортиране” означава едно от следните три неща:

- или действието по конструирането на това сортиране, тоест работата на алгоритъма,
- или резултатът от работата на алгоритъма, тоест наредбата,
- или самият алгоритъм.

От контекста би трябвало да е ясно какво точно се има предвид.

Теорема 83 и Теорема 84 са от курса Дискретни Структури. Тук са сложени за пълнота.

Теорема 83: НДУ за съществуване на топологическо сортиране

Съществува топологическо сортиране на G тогава и само тогава, когато G е даг.

Източник в G , на английски *source*, е всеки връх в G , чиято полустепен на входа е нула.
Сифон в G , на английски *sink*, е всеки връх в G , чиято полустепен на изхода е нула.

Теорема 84: Поне един източник и поне един сифон в даг

Във всеки даг има поне един източник и поне един сифон.

Следното твърдение е прекалено очевидно, за да бъде доказано формално. Да си припомним, че ако $U \subset V$, " $G - U$ " означава графа, получен от G с изтриване на върховете от U .

Наблюдение 54: Дефинираща характеристика на топологическото сортиране

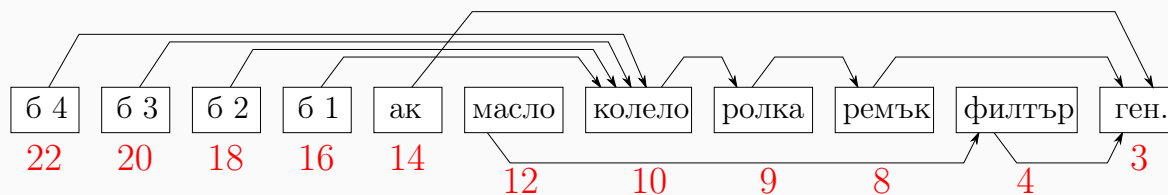
Нека G е даг и нека $h : V \rightarrow \{1, \dots, n\}$ е биекция. h е топо-сортиране тогава и само тогава, когато $\forall v \in V : v$ е източник в $G - \{h^{-1}(1), \dots, h^{-1}(h(v) - 1)\}$.

Пример за даг и негово топологическо сортиране е показан на Фигура 9.1. Примерът идва от реална житейска ситуация: смяна на електрическия генератор на автомобил, като свалянето на генератора налага предварително да се извършат други дейности, които на свой ред налагат предварително да се извършат други дейности, и така нататък. Върховете на дага са дейностите, а ребрата съответстват на непосредствените предшества между дейностите.

Фигура 9.1 : Даг и неговото топо-сортиране.



Даг от дейности с ограничения “за да извършим x , първо трябва да извършим y ”.



Топо-сортиране на дага по намаляващи времена на финализиране на DFS.

Допълнение 47: За произхода на “топологическо сортиране”.

Името “топологическо сортиране”, без съмнение, идва от “топология” – дял на математиката, възникнал като абстракция на геометрията. По думите на Henning Makhholm в тази дискусия за произхода на името:

Graph theory was originally (and still sometimes is, depending on who you ask) considered a branch of topology.

This may sound strange to people with a modern education, where “topology” means more or less “the part of mathematics that deals abstractly with continuity and limits, without using real numbers” – or at least without giving the real numbers any central position in the theory. However, earlier on, “topology” appears to have been a catch-all term for “the part of mathematics that isn’t about numbers or geometric magnitudes”. (This was before algebraists stopped pretending that algebra is necessarily about numbers). Only later did a

distinction between what we now call topology and discrete mathematics become common.

In this old usage, “topological sorting” simply means “the kind of sorting you can define without reference to comparison of numbers”.

Терминът “topological ordering” се появява в статията на Kahn от 1962 г. [86], където се казва:

In recent years much work has been done on the computational analysis of problems which can be formulated as networks with distinct elements. In particular, this class of problems include PERT (Program Evaluation Review Technique) which is used as management schedule tool. To a large extent the feasibility of network computations depends upon the ability to arrange the network information in topological order. A list in topological order has a special property. Simply expressed: proceeding from element to element along any path in the network, one passes through the list in one direction only. Stated another way, a list in topological order is such that no element appears in it until after all elements appearing on all paths leading to the particular element have been listed.

“PERT” е средство от 50-те години на XX век за управление на проекти, като описанието на задачите в даден проект задава нещо като даг от зависимости (задача X преди задача Y). За подробности вижте [статията в уикипедия](#).

Допълнение 48: Команда `tsort`.

Типичните UNIX инсталации имат средство за топологическо сортиране: командата `tsort`. Подробна информация за нея може да се получи с `info tsort`. Ето пример за използването ѝ. Това е примерът с ремонта на автомобил от Фигура 9.1. Елементите, които трябва да бъдат топологически сортирани, се изписват в двойки, които означават ограниченията кое преди кое да бъде. Тези двойки са ориентираните ребра на графа. Разполагането на двойките по редове е без значение: програмата чете списък и го разбива на двойки. Известен недостатък на представянето на графа само чрез ребрата е, че няма как да се опишат изолираните върхове, ако има такива.

```
$ tsort <<EOF
> източване-масло маслен-филтър
> маслен-филтър генератор
> отхлабване-ролка отхлабване-на-ремък
> отхлабване-на-ремък генератор
> сваляне-колело отхлабване-ролка
> болт-1 сваляне-колело
> болт-2 сваляне-колело
> болт-3 сваляне-колело
> болт-4 сваляне-колело
> EOF
```

```

болт-1
болт-2
болт-3
болт-4
източване-масло
сваляне-колело
маслен-филтър
отхлабване-ролка
отхлабване-на-ремък
генератор
$

```

9.2 Алгоритъм на Tarjan за топологическо сортиране

Ефикасен алгоритъм за топологическо сортиране на даг е показаната в тази секция модификация на DFS, предложена от Tarjan [149]. Същината на този алгоритъм е да се наредят върховете по **намаляващи** f -стойности, където $f[1..n]$ е масивът от времената на финализиране, генерирани от $\text{DFS}(G)$. Примерът на Фигура 9.1 показва точно това. Времената на откриване и финализиране са показани в червено до върховете на дага, а след това е показана и редицата от върховете, обратно сортирани по времената на финализиране.

Сортирането на върховете по намаляващи f -стойности може да стане имплицитно в процеса на работата на DFS. Топологическото сортиране се реализира чрез масив $T[1..n]$, който е глобално видим. Масивът се пълни с върхове отдясно наляво, като върхът с най-малка f -стойност е в най-дясна позиция и така нататък. Ето и псевдокод.

Алгоритъм 20: Tsort

```

TSORT(Даг  $G = (\{1, \dots, n\}, E)$ )
1   $T[1..n]$ , time и  $\ell$  са глобални
2  for  $i \leftarrow 1$  to  $n$ 
3       $\text{color}[i] \leftarrow \text{white}$ 
4       $\pi[i] \leftarrow \text{NIL}$ 
5   $\text{time} \leftarrow 0$ ,  $\ell \leftarrow n$ 
6  for  $i \leftarrow 1$  to  $n$ 
7      if  $\text{color}[i] = \text{white}$ 
8          TSORT-REC( $G, i$ )
9  return  $T[1..n]$ 

```

Алгоритъм 21: Tsort-rec

```

TSORT-REC( $G = (V, E)$ ;  $x \in V$ )
1   $\text{time}++$ 
2   $d[x] \leftarrow \text{time}$ 
3   $\text{color}[x] \leftarrow \text{gray}$ 
4  foreach  $y \in \text{adj}[x]$ 
5      if  $\text{color}[y] = \text{white}$ 
6           $\pi[y] \leftarrow x$ 
7          TSORT-REC( $G, y$ )
8   $\text{color}[x] \leftarrow \text{black}$ 
9   $\text{time}++$ 
10  $f[x] \leftarrow \text{time}$ 
11  $T[\ell] \leftarrow x$ 
12  $\ell--$ 

```

Интуитивна аргументация на коректността на алгоритъма е, че най-малката f -стойност задължително е на сифон—ако графът е даг, той има поне един сифон съгласно Теорема 84—така че върхът в $T[n]$ е сифон. Махайки сифона от дага, получаваме пак даг, така че следващата по големина f -стойност е на сифон в новия даг, така че $T[n-1]$ е сифон в новия даг, и така нататък. При това не може да се появят ребра отдясно наляво, по отношение на масива T , защото от сифоните не излизат ребра.

Ето и формално доказателство на коректността. В него не се използват сифони; за сифони говорихме в неформалната обосновка.

Теорема 85: Коректността на топологическото сортиране на Tarjan

Ако G е даг, пермутацията $\pi = \langle u_1 u_2 \dots u_n \rangle$ на върховете на G , такава че $f[u_1] > f[u_2] > \dots > f[u_n]$ след терминирането на $\text{DFS}(G)$, реализира топологическо сортиране.

Доказателство: Нека (u, v) е произволно ребро на графа. Съгласно Теорема 80, следните възможности са изчерпателни:

- $d[u] < d[v] < f[v] < f[u]$, ако (u, v) е дървесно ребро или ребро напред. Важното в случая е, че $f[u] > f[v]$. Това влече, че връх u е вляво от връх v в пермутацията: $\pi = \dots u \dots v \dots$. Ерго, реброто (u, v) не противоречи на топологическото сортиране.
- $d[v] < d[u] < f[u] < f[v]$, което обаче е невъзможно в текущия контекст! Това би означавало, че (u, v) е ребро назад, а, съгласно Теорема 82, наличието на ребро назад означава, че G е цикличен, в противоречие с допускането, че G е даг.
- $d[v] < f[v] < d[u] < f[u]$, ако (u, v) е ребро настрани. Отново, имаме $f[u] > f[v]$, така че u е вляво от v в пермутацията и реброто (u, v) не противоречи на топологическото сортиране. $\square$

Важно наблюдение, което ще използваме в Секция 9.4 е, че този алгоритъм не дава съобщение за грешка, ако входният граф не е даг, а приключва работата си и генерира масива $T[1..n]$ от върхове. Разбира се, ако G е цикличен, T не реализира топологическо сортиране.

Сложността по време на топологическото сортиране на Tarjan е $\Theta(n + m)$. Това следва от сложността по време на DFS и факта, че топологическото сортиране на Tarjan добавя само $\Theta(1)$ работа към рекурсивните викания.

Задача 58: Второ топологическо сортиране

Предложете ефикасен алгоритъм, който получава даг G като вход и или връща две различни топологически сортирания на G , ако G има повече от едно топологическо сортиране, или намира едно топологическо сортиране и връща него плюс съобщение, че няма друго топологическо сортиране, ако G има само едно топологическо сортиране. Анализирате коректността и сложността по време.

Решение: Да се намерят всички топологически сортирания на G би било най-добре, но това е нереалистично, понеже всички топо-сортирания може да са $n!$ [†]. Тук се иска нещо много по-малко: или две различни топо-сортирания, или едно топо-сортиране плюс индикация, че второ няма.

От курса по Дискретни Структури ни е известно, че ако G има Хамилтонов път, то този Хамилтонов път е единствен. Тук ще докажем нещо сходно: G има единствено топо-сортиране тстк G има Хамилтонов път.

- В едната посока е очевидно: ако G има Хамилтонов път p , то p е единственият Хамилтонов път и G има едно единствено топо-сортиране, което е точно същото като наредбата на върховете в p .

[†]Това е най-лошият случай за броя на топо-сортиранията – G няма ребра изобщо, така че всяка пермутация на върховете реализира топо-сортиране.

- В другата посока, ако топо-сортирането е единствено, то за всеки два върха u и v , които са съседни в него и u е вляво от v , трябва да е вярно, че има ребро (u, v) ; ако това не е вярно, те може да бъдат разменени, при което топо-сортирането би останало валидно, защото останалите ребра ще продължат да са “правилно вчесани” отляво надясно. Но щом за всеки два съседа в ρ има ребро от левия към десния, то топо-сортирането директно дава път, който е Хамилтонов, понеже съдържа всички върхове.

Това ни дава идея за решение на задачата. Пускаме алгоритъм за топо-сортиране, да кажем алгоритъма на Tarjan, след което за всеки два поредни върха u и v , в този ред, проверяваме дали има ребро (u, v) . Ако се окаже, че за всеки два поредни върха има такова ребро, връщаме топо-сортирането плюс индикация, че друго топо-сортиране няма. В противен случай, за първата двойка поредни върхове u и v , за която няма ребро (u, v) , генерираме второ топо-сортиране, което се получава от даденото чрез транспозиция на u и v , и връщаме двете топо-сортирания.

Коректността следва директно от доказателството горе. Да разгледаме сложността по време. Правим стандартното допускане, че графът е представен чрез списъци на съседство. Първата фаза, а именно алгоритъма на Tarjan, работи във време $\Theta(n + m)$. Но втората фаза също работи във време $\Theta(n + m)$, защото отново става дума за “преброяване” на всички списъци: за всеки връх u в топо-сортирането, който не е последният, в най-лошия случай прочитаме целия му списък, за да видим дали следващия връх в топо-сортирането е в този списък. Това е същото като “преброяването” на всички списъци, което, както знаем, става във време $\Theta(n + m)$.

Дори да се наложи да се генерира второ топо-сортиране като пермутация на върховете, това става във време $\Theta(n)$, така че сложността по време на алгоритъма остава $\Theta(n + m)$. $\square$

9.3 Алгоритъм на Kahn за топологическо сортиране

Този алгоритъм е описан още през 1962 г. [86]. Той работи, в някакъв смисъл, обратно на алгоритъма на Tarjan. Докато алгоритъмът на Tarjan върви отляво надясно, тоест от сифоните назад, алгоритъмът на Kahn върви отляво надясно, тоест от източниците напред. Наричат алгоритъма на Kahn “топологическо сортиране чрез BFS”. За да се подчертае приликата с BFS тук ползваме цветовете бял, сив и черен за маркиране на различните от гледна точка на алгоритъма състояния на върховете. Не особено съществена разлика с BFS е, че сега множеството S се инициализира с всички източници в G , докато при BFS, неговото съответствие, опашката Q се инициализира с точно един връх.

TSORT-KAHN(Дaг $G = (V, E)$, като $V = \{1, \dots, n\}$)

```

1   $S \leftarrow \emptyset$ 
2  for  $i \leftarrow 1$  to  $n$ 
3       $A[i] \leftarrow 0$ 
4  for  $x \leftarrow 1$  to  $n$ 
5      foreach  $y \in \text{adj}[x]$ 
6           $A[y]++$ 
7  for  $i \leftarrow 1$  to  $n$ 
8      if  $A[i] = 0$ 
9           $S \leftarrow S \cup \{i\}$ 
10      $\text{color}[i] \leftarrow \text{gray}$ 
11     else
12          $\text{color}[i] \leftarrow \text{white}$ 
```

```

13  k ← 0
14  while S ≠ ∅ do
15      x ← произволен елемент на S
16      S ← S \ {x}
17      k++
18      T[k] ← x
19      foreach y ∈ adj[x]
20          A[y]--
21          if A[y] = 0
22              S ← S ∪ {y}
23              color[y] ← gray
24      color[x] ← black
25  if k < n
26      return "Грешка! G не е даг."
27  else
28      return T[1 .. n]

```

Множеството S може да бъде реализирано например чрез АТД стек или опашка. Аргументацията за коректност и сложност остава валидна и в двата случая.

Коректност. Във всеки момент от работата на алгоритъма, нека B означава множеството от черните върхове, нека A_B е множеството от клетките на A , които съдържат елементите на B , и нека $A_{\bar{B}}$ е множеството от останалите клетки на A . Коректността на алгоритъма на Kahn може да бъде осмислена лесно, ако имаме предвид Наблюдение 54: алгоритъмът строи топо-сортирането, редейки отляво надясно източници от $G - B$.

Приемаме за очевиден ефектът от работата на редове 1–13, поради което го обобщаваме в следното наблюдение без доказателство.

Наблюдение 55: Инициализацията (редове 1–13) на топол. сортиране на Kahn

При първото достигане на ред 14 е вярно, че:

- $B = \emptyset$.
- S е множеството от източниците на G и се състои точно от сивите върхове.
- A съдържа полустепените на входа на върховете на G .

Теорема 86: Коректността на топологическото сортиране на Kahn

Ако входният G е даг, алгоритъмът на Kahn го сортира топологически чрез масива T . В противен случай, алгоритъмът на Kahn връща съобщение за грешка.

Доказателство: Да допуснем, че G е даг. Следното твърдение е инвариант за **while** цикъла (редове 14–24).

Инвариант 27: while-цикълът на алгоритъма на Kahn

Всеки път, когато изпълнението е на ред 14,

- ① S е множеството от източниците на $G - V$ и се състои точно от сивите върхове.
- ② V е множеството $\{T[i] \mid 1 \leq i \leq k\}$.
- ③ Нито един връх от V не е дете на връх от $G - V$.
- ④ $A_{\overline{V}}$ съдържа полустепените на входа на $G - V$.
- ⑤ $T[1 \dots k]$ съдържа топологическо сортиране на подграфа на G , индуциран от V .

База. Разглеждаме първото достигане на ред 14.

Ще покажем ①. Според Наблюдение 55, в този момент $V = \emptyset$, така че $G - V = G$ и ① става “ S е множеството от източниците на G и се състои точно от сивите върхове”, което е част от Наблюдение 55.

Ще покажем ②. От една страна, в този момент $V = \emptyset$. От друга страна, в този момент $k = 0$ заради ред 13, така че $T[1 \dots k]$ е празен.

Ще покажем ③. Тъй като $V = \emptyset$, твърдението е вярно в празния смисъл.

Ще покажем ④. Щом $V = \emptyset$, то $A_{\overline{V}} = A$ и ④ става “ A съдържа полустепените на входа на върховете на G ”, което е част от Наблюдение 55.

Ще покажем ⑤. Щом $k = 0$, то $T[1 \dots k]$ е празен. От друга страна, подграфът на G , индуциран от V , е празният граф. Ерго, ⑤ е вярно в празния смисъл. ✓

Поддръжка. Да допуснем, че инвариантът е в сила в даден момент t , в който изпълнението е на ред 14 и **while** ще бъде изпълнен поне още веднъж. Последното влече, че Q не е празна. Нека t' е моментът на следващото достигане на ред 14.

Ще покажем ①. На ред 16, x престава да е елемент на S , а на ред 24, x става черен и по този начин става елемент на V . Тогава в момента t' , x вече хем не е сив, хем не е връх на $G - V$.

Множеството от източниците в $G - V - x$ се състои точно от тези върхове, които в $G - V$ имат един единствен родител, който е x . При допускането, че ④ е в сила в момента t , това са точно тези деца y на x , за които $A[y] = 1$ преди да бъдат “обработени” от **for**-цикъла (редове 19–23). Тези деца получават A -стойност нула на ред 20, влизат в S на ред 22, а на ред 23 стават сиви. Но $G - V - x$ от момент t е $G - V$ в момент t' заради ред 24, така че ① остава в сила в t' .

Ще покажем ②. Но ② е в сила в момента t , след което, от една страна, точно един връх от сив става черен (връх x) и никой черен връх не си мени цвета, а от друга страна, k бива инкрементиран на ред 17 и, спрямо новото k , $T[k]$ получава стойност x .

Ще покажем ③. По допускане (①), в момента t , връх x е източник в $G - V$, понеже $x \in S$ в този момент. Това означава, че x не е дете на нито един връх от $G - V$ в момент t . След това към V се добавя един единствен връх, а именно x (ред 24). Очевидно в момент t' , ③ е в сила за текущото V .

Ще покажем ④. Изпълнението на тялото на **while**-цикъла засяга $A_{\overline{V}}$ по два начина. Първо, връх x , който в момент t не е в V , “влиза” в V на ред 24; ерго, в момент t , връх x е в $A_{\overline{V}}$, но

в t' вече не е в $A_{\overline{B}}$. Що се отнася само до x , твърдение ④ е истина, понеже, от една страна, x престава да е връх от $G - B$, а от друга страна, $A_{\overline{B}}$ престава да съдържа съответен нему елемент.

Второ, изпълненията на ред 20 намаляват с единица полустепеня на входа на всички деца на x . Твърдим, че в момент t , $\forall y \in \text{adj}[x] : A[y] > 0$. Това следва от факта, че всяко дете y на x е връх от $G - B$, който не е източник в $G - B$, и че $A[y]$ е полустепеня на входа на това дете съгласно ④.

- Защо y е връх от $G - B$? Защото от ③ знаем, че в момента t , нито един черен връх не е дете на връх от $G - B$, а x в t е връх от $G - B$ и y е дете на x .
- Защо y не е източник? Защото в момент t , източниците в $G - B$ са точно върховете в S съгласно ①, а източниците не са деца по дефиниция.

И така, при всяко изпълнение на ред 20 по време на текущото изпълнение на **while**-цикъла се декрементира положително число, което след декрементирането може да е най-малко нула. Очевидно стойностите $A[y]$ на всички деца y на x в момента t' са точно полустепените на входа на тези върхове в текущия $G - B$.

Ще покажем ⑤. Очевидно $T[1..k]$ нараства с един елемент между момент t и момент t' и този елемент е връх x от ред 15. Трябва да покажем, че в момент t' няма ребро с начало $T[k]$ и край някой $T[j]$, $1 \leq j \leq k - 1$. Но това следва от това, че

- същият връх, който е $T[k]$ в момент t' , е източник в $G - B$ съгласно ①
- и това, че нито един връх от B не е дете на връх от $G - B$ в момент t .

Терминация. Приемаме за очевидно това, че при терминирането на **while**-цикъла (което е момента, в който $S = \emptyset$ съгласно ред 14), $k = n$ тстк G е даг. При текущото допускане, че G е даг, имаме $k = n$ при достигането на ред 28. Съгласно част ⑤ на инварианта, $T[1..n]$ съдържа топологическо сортиране на подграфа на G , индуциран от B . Но от ② знаем, че в този момент $B = \{T[i] \mid 1 \leq i \leq k\}$, което очевидно е същото като множеството $V(G)$. И така, подграфа на G , индуциран от B , е самият G , така че масивът $T[1..n]$ съдържа топологическо сортиране на G . $\square$

Сложност по време. Алгоритъмът на Kahn има линейна сложност по време. Намирането на полустепените на входа на редове 2–6 става във време $\Theta(n) + \Theta(n + m) = \Theta(n + m)$, защото в редове 2–6 минаваме през всички списъци на съседства с константна работа върху всеки елемент. Намирането на източниците в G на редове 7–12 става във време $\Theta(n)$. Цялото изпълнение на **while**-цикъла (редове 14–24) става във време $\Theta(n + m)$, защото пак се минава през всички списъци на съседство, с константна работа върху елемент. Като цяло, сложността по време е $\Theta(n + m)$.

9.4 Алгоритъм на Kosaraju за намиране на силно свързаните компоненти

Задачата за намиране на силно свързаните компоненти на ориентиран граф има изненадващо много приложения, например за ефикасно решение на задачата 2SAT от съждителната логика (вижте Допълнение 81).

Навсякъде в тази секция $G = (V, E)$ е ориентиран граф. За по-голяма яснота на изложението няма да именуваме върховете с цели положителни числа, а с малки латински букви.

Знаем, че релацията на силна свързаност $\mathcal{SC}(G)$ е релация на еквивалентност. Множеството от нейните класове на еквивалентност означаваме с “ $\mathcal{SCC}(G)$ ” (Нотация 7). Подграфите на G , индуцирани от елементите на $\mathcal{SCC}(G)$, са силно свързаните компоненти на G .

Забележете приликата между фактор-релация (Определение 36) и фактор-граф (Определение 74).

Определение 74: Фактор-граф

Нека $R \subseteq V \times V$ е релация на еквивалентност. Нека $\mathcal{X}$ е множеството от класовете на еквивалентност на R . *Фактор-графът на G спрямо R ^a* е ориентираният граф G/R с множество от върхове $\mathcal{X}$, като за всеки два различни класа $P, Q \in \mathcal{X}$, ребро (P, Q) в G/R съществува тогава и само тогава, когато съществува връх $a \in P$ и съществува връх $b \in Q$, такива че $(a, b) \in E$.

^aНа английски терминът е *quotient graph*.

Несъществена разлика между фактор-релация и фактор-граф е, че фактор-релацията е рефлексивна, докато фактор-графът няма примки.

В тези лекционни записки ще разглеждаме фактор-графи само спрямо релацията на силна свързаност. Когато кажем “фактор-графът на G ”, имаме предвид “фактор-графът на G спрямо $\mathcal{SC}(G)$ ”.

Нотация 9: $G/\mathcal{SC}(G)$ е фактор-графът на G

Фактор-графът на G се означава с “ $G/\mathcal{SC}(G)$ ”.

Очевидно $\mathcal{SCC}(G)$ е множеството от върховете на $G/\mathcal{SC}(G)$.

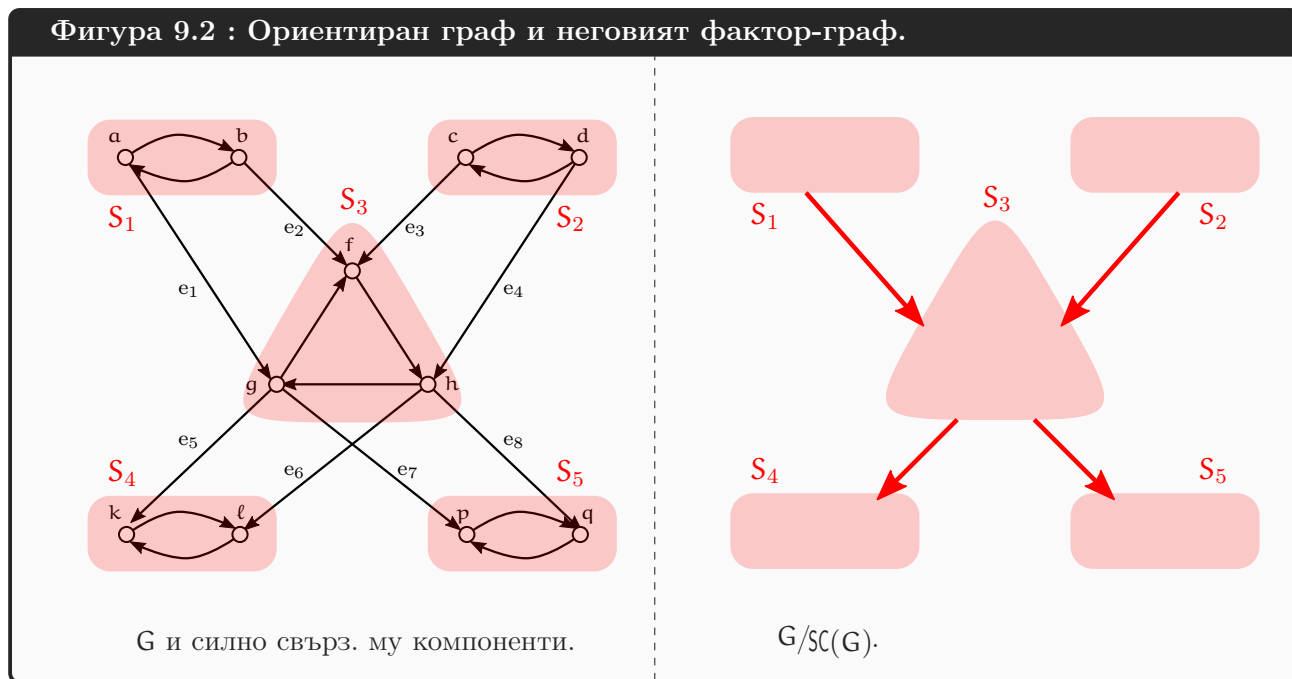
Наблюдение 56: $G/\mathcal{SC}(G)$ е даг

$G/\mathcal{SC}(G)$ е даг. Ако в $G/\mathcal{SC}(G)$ има цикъл, за всеки два различни върха X и Y от цикъла—тези върхове са различни елементи на $\mathcal{SCC}(G)$ —е вярно, че от всеки връх на X има път до всеки връх на Y в G , и от всеки връх на Y има път до всеки връх на X в G . Тогава върховете на X и Y всъщност са в една и съща силно свързана компонента на G , в противоречие с предишното допускане, че X и Y са различни елементи на $\mathcal{SCC}(G)$.

Това, което нарекохме “фактор-граф”, на английски се нарича *the condensed graph* или *the component graph* [34, стр. 617].

Фигура 9.2 показва ориентиран граф и неговият фактор-граф.

Фигура 9.2 : Ориентиран граф и неговият фактор-граф.



9.4.1 Неуспешен опит за решение чрез едно пускане на DFS

Да се опитаме да решим задачата за намиране на силно свързаните компоненти на G чрез следната малка модификация на DFS, наречена SC-DUMMY. Както знаем, силно свързаните компоненти не са само множества от върхове, а са истински графи с върхове и ребра, но за простота генерирането на силно свързаните компоненти ще се състои само в генерирането на техните множества от върхове. Идеята е, върнатото T на ред 11 да бъде $SCC(G)$.

SC-DUMMY($G = (V, E)$)

```

1  S и time са глобални
2  T ← ∅
3  foreach v ∈ V
4      color[v] ← white
5  time ← 0
6  foreach v ∈ V
7      if color[v] = white
8          S ← ∅
9          SC-DUMMY-REC(G, v)
10         T ← T ∪ {S}
11  return T

```

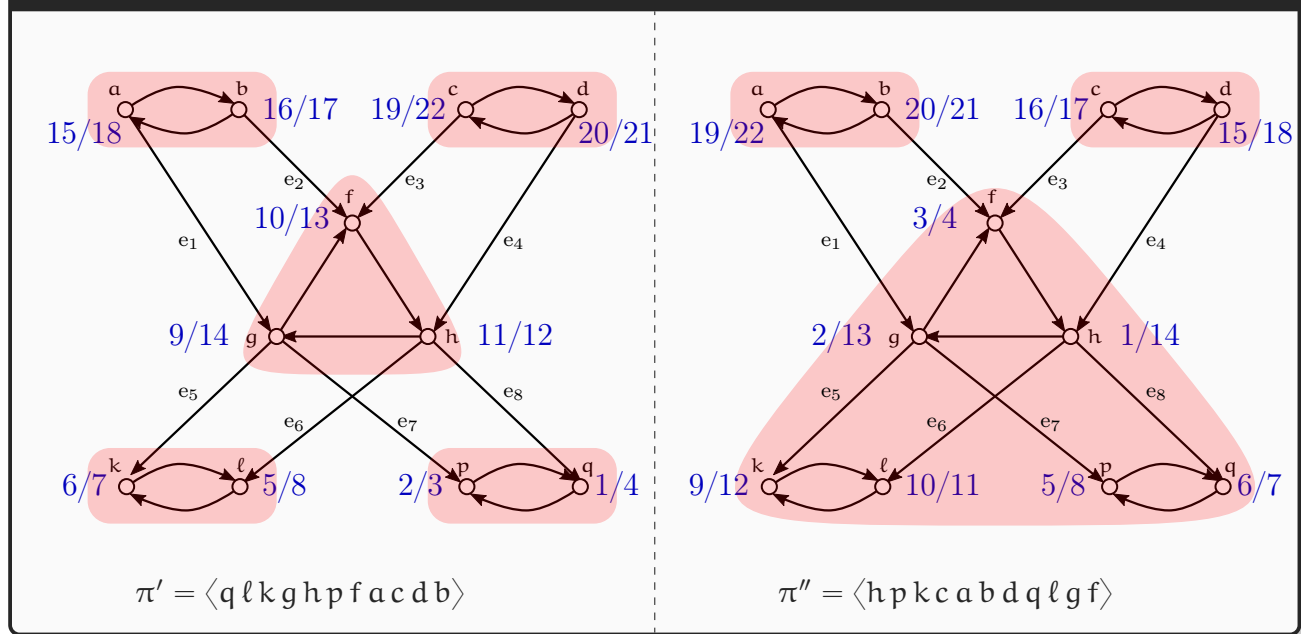
SC-DUMMY-REC($G = (V, E)$; $x \in V$)

```

1  time++
2  d[x] ← time
3  color[x] ← gray
4  S ← S ∪ {x}
5  foreach y ∈ adj[x]
6      if color[y] = white
7          SC-DUMMY-REC(G, y)
8  color[x] ← black
9  time++
10 f[x] ← time

```

Да си представим работата на SC-DUMMY върху графа G от Фигура 9.2. Ключово наблюдение е, че полученото разбиване T на ред 11 зависи от това в какъв ред се появяват върховете във **for**-цикъла на ред 6. Това наблюдение е илюстрирано на Фигура 9.3. Там е показана работата на SC-DUMMY при две различни наредби π' и π'' на върховете. π' и π'' водят до различни разбивания: ако наредбата е π' , получаваме точно $SCC(G)$, но ако наредбата е π'' , получаваме разбиване на само три дяла. Ерго, SC-DUMMY не работи коректно винаги.

Фигура 9.3 : SC-DUMMY при различни наредби π' и π'' на върховете.

Лесно се вижда защо π'' “не става”, ако искаме да намерим силно свързаните компоненти. Започвайки обхождането от връх h във време 1, обхождането “излиза” от силно свързаната компонента, индуцирана от $\{f, g, h\}$, и се “прехвърля” в две други силно свързани компоненти, а именно в тази, индуцирана от $\{p, q\}$ във време 5, и в тази, индуцирана от $\{k, l\}$, във време 9, преди да финишира върху h във време 14. Така с едно викане на SC-DUMMY-REC на ред 9, алгоритъмът открива три силно свързани компоненти и слага върховете им в S , което е недопустимо, ако искаме S на ред 10 в SC-DUMMY да съдържа върховете на точно една силно свързана компонента.

Наблюдение 57

Ако искаме след излизането от първото викане на SC-DUMMY-REC на ред 9 в SC-DUMMY, на ред 10 множеството S да съдържа върховете на **само една** силно свързана компонента, първата стойност на v на ред 6 трябва да е връх от силно свързана компонента G_1 , такава че няма нито едно ребро от връх от G_1 към връх от друга силно свързана компонента. Ако и само ако това е изпълнено, SC-DUMMY-REC не може да “излезе” от G_1 , така че $S = V(G_1)$ при първото приключване на SC-DUMMY-REC.

Да вземем за пример графа от Фигура 9.3. Ако SC-DUMMY на ред 6 разглежда върховете на графа в реда, в който са в π' , то първата стойност на v на ред 6 е q и наистина последващата работа на SC-DUMMY-REC открива върховете на само една силно свързана компонента, а именно тази, която се индуцира от $\{p, q\}$.

Определение 75: Благоприятна силно свързана компонента

Силно свързана компонента, от която обхождането не може да излезе, ще наричаме *благоприятна силно свързана компонента*.

Ребрата, които не принадлежат на нито една силно свързана компонента, са от особена важност за задачата, която решаваме. Те са ребрата, които позволяват на обхождането да се “прехвърля” от една силно свързана компонента в друга.

Определение 76: Важните ребра

Важните ребра на G са тези ребра, чиито начало и край са от различни силно свързани компоненти.

Например, на графа G от Фигура 9.2, важните ребра са именуваните ребра: $e_1, e_2, \dots, e_8$.

Наблюдение 58

Благоприятните силно свързани компоненти на G са тези силно свързани компоненти, от които не “излизат” важни ребра.

Но винаги ли има благоприятна силно свързана компонента? Отговорът е тривиално “да”, ако съобразим следното.

Наблюдение 59

Благоприятните силно свързани компоненти на G са тези силно свързани компоненти, които се индуцират от сифоните на фактор-графа $G/\mathcal{SC}(G)$. Тъй като $G/\mathcal{SC}(G)$ е даг, той има поне един сифон съгласно Теорема 84, така че G има поне една благоприятна силно свързана компонента.

За да намерим силно свързаните компоненти чрез DFS, не е достатъчно първият връх, чиято стойност придобива v на ред 6, да е от благоприятна силно свързана компонента G_1 . След излизането от викането на ред 9 трябва следващият бял връх, чиято стойност придобива v на ред 6, да е от благоприятна силно свързана компонента на $G - V(G_1)$, за да не може DFS да излезе от нея. И така нататък до края на алгоритъма.

9.4.2 Решение чрез две пускания на DFS

Първо да разгледаме един помощен алгоритъм. Той е почти същият като SC-DUMMY на стр. 469. Единствената разлика между двата алгоритъма е тази: в SC-DUMMY основният **for** цикъл (редове 6–10) се изпълнява върху **произволна** наредба на върховете, докато алгоритъмът STRCONN получава една точно определена наредба на върховете като втори аргумент L и аналогичният **for** цикъл (редове 7–11) се изпълнява върху наредбата на върховете, зададена от L .

Алгоритъм 22: StrConn

```

STRCONN( $G = (V, E)$ ;  $L[1..n]$ : перм. на в-вете)
1  (*  $V = \{1, \dots, n\}$  *)
2   $S$  и  $time$  са глобални
3   $T \leftarrow \emptyset$ 
4  for  $i \leftarrow 1$  to  $n$ 
5       $color[i] \leftarrow white$ 
6   $time \leftarrow 0$ 
7  for  $i \leftarrow 1$  to  $n$ 
8      if  $color[L[i]] = white$ 
9           $S \leftarrow \emptyset$ 
10         SC-REC( $G, L[i]$ )
11          $T \leftarrow T \cup \{S\}$ 
12 return  $T$ 

```

Алгоритъм 23: SC-rec

```

SC-REC( $G = (V, E)$ ;  $x \in V$ )
1   $time++$ 
2   $d[x] \leftarrow time$ 
3   $color[x] \leftarrow gray$ 
4   $S \leftarrow S \cup \{x\}$ 
5  foreach  $y \in adj[x]$ 
6      if  $color[y] = white$ 
7          SC-REC( $G, y$ )
8   $color[x] \leftarrow black$ 
9   $time++$ 
10  $f[x] \leftarrow time$ 

```

Сега ще разгледаме алгоритъм, който намира ефикасно силно свързаните компоненти. Открит е от Rao Kosaraju през 1978 [2]. Необходимо ни е следното определение.

Определение 77: Транспониран граф

Транспонираният граф на G е графът $G^T = (V, \{(u, v) \mid (v, u) \in E\})$.

На английски терминът е *the transpose of G* ([34, стр. 616]). Без съмнение, прилагателното “транспониран” идва оттам, че матрицата на съседство на G^T е транспонираната матрица на съседство на G .

Алгоритъмът на Kosaraju вика алгоритъма за Tarjan на стр. 462 за топологическо сортиране и използва резултата от него като втори аргумент, викайки алгоритъма STRCONN. Това, че сега G може да не е даг, **няма значение**. Забележете, че алгоритъмът на Tarjan, ако получи не-даг на входа, не дава съобщение за грешка. Нещо повече, алгоритъмът на Tarjan работи напълно нормално и върху не-даг. Той връща наредба на върховете по намаляващи стойности на време на финализиране, което може да бъде направено за всеки ориентиран граф – в крайна сметка, алгоритъмът на Tarjan е DFS плюс малко допълнителни инструкции, а DFS работи върху произволни ориентирани графи. Разбира се, ако графът не е даг, то изходът няма смисъл на топологическо сортиране на графа, защото такава не съществува, но това е друго нещо.

Алгоритъм 24: Kosaraju

```

KOSARAJU(Ориентиран граф  $G = (V, E)$ )
1   $L[1..n] \leftarrow TSORT(G)$ 
2  конструирай  $G^T$ 
3   $Q \leftarrow STRCONN(G^T, L)$ 
4  return  $Q$ 

```

Коректност. Неформално говорейки, същността на доказателството за коректност е тази: в някакъв смисъл, алгоритъмът на Tarjan топо-сортира фактор-графа $G/\mathcal{SC}(G)$ (Теорема 87). В някакъв смисъл, който ще стане ясен след Определение 79 и Определение 80, полученият на

ред 1 масив L е такъв, че при обхождане на L отляво надясно, силно свързаните компоненти се появяват като източници на $G/\mathcal{SC}(G)$. Тогава, по отношение на G^T , при обхождането на L отляво надясно[†], силно свързаните компоненти се появяват като сифони на $G^T/\mathcal{SC}(G^T)$. Тоест, като благоприятни, в контекста на G^T обаче, силно свързани компоненти.

Определение 78: argmin и argmax

Да разгледаме произволна функция $f : X \rightarrow \mathbb{R}$, където X е произволен домейн. Нека f има минимум върху X . Тогава

$$\operatorname{argmin}_{x \in X} f(x) \stackrel{\text{def}}{=} \{x \in X \mid (\forall y \in X : f(x) \leq f(y))\} \quad (9.1)$$

“ argmax ” се дефинира аналогично.

Имената идват от “*argument minimum*” и “*argument maximum*”.

Следва кратко обяснение за смисъла на тези понятия. Изразът

$$\min \{f(x) \mid x \in X\} \quad (9.2)$$

означава минималната функционална стойност върху множеството X . Ако имаме предвид такова x , върху което функцията достига минимална стойност, (9.2) не върши работа. Тогава трябва да използваме (9.1).

Тъй като f може да има много минимума върху X , има смисъл да говорим за (под)множеството от елементите на X , върху които f има минимум. Ако това множество е едно-елементно, то “ argmin ” означава неговия (единствен) елемент; аналогичното е в сила и за “ argmax ”.

Определение 79: Индуцирана наредба на силно свързаните компоненти

Нека е дадена наредба π на върховете на G . Говорим за биекция $\pi : V \rightarrow \{1, \dots, n\}$. Нека $\mathcal{SC}(G) = \{S_1, \dots, S_t\}$. Нека $\phi : \{1, \dots, t\} \rightarrow \{1, \dots, n\}$ е функцията-инекция, която връща минималния индекс в π на връх от S_i . Формално,

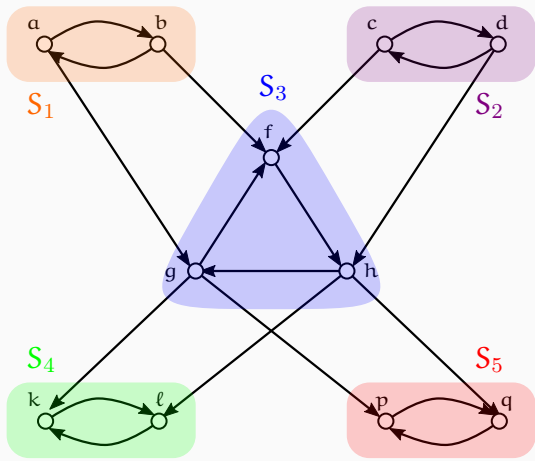
$$\forall i \in \{1, \dots, t\} : \phi(i) = \operatorname{argmin}_{u \in S_i} \pi(u)$$

Нека наредбата $\psi = \langle S_{j_1}, \dots, S_{j_t} \rangle$ е уникалната наредба на елементите на $\mathcal{SC}(G)$, за която $\phi(j_1) < \phi(j_2) < \dots < \phi(j_t)$. Казваме, че ψ е *наредбата на силно свързаните компоненти, индуцирана от π* .

Ако сме педантични, наредбата е не на силно свързаните компоненти, а на множествата от върховете на силно свързаните компоненти, но с цел по-кратко изложение може да игнорираме разликата между силно свързаните компоненти и техните множества от върхове.

Като пример да вземем графа G от Фигура 9.2. Той има единадесет върха и пет силно свързани компоненти. Фигура 9.4 показва същия граф вляво, а вдясно е показано как две различни наредби π_1 и π_2 на неговите върхове индуцират две различни наредби, съответно $\psi_1 = \langle S_5, S_4, S_3, S_1, S_2 \rangle$ и $\psi_2 = \langle S_1, S_2, S_3, S_4, S_5 \rangle$, на силно свързаните му компоненти.

[†]Викането на $\operatorname{STRCONN}(G^T, L)$ на ред 3 обхожда L отляво надясно на редове 7–11.

Фигура 9.4 : Различни наредби на V индуцират различни наредби на $\mathcal{SC}(G)$.

π_1 : $\begin{array}{cccccccccccc} & S_4 & & S_3 & & & & S_2 & & & & \\ & \text{q} & \text{l} & \text{k} & \text{g} & \text{h} & \text{p} & \text{f} & \text{a} & \text{c} & \text{b} & \text{d} \\ S_5 & & & & & & & & S_1 & & & \end{array}$

$\psi_1 = \langle S_5, S_4, S_3, S_1, S_2 \rangle$

π_2 : $\begin{array}{cccccccccccc} S_1 & & S_2 & & S_3 & & S_4 & & S_5 & & & \\ \text{a} & \text{b} & \text{d} & \text{c} & \text{h} & \text{g} & \text{k} & \text{l} & \text{p} & \text{q} & \text{f} & \end{array}$

$\psi_2 = \langle S_1, S_2, S_3, S_4, S_5 \rangle$

Определение 80: Индуцирано топо-сортиране на фактор графа

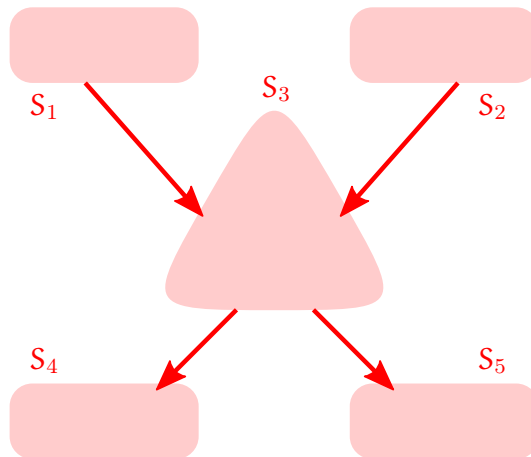
В контекста на Определение 79 и съгласно Наблюдение 54, ако е изпълнено

$$\forall k \in \{0, \dots, t-1\} : S_{j_{k+1}} \text{ е източник в } (G - (S_{j_1} \cup \dots \cup S_{j_k})) / \mathcal{SC}(G - (S_{j_1} \cup \dots \cup S_{j_k}))$$

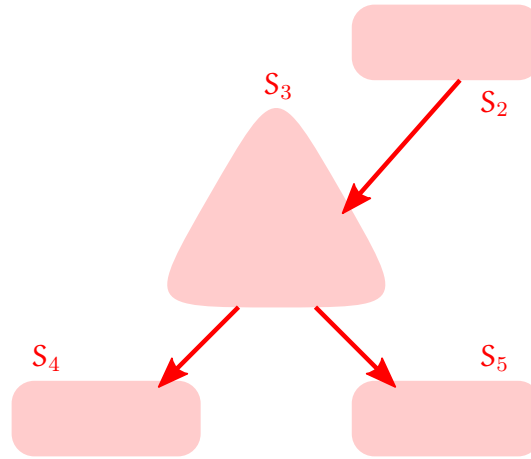
то ψ е топо-сортиране на $G/\mathcal{SC}(G)$. В такъв случай казваме, че π индуцира топо-сортиране на фактор-графа.

Да осмислим с примери Определение 80. Да разгледаме отново Фигура 9.4.

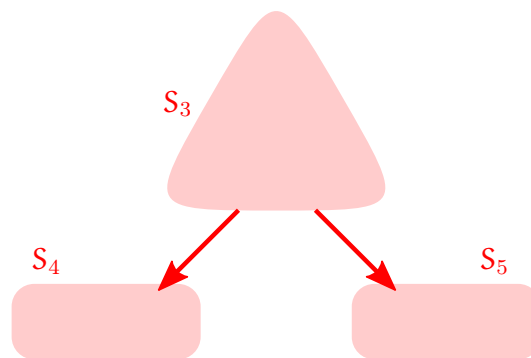
- Твърдим, че пермутацията $\pi_2 = \langle a b d c h g k l p q f \rangle$ индуцира топо-сортиране на фактор-графа. Да се убедим в това, имайки предвид, че π_2 индуцира наредбата на силно свързаните компоненти $\psi_2 = \langle S_1 S_2 S_3 S_4 S_5 \rangle$.
 - Нека $k = 0$. Тогава $S_{j_{k+1}}$ е S_{j_1} , което е S_1 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е празното множество. Наистина, S_1 е източник в $G/\mathcal{SC}(G)$.



- Нека $k = 1$. Тогава $S_{j_{k+1}}$ е S_{j_2} , което е S_2 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е S_1 . Наистина, S_2 е източник в $(G - S_1) / \mathcal{SC}(G - S_1)$.



- ♦ Нека $k = 2$. Тогава $S_{j_{k+1}}$ е S_{j_3} , което е S_3 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е $S_1 \cup S_2$. Наистина, S_3 е източник в $(G - (S_1 \cup S_2)) / \mathcal{SC}((G - (S_1 \cup S_2)))$.



- ♦ Нека $k = 3$. Тогава $S_{j_{k+1}}$ е S_{j_4} , което е S_4 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е $S_1 \cup S_2 \cup S_3$. Наистина, S_4 е източник в $(G - (S_1 \cup S_2 \cup S_3)) / \mathcal{SC}(G - (S_1 \cup S_2 \cup S_3))$.



- ♦ Нека $k = 4$. Тогава $S_{j_{k+1}}$ е S_{j_5} , което е S_5 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е $S_1 \cup S_2 \cup S_3 \cup S_4$. Наистина, S_5 източник в $(G - (S_1 \cup S_2 \cup S_3 \cup S_4)) / \mathcal{SC}(G - (S_1 \cup S_2 \cup S_3 \cup S_4))$.



- Твърдим, че пермутацията $\pi_1 = \langle q \ell k g h p f a c d b \rangle$ **не** индуцира топо-сортиране на фактор-графа. Да се убедим в това, имайки предвид, че π_1 индуцира наредбата на силно свързаните компоненти $\psi_1 = \langle S_5 S_4 S_3 S_1 S_2 \rangle$. Нека $k = 0$. Тогава $S_{j_{k+1}}$ е S_{j_1} , което е S_5 , а множеството $S_{j_1} \cup \dots \cup S_{j_k}$ е празното множество. Наистина, S_5 не е източник в $G / \mathcal{SC}(G)$.

Теорема 87: Алгоритъм 20 индуцира топо-сортиране на фактор-графа

Масивът $T[1 \dots n]$, който $\text{TSORT}(G)$ връща, индуцира топо-сортиране на $G / \mathcal{SC}(G)$.

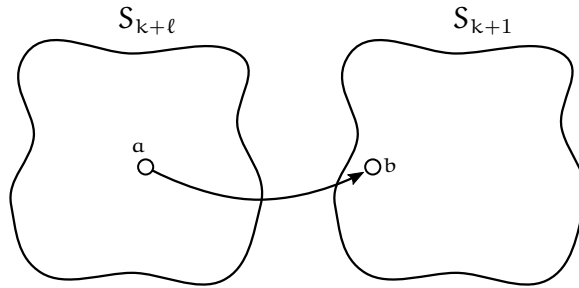
Доказателство: Разглеждаме произволно изпълнение на $\text{TSORT}(G)$. Нека направим така, че имената от Определение 79 да са в сила: изходът на $\text{TSORT}(G)$ е $\pi : V \rightarrow \{1, \dots, n\}$ и $\text{SCC}(G) = \{S_1, \dots, S_t\}$. Нека и тук $\phi : \{1, \dots, t\} \rightarrow \{1, \dots, n\}$ е дефинирана така

$$\forall i \in \{1, \dots, t\} : \phi(i) = \underset{u \in S_i}{\operatorname{argmin}} \pi(u)$$

и $\psi = \langle S_{j_1}, \dots, S_{j_t} \rangle$ е уникалната наредба на $S_1, \dots, S_t$, за която $\phi(j_1) < \phi(j_2) < \dots < \phi(j_t)$. За да избегнем двойното индексване, БОО, нека $\psi = \langle S_1, S_2, \dots, S_t \rangle$.

Нека k е произволно число от $\{0, \dots, t-1\}$. За краткост, нека $G - (S_1 \cup \dots \cup S_k)$ се нарича H . Твърдим, че S_{k+1} е източник в $H/\text{SC}(H)$.

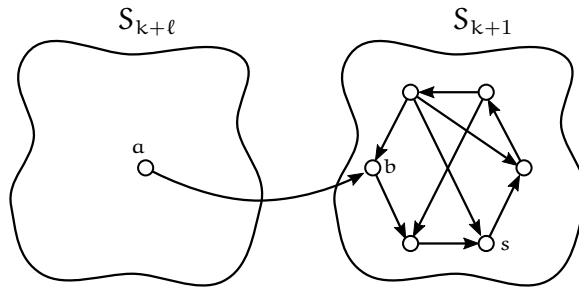
Да допуснем противното. Тогава съществува поне една $S_{k+\ell} \in \text{SCC}(H)$, където $2 \leq \ell \leq t-k$, такава че съществува връх $a \in S_{k+\ell}$ и съществува връх $b \in S_{k+1}$, такива че $(a, b) \in E(H)$.



Нека $s = \pi^{-1}(\phi(k+1))$ и $z = \pi^{-1}(\phi(k+\ell))$. На прост български, s е най-левият връх от S_{k+1} в π , а z е най-левият връх от $S_{k+\ell}$ в π . Естествено, $s \neq z$, понеже S_{k+1} и $S_{k+\ell}$ са различни множества от разбиването $\text{SCC}(G)$. Възможно е $s = b$ или $s \neq b$, $z = a$ или $z \neq a$, това е без значение за доказателството.

Първото ключово наблюдение е, че $\pi(s) < \pi(a)$. По конструкция, $\pi(z) \leq \pi(a)$, а $\pi(s) < \pi(z)$, понеже π индуцира наредба на силно свързаните компоненти, в която S_{k+1} предшества $S_{k+\ell}$, така че наистина $\pi(s) < \pi(a)$.

Повече не разглеждаме z , но разглеждаме s . Следната фигура илюстрира същественото за по-нататъшното доказателство.



Второто ключово наблюдение е, че s е достижим от a , но a не е достижим от s . И изобщо, за всеки връх $x \in S_{k+1}$ е вярно, че няма път $x \rightsquigarrow a$. Ако имаше такъв път, то S_{k+1} и $S_{k+\ell}$ нямаше да са различни елементи на $\text{SCC}(G)$.

Третото ключово наблюдение е, че $f[a] < f[s]$. Наистина, π е пермутацията, която се реализира от изхода на $\text{TSORT}(G)$, а, както знаем, $\text{TSORT}(G)$ връща масив от върховете, обратно сортиран по времената на финализиране. Щом $\pi(s) < \pi(a)$, задължително $f[s] > f[a]$.

Имайки предвид третото ключово наблюдение и Наблюдение 51, което влече $d[s] < f[s]$ и $d[a] < f[a]$, следните случаи са взаимно изключващи се и изчерпателни.

- $d[a] < f[a] < d[s] < f[s]$. Това е невъзможно, понеже s е достижим от a . Ако a е открит преди s , то s е бял в момента $d[a]$.

Нещо повече, всички върхове от S_{k+1} са бели в момента $d[a]$. Ето защо. Ако допуснем, че поне един връх от S_{k+1} бива открит преди a , то всички върхове от S_{k+1} , включително и s , ще бъдат и открити, и финализирани преди откриването на a , понеже няма път от нито един връх от S_{k+1} до a съгласно второто ключово наблюдение.

Щом s е достижим от a и всички върхове S_{k+1} са бели в момента $d[a]$, то в момента $d[a]$ съществува бял път $a \rightsquigarrow s$. Съгласно Теорема 79, a ще се окаже предшественик на s в гората на обхождането, така че, съгласно Теорема 78, в крайна сметка ще е изпълнено $d[a] < d[s] < f[s] < f[a]$, а не $d[a] < f[a] < d[s] < f[s]$.

- $d[a] < d[s] < f[a] < f[s]$. Това е невъзможно съгласно Теорема 80.
- $d[s] < d[a] < f[a] < f[s]$. Това също е невъзможно. Ако беше така, съгласно Теорема 78, s щеше да е предшественик на a в гората на обхождането. Но a не е достижим от s и няма как s да е негов предшественик.

Получените противоречия доказват, че допускането ни, че съществува поне една $S_{k+\ell} \in \text{SCC}(H)$, където $2 \leq \ell \leq t - k$, такава че съществува връх $a \in S_{k+\ell}$ и съществува връх $b \in S_{k+1}$, такива че $(a, b) \in E(H)$, е погрешно. ⚡

Тогава S_{k+1} е източник в $H/\mathcal{SC}(H)$.

Щом за всяко $k \in \{0, \dots, k-1\}$ е вярно, че S_{k+1} е източник в

$$(G - (S_1 \cup \dots \cup S_k))/\mathcal{SC}(G - (S_1 \cup \dots \cup S_k))$$

то по дефиниция $\langle S_1, S_2, \dots, S_t \rangle$ индуцира топо-сортиране на фактор-графа $G/\mathcal{SC}(G)$. $\square$

Сега започваме да разглеждаме свойствата на транспонирането по отношение на дагове и на общи ориентирани графи.

Наблюдение 60

Нека H е произволен даг. Източниците на H са точно сифоните на H^T и сифоните на H са точно източниците на H^T .

Наблюдение 61

Нека H е произволен даг и J е произволен негов индуциран подграф. Източниците на J са точно сифоните на J^T и сифоните на J са точно източниците на J^T .

Обосновката е елементарна: J също е даг, защото, ако в H няма цикли, няма как в негов подграф да се появят цикли. Щом J е даг, Наблюдение 60 дава желанния резултат веднага.

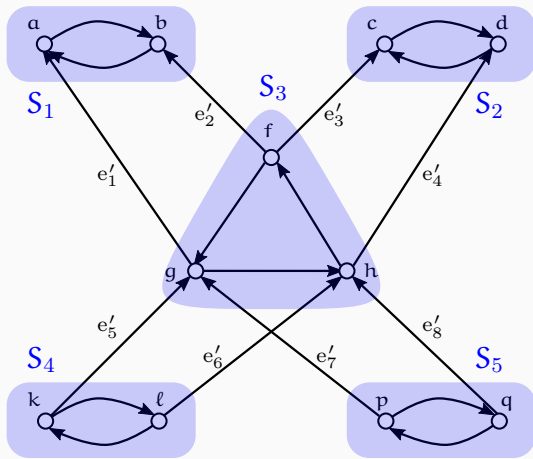
Наблюдение 62 се основава на очевидния факт, че два върха са силно свързани в G тстк са силно свързани в G^T .

Наблюдение 62

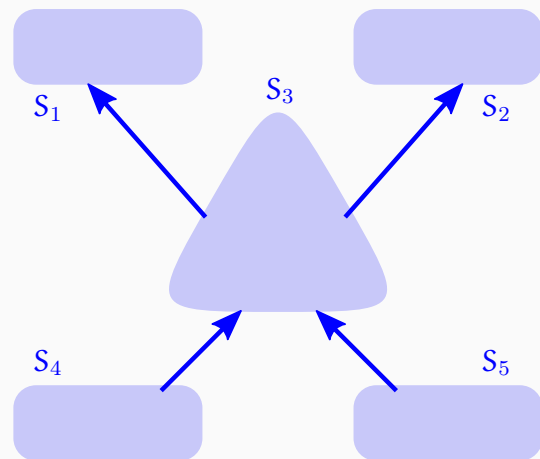
За всеки ориентиран граф G е вярно, че $\mathcal{SC}(G) = \mathcal{SC}(G^T)$, откъдето $\text{SCC}(G) = \text{SCC}(G^T)$. Нещо повече: $(G/\mathcal{SC}(G))^T = G^T/\mathcal{SC}(G^T)$. Оттук следва, че важните ребра в G са същите като в G^T , но с обратна ориентация.

Фигура 9.5 илюстрира тези наблюдения. Тя показва транспонирания граф на графа от Фигура 9.2, както и неговия фактор-граф, който очевидно е транспонираният граф на фактор-графа от Фигура 9.2. Ясно се вижда, че източниците и сифоните във фактор-графа и неговия транспониран се оказват разменени, а важните ребра в G и в G^T са същите, но с обратни посоки; примерно, $e_1 = (a, g)$ е важно ребро в G и на него съответства $e'_1 = (g, a)$, което е важно ребро в G^T , и така нататък.

Фигура 9.5 : Транспонираният граф на графа от Фигура 9.2.



Силно свързаните компоненти са същите.
Важните ребра са същите,
но с обратна ориентация.



Фактор-графът е транспониран.

Наблюдение 63 е директно очевидно следствие на Наблюдение 62 и Наблюдение 61.

Наблюдение 63

В контекста на Определение 80,

$$\forall k \in \{0, \dots, t-1\} : S_{j_{k+1}} \text{ е източник в } (G - (S_{j_1} \cup \dots \cup S_{j_k})) / \mathcal{SC}(G - (S_{j_1} \cup \dots \cup S_{j_k}))$$

тогава и само тогава, когато

$$\forall k \in \{0, \dots, t-1\} : S_{j_{k+1}} \text{ е сифон в } (G^T - (S_{j_1} \cup \dots \cup S_{j_k})) / \mathcal{SC}(G^T - (S_{j_1} \cup \dots \cup S_{j_k}))$$

Теорема 88: Коректността на Алгоритъм 24.

$\text{KOSARAJU}(G)$ намира свързаните компоненти на G в смисъл, че изходът Q е $\mathcal{SCC}(G)$.

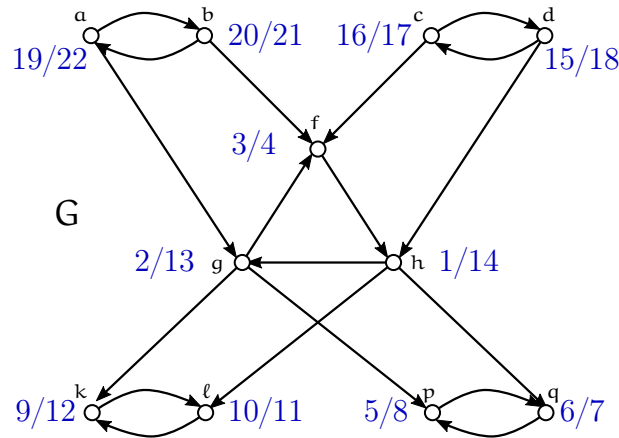
Доказателство: На ред 1 се вика $\text{TSORT}(G)$. Нека t е броят на силно свързаните компоненти. Съгласно Теорема 88, върнатият масив $L[1 \dots n]$ индуцира топо-сортиране на фактор-графа. По дефиниция (Определение 80), това означава, че L индуцира такава наредба на силно свързаните компоненти, че, ако, за кое да е $k \in \{0, \dots, t-1\}$, от G бъдат изтрети

върховете на първите k от тях, то $(k + 1)$ -вата силно свързана компонента е източник във фактор-графа на този граф, който се получава от G след въпросното изтриване на върхове.

На ред 2 се конструира транспонираният граф G^T . На ред се изпълнява $\text{STRCONN}(G^T, L)$. Но съгласно Наблюдение 63, L индуцира такава наредба на силно свързаните компоненти, че, ако, за кое да е $k \in \{0, \dots, t - 1\}$, от G^T бъдат изтрети върховете на първите k от тях, то $(k + 1)$ -вата силно свързана компонента е сифон във фактор-графа на този граф, който се получава от G^T след въпросното изтриване на върхове. На прост български, ако от G^T бъдат изтрети върховете на първите k силно свързани компоненти, то $(k + 1)$ -вата силно свързана компонента е благоприятна, и това е в сила за $0 \leq k \leq t - 1$.

От този факт и от факта, че основният цикъл на $\text{STRCONN}(G^T, L)$ (редове 7–11) обхожда върховете на G^T в реда на L , заключаваме, че при k -тото излизане от викането на SC-REC на ред 10, множеството S съдържа точно върховете на k -тата силно свързана компонента по отношение на наредбата на силно свързаните компоненти, индуцирана от L , за $1 \leq k \leq t$. $\square$

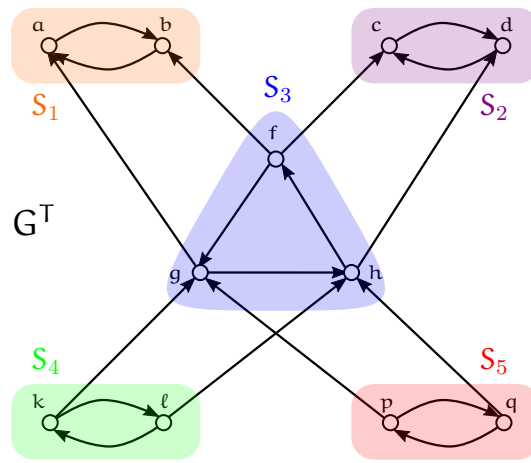
Ето пример за работата на Алгоритъм 24. Пак разглеждаме графа от Фигура 9.2. На ред 1 се изпълнява алгоритъмът на Tarjan, който по същество е DFS. Да кажем, че този DFS работи така:



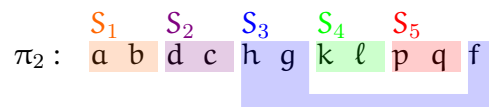
Тогава алгоритъмът на Tarjan връща

$$L = [a, b, d, c, h, g, k, l, p, q, f]$$

Нека π_2 е наредбата на върховете, която се реализира от L . Транспонираният граф G^T е този:



Очевидно π_2 индуцира наредбата на силно свързаните компоненти $\psi_2 = \langle S_1, S_2, S_3, S_4, S_5 \rangle$ (вижте пак Фигура 9.4). За удобство, ето отново как π_2 индуцира ψ_2 :



Да разгледаме изпълнението на $\text{StrConn}(G^T, L)$.

- След края на първото викане на SC-REC на ред 10, променливата S съдържа $\{a, b\}$ и на ред 11, променливата T става $\{\{a, b\}\}$.
- След края на второто викане на SC-REC на ред 10, променливата S съдържа $\{c, d\}$ и на ред 11, променливата T става $\{\{a, b\}, \{c, d\}\}$.
- След края на третото викане на SC-REC на ред 10, променливата S съдържа $\{f, g, h\}$ и на ред 11, променливата T става $\{\{a, b\}, \{c, d\}, \{f, g, h\}\}$.
- След края на четвъртото викане на SC-REC на ред 10, променливата S съдържа $\{k, l\}$ и на ред 11, променливата T става $\{\{a, b\}, \{c, d\}, \{f, g, h\}, \{k, l\}\}$.
- След края на петото викане на SC-REC на ред 10, променливата S съдържа $\{p, q\}$ и на ред 11, променливата T става $\{\{a, b\}, \{c, d\}, \{f, g, h\}, \{k, l\}, \{p, q\}\}$.

На ред 12 алгоритъмът връща $\{\{a, b\}, \{c, d\}, \{f, g, h\}, \{k, l\}, \{p, q\}\}$, което е точно $\text{SCC}(G^T)$, което е точно $\text{SCC}(G)$ заради Наблюдение 62.

Може да възникне този въпрос: дали може да опростим Алгоритъм 24 по следния начин: наместо на ред 1 да генерираме масива L от върховете в намаляващ ред по времената на финализиране и после да генерираме G^T на ред 2, за да извикаме STRCONN върху G^T и L , не може ли на ред 1 да генерираме масива $\bar{L}$ от върховете в **нарастващ ред** по времената на финализиране и после да викнем STRCONN директно върху **оригиналния** граф G и $\bar{L}$? Това би спестило генерирането на G^T . Асимптотиката на алгоритъма не би се променила, но определено алгоритъмът би станал по-прост.

Отговорът е, че **не може**. Наистина, има ориентирани графи, върху които можем да намерим силно свързаните компоненти по описания начин, спестявайки си работата по генериране на транспонирания граф. Но примерът, който току-що видяхме, е контрапример за тази идея. Да видим защо. Обратно нареденият на L масив е този

$$\bar{L} = [f, q, p, \ell, k, g, h, c, d, b, a]$$

Нека π_3 е наредбата на върховете, която се реализира от $\bar{L}$. π_3 индуцира тази наредба на силно свързаните компоненти:

$$\psi_3 = \langle S_3, S_5, S_4, S_2, S_1 \rangle$$

Ето защо, нагледно показано:

$$\bar{L}: \begin{array}{ccccccccc} S_3 & S_5 & S_4 & & S_2 & S_1 \\ f & q & p & \ell & k & g & h & c & d & b & a \end{array}$$

Ключовото наблюдение е това: ψ_3 **не е** пермутацията-инверсия на ψ_2 , въпреки че π_3 е пермутацията-инверсия на π_2 ! Наредбата ψ_3 започва със S_3 , което не е източник във фактор-графа на $G^\dagger$. Поради това, изпълнението на $\text{STRCONN}(G, \bar{L})$, започвайки обхождането от връх $f \in S_3$, ще се прехвърли в S_4 и S_5 и ще намери $S_3 \cup S_4 \cup S_5$ като една единствена силно свързана компонента. Кое е грешно.

И така, в общия случай не може да минем без генерирането на $G^\dagger$.

Сложност по време. Алгоритъмът на Kosaraju има линейна сложност по време. Работата на TSORT има линейна сложност – факт, в който вече се убедихме. Това, че сега графът-вход не е непременно даг, няма значение за това. Създаването на транспонирания граф също има линейна сложност: то може да се реализира чрез едно “минаване” през списъците на съседства на G , като, щом срещнем връх v в списъка на връх u (това означава ребро (u, v) в G), слагаме u в списъка на v в представянето на $G^\dagger$ (това има смисъл на слагане на ребро (v, u) в $G^\dagger$). И накрая, работата на STRCONN се извършва очевидно в линейно време.

9.5 Алгоритми за намиране на срязващите върхове и на мостовете

9.5.1 Фундамент

Сега разглеждаме само неориентирани графи. В тази секция, $G = (V, E)$ означава неориентиран свързан граф (без примки, не е мултиграф).

Определение 81: Срязващ връх и мост

Срязващ връх в G е всеки $u \in V$, такъв че $G - u$ има повече от една свързани компоненти. *Срязващо ребро*, още се нарича *мост*, е всяко $e \in E$, такова че $G - e$ има две свързани компоненти.

[†] S_3 не е и сифон във фактор-графа, поради което $\bar{L}$ “не става” като наредба на върховете нито по отношение на G , нито по отношение на $G^\dagger$.

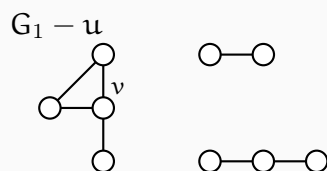
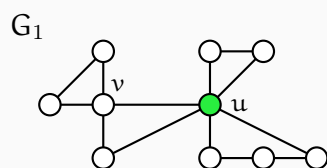
Определение 82: Срязващ връх и мост, алтернативно определение

Срязващ връх в G е всеки $u \in V$, такъв че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа u като вътрешен връх. Мост е всяко $e \in E$, такава че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа e .

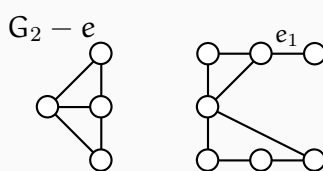
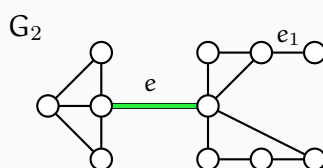
Наблюдение 64

Определения 81 и 82 са еквивалентни.

Лесно се вижда, че на всеки висящ връх в свързана компонента с повече от два върха съответства точно един срязващ връх, а именно единственият му съсед. Самият висящ връх не е срязващ. Ако свързаната компонента има точно два върха, тоест, ако е едно единствено ребро, то в нея срязващ връх няма. На всеки висящ връх съответства точно един мост, а именно единственото ребро, инцидентно с него; това остава в сила дори свързаната компонента да се състои само от това ребро. Фигура 9.6 илюстрира Определения 81 и 82. В графа G_1 , върховете u и v са срязващи, но е показано само изтриването на u . В графа G_2 , ребрата e и e_1 са мостове, но е показано само изтриването на e .

Фигура 9.6 : Срязващ връх и срязващо ребро.

u е срязващ връх (v също).



e е срязващо ребро (e_1 също).

Приложение на срязващи върхове и мостове: намиране на слаби места в мрежи. Срязващите върхове и мостовете са важни понятия, ако графът моделира някаква свързаност. Например, ако моделира комуникационна мрежа, или пътна мрежа, или електрическа мрежа.

Нека графът моделира комуникационна мрежа от компютри и жични връзки между тях. Нека мрежата е свързана в смисъл, че всеки два компютъра X и Y могат да си “говорят”; може би не директно, а през други компютри, но между X и Y има маршрут. Тогава срязващ връх в графа съответства на компютър, чиято повреда би довела до това, че мрежата би се разпадала на две или повече подмрежи, които не могат да си “говорят”: в разпаданата мрежа има компютри, между които няма маршрут. А мост отговаря на жична връзка, чиято повреда би довела до разпадане на мрежата на две подмрежи, които не могат да си “говорят”.

И така, срязващите върхове и мостовете отговарят на някакви критично важни компоненти на мрежата. Мрежа, чийто съответен граф има срязващи върхове или мостове е потенциално по-ненадеждна от мрежа, чийто съответен граф няма такива елементи. От гледна точка на надеждността, очевидно предпочитаме мрежи, чиито съответни графи нямат срязващи върхове или мостове.

Приложение на срязващи върхове: блокова декомпозиция. Започваме с няколко дефиниции.

Определение 83: Разцепване на срязващи върхове

Нека $u \in V$ е срязващ връх. Нека $G - u$ има k свързани компоненти, наречени $G_1, \dots, G_k$. Нека $z_1, \dots, z_k$ са k нови върхове – такива, които не са от V . Нека, за $1 \leq i \leq k$,

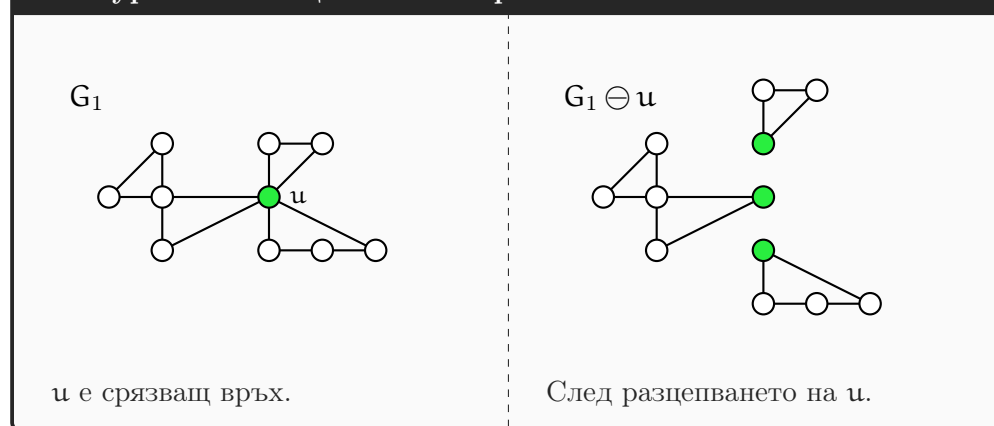
$$H_i = (V(G_i) \cup \{z_i\}, E(G_i) \cup \{(z_i, x) \mid x \in V(G_i) \cap N(u)\})$$

Да *разцепим* u означава да преобразуваме G в колекцията $H_1, \dots, H_k$. Графът-резултат означаваме с $G \ominus u$.

Очевидно $H_1, \dots, H_k$ са свързани графи.

Неформално казано, разцепването на срязващ връх u се състои в създаването на негови копия (самият u изчезва), на брой колкото са свързаните компоненти **спрямо него**, като всяко копие бива свързано с точно една от тези компоненти, с точно тези върхове в нея, с които u е бил свързан. Фигура 9.7 показва разцепване на връх – след разцепването на u се появяват три свързани компоненти.

Фигура 9.7 : Разцепване на връх.



Операцията “разцепване на срязващ връх” е асоциативна: можем да разцепваме различни върхове в каквато искаме последователност и резултатът е един и същи. Ако $u_1, \dots, u_t$ са различни срязващи върхове в G , пишем $G \ominus u_1 \ominus \dots \ominus u_t$. Резултатът от разцепването на всички срязващи върхове е предмет на следващата дефиниция.

Определение 84: Блокове в граф

Нека $G = (V, E)$ е граф. *Блок* в G е всеки максимален по включване свързан подграф, който не съдържа срязващи върхове.

Определение 85: Блокове в граф, алтернативна дефиниция

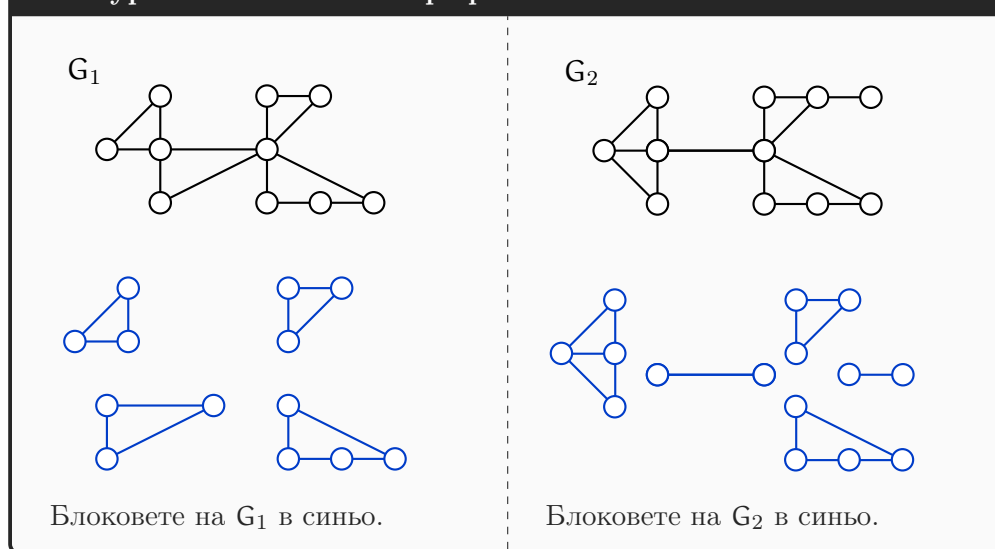
Нека $G = (V, E)$ е граф. *Блоковете на G* са тези свързани компоненти, които се получават от разцепването на всички срязващи върхове.

Наблюдение 65

Определения 84 и 85 са еквивалентни.

Фигура 9.8 илюстрира блоковете на два графа G_1 и G_2 .

Фигура 9.8 : Блокове в графи.



Има много алгоритми върху графи—например, алгоритмите за планарност—които извършват като предварителна обработка следните декомпозиции. Първо, намират свързаните компоненти, което може да стане ефикасно с BFS или DFS. Очевидно, графът е планарен тстк всяка свързана компонента е планарна. После за всяка компонента намират срязващите върхове и блоковете. Очевидно, графът е планарен тстк блоковете са планарни.

Ето друго приложение на срязващите върхове: декомпозицията на граф на блокове.

9.5.2 Алгоритъм за ефикасно намиране на срязващите върхове

Ако конструираме алгоритъм за намиране на срязващите върхове на G направо от Определение 81, той би бил следният. За всеки връх u , изтриваме u от G и с BFS или DFS проверяваме дали G остава свързан; ако остава, то u не е срязващ връх, ако ли не, то u е срязващ връх и го добавяме към множеството от срязващите върхове, после връщаме u и продължаваме по същия начин. Това би бил твърде неефекасен алгоритъм – дори да можем да трием и връщаме върхове в $\Theta(1)$: сложността му би била $\Theta(n(n + m))$.

Ефикасният алгоритъм, който ще разгледаме сега, е базиран на [144, Section 5.9.2, стр. 173–177]. Това е DFS, модифициран за намиране на срязващи върхове.

Всяко изпълнение на DFS върху свързан граф $G = (V, E)$ генерира дърво на обхождането T . Знаем, че $V(T) = V$ и $E(T) = \{e \in E \mid e \text{ е категоризирано като дървесно ребро от DFS}\}$. Разглеждаме три категории върхове на **графа G** по отношение на **дървото T** и необходимите и достатъчни условия за това, връх от дадена категория да е срязващ.

Теорема 89: Намиране на срязващите върхове: листата на дървото

В текущия контекст, нито едно листо не е срязващ връх в G .

Доказателство: Ако v е листо на T , очевидно $G - v$ има точно една свързана компонента. $\square$

Теорема 90: Намиране на срязващите върхове: коренът на дървото

В текущия контекст, коренът r на T е срязващ връх в G тстк степента на r в T е поне 2.

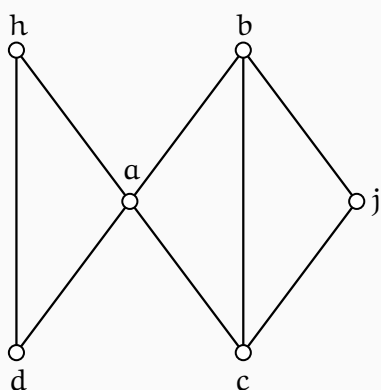
Подчертаваме, че разсъждаваме дали r е срязващ връх или не **в графа G** , а не в покриващото дърво T . Във всяко дърво, всеки връх, който не е висящ, задължително е срязващ, но не това ни интересува сега.

Доказателство: В едната посока, нека коренът r на T е срязващ връх в G . Тогава в G съществуват върхове u и v , такива че $u \neq v \neq r \neq u$ и всеки път между u и v съдържа r като междинен връх. БОО, нека u и v са съседи на r в G . Очевидно е, че в T , върховете u и v са деца на r . Щом r има поне две деца, ясно е, че степента на r в T е поне две.

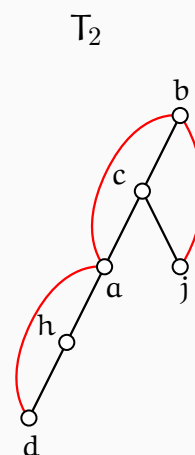
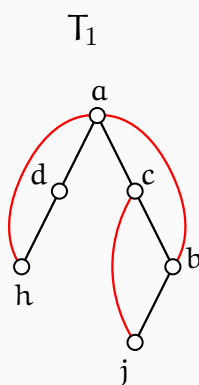
В другата посока, нека степента на r в T е поне две. Тогава r има поне две деца в T . Да ги наречем u и v . Твърдим, че всеки път p в G между u и v съдържа r като междинен връх. Наистина, ако не беше така, то u и v щяха да са върхове от едно и също поддърво $T[w]$, за някой съсед w на r , който може да е, а може и да не е, някой от u и v . $\square$

Фигура 9.9 е илюстрация на Теорема 90. Неориентираният граф е показан вляво, а вдясно са показани два възможни избора на стартов връх за DFS, които отговарят на два избора за корен на дървото на обхождането. Ако a е коренът, то той има степен в дървото две, което точно съответства на факта, че в графа връх a е срязващ. Ако b е коренът, то той има степен в дървото едно, което точно съответства на факта, че в графа b не е срязващ.

Фигура 9.9 : Кога коренът на дървото на DFS е срязващ връх.



Неориентираният граф.



Различни избори на корен водят до различни степени на корена в дървото: $d_{T_1}(a) = 2$ и $d_{T_2}(b) = 1$.

Теорема 91: Намиране на срязващите върхове: не-листа, не-корен

В текущия контекст, всеки връх u , който не е нито листо, нито коренът, е срязващ връх тстк съществува дете v на u в T , такова че в G няма нито едно ребро между връх на $T[v]$ и връх-предшественик на u в дървото.

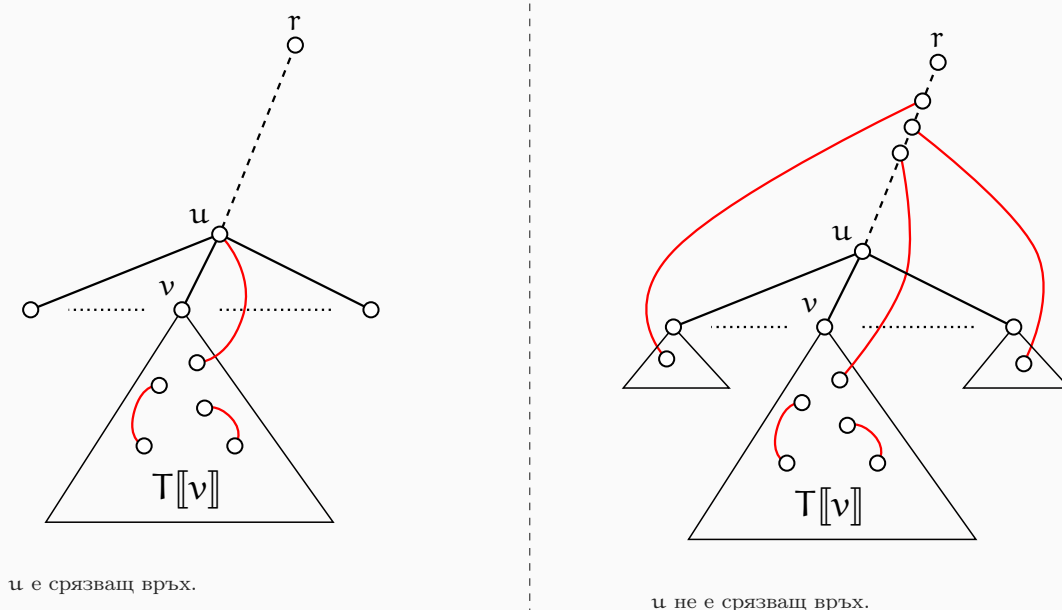
Забележете, че ако има ребро e между връх на $T[v]$ и връх-предшественик на u в дървото, то e би било ребро назад съгласно категоризацията на ребрата от DFS (Подподсекция 8.5.3.2). С други думи, e не може да е ребро от T .

Доказателство: Нека u е връх, който не е нито листо, нито коренът. В едната посока, да допуснем, че съществува дете v на u в T , такова че няма нито едно ребро между връх на $T[v]$ и връх-предшественик на u в дървото. Това влече, че за всеки $x \in V(T[v])$ и всеки $y \in V(G) \setminus (V(T[v]) \cup \{u\})$, за всеки път p в G между x и y е вярно, че u е вътрешен връх в p . Тогава u е срязващ връх в G .

В обратната посока, нека за всяко дете v на u в T има ребро назад между връх $x_v \in V(T[v])$ и предшественик y_v на u в дървото. Тогава за всеки два върха a и b в G е вярно, че в G има път $p_{a,b}$, който не съдържа u като вътрешен връх. Тогава u не е срязващ връх. $\square$

Фигура 9.10 илюстрира Теорема 91.

Фигура 9.10 : Кога не-листо, не-корен на дървото на DFS е срязващ връх.



Алгоритъм 25 имплементира тези идеи в модификация на DFS. В него няма масиви $d[1..n]$, $f[1..n]$ и $\pi[1..n]$, а също така няма и променлива $time$. DFS-ът стартира от произволен връх u , който е и коренът на дървото на обхождането T .

Да си припомним един факт за разстоянията. Разстоянията в графа и в дървото не са непременно същите. Нещо повече, за всеки $v \in V$ е вярно, че $\text{dist}_G(u, v) \leq \text{dist}_T(u, v)$. Това е така, защото T е подграф на G .

Алгоритъмът поддържа масив от нивата $\text{level}[1..n]$, като в края на работата, за всеки връх v , $\text{level}[v] = \text{dist}_T(u, v)$. С други думи, нивото на v е разстоянието между u и v в дървото на обхождането (вижте Определение 51). Пак подчертаваме, че **не става дума**

за разстояния в G ; знаем, че DFS в общия случай не изчислява разстояния в графа. Очевидно, $\text{level}[u] = 0$.

Алгоритъм 25: Find Cut Vertices

FIND CUT VERTICES($G = (V, E)$): неориентиран свързан граф)

```

1  level[1 .. n] е глобален
2   $S \leftarrow \emptyset$ 
3  foreach  $a \in V$ 
4      color[a]  $\leftarrow$  white
5  нека  $a$  е произволен връх от  $V$ 
6  FCV-REC( $G, a, 0$ )
7  return  $S$ 
```

Алгоритъм 26: FCV-rec

FCV-REC($G = (V, E), x \in V, \ell \in \mathbb{N}$)

```

1  color[x]  $\leftarrow$  gray
2  level[x]  $\leftarrow$   $\ell$ 
3  minback  $\leftarrow$  level[x]
4  if level[x] = 0
5      isroot  $\leftarrow$  TRUE
6  else
7      isroot  $\leftarrow$  FALSE
8  count  $\leftarrow$  0
9  iscutvertex  $\leftarrow$  FALSE
10 foreach  $y \in \text{adj}[x]$ 
11     if color[y] = white {
12         count++
13          $b \leftarrow$  FCV-REC( $G, y, \ell + 1$ )
14         if  $b \geq \text{level}[x]$  and ( not isroot )
15             iscutvertex  $\leftarrow$  TRUE
16         minback  $\leftarrow$  min {minback, b} }
17     if color[y] = gray {
18         if level[y] < minback and level[y]  $\neq$  level[x] - 1
19             minback  $\leftarrow$  level[y] }
20 if iscutvertex or ( isroot and count  $\geq$  2 )
21      $S \leftarrow S \cup \{x\}$ 
22     color[x]  $\leftarrow$  black
23 return minback
```

Коректност. Променливата minback има смисъл на минимално ниво (най-близо до корена), в което има връх, инцидентен с ребро, другият край на което е от $V(T[x])$.

Теорема 89, Теорема 90 и Теорема 91 са достатъчна обосновка за коректността, тъй като рекурсивната функция FCV-REC имплементира техните необходими и достатъчно условия. Функцията FCV-REC прави следното.

- Очевидно не категоризира никое листо на T като срязващ връх.

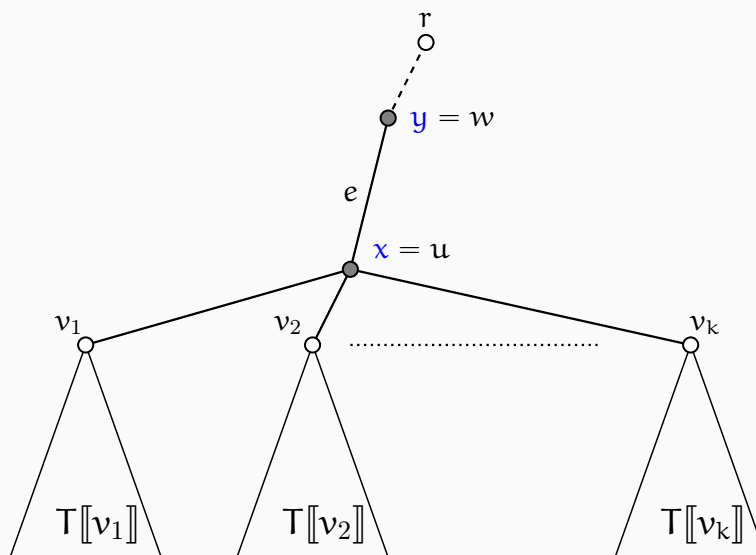
- Категоризира върхът-корен на T като срязващ връх тстк той има поне две деца в T . Това също е очевидно от редове 4, 5, 8 и 12 и второто условие в дизюнкцията на ред 20.
- За всеки друг вид връх x , FCV-REC сравнява на ред 14 следните числа
 - ♦ $\text{level}[x]$, което е разстоянието в дървото между x и корена, и
 - ♦ b , което е минималното разстояние в дървото между корена и най-близък до корена връх, такъв че между този връх и връх от $T[y]$ има ребро назад,

по всички y , които са деца на x в дървото. Ако се окаже, че за поне едно дете y е вярно, че съответното b е по-голямо или равно на $\text{level}[x]$, то връх x е срязващ съгласно Теорема 91.

При допускането, че рекурсивното викане на ред 13 връща стойност b , равна на минималното разстояние в дървото между корена и най-близък до корена връх, между който и връх от $T[y]$ има ребро назад, Теорема 91 ни дава желаната аргументация.

Има една особеност при изчисляването на minback на редове 17–19, които третират възможността y да е сив връх. Връх y е сив тстк реброто (x, y) е ребро назад, но с едно важно изключение, което сега ще разгледаме. Това изключение е y да е родителят на x в дървото. Тъй като и x , и y са променливи, които в даден момент от изпълнението на рекурсивния алгоритъм съществуват едновременно в няколко инкарнации в стека, да разгледаме момента t , в който, в последното извикване (върхът на стека) x съдържа някакъв връх u , който не е нито корена на T , нито е листо в T . Щом не е листо, връх u има деца $v_1, \dots, v_k$ в T , за някое $k \geq 1$. Щом не е коренът, връх u има точно един родител в T . Нека този родител е w . Ситуацията в момента t е илюстрирана на Фигура 9.11.

Фигура 9.11 : В алгоритъма FCV-REC: когато x съдържа не-листо, не-корен.



Променливите x и y в момента t .

И $u = x$, и $w = y$ са нарисувани като сиви върхове, а $v_1, \dots, v_k$ са нарисувани като бели върхове, защото допускаме, че моментът t е тогава, когато $u = x$ вече е станал сив на ред 1, y е взел стойност w на ред 10, но $v_1, \dots, v_k$ са все още неоткрити и поради това, бели.

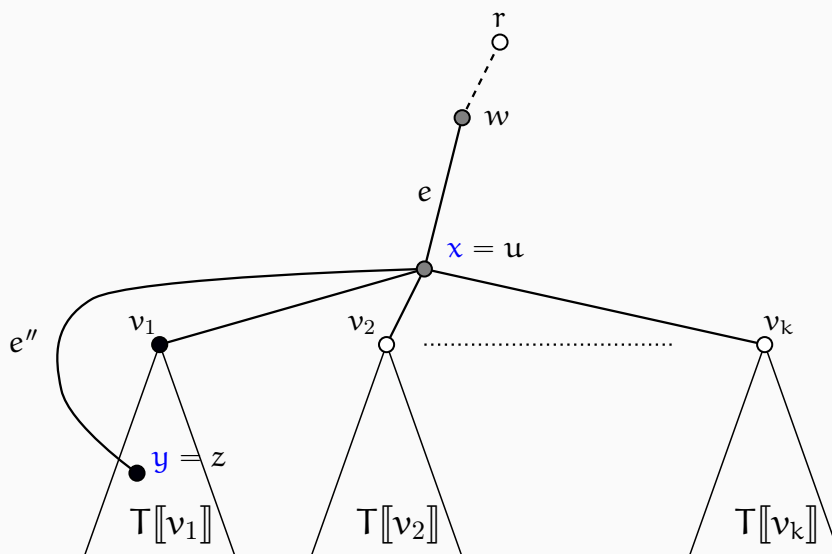
Ключовото наблюдение е, че реброто (x, y) , тоест (u, w) , **не е ребро назад**. Да наречем това ребро e . То е дървесно ребро по отношение на работата на DFS, която разглеждаме. Да си припомним, че в неориентираните графи DFS (както и BFS) “прекосява” всяко ребро два пъти, понеже се използва маркиране на върховете, а не маркиране на ребрата. Моментът t е **второто** прекосяване на това ребро. Първото му прекосяване е било в някакъв предишен момент t' , когато $x = w$ и $y = u$; в този предишен момент w е бил сив, а u е бил бял, така че за DFS това ребро е дървесно. Второто прекосяване не може да характеризира реброто по друг начин.

Това си заслужава да се повтори: реброто $e = (u, w)$ е неориентирано и бива представено в списъците на съседство както чрез появата на u в списъка на w , така и чрез появата на w в списъка на u . Но e е едно единствено ребро, понеже графът е неориентиран. Това, че в списъците му съответстват два елемента, е особеност на списъчното представяне на неориентираните графи. При работата на DFS, която разглеждаме, e бива прекосено първоначално от w (родителят на u в дървото) към u . Ерго, от гледна точка на работата на DFS, e е дървесно.

Щом $e = (u, w)$ не е ребро назад, то не трябва да се използва за изчисляване на `minback`. Алгоритъмът обаче “разглежда” всички ребра, инцидентни с връх $x = u$, което включва и e . Тъй като e не е ребро назад, а е дървесно ребро, то трябва да бъде прескочено в това разглеждане. Смисълът на `level[y] ≠ level[x] - 1` на ред 18 е точно този: да прескочи e , и само него, измежду ребрата, инцидентни с x , другият край на които е сив връх (което означава предшественик на x в дървото).

И още нещо. Алгоритъмът разглежда само белите съседи на x (условието на ред 11 е истина) и сивите съседи на x (условието на ред 17 е истина), но не и черните съседи на x . Забележете, че DFS върху неориентирани граfi може да “вижда” и черни върхове в смисъл, че променливата y , която получава за стойности съседите на x на ред 10 в псевдокода на FCV-REC, може да е връх с черен цвят. Това се вижда ясно на Фигура 9.12, която показва работата на алгоритъма върху същия граф, но в по-късен момент t'' , когато алгоритъмът е приключил с $T[v_1]$, но още не е открил $v_2, \dots, v_k$.

Фигура 9.12 : В алгоритъма FCV-REC: черните върхове се прескачат.



В момента t'' , y съдържа черен връх.

В момента t'' , връх v_1 , както и всеки друг връх от $T[v_1]$, е черен. Ако си представим ребро e'' между u и кой да връх z от $T[v_1]$, такъв че $z \neq v_1$, при второто прекосяване на e'' , връх $y = z$ е черен[†]. Алгоритъмът прескача $y = z$ в смисъл, че условията на ред 11 и ред 17 са лъжа. Извод: не може да заменим проверката за сив връх на ред 17 с **else**.

Сложност. Очевидно сложността е $\Theta(m+n)$, което следва веднага от това, че алгоритъмът обхожда еднократно списъците на съседства и върши само $\Theta(1)$ работа за всеки елемент от тях.

9.5.3 Алгоритъм за ефикасно намиране на мостовете

Подходът с брутална сила за намиране на мостовете би работил, в най-бруталния си вариант, в $\Theta(m(n+m))$: за всяко ребро e , изтриваме e от G и с BFS или DFS проверяваме дали G остава свързан; ако остава, то e не е мост, ако ли не, то e е мост и го добавяме към множеството от мостовете, после връщаме e и продължаваме по същия начин.

Задачата за намиране на мостовете е много подобна на задачата за намиране на срязващите върхове, откъдето е и приликата между алгоритмите FIND CUT VERTICES и FIND BRIDGES.

Теорема 92: Намиране на мостовете

В текущия контекст, за всяко ребро (u, v) е вярно, че то е мост тстк $DFS(G)$ го категоризира като дървесно и, ако u е бащата на v в дървото на обхождането, то няма ребро назад от никой връх на $T[v]$ към u или предшественик на u .

Доказателство: В едната посока, да допуснем, че (u, v) е дървесно ребро, като u е бащата на v , и няма ребро назад от никой връх на $T[v]$ към u или предшественик на u . Тогава за всеки $x \in V(T[v])$ и всеки $y \in V(G) \setminus (V(T[v]) \cup \{u\})$, за всеки път p в G между x и y е вярно, че (u, v) е ребро в p , което означава, че (u, v) е мост.

В обратната посока, нека има ребро назад от връх $x \in V(T[v])$ към u или предшественик на u . Тогава за всеки два върха a и b в G е вярно, че в G има път $p_{a,b}$, който не съдържа реброто (u, v) . Тогава (u, v) не е мост. $\square$

За разлика от задачата за намиране на срязващите върхове, сега не е необходимо да третираме корена, листата и останалите върхове по различни начини.

FIND BRIDGES($G = (V, E)$: неориентиран свързан граф)

- 1 level[1 .. n] е глобален
- 2 **foreach** $u \in V$
- 3 color[u] $\leftarrow$ white
- 4 u е произволен връх от V
- 5 FBR-REC($G, u, 0$)

FBR-REC($G = (V, E), x \in V, \ell \in \mathbb{N}$)

- 1 color[x] $\leftarrow$ gray
- 2 level[x] $\leftarrow$ ℓ

[†]Забележете, че реброто e'' е ребро назад, защото е било прекосено за първи път откъм z , в смисъл, че тогава x е бил z , а y е бил u ; това е било в някакъв момент между t и t'' . В t'' се случва второто прекосяване на e'' , но второто прекосяване не може да категоризира e'' по друг начин.

```

3  minback  $\leftarrow$  level[x]
4  foreach  $y \in \text{adj}[x]$ 
5      if color[y] = white {
6           $b \leftarrow \text{FBR-REC}(G, y, \ell + 1)$ 
7          if  $b > \text{level}[x]$ 
8              print (x, y)
9          else
10             minback  $\leftarrow \min \{ \text{minback}, b \}$ 
11      if color[y] = gray {
12          if  $\text{level}[y] < \text{minback}$  and  $\text{level}[y] \neq \text{level}[x] - 1$ 
13             minback  $\leftarrow \text{level}[y]$  }
14  color[x]  $\leftarrow$  BLACK
15  return minback

```

Коректност и сложност. Доказателството за коректност почива на Теорема 92. Ако рекурсивното викане на ред 6 връща коректно минималния номер на ниво, към което има ребро назад от връх от $T[y]$, съхранявайки го в b , то, съгласно Теорема 92:

- ако $b > \text{level}[x]$, тоест, ако това ниво е “под x ”, то (x, y) е мост;
- в противен случай, (x, y) не е мост. Нещо повече, в този случай има смисъл да се опитаме да подобрим (тоест, намалим) minback на ред 10. В предишния случай няма смисъл от такъв опит.

Действията на алгоритъма върху сивите съседни на x (редове 11–13) и тяхната обосновка са точно както във FIND CUT VERTICES.

Сложността по време е очевидно $\Theta(m + n)$.

Лекция 10

Намиране на минимални покриващи дървета. Алгоритми на Prim и Kruskal.

Резюме: Въвеждаме тегловни графи и задачата за намиране на минимално покриващо дърво на тегловен граф. Разглеждаме два ефикасни алгоритъма за намиране на минимални покриващи дървета: алгоритмите на Prim и Kruskal. Заради ефикасността на алгоритъма на Kruskal въвеждаме и изследваме Union-Find структури данни.

10.1 Фундамент

Определение 86: Тегловен граф

Тегловен граф е наредена двойка (G, w) от граф $G = (V, E)$ и *тегловна функция* $w : E \rightarrow \mathbb{R}$.

Говорим за *тегла на ребрата*[†]. На английски терминът за “тегловен граф” е *weighted graph*, откъдето идва и *edge weights*. Освен “тегловна”, тази функция понякога бива наричана *ценова функция*, откъдето може да говорим и за *цени на ребрата*, имайки предвид същото нещо като “тегла на ребрата”. Използваме термините *леките/евтините ребра* и *тежките/скъпите ребра* с очевидния смисъл.

Тегловните графи имат много по-голяма моделираща мощ от обикновените, не-тегловни графи. Очевидно е, че не-тегловните графи са частен случай на тегловните: просто си представяме, че всяко ребро има тегло единица.

Има ограничени случаи, в които задача върху тегловен граф може да бъде сведена до задача върху не-тегловен граф и да използваме алгоритъм за не-тегловния граф, за да я решим ефикасно върху тегловния. Пример за това е задачата за най-къси пътища в тегловен граф, където тегловната функция е доста ограничена, да кажем $w : E \rightarrow \{1, 2\}$. При това ограничение може да решим задачата не с алгоритъма на Dijkstra за най-къси пътища в тегловни графи, а с прост BFS, ако първо заменим всяко ребро с тегло две: $\bigcirc \text{---}^2 \text{---} \bigcirc$ с последователност от две ребра с тегла единици: $\bigcirc \text{---}^1 \text{---} \bigcirc \text{---}^1 \text{---} \bigcirc$. Но този подход за свеждане на задачи върху тегловни графи към задачи върху не-тегловни графи не се мащабира. Ако ограничението за теглата е $w : E \rightarrow \{1, \dots, 1\,000\,000\}$, този подход би налагал да заменим ребро с тегло един милион с път от един милион ребра, всяко с тегло единица. Това е все едно да записваме числата в унарна бройна система; големината на теглото става брой ребра. В

[†]Възможно е да се въведат тегла и на върховете, но ние ще разглеждаме тегла само на ребрата.

общия случай, този подход би довел до експоненциално нарастване на големината на примера, ако отчитаме размера на представянето на числата, или до неограничено нарастване на примера в големината на входа, ако игнорираме размера на представянето на числата. Да не говорим за проблемите на този подход при тегла, които не са цели числа или са отрицателни числа. И така, тегловните графи са строго по-мощно средство за моделиране на житейски задачи от не-тегловните графи.

Тегловни могат да бъдат както неориентирани, така и ориентирани графи. В тази лекция разглеждаме само неориентирани графи, тегловни или не.

Определение 87: Покриващ подграф

Нека $G = (V, E)$ е граф. *Покриващ подграф* на G е всеки подграф $G' = (V, E')$ на G . *Покриващо дърво* на G , съкратено ПД, е покриващ подграф на G , който е дърво.

На английски термините са съответно *spanning subgraph* и *spanning tree*. Ние няма да разглеждаме други видове покриващи подграфи освен покриващи дървета, но принципно има смисъл да се обобщи “покриващо дърво” до “покриващ подграф”. На читателя остава да съобрази, че покриващ цикъл е същото като Хамилтонов цикъл.

Наблюдение 66

За всеки граф G , G има поне едно покриващо дърво тстк G е свързан.

Съгласно формулата на Cayley, която знаем от предмета Дискретни Структури, броят на покриващите именуван дървета е най-много n^{n-2} и тази граница е точна.

Определение 88: Тегло на ПД. Минимално ПД.

Нека G е свързан тегловен граф с тегловна функция w . Нека T е ПД на G . *Теглото* на T е $w(T) = \sum_{e \in E(T)} w(e)$. Минимално ПД, съкратено МПД, е такова ПД T' , че за всяко ПД T'' е вярно, че $w(T') \leq w(T'')$.

Формално, задачата МПД, на английски MST, се дефинира така.

Изч. Задача 26: Минимално покриващо дърво (MINIMUM SPANNING TREE)

екземпляр: Наредена двойка $\langle G, w \rangle$ от неориентиран свързан граф G и тегловна функция върху ребрата му w .

решение: Минимално покриващо дърво T на G .

Това е първата оптимизационна графова задача, която разглеждаме в [Част III](#). Разглеждаме я само в тежкия вариант – на автора не е известно да се преподават алгоритми, които намират теглото на МПД, но не и самото МПД.

Следното определение е буквален превод на “safe edge” от [34, стр. 626].

Определение 89: Сигурно ребро.

Нека G е тегловен граф и $A \subseteq E$ е такова, че съществува МПД T , такова че $A \subset E(T)$. Нека e е ребро на G , което не е от A . Казваме, че e е *сигурно* за A , ако съществува МПД T' , такова че $A \cup \{e\} \subseteq E(T')$.

Общата схема на алгоритмите за конструиране на МПД се съдържа в следния псевдокод ([34, стр. 626]). МПД се определя напълно от своите ребра, така че да конструираме МПД е същото като да конструираме множеството от ребрата му.

ОБЩА СХЕМА ЗА АЛГОРИТЪМ НА МПД(G : граф, w : тегловна функция)

```

1   $A \leftarrow \emptyset$ 
2  while  $A$  не образува ПД do
3      намери ребро  $e$ , което е сигурно за  $A$ 
4       $A \leftarrow A \cup \{e\}$ 
5  return  $A$ 

```

Ясно е, че във всеки момент от работата на алгоритъм, изграден върху тази схема, ребрата от A дават ацикличен подграф на G ; в противен случай, e не би било сигурно. Алгоритъмът спира, когато множеството от върховете-краища на ребрата от A стане V . Веднага следва, че **while**-цикълът се изпълнява точно $n - 1$ пъти.

Същността на решението е в намирането на сигурно ребро по отношение на множество от ребра. Теорема 93 дава достатъчна обосновка за намирането на сигурни ребра от алгоритмите, които ще разгледаме. Първо ни трябва няколко определения.

Определение 90: Срез в граф.

Срез в граф G е всяко разбиване на $V(G)$ на два дяла. Нека $S = \{V_1, V_2\}$ е срез в G . Ребро $e = (u, v)$ *прекосява* S , ако $u \in V_1$ и $v \in V_2$. Нека $E' \subseteq E(G)$. S е *съобразен* с E' , ако нито едно ребро от E' не го прекосява. Ако G е тегловен и реброто e прекосява S , казваме, че реброто e е *леко*, ако e има минимално тегло измежду всички ребра, които прекосяват S .

В учебника на Cormen и др., термините “срез”, “реброто e прекосява среза S ”, “срезът S е съобразен с множество от ребра E ” и “леко ребро” са съответно “*cut*”, “*e crosses S*”, “*S respects E*” и “*light edge*” [34, стр. 626].

Теорема 93: Theorem 23.1 от [34, стр. 627] (МПД теоремата)

Нека $G = (V, E)$ е свързан тегловен граф с тегловна функция $w : E \rightarrow \mathbb{R}$. Нека $A \subseteq E$ е такава, че съществува МПД T , че $A \subseteq E(T)$. Нека $S = \{V_1, V_2\}$ е срез в G , който е съобразен с A . Нека $e = (u, v)$ е произволно леко ребро, прекосяващо S . Тогава e е сигурно за A .

Доказателство: Нека T е МПД, такава че $A \subseteq E(T)$. Ако $e \in E(T)$, то ние сме готови с доказателството. Да допуснем, че $e \notin E(T)$. Ще покажем, че съществува МПД T' , такава че $A \cup \{e\} \subseteq E(T')$. Добавяме e към T и получаваме уницикличен покриващ граф U . Нека C е цикълът в U . Очевидно $e \in E(C)$ и e прекосява среза S . Но съществува ребро $e' \in E(C)$, такава че $e' \neq e$, което също прекосява среза S . Изтриваме e' от U и получаваме ПД T' , различно от T . Да сравним $w(T)$ с $w(T')$. T' се получи от T с добавяне на e и изтриване на e' . Тогава

$$w(T') = w(T) + w(e) - w(e')$$

Но e е леко по конструкция, така че $w(e) \leq w(e')$, следователно $w(e) - w(e') \leq 0$. Тогава $w(T') \leq w(T)$. Но T е МПД, така че $w(T) \leq w(T')$ по дефиниция. Тогава $w(T) = w(T')$. Тогава T' също е МПД, което съдържа както ребрата от A , така и e . $\square$

10.2 Алчни алгоритми

Предложената обща идея за намиране на МПД на предната страница заедно с Теорема 93 се конкретизира до два алгоритъма, които ще разгледаме в следващите две секции. И двата алгоритъма са изградени по *алчна схема* (на английски, *greedy strategy*). Алгоритмите, изградени по тази схема, се наричат *алчни алгоритми* (*greedy algorithms*).

По правило, изчислителните задачи, които алчните алгоритми решават, са оптимизационни задачи (Определение 4). Задачата МИНИМАЛНО ПОКРИВАЩО ДЪРВО е именно оптимизационна. За нея има ефикасни алгоритмични решения, които работят итеративно, на всяка итерация добавяйки точно едно ребро към множеството от вече добавените ребра. Критерият, по който се избира ребро за добавяне, е много прост и ясен: добавя се леко ребро, прекосяващо някакъв срез (в алгоритъма на Kruskal: стига то да не образува цикъл с досега сложените ребра). Всеки такъв избор на леко ребро е *локален*, защото не се прави с оглед на “голямата картина”. Теорема 93 гарантира, че **редицата от локално оптимални избори** (на леки ребра) води до **глобално оптимално решение** (МПД).

Най-общо казано, всеки алчен алгоритъм работи така:

- започва с някакво елементарно частично решение, примерно празното множество и
- докато не е конструирано цялостно решение, добавя някакъв елемент към текущото частично решение; кой елемент да бъде добавен се взема въз основа на прост, ефикасен критерий.

Би трябвало да е ясно защо такива алгоритми се наричат алчни. Терминът “алчни” не е съвсем прецизен, понеже всеки алгоритъм, който решава оптимизационна задача, е в някакъв смисъл алчен. При алчните алгоритми става дума за някаква **късогледа алчност**, или **локална алчност**. Когато решаваме задачата за намиране на МПД, локалната алчност е печеливша стратегия. Има обаче задачи, в чиито решения проявата на локална алчност води до решение, което не е оптимално. Всички **NP**-трудни задачи са такива.

Примерно, **NP**-трудната задача KNAPSACK (това е задачата за раницата от Подсекция 12.5.3) не изглежда да е податлива на ефикасно решаване чрез алчен алгоритъм. Неоптималността на очевидния алчен алгоритъм за нея е показана на стр. 631. От друга страна, за нея има тривиален алчен алгоритъм, даващ оптимално решение: алгоритъмът, който разглежда всички подмножества на даденото множество от предмети. Той не е ефикасен, но в някакъв смисъл алчен.

Подробно теоретично изложение на фундамента на алчните алгоритми има в Секция 16.2 на [34, стр. 423] и Секция 16.4 на [34, стр. 437], както и [126, стр. 280]. Това е материал, който е далече отвъд обхвата на тези лекции.

Говорейки полуформално за алчни алгоритми и алчни схеми, авторът на тези записки е на мнение, че има неизбежен субективизъм/условност в това, което наричаме “локално оптимален избор” (което дефинира алчните схеми). От някаква гледна точка, всеки избор (на нов елемент, който да добавим в множеството) е локален. Ако искаме да прецизираме “локално оптимален избор”, трябва да говорим за максималните ресурси (време/памет), които сме склонни да отделим за извършване на този избор.

Доказателството за коректност на алчен алгоритъм може да е доста триково. Ако се прави педантично, такова доказателство трябва да покаже две неща: първо, че полученото решение е допустимо (*feasible*) и, второ, че полученото решение е оптимално. В контекста на задачата за намиране на минимално покриващо дърво, решението да е допустимо е полученият

подграф да е ПД, а решението да е оптимално е полученият подграф да е МПД. Понякога коректността на алчен алгоритъм се доказва с инвариант на цикъла, понякога се доказва с допускане на противното “Да допуснем, че еди-кой си алгоритъм не е коректен, и да разгледаме елемент x , чието добавяне към частичното решение е грешка, като x е минимален в някакъв смисъл”.

10.3 Алгоритъм на Prim

Алгоритъмът на Prim за намиране на МПД далечно прилича на обхождане в ширина. Той започва от един стартов връх s , намира най-леко ребро, инцидентно с s , да го наречем (s, u) , слага това ребро в дървото, после намира най-леко ребро, единият край на който е s или u , а другият край е връх, различен от s и u , слага го в дървото, и така нататък. В терминологията на ОБЩ ВИД НА МПД на стр. 494, за алгоритъма на Prim е вярно, че ребрата от A във всеки момент образуват едно единствено дърво. Естествено, то става ПД чак накрая, когато включи всички върхове, но във всеки момент от изпълнението то е едно дърво. Ето много общ псевдокод на алгоритъма на Prim.

PRIM GENERIC($G = (V, E)$): неор. свързан граф, w : тегл. ф-я върху G , s : стартов връх)

```

1   $V(T) \leftarrow \{s\}$ 
2   $A \leftarrow \emptyset$ 
3  while  $V \setminus V(T) \neq \emptyset$  do
4      намери леко ребро  $e = (x, y)$ , прекосяващо среза  $\{V(T), V \setminus V(T)\}$ 
5      нека  $x \in V(T)$ 
6       $V(T) \leftarrow V(T) \cup \{y\}$ 
7       $A \leftarrow A \cup \{e\}$ 
8  return  $A$ 
```

Коректността е очевидна и следва веднага от Теорема 93. От общи съображения обаче сложността не е добра. Очевидно **while**-цикълът се изпълнява $n - 1$ пъти. Потенциално проблематично е намирането на леко ребро (ред 4) на всяка итерация. Ако го правим с пълно изчерпване, тоест, минаваме през всички ребра на G и търсим ребро, прекосяващо среза с минимално тегло, само ред 4 би отнемал $\Theta(m)$ време на всяка итерация, поради което сложността на алгоритъма би била $\Theta(nm)$. Това е неприемливо бавно.

Известно подобрение би било на всяка итерация да не преглеждаме всички ребра на графа, а само тези, които прекосяват среза $\{V(T), V \setminus V(T)\}$. В асимптотичния смисъл обаче това не е никакво подобрение. В най-лошия случай, общият брой на прегледани ребра (за цялото изпълнение на алгоритъма) се оценява със сумата

$$\sum_{k=1}^{n-1} k(n-k) = \Theta(n^3)$$

тъй като има ребро между всеки връх от единия дял и всеки връх от другия дял, на всяка итерация (какъв граф трябва да е G , за да изпълнено това?). И така, отново имаме, в най-лошия случай, кубичен алгоритъм. Трябва да направим нещо по-умно, за да избираме ефикасно леко прекосяващо ребро.

В следващите две подсекции ще разгледаме два варианта на алгоритъма на Prim. Единият не използва изтънчени структури данни и ще го наречем “базов”, а другият ползва АТД

приоритетна опашка и ще го наречем “изтънчен”. Върху разредени графи изтънченият вариант е по-добър, но върху графи с брой ребра, близък до максималния, базовият алгоритъм, колкото и да е странно, е по-бърз.

10.3.1 Базов вариант на алгоритъма на Prim

Този алгоритъм следва плътно PRIM GENERIC, като намирането на леко ребро на ред 4 става във време $O(n)$, като в най-лошия случай това е $\Theta(n)$. Идеята е елементарна. За да я изложим, ще въведем класификация на върховете. Нека във всеки момент от работата на алгоритъма,

- върховете от $V(T)$ са *дървесните върхове* (на английски, *tree vertices*),
- върховете от $N(V(T))$ са *границните върхове* (на английски, *fringe vertices*),
- върховете от $V \setminus N[V(T)]$ са *неизвестните върхове* (на английски, *unseen vertices*).

Тази класификация на практика дублира класификацията от обхождането на графи на върховете на бели, сиви и черни на стр. 433, като черните върхове сега наричаме дървесни, сивите наричаме гранични, а белите са неизвестните. Причината да въвеждаме с други имена практически същата класификация е широко приетата терминология: класификацията на бели, сиви и черни е от [34], а на *tree*, *fringe* и *unseen* е например от [144, стр. 485].

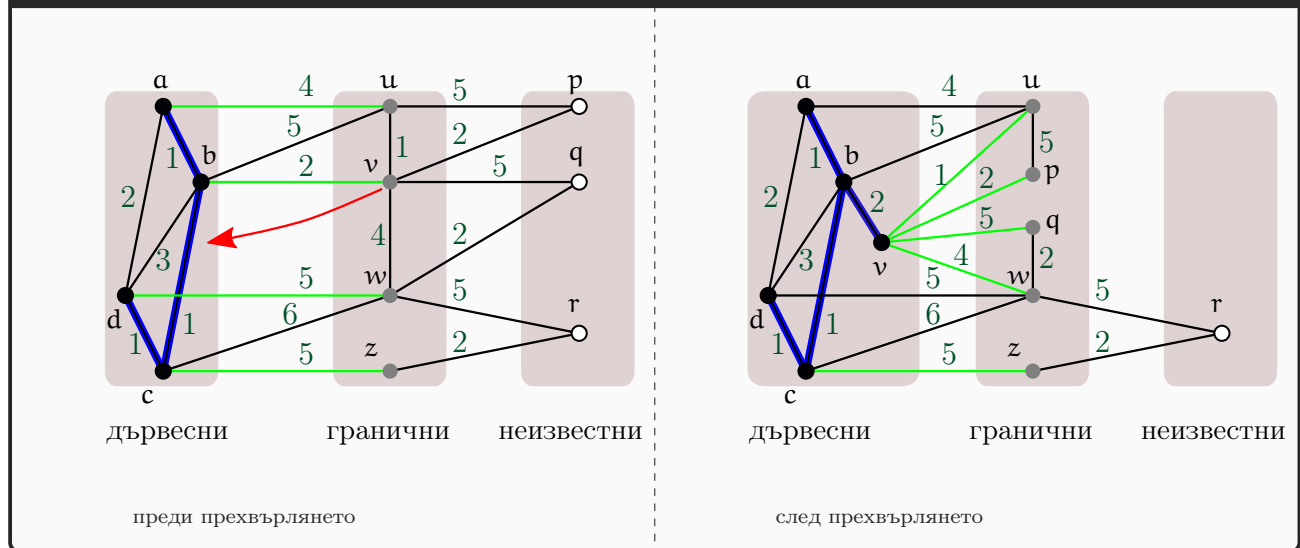
И така, сега класифицираме върховете на дървесни, гранични и неизвестни. Множеството от граничните върхове ще наричаме *границата*. На всяка итерация точно един граничен връх става дървесен. Нещо повече. Това е такъв граничен връх, който е инцидентен с леко прекосяващо ребро. В термините на PRIM GENERIC, това е връх u . За да направим избора на ред 4, достатъчно е да прегледаме всички гранични върхове; ако за всеки от тях предварително е известно теглото на най-леко инцидентно прекосяващо ребро и самото ребро, може да направим избора във време, пропорционално на размера на границата, което, в най-лошия случай е $\Theta(n)$. Иначе казано, наместо да търсим директно леко ребро, както пише на ред 4, ще търсим граничен връх, инцидентен с такова, а, когато намерим върха, в $\Theta(1)$ ще намерим и реброто, ако предварително всеки граничен връх е бил асоцииран с най-леко инцидентно прекосяващо ребро. Ако може да работим във време $\Theta(1)$ за всеки граничен връх, ще може да сведем сложността на алгоритъма до $\Theta(n^2)$.

Как да работим във време $\Theta(1)$ за произволен граничен връх? Въвеждаме понятието *кандидат-ребро*, или просто *кандидат*. По време на работата на алгоритъма, всеки граничен връх е асоцииран с точно едно кандидат-ребро и с една стойност, наречена *границно тегло*, която е теглото на кандидат-реброто. За дървесните и неизвестните върхове кандидат-ребра и гранични тегла не се дефинират.

Всеки граничен връх може да е съседен на дървесни, гранични и неизвестни върхове. След избора на ред 4 прехвърляме връх от границата в дървото. Това обаче може да наложи промени в някои кандидат-ребра и гранични тегла, за да е сигурно, че на следващата итерация, стойностите на всички кандидат-ребра и гранични тегла да са коректни. Това е показано на Фигура 10.1. Дървесните върхове са вляво, граничните са по средата, а неизвестните са вдясно. Ребрата в дървото са в синьо. Кандидат-ребрата са в ярко зелено. В лявата подфигура е показано състоянието на нещата в началото на някоя итерация. Граничните върхове са u , v , w и z със съответни гранични тегла 4, 2, 5 и 5. Върхът с минимално гранично тегло е v и той бива прехвърлен към дървесните върхове, както показва червената стрелка. След прехвърлянето (дясната подфигура), реброто (b, v) става част от дървото. Върховете, които са неизвестни съседи на v преди прехвърлянето (p и q), сега вече са гранични, като

кандидат-ребрата им са съответно (p, v) и (q, v) . Преди прехвърлянето, v е съсед на граничните върхове u и w . След прехвърлянето на v , те си остават гранични, но кандидат-ребрата им се променят на съответно (u, v) и (w, v) , понеже и двете имат по-малко тегло от съответните им кандидат-ребра (a, u) и (d, w) преди прехвърлянето. И така, дясната подфигура показва състоянието на нещата в началото на следващата итерация.

Фигура 10.1 : Прехвърляне на връх от границата в дървото при Prim.



Ето псевдокод на базовия вариант на алгоритъма на Prim. Ползва се масив $\text{status}[1..n]$, аналог на масива color от обхожданията, като $\text{status}[i] \in \{\text{дървесен}, \text{граничен}, \text{неизвестен}\}$.

МПД $\text{PRIM1}(G = (V, E)$: неор. свързан граф, w : тегл. ф-я върху G , s : стартов връх)

```

1   $V(T) \leftarrow \{s\}$ 
2   $A \leftarrow \emptyset$ 
3  foreach  $u \in V$ 
4       $\text{status}[u] \leftarrow \text{неизвестен}$ 
5   $\text{status}[s] \leftarrow \text{дървесен}$ 
6  foreach  $y \in \text{adj}[s]$ 
7       $\text{status}[y] \leftarrow \text{граничен}$ 
8   $x \leftarrow s$ 
9  while  $V \setminus V(T) \neq \emptyset$  do
10     (*  $x$  е последният сложен в дървото връх *)
11     foreach  $y \in \text{adj}[x]$ 
12         if  $\text{status}[y] = \text{граничен}$ 
13             ако  $w((x, y))$  е по-малко от граничното тегло на  $y$ ,
14                 то смени кандидат-реброто на  $y$  с  $(x, y)$ 
15         if  $\text{status}[y] = \text{неизвестен}$ 
16              $\text{status}[y] \leftarrow \text{граничен}$ 
17             кандидат-реброто на  $y$  става  $(x, y)$ 
18     намери кандидат-ребро  $e$  с минимално тегло
19      $x \leftarrow$  граничният връх на  $e$ 
20      $\text{status}[x] \leftarrow \text{дървесен}$ 
21      $V(T) \leftarrow V(T) \cup \{x\}$ 
22      $A \leftarrow A \cup \{e\}$ 
23 return  $A$ 

```

Коректност и сложност. Подробно доказателство за коректност няма да правим. Инвариант е този: при всяко достигане на на ред 9, ребрата от A представляват МПД на подграфа на G , индуциран от $V(T)$. Използвайки Теорема 93, лесно показваме, че реброто e на ред 17 е сигурно за A .

Сложността по време в най-лошия случай е $\Theta(n^2)$. Ето защо.

- Първо, работата по всички намирания на кандидат-ребра, за цялото изпълнение на алгоритъма, се извършва във време $\Theta(n^2)$. Най-лошият случай е графът да е пълен. Тогава неизвестни върхове изобщо няма, понеже всеки връх е съседен на всеки друг, така че на всяка итерация на **while**-а (ред 9) върховете на графа се разбиват само на дървесни и гранични. Тогава на всяка итерация на **while**-а се разглеждат $|V \setminus V(T)|$ (гранични) върхове, за да се намери кандидат-ребро с минимално тегло на ред 17. Тогава следната сума дава асимптотиката на общата (за цялото изпълнение на алгоритъма) работа за намирането на кандидат-ребрата:

$$\Theta(n-1) + \Theta(n-2) + \dots + \Theta(2) + \Theta(1)$$

Тривиално се пресмята, че $\Theta(n-1) + \Theta(n-2) + \dots + \Theta(1) \asymp \Theta(n^2)$.

- Освен намирането на кандидат-ребрата, алгоритъмът обхожда списъците на съседство чрез **for**-цикъла на редове 11–16. Това става във време $\Theta(n+m)$, за цялото изпълнение на алгоритъма.

Тъй като $\Theta(m+n) = O(n^2)$, имаме обща асимптотична оценка за сложността $\Theta(n^2)$. Забележе, че по отношение на най-лошия случай, тази сложност по време е оптимална, защото в най-лошия случай списъците на съседство имат размер $\Theta(n^2)$.

10.3.2 Изтънчен вариант на алгоритъма на Prim

Този вариант на алгоритъма на Prim прави бърз избор на минимално ребро, което да бъде добавено към временното дърво. Използва се АТД приоритетна опашка Q от `min` тип от върховете. Ключът на всеки връх в Q е минималното тегло на реброто, свързващо този връх с връх от временното дърво. Ако такова ребро няма, ключът е ∞ . В Q поначало се слагат всички върхове на графа с ключове ∞ , като само s има ключ 0. След това на всяка итерация Q съдържа точно върховете, които **не са в дървото**; тоест, Q съдържа точно граничните и неизвестните върхове. Граничните върхове са тези с ключове-числа, а неизвестните върхове са тези с ключове ∞ . Ключовете пишем така: ако връхът е u , неговият ключ е $u.key$. Самото МПД се реализира с масив на предшествията. За разлика от [34], този масив ще записваме като $\pi[1..n]$.

Алгоритъм 27: МПД Prim2

МПД PRIM2($G = (V, E)$): неор. свързан граф, w : тегл. ф-я върху G , s : стартов връх)

```

1  foreach  $u \in V$ 
2       $u.key \leftarrow \infty$ 
3       $\pi[u] \leftarrow Nil$ 
4   $s.key \leftarrow 0$ 
5  създай празна приоритетна min опашка  $Q$ 
6   $Q \leftarrow V$ 
7  while not isempty( $Q$ ) do
8       $x \leftarrow EXTRACT-MIN(Q)$ 
9      foreach  $y \in adj[x]$ 
10         if  $y \in Q$  and  $w((x, y)) < y.key$ 
11              $y.key \leftarrow w((x, y))$ 
12              $\pi[y] \leftarrow x$ 
13  return  $\pi[1..n]$ 

```

Този псевдокод следва плътно псевдокода от [34, стр. 634]. Две забележки по имплементацията. Първо, проверката $y \in Q$ на ред 10 може да бъде имплементирана, като всеки връх има булев флаг за принадлежност към опашката; в началото всички флагове са TRUE, а при всяко вадене на връх от опашката на ред 8, правим флага му FALSE. Второ, смяната на ключа на y на ред 11 не е просто смяна на едно число с друго, както може човек да си помисли първо. Връх y е в опашката и смяната на ключа му, тоест, намаляването на ключа му, трябва да бъде съпроводено с изпълнение на DECREASE-KEY, която е огледален аналог на INCREASE-KEY от Подсекция 5.4.4. Това има значение за анализа на сложността.

Лема 38: while-цикълът на алгоритъма на Prim

При всяко достигане на ред 7 на Алгоритъм 27:

- ① $Q \neq \emptyset \rightarrow (\exists u \in Q : u.key < \infty)$.
- ② $\forall u \in Q \setminus \{s\}$: ако $u.key < \infty$, то:
 - ① $\pi[u] \neq Nil$
 - ② $\pi[u] \notin Q$
 - ③ $u.key$ е теглото на леко ребро, инцидентно с u и прекосяващо $\{V \setminus Q, Q\}$.
- ③ $\forall u \in Q$: ако $u.key = \infty$, то съществува $v \in Q$, такъв че $v.key < \infty$ и има път между u и v , чиито върхове са само от Q .

Доказателство: Лема 38 е инвариант на цикъл. Доказателството няма фаза **Терминация**, понеже лемата казва нещо за всяко достигане на управлението на цикъла, а не за резултата от работата на алгоритъма.

База. Разглеждаме първото достигане на ред 7. Тогава $Q = V$ (ред 6), така че Q не е празна и антецедентът на ① е истина. Но консеквентът също е истина, понеже има връх във V (и оттам, в Q) със стойност key , по-малка от ∞ : това е връх s (ред 4). Тогава ① е в сила.

Да разгледаме ②. Тъй като единственият връх в Q със стойност key , която не е ∞ , е s , антецедентът на импликацията е лъжа за всяко $u \in Q \setminus \{s\}$ и оттам, импликацията е истина за всяко $u \in Q \setminus \{s\}$.

Да разгледаме ③. Истинността следва от това, че $V = Q$, $s.key < \infty$ и G е свързан. ✓

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 7, което не е последното. Нека моментът на това достигане е t , а следващото достигане е в момента t' .

Ще докажем ①. Ако след изваждането на връх от Q на ред 8, Q стане празна, антецедентът на ① става лъжа, а цялата импликация, истина. Нека след ред 8 Q не е празна. Ако в момента t съществува $z \in Q$, $z \neq x$, такъв че $z.key < \infty$, доказателството е готово. Да допуснем, че в момента t , за всеки $z \in Q$, $z \neq x$, е вярно, че $z.key = \infty$. Но част ③ от индуктивното предположение влече, че x има поне един съсед z' в Q . Очевидно на ред 11 $z'.key$ ще стане $w((x, z'))$, поради което в момент t' отново е вярно, че Q съдържа връх с key стойност, по-малка от ∞ .

Ще докажем ②. Нека в момента t , $Q_f = \{v \in Q \mid v.key < \infty\}$ и $Q_\infty = \{v \in Q \mid v.key = \infty\}$. Ще разгледаме три множества: $Q_f \setminus N(x)$, $Q_f \cap N(x)$ и $Q_\infty \cap N(x)$.

Върховете от $Q_f \setminus N(x)$ не биват засегнати от изваждането на x от Q и за тях ①, ② и ③ остават в сила в момента t' .

Разглеждаме произволен $v \in Q_f \cap N(x)$. Ако в момента t , $w((x, v)) \geq v.key$, то булевото условие на ред 10 е лъжа, когато y стане v , и редове 11–12 не се изпълняват. Тогава в момента t' , ①, ② и ③ остават в сила за v от индуктивното предположение. Ако в момента t , $w((x, v)) < v.key$, то булевото условие на ред 10 е истина, когато y стане v (ред 9), и редове 11–12 се изпълняват. Тогава $\pi[v]$ става x (ред 12) и ① и ② са в сила за v . На ред 11, $v.key$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото, инцидентно с v , прекосяващо ребро, така че и ③ е в сила.

Разглеждаме произволен $v \in Q_\infty \cap N(x)$. Когато y стане v (ред 9), булевото условие на ред 10 задължително е истина, понеже $v \in Q$ и $v.key = \infty$. После на ред 12, $\pi[v]$ става x , така че ① и ② са в сила за v . На ред 11, $v.key$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото (и единствено), инцидентно с v , прекосяващо ребро, така че и ③ е в сила.

Ще докажем ③. След изваждането на x от Q , за всеки $v \in Q_\infty \cap N(x)$, $v.key$ става $w((x, v))$ на ред 11 и v престава да е елемент на Q_∞ . От друга страна, за всеки $v \in Q_\infty \setminus N(x)$, твърдението остава в сила заради свързаността на графа. □

Теорема 94: Коректността на алгоритъма на Prim

Масивът $\pi[1..n]$, който Алгоритъм 27 връща, реализира МПД на G .

Доказателство: Нека във всеки момент от работата на алгоритъма, $G_{\overline{Q}}$ е подграфът на G , индуциран от върховете, които не са в Q . Твърди се, че съществува МПД T на G , спрямо което Инвариант 28 е в сила.

Инвариант 28: while-цикълът на МПД PRIM2

При всяко достигане на ред 7, $\{(u, \pi[u]) \mid u \in V \setminus (\{s\} \cup Q)\} = E(T) \cap E(G_{\overline{Q}})$.

База. Разглеждаме първото достигане на ред 7. В този момент $\{s\} \cup Q = V$, така че $V \setminus (\{s\} \cup Q) = \emptyset$ и множеството в лявата страна е празно. Но множеството в дясната страна също е празно, понеже всички върхове са в Q , така че $E(G_{\overline{Q}})$ е празно. Със сигурност такава МПД T съществува.

Поддръжка. Да допуснем, че инвариантът е в сила в даден момент t , в който изпълнението е на ред 7 и **while** ще бъде изпълнен поне още веднъж, което влече, че Q не е празна. Щом Q не е празна, съгласно Лема 38 е вярно, че $\exists u \in Q : u.key < \infty$.

Твърди се, че в момента t за всеки връх $v \in Q$, такъв че $v.key$ е минимален, е вярно, че реброто $(v, \pi[v])$ е леко ребро, прекосяващо среза $\{V \setminus Q, Q\}$. Но това следва директно от Лема 38. Съгласно Теорема 93, в момента t за всеки връх $v \in Q$, такъв че $v.key$ е минимален, е вярно, че реброто $(v, \pi[v])$ е сигурно за множеството $E(T) \cap E(G_{\overline{Q}})$, защото това множество е съобразено със среза $\{V \setminus Q, Q\}$.

На ред 8 върхът, който бива изваден от опашката и съхранен в променливата x е връх, който в момента t е в Q и е с минимална key стойност – при допускането, че EXTRACT-MIN работи коректно. Заклучаваме, че след изваждането на този връх от Q , множеството $\{(u, \pi[u]) \mid u \in V \setminus (\{s\} \cup Q)\}$ продължава да е равно на $E(T) \cap E(G_{\overline{Q}})$.

Терминация. При последното достигане на ред 7, опашката Q е празна. Тогава $E(G_{\overline{Q}}) = E(G)$, така че $E(T) \cap E(G_{\overline{Q}}) = E(T)$; освен това, $V \setminus (\{s\} \cup Q) = V \setminus \{s\}$; заклучаваме, че $\{(u, \pi[u]) \mid u \in V \setminus \{s\}\} = E(T)$. С други думи, МПД PRIM2 е построил МПД чрез $\pi[1..n]$. $\square$

Сложност по време. Допускаме, че Q е реализирана чрез двоична пирамида. Ще приложим подхода към анализа на сложността, който ползвахме при BFS/DFS, и ще игнорираме факта, че **while**-цикълът (редове 7–12) се изпълнява $\Theta(n)$ пъти. В израза за сложността няма множител n . Наместо да броим колко пъти се “извърта” **while**-а, ще разсъждаваме колко пъти се изпълнява **for**-цикълът на редове 9–12 за **цялото изпълнение на алгоритъма**. Както знаем, ред 9 се изпълнява $\Theta(n + m)$ пъти, защото това е просто минаване през списъците на съседство. Само че тялото на този **for** се изпълнява в най-лошия случай във време $\Theta(\lg n)$ заради имплицитното викане на DECREASE-KEY на ред 11. Оттук имаме оценка за сложността $\Theta((n + m) \lg n)$, което при свързан граф е $\Theta(m \lg n)$. Колкото и да е странно, ако ребрата на графа са много в смисъл, че $m \asymp n^2$, изтънчения вариант на алгоритъма на Prim с приоритетна опашка е по-лош от базовия вариант, в който намираме следващото ребро с просто последователно търсене. За разредени графи обаче предимството на изтънчения вариант е сериозно: $n \lg n$ срещу n^2 .

Забележете, че в това извеждане игнорирахме факта, че EXTRACT-MIN на ред 8 работи във време $\Theta(\lg n)$. Ако сумираме цялата работа на EXTRACT-MIN по време на изпълнението на алгоритъма, ще получим $\Theta(n \lg n)$. Но $\Theta(n \lg n) = O(m \lg n)$ за свързан граф, така че изразът за сложността може да бъде записан най-кратко като $\Theta(m \lg n)$.

Ако Q бъде реализирана не чрез двоична пирамида, а чрез пирамида на Fibonacci (Fibonacci heap) може сложността да се подобри, поне в асимптотичния смисъл. При пирамидите на Fibonacci, функцията DECREASE-KEY работи в амортизирано време $\Theta(1)$. Функцията EXTRACT-MIN остава със сложност $\Theta(\lg n)$ в най-лошия случай, поради което асимптотичната сложност става $\Theta(m + n \lg n)$. Събираемостта $n \lg n$ отчита работата на всички EXTRACT-MIN по време на изпълнението на алгоритъма. Пирамида на Fibonacci обаче е много сложна структура, ползването на която води до големи скрити мултипликативни константи (скрити в Θ -нотацията). Трябва данните наистина да са големи, за да има осезаема практическа полза от ползването на пирамиди на Fibonacci вместо двоични пирамиди в алгоритъма на Prim.

Пирамидите на Fibonacci имат и друго предимство пред двоичните пирамиди: те може да бъдат сливани ефикасно, в $\Theta(1)$ амортизирано време, докато двоичните пирамиди искат линейно време за сливане. Пирамидите на Fibonacci са изложени подробно в [34, стр. 505].

10.4 Алгоритъм на Kruskal

Алгоритъмът на Kruskal е ефикасен алгоритъм за намиране на МПД, който е основан на съвсем различна идея от тази на алгоритъма на Prim. Най-общо казано, първо сортираме ребрата по тегло, и после **в намаляващия ред** на теглата слагаме първите $n - 1$ ребра, за всяко от които, в този ред, е вярно, че не образува цикъл с вече сложените ребра. Ако обаче имплементираме тази идея буквално, ще получим кубичен алгоритъм.

- Сортирането на ребрата става във, и иска, време $\Theta(m \lg m)$, което е същото като $\Theta(n^2 \lg n)$, ако $m \asymp n^2$.
- В най-лошия случай се налага да проверим всички m ребра. За да се убедим в това, нека най-тежкото ребро е мост. Всеки мост задължително участва във всяко МПД, защото всеки мост участва във всяко ПД. Ерго, в най-лошия случай може да се наложи да стигнем до края на сортираната редица от ребра.

За всяко ребро в сортирана редица тестваме дали образува цикъл с вече сложените ребра. Тоест, за всяко ребро тестваме дали подграфът, индуциран от вече сложените ребра плюс него е цикличен. Ако тестът се направи с BFS или DFS, той става във време $\Theta(n)$ в най-лошия случай[†]. Така че тези тестове ще станат във време $O(nm)$, като може да се покаже, че границата е точна, тоест, $\Theta(nm)$.

При $m \asymp n^2$, това е $\Theta(n^2 \lg n) + \Theta(n^3) = \Theta(n^3)$.

Как да подобрим сложността по време? Сортирането на ребрата отнема $\Omega(m \lg n)$ време и това е неизбежно заради долната граница на сортирането (Подсекция 13.2.2)[‡]. Ерго, имаме долна граница $\Omega(m \lg n)$ за алгоритъма на Kruskal, независимо от това колко ефикасно имплементираме слагането на ребрата.

Слагането на ребрата става итеративно, като в най-лошия случай се изпълняват m итерации (ако най-тежкото ребро е мост). Имайки предвид това, целта ни е тестването за едно ребро да става във време $O(\lg n)$. Ако успеем да постигнем това, ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal; дори да постигнем тестване за едно ребро във време $o(\lg n)$, пак ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal заради сортирането в началото.

Ключовото наблюдение е, че ако имаме гора с повече от едно дърво и добавим ново ребро между два върха:

- ако те са от различни дървета на гората, няма да се образува цикъл, но тези две дървета плюс новото ребро ще станат едно дърво, а графът ще продължи да е гора;
- ако те са от едно и също дърво, ще се образува цикъл, а графът ще престане да е гора.

В светлината на това, за ефикасното тестване дали всяко ново ребро в наредбата образува цикъл е достатъчно да поддържаме разбиване на върховете на графа, като дяловете на разбиването отговарят на дърветата в гората. С други думи, за всяко дърво от гората ни трябва да знаем само кои са върховете му. Ребрата му не ни интересуват. Наистина, алгоритъмът строи множество от ребра и връща множество от ребра (чиято съвкупност е МПД), но по време на итерациите няма смисъл да поддържаме експлицитно отделните дървета в гората като върхове и ребра – интересуват ни само върховете им. Тестването дали ребро образува цикъл се свежда до това дали двата му края са в различни дялове на разбиването: ако не,

[†]А не в $\Theta(n + m)$ – защо?

[‡]За теглата на ребрата няма ограничения, така че не може да ползваме COUNTING SORT.

това ребро се прескача, ако да, то се добавя, като обаче тези дялове трябва да станат един и същи дял.

Забележете промяната в терминологията, която направихме. Поначало говорехме за тестване дали двата края на новото ребро са в различни дървета, всяко от които си има върхове и ребра, и за сливане на тези дървета плюс новото ребро. Сега говорим за тестване дали двата края на новото ребро са в различни множества (само от върхове) и сливане на тези множества; “плюс новото ребро” сега няма.

Ето псевдокод на алгоритъма на Kruskal съгласно [144]. Алгоритъмът поддържа разбиване на множеството от върховете, като в началото създава разбиването на едноелементни множества; функцията $\text{component}(u)$ връща идентификатора на дяла на разбиването, в който е връх u , а функцията $\text{identify}(\text{component}(u), \text{component}(v))$ слива дяловете, в които се намират u и v , при условие, че тези дялове са различни.

Алгоритъм 28: МПД Kruskal

МПД KRUSKAL($G = (\{1, \dots, n\}, E)$: неориентиран свързан граф, w : тегловна ф-я върху G)

```

1   $A \leftarrow \emptyset$ 
2  направи разбиването  $\{\{1\}, \{2\}, \dots, \{n\}\}$ 
3  сортирай  $E$  по тегла
4   $\text{count} \leftarrow 0$ 
5  while  $\text{count} < n - 1$  do
6      нека  $e = (u, v)$  е следващото ребро в сортираната редица
7      if  $\text{component}(u) \neq \text{component}(v)$ 
8           $A \leftarrow A \cup \{e\}$ 
9           $\text{identify}(\text{component}(u), \text{component}(v))$ 
10      $\text{count}++$ 
11 return  $A$ 
```

Коректност. Коректността може да се докаже със следния инвариант: съществува МПД T , такова че при всяко достигане на ред 5 е вярно, че $A \subseteq E(T)$. Реброто e , което избираме на ред 6, има краища u и v . Ако компонентите, в които се намират u и v , да ги наречем T_u и T_v съответно, са различни, булевото условие на ред 7 е истина и изпълнението отива на ред 8. Но щом T_u и T_v са различни, то e прекосява среза $\{V(T_u), V(G) \setminus V(T_u)\}$. Тогава, съгласно Теорема 93, реброто e е сигурно за A .

Сложност по време. За да бъде ефикасен МПД KRUSKAL, трябва проверката дали $\text{component}(u) \neq \text{component}(v)$ (ред 7) и операцията $\text{identify}(\text{component}(u), \text{component}(v))$ (ред 9) да се вършат във време $O(\lg n)$. Ако успеем да постигнем това, **while**-цикълът (редове 5–10) ще се изпълнява във време $O(m \lg n)$ и алгоритъмът ще има сложност по време $\Theta(m \lg n)$ заради сортирането (ред 3).

Всяка компонента, или дял, на разбиването трябва да има идентификатор; тоест, свое име. Нека са дефинирани две примитивни функции: Find и Union. Ако u е елемент от опорното множество (в случая, връх на графа), Find(u) връща името на дяла на разбиването, в който е u . Ако i и j са имената на два различни дяла, Union(i, j) слива тези два дяла в един и му дава име i . Ако разполагаме с такива примитиви, можем да реализираме $\text{component}(u) \neq \text{component}(v)$ така:

$\text{component}(u) \neq \text{component}(v)$

```

1  i ← Find(u)
2  j ← Find(v)
3  if i ≠ j
4      return TRUE
5  else
6      return FALSE

```

и да реализираме $\text{identify}(\text{component}(u), \text{component}(v))$ така:

$\text{identify}(\text{component}(u), \text{component}(v))$

```

1  i ← Find(u)
2  j ← Find(v)
3  Union(i, j)

```

В следващата секция ще видим как се реализират ефикасно примитивите Find и Union.

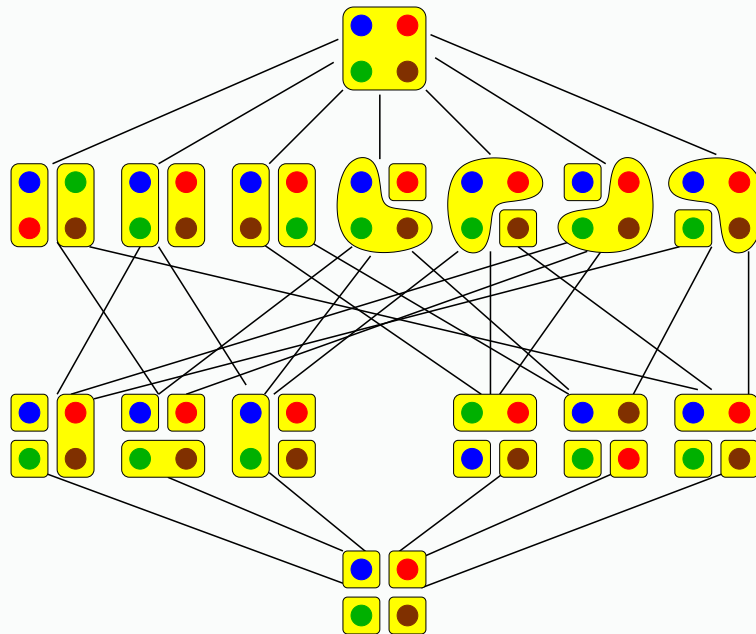
Допълнение 49: Частична наредба $\sqsubseteq$ върху разбиванията

Нека S е множество. Множеството от всички разбивания на S ще бележим с $\Pi(S)$. За всеки $P_1, P_2 \in \Pi(S)$, P_1 *рафинира* P_2 , което означаваме с $P_1 \sqsubseteq P_2$, ако

$$\forall X \in P_1 \exists Y \in P_2 : X \subseteq Y$$

Конверсно, P_2 *абстрахира* P_1 . Примерно, ако $S = \{a, b, c, d\}$, то е вярно, че $\{\{a, b\}, \{c\}, \{d\}\} \sqsubseteq \{\{a, b\}, \{c, d\}\}$. Очевидно релацията $\sqsubseteq$ е частична наредба.

Наредената двойка $(S, \sqsubseteq)$ е вид *решетка* – понятие, което няма да дефинираме тук (вижте [20] и [129] за много подробно и изчерпателно въведение). За целите ни е достатъчно да се каже, че всяка решетка има уникален минимален елемент и уникален максимален елемент. Ето как изглежда диаграмата на Hasse на $(\{a, b, c, d\}, \sqsubseteq)$ (без имената на елементите):



Минималният елемент е $\{\{a\}, \{b\}, \{c\}, \{d\}\}$: разбиването на едноелементни множества (*partition into singletons*). Максималният елемент е тривиалното разбиване $\{\{a, b, c, d\}\}$.

Връзката на всичко това с алгоритъма на Kruskal е следната. В някакъв смисъл, алгоритъмът на Kruskal се “придвижва” от минималния елемент нагоре до максималния в решетката на разбиванията, като на всяка итерация текущото разбиване абстрахира предишното. При това без да се прескачат нива в решетката, защото се сливат два дяла в един, което означава, че броят на елементите в разбиването намалява с единица.

10.5 Union-Find структури данни

10.5.1 Два крайни подхода, които не вършат работа

Да разгледаме две екстремни възможности за реализация именуването на дяловете на разбиването. Понеже говорим за дървета, нека има дялове ДЪБ и БУК.

- Всеки елемент (връх) е запис, който има поле за името на дяла на разбиването, в който се намира.

ДЪБ	ДЪБ	ДЪБ	БУК	БУК	БУК
1	2	3	4	5	6

При този начин на съхранение на имената, проверката $\text{component}(u) \neq \text{component}(v)$ става във време $\Theta(1)$. Но операцията $\text{identify}(\text{component}(u), \text{component}(v))$ е бавна. Да даваме напълно ново (което досега не е използвано) на дяла-резултат от сливането, примерно ОРЕХ, не е добра идея. Да кажем, че едното от двете използвани имена ДЪБ и БУК става името на дяла-резултат от сливането, а другото име изчезва. Да кажем, че остава името ДЪБ. Тогава след сливането имаме:

ДЪБ	ДЪБ	ДЪБ	ДЪБ	ДЪБ	ДЪБ
1	2	3	4	5	6

Очевидно при този начин на съхранение на имената, сливането на дялове става в линейно, а не в логаритмично време.

- Другата екстремна възможност е името да се пази на едно място, а от всеки елемент да има указател към него.

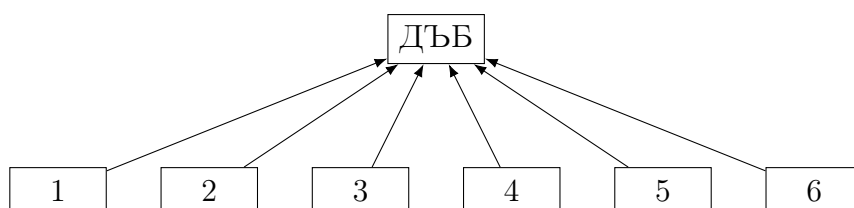


На пръв поглед, това съхранение на имената ни дава възможност да реализираме и Union, и Find в $\Theta(1)$, защото Find(u), където u е елемент, се състои от един единствен *dereference* на указателя на u , а Union(i, j) можем да реализираме, като просто сменим името j с i (в примера, БУК става ДЪБ):



Тази идея, разбира се, не работи. Показаната диаграма е “снимка” на моментно състояние на някакви компоненти, което е недостижимо: поначало компонентите имат по един елемент (защото започваме с разбиването на едноелементни дялове), така че, ако следваме тази идея, всеки елемент ще носи името на дяла си (само че на един указател разстояние) до края и Union пак ще работи в линейно време.

Ако опитаем да премахваме едното хранилище на име, така щото всеки елемент от даден дял да сочи към само едно хранилище на име, пак ще получим Union, работещ в линейно време:



Печелившата идея е да се откажем от крайностите и да възприемем междинен подход: нито “всеки елемент има свое копие на името на компонентата си”, нито “всички хранилища на имена на компоненти са на един указател разстояние от всеки елемент, през цялото време”.

10.5.2 Реализация на разбиване чрез ориентирана гора

Ориентирана гора е множество от ориентирани дървета без общи върхове. В случая става дума за антиарборесценции (Определение 69). Върховете са еднотипни: всеки връх е запис от идентификатора на елемента в множеството и указател към родителя му. Указателите на корените сочат към себе си. Подобни дървета са дърветата на обхождането в BFS/DFS, реализирани с масив на предшествие π . Козметична разлика е, че при онези дървета указателят на корена има стойност Nil, докато тук указателят на корена е “примка” към себе си. В тази подсекция, когато кажем “дърво”, разбираме антиарборесценция с указател на корена, който е примка към себе си.

Забележете това: “върховете са еднотипни”. При компонентите ДЪБ и БУК от миналата подсекция, имената на компонентите бяха съвсем различни от имената на елементите в тях. Математическият подход към описанието на дадено разбиване е точно такъв – ако е дадено множество $Z = \{z_1, \dots, z_n\}$ и разглеждаме някакво разбиване на S , типично именуване на дяловете на разбиването е, да кажем, $W_1, \dots, W_k$. В практическата реализация на разбиване обаче е лоша идея да се ползват отделни имена за дяловете. В практическите реализации, всеки дял на разбиването се идентифицира с точно един от елементите в него. За този елемент казваме, че е *лидер* на дяла на разбиването (в който се намира). В примера със Z от този параграф, всеки дял на разбиването би бил именуван, или идентифициран, с точно едно z_i от него.

Тези лидери са важни. Функцията $\text{Find}(u)$ връща лидера на компонентата, в която се намира u , а функцията $\text{Union}(i, j)$ работи само върху лидери на компоненти.

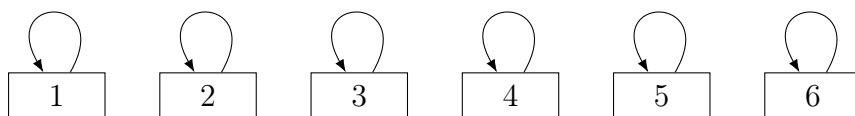
Ето псевдокодът на Find . $\pi[u]$ е родителят на u .

```

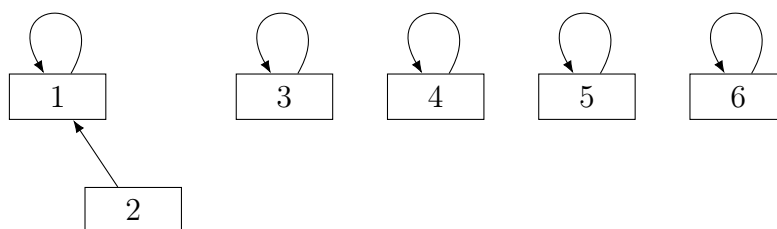
Find(u)
1  if  $u \neq \pi[u]$ 
2      return Find( $\pi[u]$ )
3  else
4      return u

```

При реализация на разбиване с ориентирана гора, всеки дял се представя чрез точно едно дърво от гората, а лидерът на дяла е коренът – това е естественият избор. Началното разбиване на едноелементни множества, което се има предвид на ред 2 от МПД KRUSKAL[†] се реализира от дървета с по един връх-корен ето така:

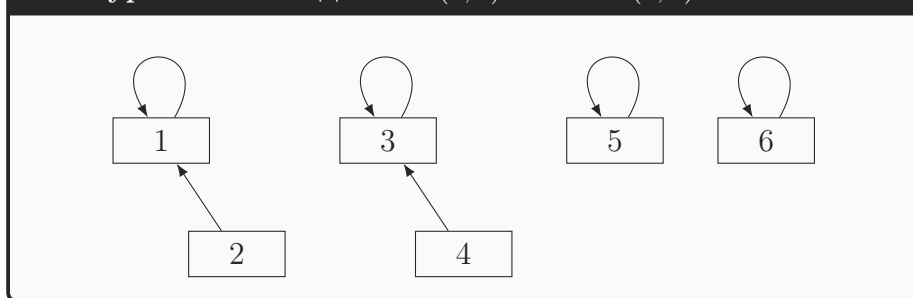


В този момент има шест компоненти, всяка с по един елемент, който е и лидерът. Find върху който и да е елемент ще върне самия него. Да кажем, че се извърши Union(1, 2). След това вече има само пет дяла, като единият е от два елемента, а именно 1 и 2. Да кажем, че лидерът на двуелементния дял е 1. Тъй като имплементираме с ориентирани дървета, трябва указателят на 2 вече да сочи към 1:



Да кажем, че следващото сливане е Union(3, 4). Резултатът е показан на Фигура 10.2.

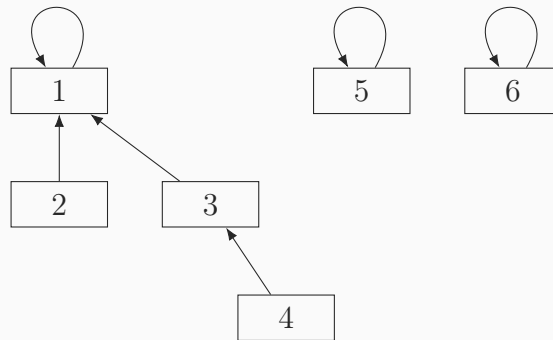
Фигура 10.3 : След Union(1, 2) и Union(3, 4).



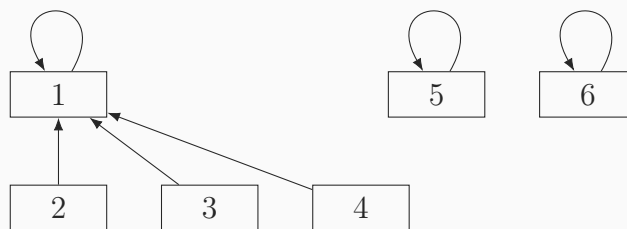
Нека следващото сливане е Union(1, 3). Викането е коректно, защото и 1, и 3 са лидери[‡]. Както се вижда на Фигура 10.3, има два начина да направим дървото-резултат от сливането.

[†]С други думи, минималният елемент на решетката от Допълнение 49.

[‡]Викане Union(1, 4) би било некоректно, защото 4 не е лидер.

Фигура 10.3 : Два варианта за Union(1,3) върху гората от Фигура 10.2.

Вариант 1.



Вариант 2.

Предимството на това, дървото с върхове 1, 2, 3 и 4 да е във Вариант 2 е, че височината му е само едно, за разлика от дървото с тези върхове във Вариант 1, което има височина две. Ако има бъдещо викане на Find(4), във Вариант 1 ще се наложи да се направят два “скока нагоре”, за да се стигне до лидера, докато във Вариант 2, Find(u) за който да е връх $u \in \{1, 2, 3, 4\}$ иска най-много един “скок нагоре”. Предимството на ниските дървета е ясно: бърза работа на Find в най-лошия случай.

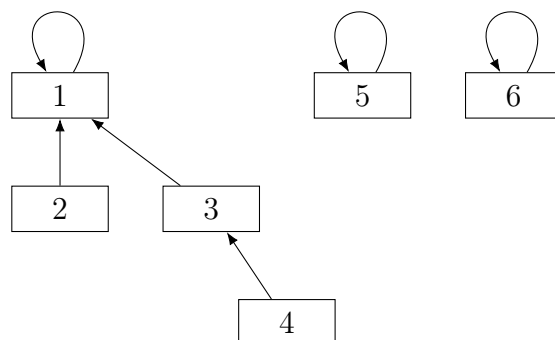
Обаче ние предпочитаме Вариант 1. Фигура 10.3 показва малки дървета, но става ясно, че ако сливаме две произволни дървета, всяко с височина едно, и резултатът трябва да е с височина едно, в най-лошия случай Union ще работи в линейно време в броя на върховете на тези две дървета, който, на свой ред, може да е $\Theta(n)$. А ние искаме Union да работи в логаритмично време. От тези съображения, Вариант 2 отпада. Вариант 1 е изключително бърз: ако лидерите са известни, сливането на дърветата (компонентите) става във време $\Theta(1)$.

Ние обаче искаме и Union, и Find да работят в не по-лошо от логаритмично време в броя на върховете в двете компоненти, при Union, и в единствената компонента, при Find. Видяхме дори как да реализираме Union в константно време. Ако извършваме обаче Union-ите безразборно, може да получим дървета с ненужно голяма височина, поради което Find да стане ненужно бавна.

Наблюдение 67

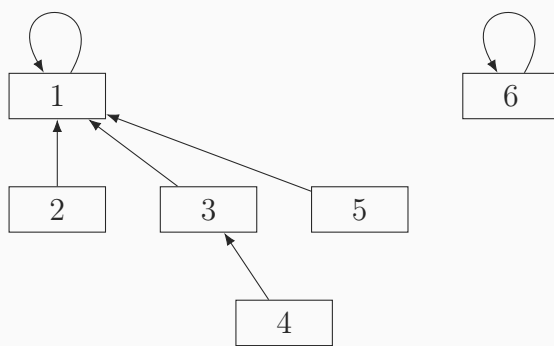
При реализацията с ориентирана гора, сложността по време на Find(u) в най-лошия случай е $\Theta(h)$, където h е височината на дървото, чийто връх е u .

Като пример, да разгледаме пак дървото от Вариант 1 на Фигура 10.3:

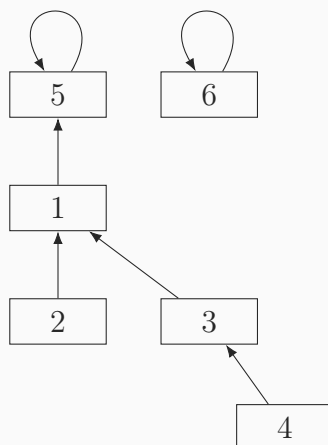


Нека извършим $\text{Union}(1, 5)$. Да разгледаме поотделно тези две възможности: връх 1 да бъде коренът на полученото дърво и връх 5 да бъде коренът на полученото дърво. Те са показани на Фигура 10.4 като съответно Вариант А и Вариант Б.

Фигура 10.4 : Два варианта за $\text{Union}(1, 5)$ върху Вар. 1 от Фиг. 10.2.



Вариант А.



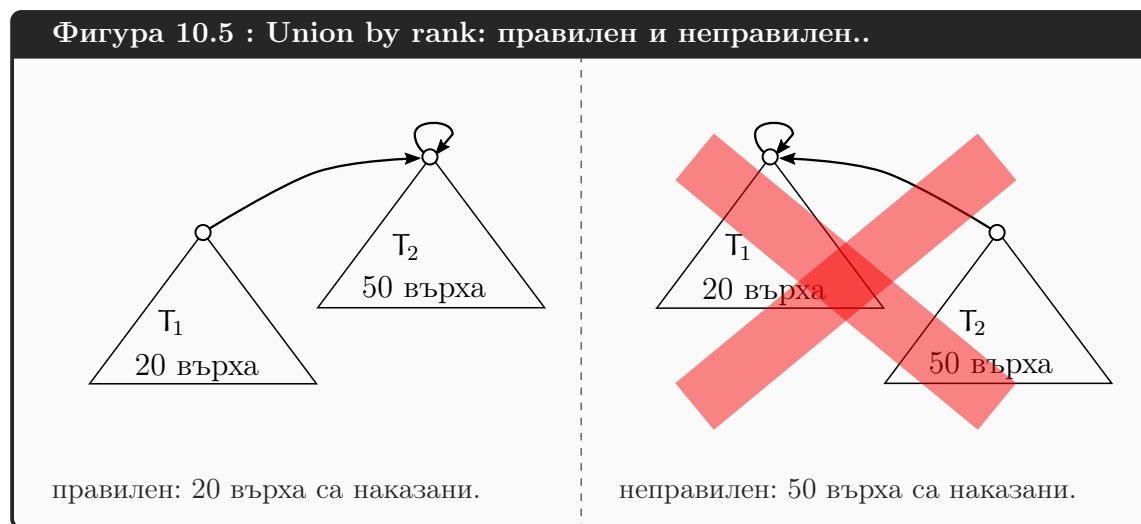
Вариант Б.

Ясно е, че изборът на 5 за нов лидер (Вариант Б) беше неудачен. Нещо повече. Серия от такива неудачни избори може да доведе до дървета с линейна височина, откъдето в най-лошия случай Find ще работи в линейно време. За да избегнем това, прилагаме две *евристики*[†].

[†]На английски терминът е *heuristic*. Свободно казано, означава добра идея за решаване на някаква задача.

Union by rank. Винаги, когато правим Union, “наказваме” върховете на едно от дърветата с увеличаване на дълбочината им с единица. Ако извършваме Union с просто пренасочване на указателя на единия корен към другия корен, това е неизбежно. Но кое от двете дървета да накажем?

Логично е да накажем дървото с по-малко върхове. В това се състои тази евристика: всяка от компонентите съдържа и информация за броя на елементите в себе си, която е *рангът* на дървото. При Union се променя указателят на корена на дървото с по-малък ранг (ако са с еднакви рангове, решението се взема произволно).



Да разгледаме примери за създаване на дървета (разбивания) чрез Union by rank. Започваме с шестнадесет върха.

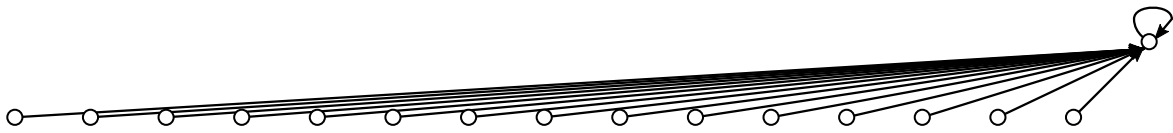


Ако сме късметлии, може сливанията да стават така:

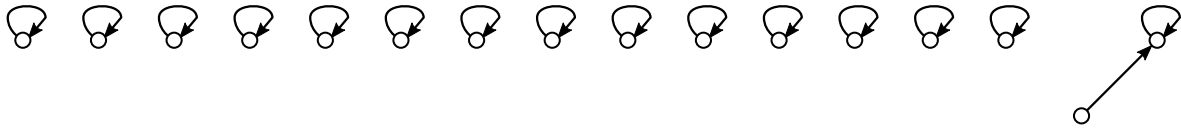


и така нататък, докато получим дърво с височина едно:

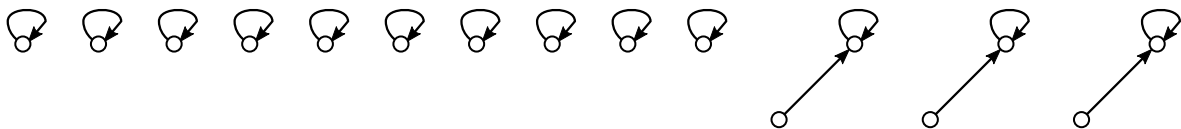
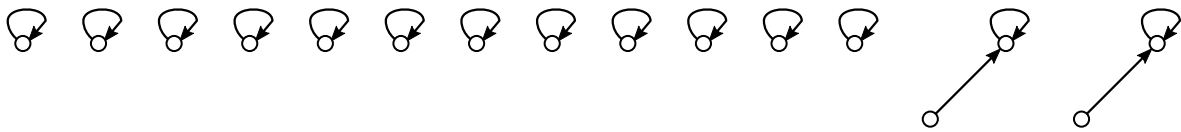
Евристиките не гарантират оптималност на решението и дори не дават гаранция за това, колко далече от оптималното е тяхното решение.



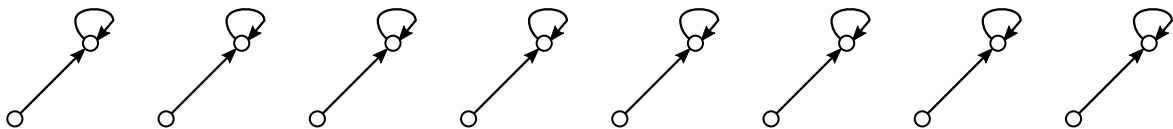
Ако не сме късметлии, може сливанията да стават по следния начин. Първото сливане от-ново е:



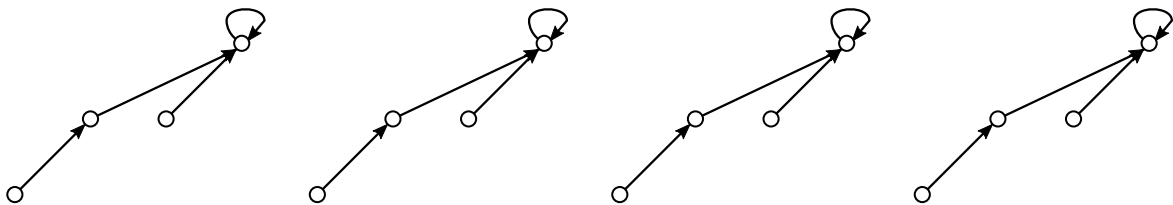
После обаче продължават така:



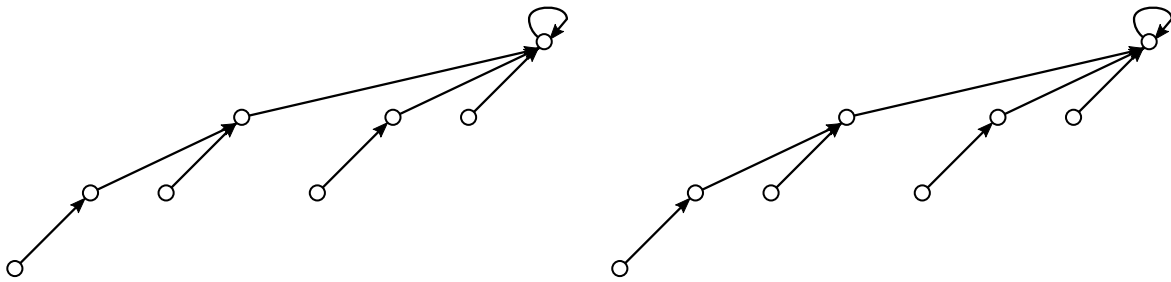
и така нататък до:



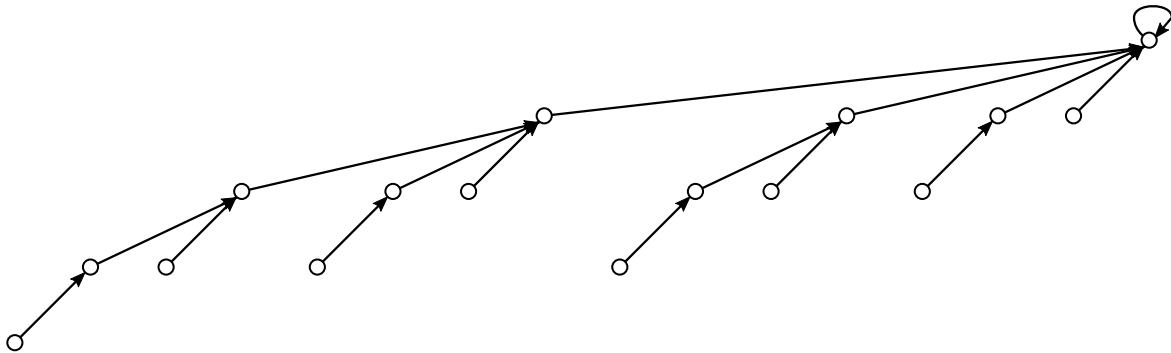
После имаме:



След това:



И накрая:



От тези илюстрации е очевидно, че чрез Union by rank може да получаваме дървета с височина и разклоненост $\lg n$. Но дали това е най-лошата (най-голямата) възможна височина? Оказва се, че да. Сега ще докажем това формално. Само че за лекота на доказателството ще вземем като ранг на дървото не броят на върховете, а височината му.

Теорема 95: Максимална височина на дърво след Union by rank

Започвайки от дървета с по един връх (разбиване на едноелементни множества), след произволна серия от изпълнения на Union by rank, за всяко дърво в колекцията от дървета е вярно, че $h = O(\lg n)$, където h е височината на дървото, а n е броят на неговите върхове.

Трябва да се подчертае отново, че n не е общият брой на върховете във всички дървета, а само в дървото, което разглеждаме.

Доказателство: Твърди се, че съществува константа $c > 0$, такава че $h \leq c \lg n$. Ще го докажем с индукция по h .

База $h = 0$. Дървото може да се състои от само един връх, ако височината му е нула, така че $n = 1$. Наистина, за всяка положителна константа е вярно, че $0 \leq c \cdot \lg 1$. ✓

Индуктивно предположение Допускаме, че съществува константа $c > 0$, така че за някое $h \geq 0$, всяко дърво, генерирано от Union by rank, $h \leq c \lg n$.

Индуктивна стъпка Разглеждаме произволно дърво с височина $h + 1$, генерирано от Union by rank, получено от две дървета с височина h . Защо две дървета с височина точно h ? Защото искаме да видим какво става, когато минаваме към височина $h + 1$, сливайки дървета с по-малки височини. Но тези по-малки височини може да са само h и h , ако искаме

да получим $h + 1$. Нека двете дървета, които сливаме, са T_1 и T_2 , съответно с n_1 и n_2 върха. Предположението е, че

$$h \leq c \lg n_1$$

$$h \leq c \lg n_2$$

БОО, нека $n_1 \geq n_2$. Тогава

$$c \lg (n_1 + n_2) \geq c \lg (n_2 + n_2) = c \lg 2n_2 = c \lg n_2 + c \geq \quad // \text{ съгласно инд. предположение} \\ h + c \geq h + 1$$

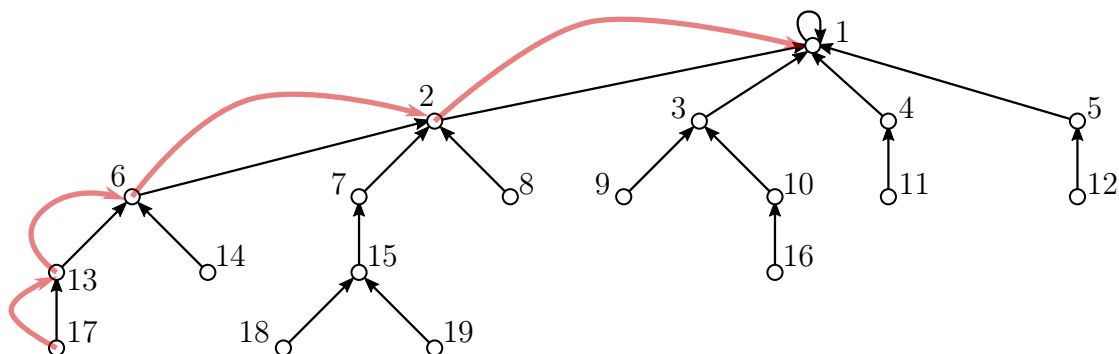
Има такава константа, да кажем $c = 1$. □

Показахме, че Union by rank генерира дървета с най-много логаритмична в броя на върховете им височина. Това означава, че Find върху такива дървета работи в не по-лошо от логаритмично време. На свой ред, това означава, че в МПД KRUSKAL проверката дали $\text{component}(u) \neq \text{component}(v)$ (ред 7) и операцията $\text{identify}(\text{component}(u), \text{component}(v))$ (ред 9) се изпълняват във време $O(\lg n)$. Оттук заключаваме, че МПД KRUSKAL има сложност по време $\Theta(m \lg n)$, като само сортирането ред 3 дава долна граница $\Omega(m \lg n)$.

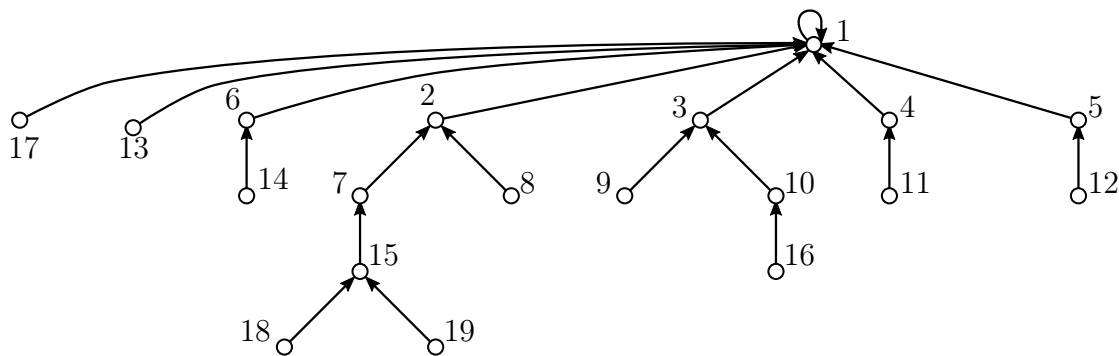
Path compression. Да си припомним текущия псевдокод на Find.

```
Find(u)
1  if u ≠ π[u]
2    return Find(π[u])
3  else
4    return u
```

Още едно подобрение (евристика), която можем да направим към Union-Find структурите, за да ги направим още по-ефикасни, е при всяко изпълнение на Find, при всяко “катериене нагоре” в дървото за достигане до корена, върховете, през които минаваме, да ги наречем $u_1, \dots, u_k$, да бъдат направени деца на корена. Това няма да натовари значително Find, защото той така или иначе трябва да измине пътя до корена – защо да не направи дървото “по-сплескано”, така че при следващи изпълнения на функцията Find върху някой от $u_1, \dots, u_k$, тя да прави точно един “скок нагоре”, за да стигне до корена? Ето пример на дърво в Union-Find структура плюс илюстрация в червено на “скоковете” на Find(17).



Правим всеки от върховете 17, 13 и 6 дете на корена 1.



Ето и псевдокод на подобрения Find.

Find with path compression(u)

```

1  if  $u \neq \pi[u]$ 
2     $\pi[u] \leftarrow \text{Find}(\pi[u])$ 
3  return  $\pi[u]$ 

```

Анализът на сложността на Union и Find операциите при използване на union by rank и path compression евристиките е сложен и нетривиален. В [34, стр. 573] е показано, без стриктно формално доказателство, че при m операции Union или Find върху n елементно опорно множество, в каквато и да е последователност, сумарната сложност по време е $O(m \cdot \alpha(n))$, където $\alpha(n)$ е **изключително бавно** растяща функция, наречена *обратната функция на функцията на Ackermann*. Теоретично говорейки, $\alpha(n)$ е неограничено нарастваща функция на n , но от практическа гледна точка, $\alpha(n) \leq 4$ за всяка стойност на n , която може да възникне на практика. Така че с пълно основание можем да мислим, че $\alpha(n)$ е константа на практика и че сложността на m операции е $O(m)$, което означава сложност $O(1)$ в амортизирания смисъл.

Това подобрение на сложността на Union и Find не води до подобрение на асимптотичната оценка за сложността на алгоритъма на Kruskal. Тя си остава $\Theta(m \lg n)$ заради сортирането. Но Union-Find структурите се ползват не само за реализация на гората в алгоритъма на Kruskal, а и винаги, когато трябва бързо да се установява свързаност между елементи и бързо да се сливат компоненти, така че възможността да правим всяка от тези операции в константно амортизирано време е нещо забележително и полезно.

Допълнение 50: Историята на МПД: Borůvka, Prim, Kruskal, anti-Kruskal

Всички източници, третиращи историята на алгоритмите за конструиране на МПД, посочват като първооткривател на такъв алгоритъм чешкият математик Otakar Borůvka, който през далечната 1926 г. предлага следното [26]. Поддържаеме покриваща гора, от която ще построим МПД чрез добавяне на ребра. Добавянето на ребра слива елементи на гората. Ребра се добавят, докато гората не се превърне в едно единствено дърво, което е желаното МПД. В началото всеки връх на графа е едно дърво в гората. След това се действа итеративно. На всяка итерация, за всяко дърво T от гората намираме ребро с минимално тегло, инцидентно с връх от $V(T)$. После всички тези ребра се

добавят към гората, което води до сливания на дървета. Ако след това остане само едно дърво, алгоритъмът връща него и приключва, в противен случай продължава със следващата итерация.

Алгоритъмът прилича на алгоритъма на Kruskal, но не е същият. В МПД KRUSKAL на стр. 504 на всяка итерация се добавя най-много едно ребро, така че или две дървета от гората се сливат в едно, или не слива нищо, докато алгоритъмът на Borůvka на всяка итерация добавя множество ребра. В екстремален случай, алгоритъмът на Borůvka може да приключи след първата итерация, сливайки всички върхове в едно единствено дърво. Лесно се вижда, че броят на дърветата намалява наполовина или повече с всяка итерация, така че броят на итерациите е $O(\lg n)$. Тъй като се разглеждат всички ребра, сложността по време на алгоритъма на Borůvka е $O(m \lg n)$, ако действията по намиране на леките ребра, добавянето им и последващите сливания се вършат ефикасно [46, стр. 262]. Нещо повече, Jeff Erickson казва [46, стр. 262–263], че алгоритъмът на Borůvka е най-добрият алгоритъм за намиране на МПД на практика, като сред неговите предимства са

1. това, че в най-лошия случай броят на дърветата намалява наполовина, откъдето е множителят $\lceil \log_2 n \rceil$ в израза за сложността, но на практика се случва намаляването да е много повече от наполовина, което сваля този множител значително;
2. това, че за широк клас графи, включително планарните, алгоритъмът на Borůvka може да се имплементира със сложност $O(m)$;
3. това, че алгоритъмът на Borůvka може да се имплементира ефикасно като паралелен алгоритъм, защото намирането на най-леки ребра, инцидентни с дърветата на гората, може да се прави едновременно, стига да разполагаме с подходяща машинария.

Оригиналната статия на Borůvka [26] е на чешки, но има свободно достъпен превод на английски [121].

Историята на алгоритмите за МПД е прекалено сложна, с много автори, открили независимо един от друг едни и същи факти или алгоритми. Тук ще споменем само един фрагмент от тази история.

Статията на Kruskal [98], която излага на вниманието на специалистите алгоритъма, който днес наричаме “алгоритъма на Kruskal”, всъщност описва три различни алгоритъма за построяване на МПД. Алгоритъмът на Kruskal е само един от тях. Освен него, там е описан и алгоритъмът на Prim (поне днес го наричаме така), а също така и един алгоритъм, който някои автори наричат ANTI-KRUSKAL. Статията [98] цитира една единствена статия, а именно [26], но още в началото Kruskal заявява, че предложението от Borůvka алгоритъм е ненужно сложен (не като висока сложност по време, а като описание и за имплементация):

The construction given in [l] is unnecessarily elaborate.

Кое е в явно противоречие с мнението на Jeff Erickson, че на практика алгоритъма на Borůvka превъзхожда Prim и Kruskal.

Известен интерес представлява алгоритъмът ANTI-KRUSKAL. Той разглежда ребрата в ненарастващ ред по тегла (точно обратното на МПД KRUSKAL), като за всяко ребро е в този ред, ако изтриването на е от графа не води до несвързаност (с други думи,

ако e не е мост в текущия граф), изтриваме e , в противен случай го прескачаме. Този алгоритъм още е известен като *reverse-delete algorithm*. Неговата коректност се доказва сравнително лесно. Сложността му по време е $O(m(\lg n)(\lg \lg n)^3)$, като това е най-добрата известна долна граница, която се доказва с нетривиално сложен анализ.

Лекция 11

Намиране на най-къси пътища.

Алгоритми на Dijkstra, върху дагове и на Bellman-Ford.

Резюме: Въвеждаме задачата за намиране на най-къси пътища в ориентирани тегловни графи в няколко варианта. Доказваме, че най-къс път се състои от най-къси подпътища. Разглеждаме представянето на всички най-къси пътища от един връх чрез едно единствено дърво. Разглеждаме проблемите, възникващи при наличие на ребра с отрицателни тегла. Разглеждаме три ефикасни алгоритъма за намиране на най-къси пътища от даден връх до всички останали: алгоритъмът на Dijkstra, алгоритъмът върху дагове и алгоритъмът на Bellman-Ford. Подробно доказваме коректността на всеки от тях и изследваме сложността им по време.

11.1 Фундамент

11.1.1 Базови дефиниции

В тази лекция по подразбиране разглеждаме ориентирани тегловни графи. Ако в даден момент разглеждаме неориентирани графи, това ще се каже изрично. Примки не се допускат, понеже те или нямат отношение към най-късите пътища, ако теглата им са положителни, или представляват отрицателни цикли, ако теглата им са отрицателни – а отрицателните цикли са “проклятието” на задачата за най-къси пътища, както ще видим нататък. Не се допускат и паралелни ребра, тъй като от всеки сноп паралелни ребра, за най-късите пътища значение има само най-леко ребро. Така че и мултиграфи не се допускат.

По подразбиране $G = (V, E)$ е ориентиран тегловен граф с тегловна функция $w : E \rightarrow \mathbb{R}$.

Конвенция 12

В тази лекция, казвайки “път”, имаме предвид път, който не е непременно прост.

Щом пътищата не са непременно прости, в ориентиран граф с цикли, $\text{Paths}(u, v)$ (може да) е безкрайно множество. В неориентиран граф, всяко ребро задава безкрайна колекция от пътища, така че $\text{Paths}(u, v)$ е безкрайно множество, освен особените случаи на липса на пътища между u и v , както и на $u = v$ като изолиран връх.

Определение 91: Тегло на път

Теглото на пътя p е $w(p) = \sum_{e \in E(p)} w(e)$.

Определение 92: Отрицателен цикъл

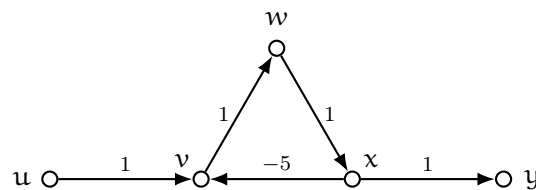
Нека G е тегловен граф. Ако G е ориентиран, *отрицателен цикъл* е всеки прост цикъл в G , който има отрицателна сума от теглата на ребрата. Ако G е неориентиран, отрицателен цикъл е всеки прост цикъл в G , който има отрицателна сума от теглата на ребрата, както и всяко ребро с отрицателно тегло.

Наблюдение 68: Проблем при отрицателните цикли

Ако теглата са само положителни, “най-къс път от u до v ” е добре дефинирано понятие. Обаче за всеки отрицателен цикъл c , за всеки път p , който съдържа c , съществува път p' , такъв че $w(p') < w(p)$, като p' “минава през” c повече пъти от p . Ерго, ако поне един път от $\text{Paths}(u, v)$ съдържа поне един връх от отрицателен цикъл, не можем да дефинираме “най-къс път от u до v ”.

Ако няма отрицателни цикли, този проблем не съществува дори при наличие на отрицателни тегла.

Като пример, да разгледаме следния граф:



Ако разглеждаме само прости пътища, най-късият път (той е и единствен) $u \rightsquigarrow y$ е пътят u, v, w, x, y с тегло 4. Ако обаче разглеждаме пътища, които не са непременно прости, най-къс път $u \rightsquigarrow y$ няма, защото всяко минаване през цикъла v, w, x, v добавя $1 + 1 - 5 = -3$ към теглото на пътя:

u, v, w, x, v, w, x, y има тегло 1

$u, v, w, x, v, w, x, v, w, x, y$ има тегло -2

$u, v, w, x, v, w, x, v, w, x, v, w, x, y$ има тегло -5

и така нататък.

Наблюдение 69: Когато няма отрицателни цикли

При липса на отрицателни цикли, за всеки два върха u и v , всеки най-къс път от u до v задължително е прост път.

Определение 93: Теглото на най-къс път

Теглото на най-къс път от u до v е

$$\delta(u, v) = \begin{cases} \infty, & \text{ако } \text{Paths}(u, v) = \emptyset \\ -\infty, & \text{ако } \text{Paths}(u, v) \neq \emptyset \text{ и } \exists p \in \text{Paths}(u, v) : p \text{ съдържа отрицателен цикъл} \\ \min \{w(p) \mid p \in \text{Paths}(u, v)\}, & \text{ако } \text{Paths}(u, v) \neq \emptyset \text{ и } \neg \exists p \in \text{Paths}(u, v) : p \text{ съд. отр. цик.} \end{cases}$$

Причината да дефинираме като ∞ теглото на най-къс път при липса на път е очевидна. Причината да дефинираме като $-\infty$ теглото на най-къс път при наличие на начин да се мине през отрицателен цикъл е Наблюдение 68. И така, $\delta(u, v) \in \mathbb{R} \cup \{-\infty, \infty\}$.

Има леко терминологично несъответствие: щом говорим за “тегло на път”, би трябвало да казваме “най-лек път” за път с минимално тегло, а не “най-къс път”. Но както “тегло на път”, така и “най-къс път” (в тегловния смисъл) вече са широко приети както на български, така и на английски, като съответно се казва “weight of path” и “shortest path weight”, така че в тези лекции ще се съобразим с това. И така, ще се придържаме към следната езикова конвенция.

Конвенция 13

В контекста на задачата, която разглеждаме, “най-къс път” е синоним на “път с най-малко тегло”. Ако имаме предвид броя на ребрата в пътя, ще го кажем експлицитно.

Конвенция 14

Тегловните функции, които ще разглеждаме, или имат кодомейн $\mathbb{R}^+$, или имат кодомейн $\mathbb{R}$, но в G няма отрицателни цикли. Поради това никъде няма да дефинираме пътищата като прости пътища. Те ще се оказват прости вследствие на отсъствието на отрицателни цикли, предвид Наблюдение 69.

11.1.2 Варианти на задачата

Известни са следните варианти на задачата за най-къс път в тегловен граф.

Изч. Задача 27: SINGLE PAIR SHORTEST PATH

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w , начален връх s , краен връх t .

решение: $\delta(s, t)$

Изч. Задача 28: SINGLE SOURCE SHORTEST PATHS

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w , начален връх s .

решение: $\forall v \in V : \delta(s, v)$.

Изч. Задача 29: SINGLE DESTINATION SHORTEST PATHS

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w , краен връх t .

решение: $\forall v \in V : \delta(v, t)$.

Изч. Задача 30: ALL PAIRS SHORTEST PATHS**екземпляр:** Ориентиран граф $G = (V, E)$, тегловна функция w .**решение:** $\forall u, v \in V : \delta(u, v)$.

SINGLE PAIR SHORTEST PATH изглежда алгоритмично най-лесна. Както ще се убедим обаче, в най-лошия случай, за да изчислим теглото на най-лек път $s \rightsquigarrow t$, се налага да изчислим и теглото на най-лек път $s \rightsquigarrow v$, за всеки $v \in V$. Ерго, в най-лошия случай, SINGLE PAIR SHORTEST PATH е трудна колкото SINGLE SOURCE SHORTEST PATHS.

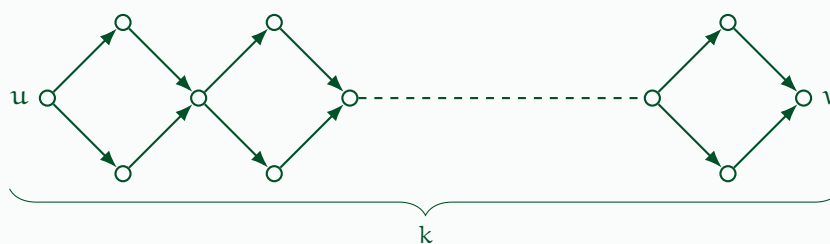
Ние ще разгледаме подробно SINGLE SOURCE SHORTEST PATHS, като за краткост ще я наричаме SSShP. Ние ще решим SSShP директно в тежкия вариант.

SINGLE DESTINATION SHORTEST PATHS има алгоритмичната трудност на SINGLE SOURCE SHORTEST PATHS, защото SINGLE DESTINATION SHORTEST PATHS се превръща в SINGLE SOURCE SHORTEST PATHS при обръщане на посоките на ребрата; тоест, на транспониране на графа.

ALL PAIRS SHORTEST PATHS ще бъде разгледана подробно. За краткост я наричаме APShP. Очевидно APShP се редуцира лесно до SSShP, защото, ако имаме алгоритъм за SSShP, можем да го пуснем от всеки връх и да намерим решение. Но има интересни ефикасни алгоритми за APShP, които решават само APShP в смисъл, че нямат ограничена версия, която да решава SSShP по-ефикасно. Ние ще разгледаме и изследваме такъв алгоритъм в Лекция 12, Подсекция 12.1.4. Причината да отложим разглеждането му за Лекция 12 е, че той е конструиран по схемата **Динамично Програмиране**. За да бъде разбран добре, трябва читателят да има представа от динамично програмиране.

Допълнение 51: Генерирането на всички най-къси пътища е неефикасно.

В реални приложения може да е много полезно да бъдат получени всички най-къси пътища от връх до друг връх. Имайки всички най-къси пътища, може да изберем някой от тях по друг критерий. За съжаление, в най-лошия случай, броят на най-късите пътища $u \rightsquigarrow v$ може да е експоненциален в n , в което може да се убедим с този прост пример (приемете, че теглата са единици):



Виждаме k на брой подграфи, да ги наречем *ромбовете*, като всеки ромб е . Ромбовете са “слепени” в редица. Очевидно на фигурата има 2^k пътя от u до v , всеки от тях с дължина $2k$, защото за всеки ромб, може да минем или “отгоре”, или “отдолу”. И тъй като е възможно $k \asymp n$, в графа може да има $\Omega(2^n)$ пътя от u до v . Следователно, няма ефикасен алгоритъм, който строи всички пътища, защото само извеждането им на изхода на алгоритъма отнема време $\Omega(2^n)$.

11.1.3 Ключово свойство на най-късите пътища: оптимална структура се състои от оптимални подструктури

Следният резултат е Lemma 24.1 в [34, стр. 645].

Теорема 96: Най-къс път се състои от най-къси подпътища

Нека G е ориентиран тегловен граф, s и t са върхове в него и p е най-къс път от s до t . Нека $u, v \in V(p)$, като u предхожда v от s към t в p , ако u и v са различни^a. Нека подпътят на p от u до v се казва q :

$$p = s \rightsquigarrow \underbrace{u \rightsquigarrow v}_q \rightsquigarrow t$$

Тогава q е най-къс път от u до v .

^aНе е необходимо s, t, u и v да са четири различни върха; в най-екстремния вариант, може $s = t = u = v$ и пак остава вярно, че най-къс път се състои от най-къси подпътища.

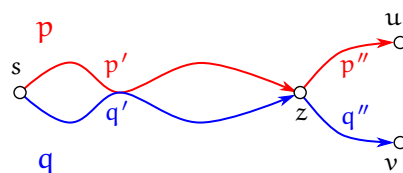
Доказателство: Ако допуснем, че има път q' , такъв че $u \rightsquigarrow v$ и $w(q') < w(q)$, веднага заключаваме, че p не е най-къс път от s до t , понеже замяната на q с q' в p намалява $w(p)$ с $w(q) - w(q')$. $\square$

В доказателството на Теорема 96 се възползваме от това, че пътищата не са непременно прости. Ако настоявахме пътищата да са прости, то вмъкването на q' на мястото на q може да е проблематично, понеже няма гаранции, че q' и p нямат други общи върхове освен u и v ; ерго, след вмъкването, целият път може да не е прост. Ако теглата са положителни, можем “да изрежем” общите части на p и q' и да получим път p'' , който е с дори още по-малко тегло, което пак ни дава желаното противоречие. Обаче, ако има отрицателни тегла, трябва да покажем, че това “изрязване” не увеличава теглото на пътя, което прави доказателството по-тежко.

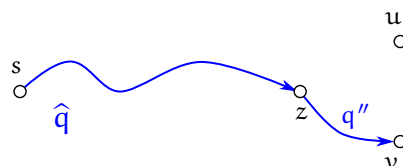
11.1.4 Дърво на най-късите пътища

Ако искаме да изчислим най-къс път $s \rightsquigarrow t$, паметта, която е необходима за записването на пътя е $\Theta(n)$ в най-лошия случай, защото, в най-лошия случай, дължината на пътя е $\Theta(n)$ и не можем да избегнем записването на всеки връх от него. На пръв поглед, ако искаме да запишем по един най-къс път $s \rightsquigarrow v$ за всеки $v \in V$, ще ни трябва общо $O(n^2)$ памет, като може да се покаже, че асимптотичната горна граница $O(n^2)$ е точна; ерго, имаме право да кажем, че ще ни трябва $\Theta(n^2)$ памет.

Но това е само на пръв поглед. Всъщност, можем “да минем” само с $\Theta(n)$ памет за всички пътища, защото можем да ги представим с ориентирано кореново дърво-арборесценция, която може да се представи с масив на предшествията $\pi[1..n]$, също както при BFS и DFS. Ще покажем, че най-къси пътища от s могат да се представят с една арборесценция с корен s . Нека p и q са най-къси пътища от s съответно до u и v , където u и v са различни върхове. Ако единственият общ връх на p и q е s , няма какво да се показва. Нека p и q имат поне един общ връх освен s . Нека z е най-отдалеченият в p и в q връх от s , който е общ за p и q . Очевидно не може $z = u = v$, понеже u и v са различни. Ако $z = u \neq v$ или $z = v \neq u$, няма какво повече да показваме; това са случаите, в които съответно p е част от q или q е част от p . Остава да разгледаме случая, в който $z \neq u$ и $z \neq v$. Нека подпътят на p от s до z е p' , а подпътят на q от s до z е q' :

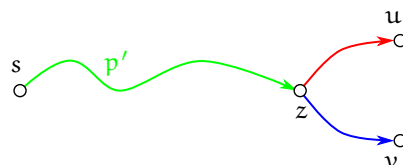


Твърдим, че $w(p') = w(q')$. Да допуснем противното. БОО, нека $w(p') < w(q')$. Тогава в q подменяме q' с p' :



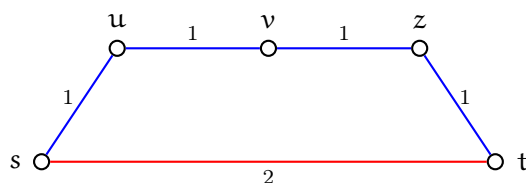
и получаваме път $\hat{q}$ от s до v , такъв че $w(\hat{q}) < w(q)$, в противоречие с допускането, че q е най-къс път от s до v .

Щом $w(p') = w(q')$, може и в p , и в q да използваме само единият от тях, да кажем p' ; по този начин, получаваме най-къси пътища от s до u и от s до v , които се състоят от един общ подпът от s до z , след което се “разделят” и повече не се събират:



Ако си направим същото нещо за всеки два върха, достижими от s , ще получим арборесценция с корен s , което е дървото на най-късите пътища от s .

Задачата за най-късите пътища и задачата за МПД са различни. Естествено, те са задачи върху различни видове графи, като МПД е върху неориентирани тегловни графи, а най-късите пътища са върху ориентирани тегловни графи, но съществената разлика не е в това. Дори да решаваме задачата за най-късите пътища върху неориентирани тегловни графи, тя остава принципно различна от задачата за МПД. Ето малък пример:



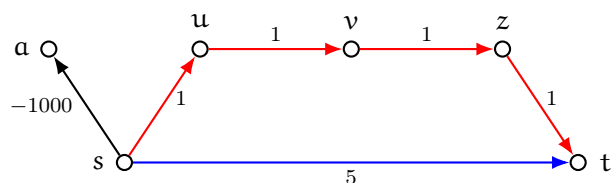
Най-късият път между s и t е реброто (s, t) с тегло 2 (в червено). Но МПД-то (то е само едно) се състои от четирите сини ребра, всяко с тегло 1. Ако си представим работата на алгоритъма на Kruskal върху този граф, сортирайки ребрата, той ще постави в началото четирите ребра с тегло 1 и след това реброто с тегло 2. Когато започне да слага ребра, той ще сложи четирите ребра с тегло 1 и ще спре, без да достигне до реброто с тегло 2, което реализира най-късия път.

Наблюдение 70

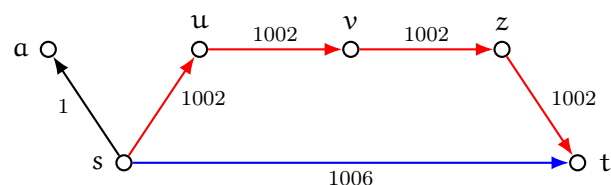
Задачите за намиране на МПД и намиране на дърво на най-късите пътища върху неориентирани тегловни графи са принципино различни. Всяко МПД, за всеки два върха s и t , дава (уникален) път p между s и t , но в общия случай p не е най-къс път в графа между s и t .

Друга важна разлика между тези две задачи е възможността да се “отървем” от отрицателни тегла. Когато дефинирахме задачата за МПД, дефинирахме теглата като реални числа, оставяйки възможността те да са отрицателни. Наистина, нищо не пречи теглата да са отрицателни в Теорема 93 или в алгоритмите на Prim и Kruskal. Ако обаче по някаква причина не искаме отрицателни тегла там, лесно може да се отървем от тях, като намерим най-малкото отрицателно тегло x и после добавим $|x| + 1$ към теглата на всички ребра. Те ще станат положителни, а МПД-тата на графа ще останат **същите**.

Този трик не работи при най-късите пътища. Ако “повдигнем” теглата достатъчно, така че всички те да станат положителни, дървото на най-късите пътища може да се промени. Ето малък пример за това, този път с ориентиран граф:



Очевидно най-късият път от s до t се състои от червените ребра и има тегло 4. Ако обаче добавим 1001 към теглото на всяко ребро, така че теглата да станат само положителни:



пътят от червените ребра става с тегло 4008, а този със синьото ребро, само 1006, така че сега той е най-късият път от s до t .

Наблюдение 71

Нека G е тегловен неориентиран граф с тегловна функция $w : E \rightarrow \mathbb{R}$. Нека $w' : E \rightarrow \mathbb{R}$ е друга тегловна функция, като $\forall e \in E : w'(e) = w(e) + c$, където c е произволна реална константа. МПД-тата на G по отношение на w и по отношение на w' са едни и същи. Ако обаче G тегловен ориентиран граф с тегловна функция $w : E \rightarrow \mathbb{R}$ и $w' : E \rightarrow \mathbb{R}$ е друга тегловна функция, като $\forall e \in E : w'(e) = w(e) + c$, където c е произволна реална константа, дървото на най-късите пътища по отношение на произволен $s \in V(G)$ може да бъде различно за w и за w' .

11.1.5 Пак за отрицателните тегла

Както се убедихме, отрицателните тегла представляват значителен проблем за задачата за най-късите пътища (за разлика от задачата за МПД, където отрицателните тегла нямат никакво значение). При наличие на отрицателни тегла ние може дори да не сме в състояние да дефинираме “най-къс път от s до t ”, ако поне един път от s до t съдържа отрицателен цикъл.

Това се дължи на факта, че разглеждаме пътища, които не са непременно прости. Ако пътищата не са непременно прости, то всяко “завъртане” през отрицателен цикъл дава още по-къс път. Не може ли да постулираме, че разглеждаме само прости пътища, и по този начин да няма възможност да се “въртим” в циклите поначало? Отговорът е, че теоретично можем да го направим, но алгоритмично не можем да го имплементираме по ефикасен начин. Нашите алгоритми за най-къси пътища не правят проверки дали изградените пътища са прости или не. Нашите алгоритми

- или конструират пътища, които се оказват прости (вижте Наблюдение 69; в алгоритъма на Dijkstra теглата са положителни, така че това е в пълна сила),
- или установяват, че в поне един път измежду тези, които биват конструирани, има повторение на върхове (примерно, алгоритъмът на Bellman-Ford “усеща” повторение на върхове) и терминират със съобщение за грешка; тоест, алгоритъмът отказва да търси най-къси пътища след установяване на съществуване на отрицателен цикъл.

В Секция 12.11 е показана схема, по която може да се построи алгоритъм за най-дълги пътища, при който пътищата задължително са прости, понеже преди добавяне на връх към път се прави проверка дали този връх вече не се среща в този път. За съжаление, ако конструираме алгоритъм по схемата от Секция 12.11, той би бил с експоненциална сложност по време.

Силна улика за това, че не може да решим проблема с отрицателните цикли с елементарни средства—примерно, постулиране, че не разглеждаме други пътища освен прости—е фактът, че задачата за най-къси ориентирани пътища е същата като задачата за най-дълги ориентирани пътища (Задача 57) при обръщане на знака на теглата. Иначе казано, ако G е ориентиран тегловен граф, върху който са дефинирани тегловни функции $w, w' : E \rightarrow \mathbb{R}$, като $\forall e \in E(G) : w(e) = -w'(e)$, то, за всеки $s, t \in V(G)$, p е най-къс път $s \rightsquigarrow t$ по отношение на w тстк p е най-дълъг път $s \rightsquigarrow t$ по отношение на w' . Задачата за най-дълги пътища, както в ориентирания, така и в неориентирания вариант, е **NP**-трудна, което означава, че почти сигурно за нея няма ефикасен алгоритъм. Какво означава “**NP**-трудна” и защо задачата за най-дългите пътища е такава, ще видим в Лекция 14, Лекция 15 и Лекция 16.

Допълнение 52: За структурата на най-дългите пътища

За задачата за най-дългите пътища не е вярно, че оптималната структура се състои от оптимални подструктури (сравнете с Теорема 96). Това обяснява драстичната разлика между ефикасността на най-добрите известни алгоритми за най-къси пътища и за най-дълги пътища. Ето и задачата като оптимизационна задача в неориентирания, нетегловен, олекотен вариант.

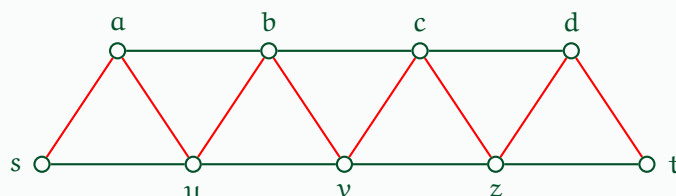
Изч. Задача 31: LONGEST PATH МЕЖДУ ДВА ВЪРХА, ОПТИМИЗАЦИОННА ВЕРСИЯ

екземпляр: Неориентиран граф $G = (V, E)$, върхове s и t от V .

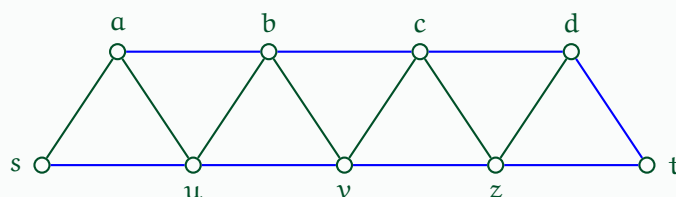
решение: Дължина на най-дълъг прост път между s и t в G .

Забележете, че в задачата за най-дългите пътища задължително разглеждаме само прости пътища. Иначе, дори в нетегловния вариант, задачата е лошо дефинирана, понеже съществуването на цикъл в ориентиран граф или на ребро в неориентиран граф влече съществуването на неограничено дълги пътища, които не са прости.

Лесно е да намерим пример, в който най-дългият път не се състои от най-дълги подпътища:



Най-дългият път между s и t е с дължина 8 и е нарисован с червено. В него има подпът между s и a с дължина 1. Но най-дългият път между s и a е с дължина 8, както се вижда на следната фигура (в синьо):



Очевидно не е вярно, че най-дългите пътища се състоят от най-дълги подпътища.

След всички изложени проблеми, които възникват при отрицателни тегла на ребрата, възниква въпроса, а защо не се откажем от отрицателни тегла поначало? Наистина, ако теглата моделират някакви физически величини, които по природа да положителни, примерно закъснения във времето или разстояния в реалния свят, няма как да се появят отрицателни тегла; отрицателно закъснение, например, означава пътуване назад във времето. Обаче има важни практически задачи, в които е удобно да се работи с абстракцията на отрицателните числа. Примерно, ако моделираме някакви потоци от стоки или пари, можем да кажем, че от сметката на X са прехвърлени 100 лева в сметката на Y , като кажем, че от сметката на Y са прехвърлени -100 лева в сметката на X . Така че, ако се откажем от отрицателни тегла

изобщо, графите ни ще станат със строго по-малка моделираща мощ.

Допълнение 53: Намиране на арбитражи чрез отрицателни цикли

Арбитраж, на английски *arbitrage*, е термин от финансите и търговията, който означава, най-общо казано, купуване на нещо, последвано почти веднага от продаването на същото нещо (по всяка вероятност, на друг пазар), като, естествено, купуването е на ниска цена, а продаването, на висока цена.

В случая става дума за търговия на валути. В този контекст, арбитраж е почти едновременно купуване и продаване на ценни книжа или валути на различни пазари с цел печалба от разминавания в курсовете. Като пример, да разгледаме валутите евро **EUR**, български лев **BGN**, унгарски форинт **HUF**, чешка крона **CZK** и монголски тугрик **MNT**. Дадени са обменните курсове в квадратна матрица $M[i, j]$, като $M[i, j]$ има смисъл на количество валута j , която можем да закупим за единица валута i . Матрицата задължително има свойството $M[i, j] = \frac{1}{M[j, i]}$ при $i \neq j$ – обратното би било арбитраж върху само две валути. Главният диагонал очевидно е от единици. Ето матрица с курсовете на тези валути^a.

	EUR	BGN	HUF	CZK	MNT
EUR	1	1.95583	352.604	27.6033	3 035.51
BGN	0.51129	1	180.241	14.1150	1 551.88
HUF	0.0028360	0.0055481	1	0.0783177	8.60872
CZK	0.036227	0.0055481	12.7685	1	110.009
MNT	0.00032943	0.00064437	0.116161	0.0090834	1

Нека продадем и купим валути в тази последователност:

EUR $\rightarrow$ BGN $\rightarrow$ MNT $\rightarrow$ CZK $\rightarrow$ HUF $\rightarrow$ EUR

Съгласно данните от матрицата, за всяко евро, с което започваме, получаваме

$$1.95583 \times 1551.88 \times 0.0090834 \times 12.7685 \times 0.0028360 \approx 0.99835$$

евро. Губим съвсем малко, по-малко от 0.2%. Важното е, че не печелим. Това е напълно естествено – в реалния свят не може да се печели толкова елементарно.

Сега да манипулираме курсовете. Променяме курсовете между **CZK** и **HUF**, както е показано в червено.

	EUR	BGN	HUF	CZK	MNT
EUR	1	1.95583	352.604	27.6033	3 035.51
BGN	0.51129	1	180.241	14.1150	1 551.88
HUF	0.0028360	0.0055481	1	0.063403	8.60872
CZK	0.036227	0.0055481	15.77188	1	110.009
MNT	0.00032943	0.00064437	0.116161	0.0090834	1

Продаваме и купуваме валути в същата последователност:

EUR $\rightarrow$ BGN $\rightarrow$ MNT $\rightarrow$ CZK $\rightarrow$ HUF $\rightarrow$ EUR

но сега вече ползваме манипулираните курсове. За всяко евро, с което започваме, получаваме

$$1.95583 \times 1551.88 \times 0.0090834 \times 15.77188 \times 0.0028360 \approx 1.2332$$

евро. С други думи, печалба от над 23% заради извършената манипулация.

Виждаме, че някои матрици позволяват арбитраж с печалба. Задача е, при дадена матрица M да се намери дали има арбитраж с печалба. По-подробно казано, да се определи дали има редица от валути $c_{i_1}, c_{i_2}, \dots, c_{i_k}$, такава че

$$M[i_1, i_2] \times M[i_2, i_3] \times \dots \times M[i_{k-1}, i_k] \times M[i_k, i_1] > 1 \quad (11.1)$$

Това е Problem 24.3 в [34, стр. 679]. Тази задача има добре известно ефикасно решение, което я свежда до задачата за намиране на отрицателен цикъл в граф. Логаритмуваме двете страни в (11.1) и получаваме:

$$\lg(M[i_1, i_2]) + \lg(M[i_2, i_3]) + \dots + \lg(M[i_{k-1}, i_k]) + \lg(M[i_k, i_1]) > 0 \quad (11.2)$$

което е същото като

$$-\lg(M[i_1, i_2]) - \lg(M[i_2, i_3]) - \dots - \lg(M[i_{k-1}, i_k]) - \lg(M[i_k, i_1]) < 0 \quad (11.3)$$

Сведохме задачата за намиране на арбитраж до задачата за намиране на отрицателен цикъл в пълен ориентиран тегловен граф, чиито върхове са валутите, а теглото на реброто (i, j) е $-\lg(M[i, j])$. Алгоритъмът на Bellman-Ford може да открива наличието на отрицателни цикли и с малка модификация може да намира отрицателен цикъл.

^aКурсовете са истинските курсове за тези валути на 18 май 2020 г.

11.1.6 Релаксация

Става дума за задачата за най-късите пътища в SSShP варианта. Даден е някакъв стартов връх s . Според [34, стр. 647], *релаксация* е техника за подобряване на *оценката* (*estimate* на английски), по отношение на произволен връх $u \in V(G)$, на теглото на най-къс път $s \rightsquigarrow u$. Както ще видим надолу, тази оценка е всъщност горна граница, която в алгоритмите, които ще разгледаме, започва с “ ∞ ” (тоест, нищо не знаем) и става все по-и по-добра, докато не стане точна (горна граница) и остане така до края на алгоритъма.

Тази оценка за връх u може да бъде записана като $u.d$. Алтернативно, може да има масив от оценките $d[1 \dots n]$ и оценката за u да е $d[u]$. Във всеки случай, това е реално число, което има смисъл на “най-ниската горна граница за минималното тегло на $s \overset{p}{\rightsquigarrow} u$, която можем да дадем въз основа на наличната информация”.

Както вече казахме, ние искаме не просто дължините на най-къси пътища, а и самите пътища, реализирани чрез масив на предшествия. На свой ред, тези стойности (идентификаторът на родителя в дървото) може да се записват като $u.\pi$, а може да има масив $\pi[1 \dots n]$, като $\pi[u]$ е родителят на u в дървото (родителят на корена е Nil).

Оценките се инициализират от следната процедура, която ще наричаме кратко “ISS”, което идва от Initialize-Single-Source.

ISS(G : ориентиран тегловен граф, s : връх в G)

```

1  foreach  $v \in V(G)$ 
2       $v.d \leftarrow \infty$ 
3       $v.\pi \leftarrow \text{Nil}$ 
4   $s.d \leftarrow 0$ 

```

Самата релаксация става така. Контекстът е, както очакваме, ориентиран тегловен граф G и тегловна функция w .

RELAX($(x, y) \in E(G)$)

```

1  if  $y.d > x.d + w((x, y))$ 
2       $y.d \leftarrow x.d + w((x, y))$ 
3       $y.\pi \leftarrow x$ 

```

Смисълът е напълно ясен: опитваме да намерим път с по-малко тегло $s \rightsquigarrow y$ от този, който е оптимален до момента. Стойността $y.d$ има смисъл на дължината на най-къс път $s \rightsquigarrow y$ според информацията до момента. Опитваме да намерим по-къс път до y , ползвайки x като предпоследен връх. Ако се окаже, че $y.d \leq x.d + w((x, y))$, то опитът е неуспешен и просто прескачаме. Ако обаче $y.d > x.d + w((x, y))$, наистина има такъв по-къс път—при допускането, че $x.d$ е смислена и π -стойностите реализират дърво на пътища—така че подменяме $y.d$ и $y.\pi$ по адекватен начин.

Изборът на името “релаксация” може би изглежда странен. Чрез тази примитивна процедура ние всъщност **затягаме** горната граница (оценката) за дължината на най-къс път, а не я отпусваме. Според [34, стр. 647], произходът на името “релаксация” е следният. Резултатът от RELAX((x, y)) може да бъде разглеждан като релаксиране на ограничението (constraint) $y.d \leq x.d + w((x, y))$; съгласно неравенството на триъгълника, това ограничение трябва да е в сила, ако $y.d = \delta(s, y)$ и $x.d = \delta(s, x)$. С други думи, ако е вярно, че $y.d \leq x.d + w((x, y))$, то няма защо да се притесняваме за удовлетворяването на това ограничение, така че можем да го релаксираме.

Свойства на релаксацията. В сила са следните твърдения. Всички те са от [34, стр. 650]

Лема 39: Неравенство на триъгълника

За всяко ребро $(x, y) \in E(G)$ е изпълнено

$$\delta(s, y) \leq \delta(s, x) + w((x, y)) \quad (11.4)$$

Доказателство: Ако няма път $s \rightsquigarrow x$, то $\delta(s, x) = \infty$ и дясната страна на (11.4) е ∞ . Неравенството остава в сила независимо от това, дали няма път $s \rightsquigarrow y$, което влече, че лявата страна е ∞ , или има път $s \rightsquigarrow y$, което влече, че лявата страна е или число, или $-\infty$.

Нека има път $s \rightsquigarrow x$. Ако съществува поне един $s \rightsquigarrow x$, който съдържа отрицателен цикъл, то $\delta(s, x) = -\infty$. Но тогава очевидно съществува $s \rightsquigarrow y$, който съдържа отрицателен цикъл, така че и $\delta(s, y) = -\infty$ и (11.4) е в сила.

Нека нито един $s \rightsquigarrow x$ не съдържа отрицателен цикъл. Тогава $-\infty < \delta(s, x) < \infty$. Нека p е най-къс път $s \rightsquigarrow x$. Тогава $w(p) = \delta(s, x)$. Ако съществува поне един $s \rightsquigarrow y$, който съдържа отрицателен цикъл, то $\delta(s, y) = -\infty$ и (11.4) е в сила.

Нека нито един $s \rightsquigarrow y$ не съдържа отрицателен цикъл. Нека e е реброто (x, y) . Нека $p' = p, e, y$. Забелязваме, че $s \rightsquigarrow y$. Нещо повече: $w(p') = w(p) + w((x, y)) = \delta(s, x) + w((x, y))$. Очевидно $\delta(s, y) \leq w(p')$. Тогава $\delta(s, y) \leq \delta(s, x) + w((x, y))$. $\square$

Лема 40: Горна граница и точна горна граница

Нека G е инициализиран с $ISS(G, s)$. Тогава, за всеки връх $v \in V(G)$, $v.d \geq \delta(s, v)$, след произволна редица от релаксации на ребрата на G . Нещо повече, при достигане на равенство, то не се променя след произволна редица от релаксации на ребрата на G .

Сравнете Лема 40 с Лема 35.

Доказателство: Ще докажем, че $v.d \geq \delta(s, v)$ за всеки връх v , след произволна редица от релаксации на ребра, по индукция по броя на релаксациите.

Базата е за нула релаксации, тоест, d -стойностите са както са генерирани от ISS , а именно $s.d = 0$ и $v.d = \infty$, ако $v \neq s$. Вярно е, че $s.d \geq \delta(s, s)$, защото $s.d = 0$ заради ISS , а $\delta(s, s) = 0$ или $\delta(s, s) = -\infty$ в зависимост от това, дали s не е, или е, върху отрицателен цикъл. И в двата случая е вярно, че $s.d \geq \delta(s, s)$. Нека $v \neq s$. Вярно е, че $v.d \geq \delta(s, v)$, защото $v.d = \infty$ заради ISS , а $\delta(s, v) \in \mathbb{R} \cup \{-\infty, \infty\}$.

Да допуснем, че твърдението е вярно след някаква произволна серия от релаксации. Разглеждаме $RELAX((u, v))$, за някое $(u, v) \in E(G)$, веднага след тази серия. Единствената d -стойност, която може да се промени от $RELAX((u, v))$, е $v.d$. Ако тя изобщо се промени, промяната ще е такава:

$$\begin{aligned} v.d &= u.d + w((u, v)) \quad // \text{ защото } RELAX \text{ работи така} \\ &\geq \delta(s, u) + w((u, v)) \quad // \text{ от индуктивното предположение за } u \\ &\geq \delta(s, v) \quad // \text{ от Лема 39} \end{aligned}$$

За да се убедим, че, ако веднъж се постигне равенство $v.d = \delta(s, v)$, то остава в сила във всеки момент след това, да съобразим, че

- няма как да се получи $v.d < \delta(s, v)$ заради вече доказанния факт, че $v.d \geq \delta(s, v)$ винаги, и
- няма как да се получи $v.d > \delta(s, v)$ след $v.d = \delta(s, v)$, защото $RELAX$ никога не увеличава d -стойността на края на релаксираното ребро. $\square$

Лема 41: При липса на пътища

Нека v е връх, такъв че няма $s \rightsquigarrow v$. Тогава, след произволна редица от $RELAX$ върху ребрата на графа е вярно, че $v.d = \infty$.

Доказателство: Съгласно Лема 40, $v.d \geq \delta(s, v)$, след произволна редица от релаксации на ребрата. Но $\delta(s, v) = \infty$ съгласно Определение 93. $\square$

Лема 42: Конвергенция

Нека p е най-къс път $s \rightsquigarrow v$ и $u \neq v$ е предпоследният връх в p . Ако $u.d = \delta(s, u)$ в някакъв момент преди релаксирането на реброто (u, v) , то $v.d = \delta(s, v)$ във всеки момент след това релаксиране.

Доказателство: Съгласно Лема 40, ако веднъж се постигне равенство $u.d = \delta(s, u)$, то се запазва нататък. В частност, след релаксирането на (u, v) , в сила е

$$\begin{aligned} v.d &\leq u.d + w((u, v)) && // \text{очевидно от кода на RELAX} \\ &= \delta(s, u) + w((u, v)) && // \text{както вече казахме} \\ &= \delta(s, v) && // \text{Теорема 96} \end{aligned}$$

Покажахме, че $v.d \leq \delta(s, v)$ след релаксирането на (u, v) . Но $v.d \geq \delta(s, v)$ съгласно Лема 40). Тогава $v.d = \delta(s, v)$ и това равенство остава в сила до края (пак Лема 40). $\square$

11.2 Алгоритъмът на Dijkstra в два варианта

Ще разгледаме един от най-популярните алгоритми за намиране на най-къс път в ориентиран граф: алгоритъмът на Dijkstra. Нека тегловната функция е $w : E \rightarrow \mathbb{R}^+$. Изискването да няма отрицателни тегла е съществено! Алгоритъмът на Dijkstra **не работи коректно**, ако има отрицателни тегла, дори да няма отрицателни цикли. Намерете сами малък пример за тегловен граф с точно едно ребро с отрицателно тегло, в който няма отрицателни цикли и въпреки това алгоритъмът на Dijkstra не работи коректно върху този граф.

Ще разгледаме два варианта на алгоритъма на Dijkstra: базов и изтънчен. Базовият вариант намира следващия връх x с последователно търсене, а изтънченият вариант ползва АТД приоритетна опашка, от която бързо изважда следващия връх x . Забележете аналогията с алгоритъма на Prim, който също беше имплементиран в два варианта—МПД PRIM1 на стр. 498 и МПД PRIM2 на стр. 500—с подобна разлика между тях.

Допускаме БОО, че целият G е достижим от s .

SSSNP DIJKSTRA1(G : ориентиран тегловен граф, w : тегл. ф-я върху G , s : стартов връх)

```

1  ISS( $G, s$ )
2   $D \leftarrow \emptyset$ 
3  while  $V \setminus D \neq \emptyset$  do
4      избери  $x \in V \setminus D$ , такъв че  $x.d$  е минимална
5       $D \leftarrow D \cup \{x\}$ 
6      foreach  $y \in \text{adj}[x]$ 
7          RELAX( $(x, y)$ )
```

SSSNP DIJKSTRA2(G : ориентиран тегловен граф, w : тегл. ф-я върху G , s : стартов връх)

```

1  ISS( $G, s$ )
2   $D \leftarrow \emptyset$ 
3  създай празна приоритетна min опашка  $Q$ 
4   $Q \leftarrow V$ 
5  while not isempty( $Q$ ) do
6       $x \leftarrow \text{EXTRACT-MIN}(Q)$ 
7       $D \leftarrow D \cup \{x\}$ 
8      foreach  $y \in \text{adj}[x]$ 
9          RELAX( $(x, y)$ )
```

Допълнение 54: За подобие между МПД PRIM2 и SSSH P DIJKSTRA2.

Да разгледаме псевдокодовете на изтънчените варианти на алгоритмите на Prim и Dijkstra един до друг. Има следните разлики между показаните тук псевдокодове и тези, които вече видяхме съответно на стр. 500 и на предната страница.

- За удобство тук описваме Dijkstra подробно, без ISS и RELAX.
- Множеството D не се ползва истински от алгоритъма на Dijkstra в изтънчения вариант. То присъства в псевдокода (редове 2 и 7) за удобство в доказателството за коректност.
- На ред 10 има проверка дали $y \in Q$. Това не е съществено за коректността, въпреки че на практика има смисъл. Върховете извън Q са точно върховете от досега изграденото дърво. Нас ни интересуват само ключовете на върховете от Q , понеже това са върховете извън дървото и алгоритъмът “решава” кой от тях да добави към дървото въз основа на ключовете им. Ако $y \notin Q$, няма смисъл да променяме ключа на y , защото този връх е вече в дървото и неговият ключ няма да се използва повече за променяне (намаляване) на ключове на върхове извън дървото.

Поради това тук на ред 10 не правим проверка дали $y \in Q$.

МПД PRIM2(G, w, s)

```

1  foreach  $v \in V(G)$ 
2       $v.key \leftarrow \infty$ 
3       $\pi[v] \leftarrow Nil$ 
4   $s.key \leftarrow 0$ 
5  празна приор. min опашка  $Q$ 
6   $Q \leftarrow V$ 
7  while not isempty( $Q$ ) do
8       $x \leftarrow EXTRACT-MIN(Q)$ 
9      foreach  $y \in adj[x]$ 
10         if  $y.key > w((x, y))$ 
11              $y.key \leftarrow w((x, y))$ 
12          $\pi[y] \leftarrow x$ 
```

SSSH P DIJKSTRA2(G, w, s)

```

1  foreach  $v \in V(G)$ 
2       $v.d \leftarrow \infty$ 
3       $v.\pi \leftarrow Nil$ 
4   $s.d \leftarrow 0$ 
5  празна приор. min опашка  $Q$ 
6   $Q \leftarrow V$ 
7  while not isempty( $Q$ ) do
8       $x \leftarrow EXTRACT-MIN(Q)$ 
9      foreach  $y \in adj[x]$ 
10         if  $y.d > x.d + w((x, y))$ 
11              $y.d \leftarrow x.d + w((x, y))$ 
12          $y.\pi \leftarrow x$ 
```

Единствената съществена разлика между двата алгоритъма е числото, с което се сравнява ключът на y на ред 10. В алгоритъма на Prim това число е теглото на реброто (x, y) . В алгоритъма на Dijkstra това число е **сумата** от теглото на (x, y) и ключа на x . Ерго, алгоритъмът на Prim взема своите решения въз основа на предвалително фиксирани числа – теглата на ребрата не се менят по време на работата на алгоритъма. Докато алгоритъмът на Dijkstra взема решенията въз основа на числа, които (може да) се менят и по-точно, да намаляват, от итерация на итерация.

Коректност и сложност. Въпреки повърхностната прилика между алгоритъма на Prim и алгоритъма на Dijkstra, вторият има значително по-трудно доказателство за коректност, защото в неговия контекст няма аналог на Теорема 93. Ще докажем коректността само за

олекотения вариант на задачата; тоест, само за d -стойностите. Приемаме, че от това доказателство лесно следва и коректността на π -стойностите, които реализират дървото на най-късите пътища (тежкият вариант на задачата).

Текущото допускане е, че целият G е достижим от s . При това положение смятаме за очевидно, че при всяко достигане на **while**-а, ако $V \setminus D$ не е празно, в него има поне един връх, чиято d -стойност е по-малка от безкрайност. Ерго, не се съмняваме, че всеки връх v ще бъде обходен в смисъл, че при някоя итерация на **while**-а, x ще получи стойност v и по този начин v ще “влезе” в D . Това, което остава да покажем, е, че в този момент стойността $v.d$ е точно равна на разстоянието от s до v .

Нека DIJKSTRA означава кой да е от алгоритмите SSSHP DIJKSTRA1 и SSSHP DIJKSTRA2.

Теорема 97: Коректността на алгоритъма на Dijkstra

За всеки ориентиран граф G , за всяка тегловна функция w над него, за всеки избор на стартов връх s е вярно, че след термилирането на DIJKSTRA(G, w, s):

$$\forall v \in V(G) : v.d = \delta(s, v)$$

Доказателство: Ще докажем, че за всеки връх $v \in V(G)$, в момента, в който v влиза в D е вярно, че $v.d = \delta(s, v)$. Това влече верността на теоремата заради Лема 40.

Да допуснем противното: за поне един $v \in V(G)$ е вярно, че $v.d \neq \delta(s, v)$ в момента, в който v влиза в D . Нещо повече: нека v е **първият** връх, който “влиза” в D с d -стойност, различна от минималното тегло на път от s до него. От Лема 40 знаем, че ако графът е инициализиран с ISS(G, s), то след произволна редица от релаксации е вярно, че $\forall v \in V(G) : v.d \geq \delta(s, v)$. Тогава допускането става $v.d > \delta(s, v)$. Очевидно $v \neq s$, защото $s.d = 0$ и, при положителни тегла, $\delta(s, s) = 0$.

Разглеждаме тази итерация, по време на която в D “влиза” връх v . В началото на въпросната итерация, нека t е моментът точно преди v да “влезе” в D . В момента t е вярно, че $v.d > \delta(s, v)$; ако допуснем, че $v.d = \delta(s, v)$ в момента t , то $v.d > \delta(s, v)$ би било вярно и при термилирането на алгоритъма съгласно Лема 40.

Щом $v \neq s$, то $D \neq \emptyset$ в момента t . По допускане, съществува $s \rightsquigarrow v$. Разглеждаме най-къс $s \xrightarrow{p} v$. Щом теглата са положителни, такъв съществува.

В момента t е вярно, че първият връх на p (това е s) е в D , а последният връх (това е v) не е в D . Дефинираме u като най-близкият до s връх в p , който в момента t не е в D . След малко ще видим, че $u \neq v$. Нека върхът преди u в p се казва a ; възможно е $a = s$, възможно е и $a \neq s$, важното е, че такъв a съществува. Забележете, че в момента t , всички върхове на p от s до a включително са в D , а u е първият връх в p , в посока s към v , който не е в D :

$$p = \underbrace{s \dots a}_{\text{в } D} \underbrace{u}_{\text{не е в } D} \dots \underbrace{v}_{\text{не е в } D}$$

Ще докажем, че $u.d = \delta(s, u)$ в момента t . Знаем, че в момента t , връх a се намира в D . Връх a е бил сложен в D на някоя предишна итерация, в някакъв момент $t' < t$. Следва, че в момента t' е изпълнено $a.d = \delta(s, a)$, тъй като v е първият връх, върху който алгоритъма греши. Съгласно Лема 40, във всеки момент след t' продължава да е вярно, че $a.d = \delta(s, a)$. Но на итерацията, в която a е сложен в D , реброто (a, u) е било релаксирано. За да се убедите в това, вижте псевдокода. Прилагаме Лема 42 и заключаваме, че

$$u.d = \delta(s, u) \tag{11.5}$$

е изпълнено във всеки момент след това релаксиране, включително и в момента t . От това следва, че $u \neq v$, защото за v знаем, че $v.d > \delta(s, v)$ в момента t .

Съгласно Теорема 96, $\delta(s, u) < \delta(s, v)$, тъй като теглата са положителни, а s, u и v са наредени в този ред върху най-къс път от s до v . По допускане, $\delta(s, v) < v.d$, така че

$$\delta(s, u) < \delta(s, v) < v.d \quad (11.6)$$

От (11.5) и (11.6) заключаваме, че в момента t :

$$u.d = \delta(s, u) < \delta(s, v) < v.d \quad (11.7)$$

Ключовото наблюдение е, че и u , и v са върхове **извън** D в момента t и причината алгоритъмът да избере v , а не u , като връх, който да влезе в D , е една единствена:

$$v.d \leq u.d \quad (11.8)$$

От (11.7) и (11.8) заключаваме, че в момента t :

$$v.d \leq u.d = \delta(s, u) < \delta(s, v) < v.d \quad (11.9)$$

Това, разбира се, е невъзможно. ⚡

□

Сложността по време е същата, в асимптотичния смисъл, като на съответните имплементации на Prim. Базовата имплементация има сложност $\Theta(n^2)$ в най-лошия случай, а изтънчената има сложност $\Theta((n + m) \lg n)$ в най-лошия случай, ако Q е имплементирана с двоична пирамида.

Свойство на алгоритъма на Dijkstra: върховете влизат в дървото в реда на разстоянията от s . Важно свойство на алгоритъма на Dijkstra е, че върховете влизат в дървото на най-късите пътища реда на своите разстояния от s . Както знаем от Теорема 97, разстоянията са равни на d -стойностите, така че ще докажем твърдението за d -стойностите.

Лема 43

При всяко достигане на ред 3 на SSSHP DIJKSTRA1, за всеки връх $u \in D$ и всеки връх $v \in V \setminus D$ е вярно, че $u.d \leq v.d$.

Доказателство: Както в Лема 35, и тук няма смисъл да формулираме инвариант, понеже не се интересуваме от крайното състояние на нещата при терминирането, а разглеждаме произволен момент от изпълнението, така че формулировката на лемата е инвариантът.

Базата на индукцията е първото достигане на ред 3. В този момент $D = \emptyset$ и твърдението е вярно в празния смисъл (vacuously). ✓

Разглеждаме произволно достигане на ред 3, което не е последното. Да означим с D' множеството D в този момент. Да допуснем, че за всеки връх $u \in D'$ и всеки връх $v \in V \setminus D'$ е вярно, че $u.d \leq v.d$. На ред 4 избираме x като връх с минимална d -стойност измежду върховете на $V \setminus D'$; тоест,

$$\forall v \in (V \setminus D') \setminus \{x\} : x.d \leq v.d$$

Тогава за този момент е изпълнено

$$\forall u \in D' \quad \forall v \in (V \setminus D') \setminus \{x\} : u.d \leq x.d \leq v.d \quad (11.10)$$

На ред 5, x става елемент на D . Да означим с D'' множеството D в този момент. Очевидно $D'' = D' \cup \{x\}$ и $V \setminus D'' = (V \setminus D') \setminus \{x\}$. От това и (11.10) следва, че

$$\forall u \in D'' \quad \forall v \in V \setminus D'' : u.d \leq v.d$$

След това се изпълнява вътрешният цикъл (редове 6–7).

- Ако връх y е в D'' , то $y.d$ не се променя от изпълнението на $\text{RELAX}(x, y)$, защото не е възможно $y.d$ да е по-голямо от $x.d + w((x, y))$. На свой ред, това е така, понеже в противен случай $y.d$ щеше да бъде променено от $\text{RELAX}(x, y)$, а от доказателството на Теорема 97 знаем, че за всеки връх, при влизането в D , неговата d -стойността е равна на претегленото разстояние от s , и тази стойност не се мени повече от алгоритъма (Лема 40).
- Ако връх y е във $V \setminus D''$, то $y.d$ потенциално може да намалее при изпълнението на $\text{RELAX}(x, y)$ и да стане $x.d + w((x, y))$. Но тъй като $w((x, y)) > 0$, предвид (11.10), продължава да е вярно, че за всеки връх $u \in D''$ и всеки връх $v \in V \setminus D''$ е вярно, че $u.d \leq v.d$.

Заклучаваме, че при следващото достигане на ред 3, твърдението е в сила. □

Следствие 30 по отношение на алгоритъма на Dijkstra е аналога на Следствие 27 по отношение на BFS.

Следствие 30

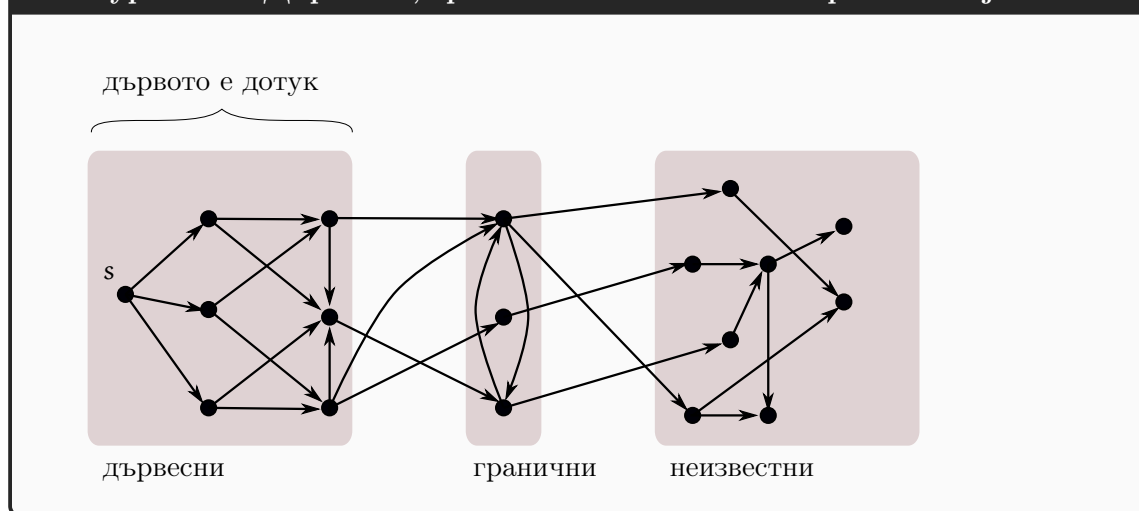
За всеки два различни върха u и v , ако по време на работата на алгоритъма на Dijkstra u влиза в D преди v , то $u.d \leq v.d$.

Да разгледаме едно класифициране на върховете на графа, което възниква по време на работата на алгоритъма на Dijkstra. Да кажем, че при всяко достигане на ред 3 по време на работата на SSSHP DIJKSTRA1, T е досега построеното дърво; неговите върхове са точно върховете в D . Тогава

- върховете от $V(T)$ са *дървесните върхове* (на английски, *tree vertices*),
- върховете от $N(V(T))$ са *границните върхове* (на английски, *fringe vertices*),
- върховете от $V \setminus N[V(T)]$ са *неизвестните върхове* (на английски, *unseen vertices*).

Забележете, че това класифициране е буквално същото като класифицирането на върховете по време на работата на алгоритъма на Prim (Подсекция 10.3.1). Някои от тези множества може да са празни, примерно преди първата итерация има само неизвестни върхове, а към края на алгоритъма неизвестни върхове няма (ако всички върхове са достижими от s), но общата картина е точно тази, която е показана на Фигура 11.1: има частично построено дърво T , което е кореново дърво с корен s , границата се състои от децата на дървесните върхове извън дървото, а отвъд границата са неизвестните върхове.

Фигура 11.1 : Дървесни, гранични и неизвестни върхове в Dijkstra.



Забележете приликата между Фигура 11.1, отнасяща се до алгоритъма на Dijkstra, Фигура 10.1, отнасяща се до алгоритъма на Prim и Фигура 8.1, отнасяща се до BFS. Това, че графът на Фигура 10.1 е неориентиран (става дума за МПД), а на Фигура 11.1 и Фигура 8.1 графите са ориентирани, е без значение. И в трите случая има върхове, с които сме приключили (дървесни), такива, които още не сме видели (неизвестни), и граница между тези два вида.

Когато не всички върхове са достижими от стартовия връх. Ако не е дадено, че всеки връх на G е достижим от s , в базовия вариант SSSHP DIJKSTRA1 може да модифицираме ред 3 така:

```
while  $\exists u \in V \setminus D : u.d < \infty$ 
```

а в изтънчения вариант SSSHP DIJKSTRA2 може да модифицираме ред 5 така:

```
while not isempty(Q) and minimum(Q) <  $\infty$ 
```

По този начин се гарантира, че не се опитваме да слагаме в D върхове с d -стойност безкрайност. От Лема 41 следва, че върховете, които не са достижими от s , имат d -стойности безкрайност в края на алгоритъма. Нещо повече: върховете, които не са достижими от s , са точно тези, които имат d -стойности безкрайност в края на алгоритъма – приехме за очевидно, че всеки достижим връх получава крайна d -стойност, влизайки в D .

И така, ако на дадено достигане на **while**-а множеството $V \setminus D$ съдържа само върхове с d -стойност безкрайност, то това са точно върховете, недостижими от s , и в този момент алгоритъмът трябва да терминира. Да слагаме в D върхове, недостижими от s , е груба грешка.

11.3 Най-къси пътища в тегловни дагове

Ако G е тегловен даг, може да използваме по-ефикасен алгоритъм от алгоритъма на Dijkstra за задачата SSShP.

DAG SSSHP(G : даг, w : тегловна функция върху G , s : стартов връх)

```
1  TOPOSORT( $G$ )
```

```

2  ISS(G, s)
3  foreach  $x \in V(G)$  в топологичната наредба
4      foreach  $y \in \text{adj}[x]$ 
5          RELAX( $(x, y)$ )

```

Ще покажем коректността на този алгоритъм. С учебна цел ще направим две доказателства: първото е “традиционното” доказателство (примерно, [34, Theorem 24.5, стр. 256–257]), което е по индукция по топологичната наредба на върховете отляво надясно, а второто е с инвариант на цикъла. Първото е по-кратко, а второ е пример за нетривиално доказателство с инвариант.

Теорема 98: Коректността на алгоритъма DAG SSSHP

За всеки даг G , за всяка тегловна функция w над него, за всеки избор на стартов връх s е вярно, че след термилирането на DAG SSSHP(G, w, s):

$$\forall v \in V(G) : v.d = \delta(s, v) \quad (11.11)$$

Първо доказателство: Нека T е резултатът от топо-сортирането на ред 1. Първо забелязваме, че (11.11) е в сила за върховете вляво от s в T : от една страна, за всеки връх v вляво от s е вярно, че $\delta(s, v) = \infty$, понеже няма път $s \rightsquigarrow v$ в дага, а от друга страна, във всеки момент от работата на алгоритъма е вярно, че $v.d = \infty$ заради Лема 41. ✓

Сега забелязваме, че (11.11) е вярно за s . Наистина, $\delta(s, s) = 0$ при липса на отрицателни цикли, а $s.d$ става 0 след изпълнението на ред 2 и остава така до края на алгоритъма (Лема 40). ✓

И накрая разглеждаме произволен връх v вдясно от s в T . Важно наблюдение е, че вътрешните върхове на всеки път от $\text{Paths}(s, v)$ се намират строго между s и v в топологичната наредба. Нека

$$U = \{u \in V(G) \mid \exists p \in \text{Paths}(s, v) : u \text{ е предпоследният връх на } p\}$$

Забележете, че е възможно $s \in U$. Очевидно v е достижим от s тстк $U \neq \emptyset$. Нека v' е върхът непосредствено вляво от v в T . Да разгледаме всички викания на RELAX, когато x взема стойностите от $\{s, \dots, v'\}$. Очевидно е, че по отношение на връх v , RELAX реализира

$$v.d = \min_{u \in U} \{u.d + w((u, v))\} \quad (11.12)$$

От друга страна обаче, очевидно

$$\delta(s, v) = \min_{u \in U} \{\delta(s, u) + w((u, v))\} \quad (11.13)$$

При допускането, че $\forall u \in U : u.d = \delta(s, u)$, от (11.12) и (11.13) заключаваме, че $v.d = \delta(s, v)$ в момента, в който x стане v . Това остава вярно дори когато $U = \emptyset$; тоест, когато v не е достижим от s . В такъв случай както $\delta(s, v) = \infty$, така и $v.d = \infty$, съответно от (11.12) и (11.13) при $U = \emptyset$, предвид това, че идентитетът на минимума е ∞ . ✓

Второ доказателство: По отношение на цикъла на редове 3–5, нека за всеки $u \in V(G) \setminus \{s\}$, “ $\mathcal{P}(u)$ ” означава множеството от пътищата от s до u , чиито **предпоследен** връх е вляво от x . Забележете, че това има смисъл само в контекста на вече генерираната на ред 1 топо-наредба. Също така забележете, че по отношение на един и същи връх v , за различни достигания на

ред 3, $\mathcal{P}(v)$ може да е различно множество, защото се дефинира по отношение на променливата x . Следното твърдение е инвариант за цикъла на редове 3–5.

Инвариант

При всяко достигане на ред 3, за всеки $v \in V(G) \setminus \{s\}$ е вярно, че $v.d$ съдържа дължината на най-къс път от $\mathcal{P}(v)$.

База. Когато изпълнението е на ред 3 за първи път, в сила е $\mathcal{P}(v) = \emptyset$ за всеки $v \in V(G) \setminus \{s\}$. Това е вярно независимо от това дали s е първият връх в топо-наредбата или не, понеже върхове вляво от x няма. От друга страна, $v.d = \infty$ за всеки $v \in V(G) \setminus \{s\}$. Базата “излиза”. ✓

Поддръжка. Да допуснем, че твърдението е вярно за някое достигане на ред 3, което не е последното. Да разгледаме преминаването през тялото на цикъла. Помним, че всички деца на x се намират вдясно от x ; това е свойство на топо-наредбата.

Съгласно индуктивното предположение, текущото $x.d$ е дължината на най-къс път от s до x , чийто предпоследен връх е вляво от x .

- Първо да допуснем, че такъв път няма. Тоест, $\mathcal{P}(x) = \emptyset$. В този случай връх x е недостижим от връх s , понеже предпоследният връх в $s \rightsquigarrow x$ не може да е нито x , нито вдясно от x . Шом x е недостижим от s , то x не може да се явява предпоследен връх по път от s до никой връх $y \in \text{adj}[x]$. Тогава за всеки $y \in \text{adj}[x]$ е вярно, че дължината на най-къс път от s до y с предпоследен връх вляво от x е равна на дължината на най-къс път от s до y с предпоследен връх вляво от, или съвпадащ, с x . Тогава, съгласно индуктивното предположение, за всеки $y \in \text{adj}[x]$ е вярно, че $y.d$ **преди** изпълнението на тялото на цикъла съдържа дължината на най-къс път от s до y с предпоследен връх вляво от, или съвпадащ, с x .

Съгласно индуктивното предположение, в този случай $x.d = \infty$. От това веднага следва, че при изпълнението на тялото на цикъла, $\text{RELAX}(x, y)$ не води до промяна на никое $y.d$. Но тогава **след** изпълнението на тялото на цикъла, за всеки $y \in \text{adj}[x]$ е вярно, че $y.d$ съдържа дължината на най-къс път от s до y с предпоследен връх вляво от, или съвпадащ, с x . Тъй като тялото на цикъла не може да промени други d -стойности освен тези на върховете от $\text{adj}[x]$, имаме право да кажем, че **след** изпълнението на тялото на цикъла, за всеки $v \in V(G) \setminus \{s\}$ е вярно, че $v.d$ съдържа дължината на най-къс път от s до v с предпоследен връх вляво от, или съвпадащ с, x .

След това променливата x взема (ред 3) стойността на следващия връх в топо-наредбата. По отношение на новата стойност на x , за всеки $v \in V(G) \setminus \{s\}$ е вярно, че $v.d$ съдържа дължината на най-къс път от $\mathcal{P}(v)$. Инвариантът се запазва в този случай.

- Сега да допуснем, че такъв път има. Тоест, $\mathcal{P}(x) \neq \emptyset$. В този случай връх x е достижим от връх s , така че x се явява предпоследен връх по път от s до всеки връх $y \in \text{adj}[x]$. Тогава за всеки $y \in \text{adj}[x]$ е вярно, че **преди** изпълнението на тялото на цикъла, дължината на най-къс път от s до y с предпоследен връх вляво от, или съвпадащ с, x е минимума на следните две:
 - ♦ дължината на най-къс път от s до y с предпоследен връх вляво от x ,
 - ♦ дължината на най-къс път от s до y с предпоследен връх x .

Но тогава **след** изпълнението на тялото на цикъла, за всяко $y \in \text{adj}[x]$ е вярно, че $y.d$ съдържа дължината на най-къс път от s до y с предпоследен връх вляво от, или

съвпадащ с, x . За да съобразим, че е така, достатъчно е да имаме предвид, че съгласно индуктивното предположение, $y.d$ преди изпълнението на тялото на цикъла съдържа именно дължината на най-къс път от s до y с предпоследен връх вляво от x , а **по време** на изпълнението на тялото на цикъла, RELAX на ред 5 прави така, че новата стойност на $y.d$ е минимума от старото $y.d$ и $x.d + w(x, y)$, като последното е дължината на най-къс път от s до y с предпоследен връх x .

Щом **след** изпълнението на тялото на цикъла, за всяко $y \in \text{adj}[x]$ е вярно, че $y.d$ съдържа дължината на най-къс път от s до y с предпоследен връх вляво от, или съвпадащ с, x , и освен това изпълнението на тялото на цикъла не засяга други d -стойности освен тези на върховете от $\text{adj}[x]$, имаме право да кажем, че **след** изпълнението на тялото на цикъла, за всеки $v \in V(G) \setminus \{s\}$ е вярно, че $v.d$ съдържа дължината на най-къс път от s до v с предпоследен връх вляво от, или съвпадащ с, x .

След това променливата x взема (ред 3) стойността на следващия връх в топо-наредбата. По отношение на новата стойност на x , за всеки $v \in V(G) \setminus \{s\}$ е вярно, че $v.d$ съдържа дължината на най-къс път от $P(v)$. Инвариантът се запазва в този случай.

Терминация. При последното достигане на ред 3, x съдържа стойността на най-десния връх в топо-наредбата. Тогава за всеки $v \in V(G)$ е вярно, че $P(v)$ е точно множеството от пътищата от s до v , тъй като предпоследният връх в който да е от тези пътища трябва да е вляво от най-десния връх. Тогава инвариантът казва: за всеки $v \in V(G) \setminus \{s\}$, $v.d$ съдържа дължината на най-къс път от s до v . $\square$

Да разгледаме сложността по време. Тя е $\Theta(n + m)$, защото и топологическото сортиране, и вложеният **for**-цикъл имат такава сложност по време.

Предимствата на DAG SSSHP пред алгоритъма на Dijkstra върху даг са следните:

- DAG SSSHP е по-бърз, както на практика, така и в асимптотичния смисъл. Сложността $\Theta(n + m)$ е оптимална, в асимптотичния смисъл, и е по-добра от сложностите на DIJKSTRA, както в базовия вариант, така и в изтънчения вариант, дори приоритетната опашка да е реализирана с пирамида на Fibonacci.
- DAG SSSHP работи без проблеми дори ако има отрицателни тегла. В даговете няма цикли поначало, така че няма опасност от отрицателен цикъл. Както се каза, DIJKSTRA не е използваем при отрицателни тегла. BELLMAN-FORD “се справя” с отрицателни тегла, но той е драстично по-бавен от DAG SSSHP.
- DAG SSSHP може да намира и най-дълги пътища с една тривиална модификация: обръщане на посоката на неравенството в RELAX. Забележете, че при даговете е вярно, че най-дълъг път се състои от най-дълги подпътища (аналогът на Теорема 96): ако заменим подпът с по-дълъг подпът, ще получим по-дълъг път без опасност от повтаряне на върхове. За да се убедим в това, достатъчно е да разгледаме дага в контекста на топологично сортиране.

От друга страна, DIJKSTRA не може да бъде трансформиран в алгоритъм за намиране на най-дълги пътища с просто обръщане на посоката на неравенството в RELAX и на $\min$ в $\max$ по отношение на избора на нов връх, който да влезе в D (в изтънчената реализация това би било смяна на типа на приоритетната опашка от $\min$ в $\max$), дори графът да е даг.

11.4 Алгоритъмът на Bellman-Ford

Този алгоритъм намира най-къси пътища в тегловни графи, в които може да има отрицателни тегла, стига да няма отрицателни цикли. Ако има поне един отрицателен цикъл, алгоритъмът установява това и връща съответна индикация FALSE; в такъв случай намерените дължини на най-къси пътища следва да се игнорират, защото поне една от тях е “фалшива”. Ако върне TRUE, то отрицателни цикли няма и намерените дължини на най-къси пътища са “истински”.

С малка модификация, алгоритъмът може да върне и един отрицателен цикъл, в случай че се установи наличие на отрицателни цикли.

BELLMAN-FORD(G : ориентиран тегловен граф, w : тегловна функция върху G , s : стартов връх)

```

1  ISS( $G, s$ )
2  for  $i \leftarrow 1$  to  $n - 1$ 
3      foreach  $(x, y) \in E(G)$ 
4          RELAX( $(x, y)$ )
5  foreach  $(x, y) \in E(G)$ 
6      if  $y.d > x.d + w((x, y))$ 
7          return FALSE
8  return TRUE
```

Коректност и сложност. Самият алгоритъм е на редове 1–4, което го прави изключително компактен. Кодът на редове 5–7 е проверка за наличие на отрицателен цикъл.

Ще направим повърхностно доказателство за коректност. БОО, нека целият G е достижим от s . Първо да допуснем, че отрицателни цикли няма. Тогава дърво на най-късите пътища с корен s е добре дефинирано. Нека T е такова дърво. T има височина най-много $n - 1$, което обяснява и “ $n - 1$ ” на ред 2. Инвариант на **for**-цикъла на редове 2–4 е, при всяко достигане на ред 2, за всеки връх $v \in V(G)$, който се намира на ниво $i - 1$ в T (ниво е множеството от върховете в T на едно и също разстояние от корена, **като брой ребра**), $v.d = \delta(s, v)$. Базата е за $i = 1$, тоест, $i - 1 = 0$, тоест, върховете от нулевото ниво, тоест, само връх s ; очевидно твърдението е вярно за връх s . В индуктивната стъпка, ако е вярно за върховете от ниво k , прилагаме Лема 42 и заключаваме, че е вярно и за върховете от ниво $k + 1$.

Лесно се вижда, че ако G няма отрицателни цикли, така че най-къси пътища са дефинирани от s до всеки връх, ако продължим да изпълняваме цикъла на редове 3–4 и след $i = n - 1$, няма да има промяна в нито една d -стойност на връх. Иначе казано, нито едно ребро не може да бъде релаксирано с промяна (намаляване) на d -стойността на крайния му връх, понеже се е стигнало до състояние $\forall v \in V(G) : v.d = \delta(s, v)$ и това е окончателно съгласно Лема 40. Това показва и коректността на проверката за отрицателни цикли на редове 5–7: ако поне едно ребро (x, y) е “релаксируемо”, което е същото като булевото условие на ред 6 да е истина, със сигурност има отрицателен цикъл. В такъв случай, колкото и пъти да продължим да изпълняваме редове 3–4 отвъд $i = n - 1$, ще получаваме “подобрения” на d -стойността на някой връх, за който е вярно, че от s до него има път, съдържащ отрицателен цикъл. $\square$

Сложността по време, очевидно, е $\Theta(nm)$, което означава, че BELLMAN-FORD е кубичен алгоритъм в най-лошия случай.

Част IV

Динамично програмиране

Лекция 12

Същност на схемата Динамично Програмиране и примери за алгоритми по нея.

Резюме: Започваме с прости примери като задачата на опашката пред касата, пресмятане на числа на Fibonacci и най-къси пътища в тегловни графи във варианта APShP. Придобивайки интуиция от примерите, разглеждаме основата на алгоритмичната схема **Динамично Програмиране**. Продължаваме с други примери за нейното успешно прилагане: задачи за ефикасно броене на комбинаторни структури, задачи за оптимално скобуване, задачи за оптимални подмножества, задачи за оптимални поредици, **NP**-трудни задачи върху дървета. Въвеждаме понятието “псевдополиномиална сложност” в контекста на задачите върху множества. Споменаваме мемоизацията като алтернативен подход на динамичното програмиране. Накрая стигаме до ограниченията на динамичното програмиране: неефикасен алгоритъм, построен по схемата **Динамично Програмиране**, за **NP**-трудната задача за намиране на най-дълъг път в граф.

12.1 Въведение с примери

Алгоритмичната схема **Динамично Програмиране** е много важна и полезна, но е и неестествена за неопитните дизайнери на алгоритми. Затова изложението започва с няколко примера за ефикасни алгоритми, конструирани чрез **Динамично Програмиране**. Чак след като придобием достатъчно интуиция от тези примери, ще разгледаме теоретичната основа в Секция 12.2. Мнението на автора е, че да се започва с теоретични обяснения преди разглеждането на достатъчно примери е безполезно.

12.1.1 Задачата за опашката пред касата

Дадена е каса за билети. Пред касата има опашка от n човека $h_1, h_2, \dots, h_n$, в този ред. Известно е, че за $1 \leq i \leq n$, h_i би се забавил(а) d_i минути на касата, пазарувайки билетите си. Известно е, че за $1 \leq i \leq n-1$, h_i и h_{i+1} , очевидно съседни в опашката, може да се комбинират и да си напазаруват билетите заедно, като това ще отнеме c_i минути. При дадени времена на индивидуално пазаруване на билети $d_1, d_2, \dots, d_n$ и времена на комбинирано пазаруване $c_1, c_2, \dots, c_{n-1}$, олекотеният вариант на задачата е да се пресметне минималното време, за което цялата опашка може да си купи билетите, а тежкият вариант е да се определи кой с кого да се комбинира, за да мине опашката за минимално време. “Олекотеният вариант” и

“тежкия вариант” са понятия, които въведохме на стр. 13.

Ако нямаше възможност за комбиниране на съседни в опашката, отговорът би бил просто $\sum_{i=1}^n d_i$. Но при дадените възможности за комбиниране може да има по-бърз начин. Примерно, ако $c_1 < d_1 + d_2$, има смисъл h_1 и h_2 да се комбинират, наместо да минават поотделно. Забележете, че не всички комбинирания са възможни: примерно, ако h_1 и h_2 са комбинирани заедно през касата, възможността h_2 да се комбинира с h_3 отпада; и обратно, ако h_2 и h_3 са комбинирани, h_1 трябва да мине през касата сам(а). Следователно, ако $c_1 < d_1 + d_2$ и $c_2 < d_2 + d_3$, можем веднага да кажем, че някакво комбиниране сред първите трима души трябва да има, но *a priori* не е ясно какво: дали h_1 с h_2 , или h_2 с h_3 .

Задачата има решение с “груба сила”. Забелязваме, че всяко решение се определя еднозначно от това, кои съседни в опашката се комбинират по двойки. Останалите хора—тези извън двойките—минават през касата индивидуално с дадените си индивидуални времена. С други думи, може да дефинираме всяко решение чрез булев (характеристичен) масив с дължина $n - 1$, да кажем $B[1..n-1]$, като $B[i] = 1$ означава, че h_i се комбинира с h_{i+1} , а $B[i] = 0$ означава, че h_i не се комбинира с h_{i+1} . Този масив не може да съдържа съседни единици, тъй като всеки човек от опашката, който потенциално може да участва в две двойки (това са всички хора без първия и последния), в дадено решение може да участва в най-много една от тях. Следователно, всички възможни комбинирания **не са** 2^{n-1} , а са толкова, колкото са булевите вектори с дължина $n - 1$ без съседни единици. Известно е, че тези вектори са F_{n+1} на брой, където F_n е n -тото число на Фибоначи.

Допълнение 55: Колко са булевите вектори без съседни единици

Нека F_m е m -тото число на Фибоначи. Нека T_m е броят на всички булеви вектори с дължина m , в които няма съседни единици. Ще покажем, че $T_m = F_{m+2}$ за всяко $m \geq 1$. Очевидно $T_1 = 2 = F_3$, а $T_2 = 3 = F_4$, защото от четирите булеви вектора с дължина 2, само векторът 11 не отговаря на условието.

За $m > 2$, съобразяваме, че всеки такъв вектор може да започва с единица, но тогава вторият му елемент задължително е нула и следва булев вектор с дължина $m - 2$ без съседни единици, или да започва с нула, като след нея има булев вектор с дължина $m - 1$ без съседни единици. По принципа на разбиването, $T_m = T_{m-1} + T_{m-2}$. Доказахме, че $T_m = F_{m+2}$ за всяко $m \geq 1$.

Тъй като $F_n \asymp \phi^n$, където $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6$, решението с груба сила е експоненциален алгоритъм, само че по отношение на експонента, чиято основа е по-малка от 2. Въпреки че основата е по-малка от 2, все пак е число, по-голямо от 1, което прави подходът неефикасен на практика.

Олекотеният вариант. Ефикасно решение за олекотения вариант може да се получи със следните съображения. Да си представим, че разглеждаме последователно редиците

$$\begin{aligned} s_1 &= \langle h_1 \rangle \\ s_2 &= \langle h_1, h_2 \rangle \\ s_3 &= \langle h_1, h_2, h_3 \rangle \\ &\dots \\ s_n &= \langle h_1, h_2, \dots, h_n \rangle \end{aligned}$$

и се опитваме да получим оптимума за s_k чрез оптимумите за $s_{k-1}, s_{k-2}, \dots, s_1$. Ако успеем в това, то оптимумът за s_n е търсеният отговор.

Оптимумът за s_1 е просто d_1 , защото един човек не може да се комбинира с никого. Оптимумът за s_2 е $\min\{d_1 + d_2, c_1\}$, защото или h_1 и h_2 минават индивидуално за време $d_1 + d_2$, или се комбинират и минават за време c_1 , а ние искаме минимума от тези времена. Дотук всичко е просто и ясно. Ключово за постигане на правилно разбиране е ефикасното пресмятане на оптимума за s_3 . Ако изчислим оптимума за s_3 като $\min\{d_1 + d_2 + d_3, c_1 + d_3, d_1 + c_2\}$, защото:

- може и тримата да минат индивидуално за време $d_1 + d_2 + d_3$,
- може първите двама да се комбинират, оставяйки третия сам, което означава време $c_1 + d_3$,
- може първият да мине сам, а вторият и третият да се комбинират, което означава време $d_1 + c_2$

ще сбъркаме. Математически е точно така, но от алгоритмична гледна точка това води до катастрофа! Ако продължим в същия дух, оптимумът за s_4 е минимумът на 5 числа, оптимумът за s_5 е минимумът на 8 числа, оптимумът за s_6 е минимумът на 13 числа, и така нататък. При този подход, оптимумът за s_k е минимумът на F_{k+1} числа. Но това е решението с груба сила, което вече коментирахме и отхвърлихме. По този начин няма да получим ефикасен алгоритъм.

Добрата идея за s_3 идва от едно съвсем просто съображение: третият човек може или да мине индивидуално, или да се комбинира с втория. Друга възможност няма.

- Ако h_3 мине индивидуално, оптимумът за s_3 е d_3 плюс оптимума за s_2 . Когато сме стигнали до s_3 , ние вече имаме оптимална стойност за s_2 . **Ние вече сме изчислили** дали тази стойност е $d_1 + d_2$ или c_1 , пресмятайки оптимума за s_2 . Няма смисъл да изчисляваме оптимума за s_2 отново, правейки изчисленията за s_3 .
- Ако h_3 се комбинира с h_2 , решението е c_2 плюс оптимума за s_1 , който вече имаме.

Тъй като по отношение на s_3 няма други възможности (освен h_3 да мине индивидуално и h_3 да се комбинира с h_2), оптимумът за s_3 е минимумът на следните две стойности:

- d_3 плюс оптимума за s_2 и
- c_2 плюс оптимума за s_1 .

Аналогично, оптимумът за s_4 е минимумът на:

- d_4 плюс оптималното решение за s_3 и
- c_3 плюс оптималното решение за s_2 .

Да обобщим. Оптимумът за s_i , $3 \leq i \leq n$, е минимумът на:

- d_i плюс оптимума за s_{i-1} и
- c_{i-1} плюс оптимума за s_{i-2} .

Накратко, ако оптимумът за s_i е $\text{opt}(s_i)$, то в сила е

$$\text{opt}(s_i) = \begin{cases} d_1, & \text{ако } i = 1 \\ \min \{d_1 + d_2, c_1\}, & \text{ако } i = 2 \\ \min \{d_i + \text{opt}(s_{i-1}), c_{i-1} + \text{opt}(s_{i-2})\}, & \text{ако } i \geq 3 \end{cases} \quad (12.1)$$

Това уравнение е коректно, но от практически съображения по конструиране на алгоритъма ще го препишем в следния (еквивалентен) вид, въвеждайки фиктивна празна редица $s_0 = \langle \rangle$, чийто оптимум е нула.

$$\text{opt}(s_i) = \begin{cases} 0, & \text{ако } i = 0 \\ d_1, & \text{ако } i = 1 \\ \min \{d_i + \text{opt}(s_{i-1}), c_{i-1} + \text{opt}(s_{i-2})\}, & \text{ако } i \geq 2 \end{cases} \quad (12.2)$$

(12.2) може да се имплементира като ефикасен алгоритъм. Ако изчисляваме оптимумите за $s_2, s_3, \dots, s_n$ в този ред и ги съхраняваме в масив, то, за $2 \leq i \leq n$, при изчисляването на оптимума за s_i ние разполагаме с оптимумите за s_{i-1} и s_{i-2} , прочитайки ги от масива във време $\Theta(1)$.

```

ALG QUEUE PROCESSING( $d[1..n], c[1..n-1]$ )
1  създай масив от числа  $\text{opt}[0..n]$ 
2   $\text{opt}[0] \leftarrow 0$ 
3   $\text{opt}[1] \leftarrow d[1]$ 
4  for  $i \leftarrow 2$  to  $n$ 
5       $\text{opt}[i] \leftarrow \min(d[i] + \text{opt}[i-1], c[i-1] + \text{opt}[i-2])$ 
6  return  $\text{opt}[n]$ 

```

Коректността на алгоритъма е доста очевидна. Това е характерно за алгоритмите, изградени по схемата **Динамично Програмиране** – алгоритъмът е директна имплементация на някаква рекурсия, в случая (12.2), и, ако сме убедени в коректността на рекурсията, коректността на алгоритъма е (почти) очевидна. В контраст с това, алгоритмите, изградени по алчната схема, по правило имат нетривиални доказателства за коректност, примерно с инварианти на цикли. Разбира се, ние можем да докажем формално, педантично и прецизно коректността на ALG QUEUE PROCESSING с инвариант на цикъла “при всяко достигане на ред 4, $\text{opt}[i-1]$ и $\text{opt}[i-2]$ съдържат съответно оптимумите за s_{i-1} и s_{i-2} ”, но няма да го направим.

Сложността по време е $\Theta(n)$.

Тежкият вариант. Сега да разгледаме задачата в тежкия вариант, в който се иска и някое оптимално решение, а не просто цената на такова. Идеята е много проста: всяка от цените $\text{opt}[2], \text{opt}[3], \dots, \text{opt}[n]$ е изчислена като минимум от две суми на ред 5. За всеки $\text{opt}[i]$ записваме индекса на елемента от масива, който участва в сумата, чрез която сме изчислили $\text{opt}[i]$. Този индекс е или $i-1$, или $i-2$. След получаването на $\text{opt}[n]$, проследяваме отдясно наляво веригата от индекси и това дава едно оптимално решение. Такова проследяване на английски се нарича *traceback*. На български ще казваме *обратно проследяване*.

Да разсъждаваме отдясно наляво. Стойността на $\text{opt}[n]$ е равна на минимума на тези двете: $d[n] + \text{opt}[n-1]$ и $c[n-1] + \text{opt}[n-2]$ (ред 5).

- Да допуснем, че $d[n] + \text{opt}[n-1] < c[n-1] + \text{opt}[n-2]$. Разговорно, това означава, че h_n трябва да мине сам(a). Формално, това означава, че непосредственото изчисляване на $\text{opt}[n]$ става чрез $\text{opt}[n-1]$.
- Да допуснем, че $d[n] + \text{opt}[n-1] > c[n-1] + \text{opt}[n-2]$. Разговорно, това означава, че h_n трябва да се комбинира с h_{n-1} . Формално, това означава, че непосредственото изчисляване на $\text{opt}[n]$ става чрез $\text{opt}[n-2]$.
- Да допуснем, че $d[n] + \text{opt}[n-1] = c[n-1] + \text{opt}[n-2]$. Разговорно, това означава, че е все едно дали h_n минава сам(a), или се комбинира с h_{n-1} . Формално, това означава, че непосредственото изчисляване на $\text{opt}[n]$ става чрез кой да е от $\text{opt}[n-1]$ и $\text{opt}[n-2]$. В този случай избираме произволно **точно единия** от $\text{opt}[n-1]$ и $\text{opt}[n-2]$ като елементът, участващ в непосредственото изчисляване на $\text{opt}[n]$. Забележете, че не избираме и двата, защото искаме да получим едно решение, а не всички решения.

Във всеки случай, $\text{opt}[n-1]$ да участва непосредствено в изчисляването на $\text{opt}[n]$ е същото като h_n да **не се** комбинира с h_{n-1} ; съответно, $\text{opt}[n-2]$ да участва непосредствено в изчисляването на $\text{opt}[n]$ е същото като h_n да **се** комбинира с h_{n-1} .

Разполагайки с елемента $\text{opt}[j]$, участващ в непосредственото изчисляване на $\text{opt}[n]$, действаме по същия начин за него, стига j да е достатъчно голямо – установяваме кой от предните елементи участва в непосредственото изчисляване на $\text{opt}[j]$. И така нататък, докато не стигнем до началото на масива opt . По този начин, тръгвайки от $\text{opt}[n]$ назад, създаваме верига от индекси, която ни казва—за някое оптимално решение—кои хора минават сами и кои се комбинират в двойки. Това е обратното проследяване.

Ето пример. Нека $n = 5$ и

$$d = [5, 6, 3, 5, 9]$$

$$c = [7, 8, 4, 10]$$

Тогава

$$\text{opt}[0] \leftarrow 0$$

$$\text{opt}[1] \leftarrow d[1] = 5$$

$$\text{opt}[2] \leftarrow \min(d[2] + \text{opt}[1], c[1] + \text{opt}[0]) = \min(6 + 5, 7 + 0) = 7$$

$$\text{opt}[3] \leftarrow \min(d[3] + \text{opt}[2], c[2] + \text{opt}[1]) = \min(3 + 7, 8 + 5) = 10$$

$$\text{opt}[4] \leftarrow \min(d[4] + \text{opt}[3], c[3] + \text{opt}[2]) = \min(5 + 10, 4 + 7) = 11$$

$$\text{opt}[5] \leftarrow \min(d[5] + \text{opt}[4], c[4] + \text{opt}[3]) = \min(9 + 11, 10 + 10) = 20$$

В червено са сумите-минимуми. Те дават индексите на предните елементи във веригата. Виждаме, че $\text{opt}[5]$ се получава от $\text{opt}[4]$, на свой ред $\text{opt}[4]$ се получава от $\text{opt}[2]$, а на свой ред $\text{opt}[2]$ се получава от $\text{opt}[0]$. Веригата от индекси е $\langle 0, 2, 4, 5 \rangle$. Ако четем отляво надясно (това отговаря на посоката, в която опашката минава през касата), има два елемента, при които увеличението към следващия е 2; това са точно комбиниранията в текущото решение. От друга страна, има един елемент, при който увеличението към следващия е 1; това отговаря на индивидуално минаване през касата в текущото решение. По-подробно,

- увеличението с 2 от 0 на 2 означава, че h_1 се комбинира с h_2 с време 7,
- увеличението с 2 от 2 на 4 означава, че h_3 се комбинира с h_4 с време 4,

- увеличението с 1 от 4 на 5 означава, че h_5 минава самостоятелно за време 9.

Общото време е $7 + 4 + 9 = 20$ и това е оптимално.

Има още едно оптимално решение. При изчисляването на $\text{opt}[5]$ взехме минимума на равни стойности. Можем да вземем $c[4] + \text{opt}[3]$ и това ще даде веригата $\langle 0, 2, 3, 5 \rangle$, която дава друго оптимално решение: h_1 се комбинира с h_2 с време 7, h_3 минава самостоятелно с време 3, h_4 се комбинира с h_5 с време 10; общото време е $7 + 3 + 10 = 20$ и това също е оптимално.

Наблюдение 72: Всички оптимални решения може да са прекалено много

Не трябва да се опитваме да получим всички оптимални решения, освен ако не сме сигурни, че те са малко на брой. В най-лошия случай, оптималните решения са поне експоненциално много в размера на екземпляра. Ерго, няма как да генерираме ефективно всички оптимални решения. В тази задача, такъв най-лош случай е $d[i] = 1$ за $1 \leq i \leq n$ и $c[i] = 2$ за $1 \leq i \leq n - 1$. Ако е така, то всяко легално комбиниране в опашката дава оптимално решение с тегло n , а броят на комбиниранията, както видяхме, е $\Theta(\phi^n)$.

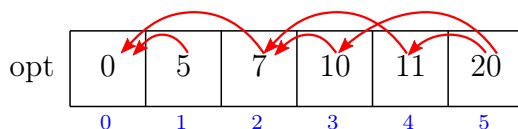
Следният алгоритъм връща както оптимална цена, така и **едно** оптимално решение.

```

ALG QUEUE PROCESSING-1( $d[1 \dots n]$ ,  $c[1 \dots n - 1]$ )
1  създай масив от числа  $\text{opt}[0 \dots n]$  и масив от индекси  $J[0 \dots n]$ 
2   $\text{opt}[0] \leftarrow 0$ ,  $J[0] \leftarrow \text{Nil}$ 
3   $\text{opt}[1] \leftarrow d[1]$ ,  $J[1] \leftarrow 0$ 
4  for  $i \leftarrow 2$  to  $n$ 
5      if  $d[i] + \text{opt}[i - 1] \leq c[i - 1] + \text{opt}[i - 2]$ 
6           $\text{opt}[i] \leftarrow d[i] + \text{opt}[i - 1]$ 
7           $J[i] \leftarrow i - 1$ 
8      else
9           $\text{opt}[i] \leftarrow c[i - 1] + \text{opt}[i - 2]$ 
10          $J[i] \leftarrow i - 2$ 
11 return ( $\text{opt}[n]$ ,  $J$ )

```

Можем да мислим за множеството от оптималните решения по следния начин. Да, това множество може да е грамадно и да не е ефективно изчислимо, но тук говорим концептуално, а не за генерирането му. Да си представим ориентиран граф G , в който върховете са клетките на $\text{opt}[0 \dots n]$, а ребро $(\text{opt}[i], \text{opt}[k])$ има тегло стойността на $\text{opt}[i]$ може да се изчисли чрез сума, в която участва $\text{opt}[k]$. Очевидно, $k \in \{i - 1, i - 2\}$, поради което G е даг. Също така, $\text{opt}[n]$ е източник, но може да има и други източници. За всяко $i \geq 1$, поне едно от ребрата $(\text{opt}[i], \text{opt}[i - 1])$ и $(\text{opt}[i], \text{opt}[i - 2])$ е налице, поради което връх $\text{opt}[0]$ е единственият сифон. Ключовото наблюдение е, че всеки максимален по включване път в G с начало $\text{opt}[n]$ (очевидно крайт е $\text{opt}[0]$) отговаря на точно едно оптимално решение, и обратно, всяко оптимално решение отговаря на такъв път в дага. Да се направи обратно проследяване, за да се генерира едно оптимално решение, е същото като да се намери път с начало $\text{opt}[n]$ и край $\text{opt}[0]$ в G . Ето въпросният даг за примера, който видяхме преди малко:



И едно предупреждение. Следният алгоритъм за тежкия вариант на задачата е **НЕКОРЕК-ТЕН**. Идеята е решението да бъде представено чрез булев масив $B[1..n-1]$, като $B[i] = 1$ означава, че h_i и h_{i+1} се комбинират, а $B[i] = 0$ означава обратното, за $1 \leq i \leq n-1$. Тази идея—решението да се представи чрез характеристичен вектор, а не чрез обратно проследяване—не е лоша сама по себе си, но трябва да се имплементира по друг начин.

ALG QUEUE PROCESSING WRONG($d[1..n]$, $c[1..n-1]$)

```

1  създай масиви  $opt[1..n]$ ,  $B[1..n-1]$ 
2   $opt[1] \leftarrow d[1]$ 
3  if  $d[1] + d[2] < c[1]$ 
4       $B[1] \leftarrow 0$ 
5       $opt[2] \leftarrow d[1] + d[2]$ 
6  else
7       $B[1] \leftarrow 1$ 
8       $opt[2] \leftarrow c[1]$ 
9  for  $i \leftarrow 3$  to  $n$ 
10     if  $d[i] + opt[i-1] < c[i-1] + opt[i-2]$ 
11          $B[i-1] \leftarrow 0$ 
12          $opt[i] \leftarrow d[i] + opt[i-1]$ 
13     else
14          $B[i-1] \leftarrow 1$ 
15          $opt[i] \leftarrow c[i-1] + opt[i-2]$ 
16  return ( $B, opt$ )

```

Ето контрапример. Нека $n = 3$, $d = [100, 150, 160]$ и $c = [50, 10]$. Този алгоритъм връща $B = [1, 1]$, което не е допустимо решение, понеже съдържа съседни единици.

Ние обаче може да познаем веднага, че алгоритъмът е некоректен, дори без сме намерили контрапример. Алгоритъмът няма как да е коректен, защото той работи с едно “премитане” на входа отляво надясно и фиксира $B[i]$ на 0 или 1, без да е прочел входните данни отвъд $B[i]$. От общи съображения е ясно, че решението дали двама съседни в опашката да се комбинират или да не се комбинират може да се вземе само като глобално решение в контекста на целия вход.

Ето една важна разлика между алгоритмите по схемата **Динамично Програмиране** и алгоритмите по алчната схема. При алчната схема решенията се вземат локално. При **Динамично Програмиране** решенията се вземат глобално, в контекста на целия вход.

12.1.2 Числата на Fibonacci

Задачата е, по дадено n , да се изчисли F_n , където

$$F_n = \begin{cases} 0, & \text{ако } n = 0, \\ 1, & \text{ако } n = 1, \\ F_{n-1} + F_{n-2}, & \text{ако } n > 1 \end{cases} \quad (12.3)$$

При големи стойности на n бихме имали проблеми с представянето на F_n , ако ползваме стандартен език за програмиране като C, защото F_n ще препълва отредената му памет. Да игнорираме това допълнително усложнение, което няма общо с основната мисъл тук; или да допуснем, че ползваме *библиотека*, осигуряваща работа с цели числа с произволна големина.

Ако имплементираме това рекурентно уравнение като рекурсивна програма на C директно, получаваме фрагмент от този вид:

```
unsigned fib(unsigned n){
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return
        fib(n-1) + fib(n-2);
}
```

Когато авторът на записките пусна тази програма с вход 10, изпълнението отне около 0.003 секунди. С вход 45 изпълнението отне около 8.937 секунди. С вход 47 изпълнението отне около 23.273 секунди. Явно това е програма с експоненциална сложност по време.

Не е трудно да се досетим защо е така. Изпълнението на `fib(45)` вика първо `fib(44)`; на свой ред, `fib(44)` вика `fib(43)` и `fib(42)`, и така нататък. След излизането от `fib(44)` се прави викане на `fib(43)`, което обаче “не знае”, че тази стойност вече е била пресметната и започва да я смята наново, като вика `fib(42)` и `fib(41)`, и така нататък. Това ненужно повторно изчисление се случва навсякъде в дървото на рекурсията, а не само в корена. За `fib(45)`, началните условия се достигат точно 1836311903 пъти, а редът `fib(n-1) + fib(n-2)`; се достига точно 1836311902 пъти. С други думи, дървото на рекурсията има 1836311903 листа и 1836311902 вътрешни върхове. И всичко това е за пресмятането на само 46 различни стойности $F_0, \dots, F_{45}$.

Ето решение, изградено по схемата **Динамично Програмиране**. Следният програмен фрагмент на C изчислява числа на Фибоначи, като помни досега изчислените стойности и не върши излишни изчисления отново и отново:

```
unsigned A[n+1];
int i;
A[0] = 0;
A[1] = 1;
for (i = 2; i <= n; i++) {
    A[i] = A[i-1] + A[i-2];
}
return A[n];
```

Изпълнението на този фрагмент за вход 47 е светкавично: 0.003 секунди.

От практическа гледна точка, проблем с този код е препълването на променливите, което настъпва дори за скромни стойности на n заради експоненциалното нарастване на числата на Фибоначи; феномен, който вече споменахме. Ако не се ползва библиотека за работа с цели числа с произволна големина, тази програма връща 3 742 884 757 при вход 200 върху 64 битов компютър. В действителност,

$$F_{200} = 280\,571\,172\,992\,510\,140\,037\,611\,932\,413\,038\,677\,189\,525$$

Допълнение 56: Най-бързото пресмятане на числата на Fibonacci

Известен е още по-бърз начин за пресмятане на числата на Fibonacci, който се основава на следния факт. Да разгледаме матрицата $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ и няколко нейни степени:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^4 = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^5 = \begin{bmatrix} 3 & 5 \\ 5 & 8 \end{bmatrix}$$

Изглежда, че числата на Fibonacci се появяват в тази редица от матрични степени.

Теорема 99: Числата на Fibonacci в редица от матрични степени

$$\text{За всяко } k \in \mathbb{N}^+ \text{ е изпълнено } \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k = \begin{bmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{bmatrix}.$$

Доказателство: Базата е $k = 1$. Очевидно е вярно, че

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_0 & F_1 \\ F_1 & F_2 \end{bmatrix}$$

Индуктивното допускане е, че $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{k-1} = \begin{bmatrix} F_{k-2} & F_{k-1} \\ F_{k-1} & F_k \end{bmatrix}$. Тогава

$$\begin{aligned} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{k-1} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} F_{k-2} & F_{k-1} \\ F_{k-1} & F_k \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} 0 \cdot F_{k-2} + 1 \cdot F_{k-1} & 1 \cdot F_{k-2} + 1 \cdot F_{k-1} \\ 0 \cdot F_{k-1} + 1 \cdot F_k & 1 \cdot F_{k-1} + 1 \cdot F_k \end{bmatrix} = \begin{bmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{bmatrix} \quad \square \end{aligned}$$

На пръв поглед пресмятането на F_n чрез Теорема 99 ползва $n-1$ матрични умножения, всяко от които се извършва във време $\Theta(1)$, поради което ефикасността е асимптотично същата като пресмятането на рекурсията, дефинираща числата на Fibonacci, с използване на **Динамично Програмиране**. Работата е там, че $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$ може да бъде получена само с $\Theta(\lg n)$ матрични умножения по следния начин. Първо да допуснем, че n е точна степен на двойката, тоест, $n = 2^m$ за някое естествено m ; тоест, $m = \lg n$.

Правим следните изчисления.

$$B_1 \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$B_2 \leftarrow B_1 \cdot B_1 \quad // \quad B_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2$$

$$B_3 \leftarrow B_2 \cdot B_2 \quad // \quad B_3 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^4$$

$$B_4 \leftarrow B_3 \cdot B_3 \quad // \quad B_4 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^8$$

...

$$B_{m-2} \leftarrow B_{m-3} \cdot B_{m-3} \quad // \quad B_{m-2} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n/4}$$

$$B_{m-1} \leftarrow B_{m-2} \cdot B_{m-2} \quad // \quad B_{m-1} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n/2}$$

$$B_m \leftarrow B_{m-1} \cdot B_{m-1} \quad // \quad B_m = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

Търсената A^n е B_m . Използвахме само $\lg n$ матрични умножения, за да намерим желаната матрица. Тази техника на бързо намиране на матрична степен се нарича *повтаряне на повдигането на квадрат*. На английски е *repeated squaring*. Ние я ползваме на стр. 562 в алгоритъм за намиране на най-къси пътища чрез матрично умножение.

Дори n да не е точна степен на двойката, можем да намерим A^n с $\leq 2 \lceil \lg n \rceil$ матрични умножения. Знаем, че n се представя по уникален начин като сума от точни степени на двойката. Да кажем, $n = 2^{m_1} + \dots + 2^{m_k}$. Тогава $A^n = B_{m_1} \cdot \dots \cdot B_{m_k}$, като в процеса на повдигане на квадрат ние със сигурност сме получили $B_{m_1}, \dots, B_{m_k}$. Ето пример.

Искаме да намерим F_{200} . За целта пресмятаме $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{200}$. Но числото 200 като сума от точни степени на двойката е $200 = 128 + 64 + 8$. Тогава

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{200} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{128} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{64} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^8$$

Чрез repeated squaring получаваме $B_2, B_4, B_8, B_{16}, B_{32}, B_{64}$ и B_{128} . В частност,

$$B_8 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^8 = \begin{bmatrix} 13 & 21 \\ 21 & 34 \end{bmatrix}$$

$$B_{64} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{64} = \begin{bmatrix} 6\,557\,470\,319\,842 & 10\,610\,209\,857\,723 \\ 10\,610\,209\,857\,723 & 17\,167\,680\,177\,565 \end{bmatrix}$$

$$B_{128} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{128} = \begin{bmatrix} 155\,576\,970\,220\,531\,065\,681\,649\,693 & 251\,728\,825\,683\,549\,488\,150\,424\,261 \\ 251\,728\,825\,683\,549\,488\,150\,424\,261 & 407\,305\,795\,904\,080\,553\,832\,073\,954 \end{bmatrix}$$

Тогава

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{200} = B_8 \cdot B_{64} \cdot B_{128} = \begin{bmatrix} 173\,402\,521\,172\,797\,813\,159\,685\,037\,284\,371\,942\,044\,301 & 280\,571\,172\,992\,510\,140\,037\,611\,932\,413\,038\,677\,189\,525 \\ 280\,571\,172\,992\,510\,140\,037\,611\,932\,413\,038\,677\,189\,525 & 453\,973\,694\,165\,307\,953\,197\,296\,969\,697\,410\,619\,233\,826 \end{bmatrix}$$

И наистина

$$F_{200} = 280\,571\,172\,992\,510\,140\,037\,611\,932\,413\,038\,677\,189\,525$$

Каква е сложността по време на този начин на пресмятане на F_n ? Броят на матричните умножения е $\Theta(\lg n)$, матриците са с константни размери 2×2 , следователно сложността по време е $\Theta(\lg n)$, където n е големината на числото от входа (или, еквивалентно, n е големината на входа, но числото-вход е записано в унарна бройна система). Този извод е в сила при допусканията, че всички числа имат представяне с константен размер и операциите събиране и умножение върху тях се изпълняват в константно време. Тези допускания са стандартни за курса по алгоритми и ние ги ползваме навсякъде, но при пресмятането на F_n , ако n е прекалено голямо, въпросните допускания стават твърде нереалистични. Лесно може да се тества описаният алгоритъм в среда, позволяваща работа с числа с произволна големина като Maple (TM), и да се види, че сложността не е пропорционална на $\lg n$, а е доста по-висока, когато n е наистина голямо. Ако искаме да имаме смислен резултат за сложността като функция на n , трябва да разглеждаме всяко F_k , което се получава като междинен или краен резултат, като стринг с дължина $\Theta(\lg k)$, а не като атомарен обект с константна големина. В този усложнен модел, елементарните операции събиране и умножение не се извършват в константно време.

Както е посочено в [45, стр. 103–104], има и друг начин, може би по-практичен, за пресмятане на F_n чрез $\Theta(\lg n)$ междинни числа на Fibonacci. В основата е следното твърждение.

Теорема 100: Едно твърждение върху числа на Fibonacci

Нека $n \in \mathbb{N}^+$ и $m \in \mathbb{N}$. В сила е

$$F_{n+m} = F_m F_{n-1} + F_n F_{m+1}$$

Доказателство: Ще използваме Теорема 99. Нека $m, n \in \mathbb{N}^+$. Съгласно Теорема 99, в сила е

$$\begin{aligned} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^m &= \begin{bmatrix} F_{m-1} & F_m \\ F_m & F_{m+1} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n &= \begin{bmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n+m} &= \begin{bmatrix} F_{n+m-1} & F_{n+m} \\ F_{n+m} & F_{n+m+1} \end{bmatrix} \end{aligned}$$

Но умножението на матрици е асоциативно, така че

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n+m} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^m \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

Тогава

$$\begin{bmatrix} F_{n+m-1} & F_{n+m} \\ F_{n+m} & F_{n+m+1} \end{bmatrix} = \begin{bmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{bmatrix} \cdot \begin{bmatrix} F_{m-1} & F_m \\ F_m & F_{m+1} \end{bmatrix}$$

Извършваме умножението вдясно и получаваме

$$\begin{bmatrix} F_{n+m-1} & F_{n+m} \\ F_{n+m} & F_{n+m+1} \end{bmatrix} = \begin{bmatrix} F_{n-1}F_{m-1} + F_nF_m & F_{n-1}F_m + F_nF_{m+1} \\ F_nF_{m-1} + F_{n+1}F_m & F_nF_m + F_{n+1}F_{m+1} \end{bmatrix}$$

Но тогава $F_{n+m} = F_{n-1}F_m + F_nF_{m+1}$ за всички цели положителни m и n . Забелязваме, че при $m = 0$ този израз става

$$F_n = F_{n-1}F_0 + F_nF_1 = F_{n-1} \cdot 0 + F_n \cdot 1 = F_n$$

Ерго, тъждеството остава в сила и при $m = 0$. Тогава $n \in \mathbb{N}^+$ и $m \in \mathbb{N}$, в сила е

$$F_{n+m} = F_mF_{n-1} + F_nF_{m+1}.$$

□

Следствие 31

Нека $n \in \mathbb{N}^+$. В сила е

$$\begin{aligned} F_{2n-1} &= F_{n-1}^2 + F_n^2 \\ F_{2n} &= F_n(2F_{n-1} + F_n) \end{aligned}$$

Доказателство: Знаем, че $F_{n+m} = F_mF_{n-1} + F_nF_{m+1}$ за $n \in \mathbb{N}^+$ и $m \in \mathbb{N}$. Разглеждаме частния случай, в който $m = n - 1$. Тъждеството става

$$F_{n+n-1} = F_{n-1}F_{n-1} + F_nF_{n-1+1}$$

което е същото като

$$F_{2n-1} = F_{n-1}^2 + F_n^2$$

Сега разглеждаме частния случай, в който $m = n$. Тъждеството става

$$F_{n+n} = F_nF_{n-1} + F_nF_{n+1}$$

Но $F_{n+1} = F_n + F_{n-1}$, така че

$$F_{n+n} = F_nF_{n-1} + F_n(F_n + F_{n-1})$$

което е същото като

$$F_{2n} = F_n(2F_{n-1} + F_n).$$

□

Следствие 31 обосновава директно следния алгоритъм за изчисляване на F_n . Алгоритъмът връща наредена двойка, единият елемент от която е F_n .

FIBONACCI FAST($n \in \mathbb{N}^+$)

```

1  if  $n = 1$ 
2    return (0, 1)
3   $m \leftarrow \lfloor \frac{n}{2} \rfloor$ 
4   $(i, j) \leftarrow \text{FIBONACCI FAST}(m)$ 
5   $p \leftarrow i^2 + j^2$ 
6   $q \leftarrow j(2i + j)$ 
7   $t \leftarrow p + q$ 
8  if iseven( $n$ )
9    return ( $p, q$ )
10 else
11   return ( $q, t$ )
```

Удачно е да се имплементира в среда, която поддържа работа с цели числа с произволни големина, за да не се получат препълвания на променливи. Такава среда е Maple(TM). Ако го имплементираме на Maple(TM), можем да пресметнем F_{1000} за части от секундата.

$F_{1000} = 434665576869374564356885276750406258025646605173717804024817290895$
 $365554179490518904038798400792551692959225930803226347752096896232$
 $398733224711616429964409065331879382989696499285160037044761377951$
 66849228875

Работата на FIBONACCI FAST(1000) е определено по-бърза от изчисляването на F_{1000} чрез алгоритъма на стр. 549, изграден по схемата **Динамично Програмиране** и имплементиран на Maple(TM).

И накрая, едно предупреждение, което може би е излишно, но е добре да се спомене експлицитно. Добре известна е формулата на Binet за n -тото число на Fibonacci

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

Изведохме я в курса Дискретни Структури, решавайки рекурентното уравнение (12.3). Ако гледаме на формулата на Binet като на алгоритъм за пресмятане на F_n , дали той е по-бърз от разгледаните досега бързи алгоритми за пресмятане на F_n ? Отговорът е, че зависи какви допускания правим. При **силно нереалистичните** допускания, че

- операцията “повдигане на константа на степен n ” (*exponentiation* на английски) се извършва в константно време независимо от големината на променливата n и
- наличието на ирационалното число $\sqrt{5}$ не пречи на изчислението, в смисъл, че хем

не го апроксимираме с рационално число (за да не губим точност), хем операциите “повдигане на $1 + \sqrt{5}$ на степен n ” и “деление на $\sqrt{5}$ ” се вършат в константно време без загуба на точност,

този алгоритъм работи в константно време. Но изброените допускания са изключително нереалистични. Наистина, в тези лекции допуснахме, че основните операции върху числа събиране, умножение и повдигане на променлива n на константна степен се вършат в константно време независимо от големините на числата, но тези допускания са горе-долу реалистични и моделират адекватно доста изчисления, възникващи на практика. Допускането, че a^n , където $a > 1$ е константа, а n е променлива, се пресмята в константно време, е на съвсем друго ниво на нереалистичност; заради експлозивното нарастване на експоненциалната функция, това допускане е адекватно само за съвсем малки стойности на n .

Наличието на радикала $\sqrt{5}$ във формулата на Binet усложнява нещата допълнително. Ако апроксимираме $\sqrt{5}$ с число с плаваща запетая, ще загубим точност. Софтуерни пакети за компютърна алгебра като Maple(TM) могат да извършат изчислението $(1 + \sqrt{5})^n$ без загуба на точност, представяйки вътрешно $\sqrt{5}$ като специален обект, който не е нито integer, нито float. Maple(TM) умее да манипулира този обект ефикасно:

- този обект, повдигнат на четна степен $2k$ (k е естествено число), става 5^k , а повдигнат на нечетна степен $2k + 1$, става $5^k\sqrt{5}$,
- ако в процеса на изчислението се получи сума от вида $a\sqrt{5} - a\sqrt{5}$, тя се замества с нула,
- ако в процеса на изчислението се получи дроб от вида $\frac{a\sqrt{5}}{\sqrt{5}}$, тя се замества с a .

Ерго, за конкретна стойност на n , Maple(TM) може да изчисли F_n чрез формулата на Binet, получавайки две суми от по $n + 1$ събираеми. Това са двата числителя, които след всички опростявания съдържат известен брой появи на $\sqrt{5}$. След събирането на дробите доста събираеми се канцелират взаимно, появява се едно $\sqrt{5}$ пред скоби, което се канцелира с $\sqrt{5}$ в знаменателя и това, което остава, е търсеното F_n . Допълнително забавяне ще има заради това, че развиването на $(1 + \sqrt{5})^n$ и на $(1 - \sqrt{5})^n$ налага изчисляване на биномни коефициенти.

Накратко, на практика никога няма да използваме формулата на Binet за пресмятане на числа на Fibonacci.

12.1.3 Боядисване на къщи в редица

Дадени са n къщи $h_1, h_2, \dots, h_n$, които са една до друга в редица, в точно този ред. Всяка къща трябва да бъде боядисана в един от цветовете червен, зелен или син. Никои две съседни къщи не трябва да са в един и същи цвят. За всяко $i \in \{1, \dots, n\}$ са известни

- цената $c_r[i]$ за боядисване на h_i в червен цвят,
- цената $c_g[i]$ за боядисване на h_i в зелен цвят,
- цената $c_b[i]$ за боядисване на h_i в син цвят.

Иска се колкото е възможно по-ефикасен алгоритъм за изчисляване на минималната цена за боядисване на всички къщи. Достатъчно е алгоритъмът да намира само цената, а не и коя къща в какъв цвят да бъде.

В тази задача има линейна наредба върху къщите, също както в задачата за опашката пред касата хората бяха наредени линейно. В задачата за опашката пред касата има имплицитна посока върху линейно наредените хора заради начина, по който опашката минава във времето. В текущата задача няма посока върху линейната наредба на къщите, така че можем да разсъждаваме от който искаме край в посока към другия край. За определеност, нека започнем разсъжденията с h_n . Очевидно, h_n трябва да бъде боядисана или в червено, или в зелено, или в синьо.

- Ако h_n бъде боядисана в червено, минималната цена за всички къщи е $c_r[n]$ плюс минималната цена за боядисване на $h_1, \dots, h_{n-1}$, като h_{n-1} може да е зелена или синя, но не и червена.
- Ако h_n бъде боядисана в зелено, минималната цена за всички къщи е $c_g[n]$ плюс минималната цена за боядисване на $h_1, \dots, h_{n-1}$, като h_{n-1} може да е червена или синя, но не и зелена.
- Ако h_n бъде боядисана в синьо, минималната цена за всички къщи е $c_b[n]$ плюс минималната цена за боядисване на $h_1, \dots, h_{n-1}$, като h_{n-1} може да е червена или зелена, но не и синя.

Нека $opt_r[i]$ е минималната цена за боядисване на къщи $h_1, \dots, h_i$ по такъв начин, че h_i е червена. Нека $opt_g[i]$ е минималната цена за боядисване на къщи $h_1, \dots, h_i$ по такъв начин, че h_i е зелена. Нека $opt_b[i]$ е минималната цена за боядисване на къщи $h_1, \dots, h_i$ по такъв начин, че h_i е синя. Използвайки тези нотации, можем да препишем горните разсъждения така.

- Ако h_n е червена, минималната цена за всички къщи е $c_r[n] + \min \{opt_g[n-1], opt_b[n-1]\}$.
- Ако h_n е зелена, минималната цена за всички къщи е $c_g[n] + \min \{opt_r[n-1], opt_b[n-1]\}$.
- Ако h_n е синя, минималната цена за всички къщи е $c_b[n] + \min \{opt_r[n-1], opt_g[n-1]\}$.

Вече имаме ясна идея за решение: ако $i > 1$, ще изразим оптималната цена за всяка къща от първата до i -тата, за всеки от цветовете, чрез оптималните цени на къщите от първата до $(i-1)$ -вата. А ако $i = 1$, оптималните цени са просто цените за боядисването на първата къща във всеки от цветовете.

$$opt_r[1] = c_r[1] \quad (12.4)$$

$$opt_g[1] = c_g[1] \quad (12.5)$$

$$opt_b[1] = c_b[1] \quad (12.6)$$

$$opt_r[i] = c_r[i] + \min \{opt_g[i-1], opt_b[i-1]\} \text{ за } i \geq 2 \quad (12.7)$$

$$opt_g[i] = c_g[i] + \min \{opt_r[i-1], opt_b[i-1]\} \text{ за } i \geq 2 \quad (12.8)$$

$$opt_b[i] = c_b[i] + \min \{opt_r[i-1], opt_g[i-1]\} \text{ за } i \geq 2 \quad (12.9)$$

Търсеният отговор е $\min \{opt_r[n], opt_g[n], opt_b[n]\}$.

Превръщането на (12.4), ..., (12.9) в алгоритъм по схемата **Динамично Програмиране** е тривиално.

Боядисай Евтино($c_r[1, \dots, n]$, $c_g[1, \dots, n]$, $c_b[1, \dots, n]$)

```

1   $\text{opt}_r[1] \leftarrow c_r[1]$ 
2   $\text{opt}_g[1] \leftarrow c_g[1]$ 
3   $\text{opt}_b[1] \leftarrow c_b[1]$ 
4  for  $i \leftarrow 2$  to  $n$ 
5       $\text{opt}_r[i] \leftarrow c_r[i] + \min(\text{opt}_g[i-1], \text{opt}_b[i-1])$ 
6       $\text{opt}_g[i] \leftarrow c_g[i] + \min(\text{opt}_r[i-1], \text{opt}_b[i-1])$ 
7       $\text{opt}_b[i] \leftarrow c_b[i] + \min(\text{opt}_r[i-1], \text{opt}_g[i-1])$ 
8  return  $\min(\text{opt}_r[n], \text{opt}_g[n], \text{opt}_b[n])$ 

```

Коректността е очевидна предвид коректността на (12.4), ..., (12.9). Сложността и по време, и по памет е $\Theta(n)$.

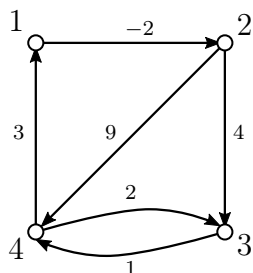
12.1.4 Най-къс път за всяка двойка върхове

Разглеждаме задачата за най-къси пътища в графи във варианта APShP (Лекция 11): за всеки два върха u и v в тегловен ориентиран граф, търсим най-къс път $u \rightsquigarrow v$. Едно решение е да викаме от всеки връх DIJKSTRA или BELLMAN-FORD. Но има интересен алгоритъм за варианта APShP, който си заслужава да бъде разгледан подробно. Това е алгоритъмът на Floyd-Warshall. Освен че е интересен от теоретична гледна точка, алгоритъмът на Floyd-Warshall допуска отрицателни тегла и има сложност по време $\Theta(n^3)$, така че е по-бърз от n -кратното пускане на BELLMAN-FORD, което би било със сложност $\Theta(n^4)$.

В тази секция графите се представят не със списъци на съседство, а с матрици от теглата. $G = (\{1, \dots, n\}, E)$ е графът, който разглеждаме, и той е представен с $n \times n$ матрица от теглата, наречена W , като

$$W[i, j] = \begin{cases} 0, & \text{ако } i = j, \\ \text{теглото на реброто } (i, j), & \text{ако има такова и } i \neq j \\ \infty, & \text{ако няма ребро } (i, j) \text{ и } i \neq j \end{cases}$$

Теглата са реални числа. Ето пример:



W	1	2	3	4
1	0	-2	∞	∞
2	∞	0	4	9
3	∞	∞	0	1
4	3	∞	2	0

Разговорно казано, смисълът на $W[i, j]$ е минималната цена на **директно отиване** от връх i във връх j . Цената на това отиване очевидно е

- ако $i = j$, то нула,
- ако $i \neq j$ и има ребро (i, j) , то цената на (i, j) ,
- ако $i \neq j$ и няма ребро (i, j) , то безкрайност.

Става дума именно за директно отиване—без междинни върхове—а не за най-къс път. В примера, директното отиване от връх 2 във връх 4 има цена 9 и затова $W[2, 4] = 9$; това, че има път от 2 до 4 с цена 5, няма значение, тъй като този по-къс път съдържа връх 3 като вътрешен връх.

На първо време се фокусираме върху изчисляването на теглата на най-къси пътища. Когато постигнем това, лесно ще добавим код, който изчислява самите пътища. И така, изходът е квадратна матрица $n \times n$, наречена D , като $D[i, j]$ съдържа теглото на най-къс път $i \rightsquigarrow j$.

12.1.4.1 Алгоритъм, реализиращ матрично умножение

Според [34, стр. 706], този алгоритъм е фолклор. Това означава, че авторството му е неясно и загубено в миналото. Той е първият нетривиален алгоритъм по алгоритмичната схема **Динамично Програмиране**, който ще разгледаме. Ще го конструираме бавно и с подробни обяснения.

Нека i и j са произволни върхове в графа. Да въведем следната характеристика на най-къс път $i \rightsquigarrow j$: p има някакъв брой ребра. Ето разсъждения, които започват от този абсолютно очевиден факт и достигат до идея за рекурсивна декомпозиция (понятие, което предстои да обясним подробно в Подсекция 12.2.2).

Разглеждаме произволен най-къс път $i \rightsquigarrow j$. Той има някакъв брой ребра. Да кажем, че има k ребра, където $k \in \{0, \dots, n-1\}$. // това е тривиално ...

Променяме леко формулировката: за някое k , този най-къс път $i \rightsquigarrow j$ има не повече от k ребра. // очевидно – е, и?

Продължаваме: всеки най-къс път $i \rightsquigarrow j$, който има не повече от k ребра, където $k \geq 1$, се състои от най-къс път $i \rightsquigarrow \ell$, който има не повече от $k-1$ ребра, за някой връх ℓ , плюс реброто (ℓ, j) . // аха, това е!

Това, че подпътят от i до ℓ е оптимален, следва веднага от Теорема 96.

Ето подробна рекурсивна декомпозиция. Нека $i, j \in \{1, \dots, n\}$ са произволни.

- Дължината на най-къс път $i \rightsquigarrow j$ с не повече от 0 ребра е или 0, ако $i = j$, или ∞ , ако $i \neq j$. Това е базата на рекурсията.
- Дължината на най-къс път $i \rightsquigarrow j$ с не повече от k ребра, за $k \geq 1$, е минимумът на тези две неща:

1. дължината на най-къс път $i \rightsquigarrow j$ с не повече от $k-1$ ребра

2. минимума на следните n на брой суми:

дължината на най-къс път $i \rightsquigarrow 1$ с не повече от $k-1$ ребра плюс теглото $W[1, j]$ на реброто $(1, j)$,

дължината на най-къс път $i \rightsquigarrow 2$ с не повече от $k-1$ ребра плюс теглото $W[2, j]$ на реброто $(2, j)$,

...

дължината на най-къс път $i \rightsquigarrow n$ с не повече от $k-1$ ребра плюс теглото $W[n, j]$ на реброто (n, j) .

Ето схема за изчисление, базирана на тази рекурсивна декомпозиция. За всяко $k \in \{0, \dots, n-1\}$, нека $D^{(k)}$ е $n \times n$ матрица от дължините на най-къси пътища с не повече от k ребра.

Тогава $D^{(k)}[i, j]$ дължината на най-къс път с не повече от k ребра от i до j . Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)}[i, j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases}$$

$$D^{(k)}[i, j] = \min \{D^{(k-1)}[i, j], \min \{D^{(k-1)}[i, \ell] + W[\ell, j] \mid 1 \leq \ell \leq n\}\}, \text{ за } k > 0$$

Това може да бъде опростено до

$$D^{(0)}[i, j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases} \quad (12.10)$$

$$D^{(k)}[i, j] = \min \{D^{(k-1)}[i, \ell] + W[\ell, j] \mid 1 \leq \ell \leq n\}, \text{ за } k > 0 \quad (12.11)$$

понеже $D^{(k-1)}[i, j]$ е елемент на множеството $\{D^{(k-1)}[i, \ell] + W[\ell, j] \mid 1 \leq \ell \leq n\}$. За да се убедим в това, да съобразим, че при $\ell = j$, $D^{(k-1)}[i, \ell] + W[\ell, j]$ става $D^{(k-1)}[i, j] + W[j, j]$, което е $D^{(k-1)}[i, j]$, понеже $W[j, j] = 0$.

Решението е матрицата $D^{(n-1)}$, тъй като простите пътища имат $\leq n - 1$ ребра.

Изразите (12.10) и (12.11) са на практика **ефективен** рекурсивен алгоритъм: за всеки i и j , $D^{(n-1)}[i, j]$ се пресмята чрез рекурсивни викания на $D^{(n-2)}[i, \ell]$, за $1 \leq \ell \leq n$, а стойностите $W[\ell, j]$ са част от входа. Този алгоритъм обаче е изключително **неефикасен**! Дървото на рекурсията му има разклоненост n и височина $n - 1$, което означава, че сложността му е $\Omega(n^{n-1})$. За всички входове с нетривиална големина, каквито възникват при моделиране на реалността, такъв алгоритъм е безполезен.

Но стойностите, които трябва да бъдат изчислени общо, не са необятно много. Всички матрици са $D^{(0)}$, $D^{(1)}$, ..., $D^{(n-1)}$. Това са n матрици. Всяка от тях има n^2 клетки. Ерго, в (12.10) и (12.11) става дума за пресмятане на само n^3 матрични клетки, като всяко пресмятане, което не е начално условие, е намиране на минимума на n суми, всяка от които е на две числа. Възниква въпросът, защо има алгоритъм със сложност $\Omega(n^{n-1})$, който пресмята само n^3 неща? Отговорът е, че това се случва, понеже **различни** рекурсивни викания пресмятат **една и съща** стойност. Тези викания са напълно отделни и “не знаят” едно за друго, поради което една стойност, веднъж пресметната, се губи и после се пресмята отново и отново[†].

Примерно, при $n = 300$, пресмятането на $D^{(299)}[1, 300]$ ползва тези стойности:

$$D^{(298)}[1, 1], D^{(298)}[1, 2], D^{(298)}[1, 3], \dots, D^{(298)}[1, 300]$$

Ясно е, че $D^{(298)}[1, 1]$ е 0, но останалите трябва да се сметнат. На свой ред, пресмятането на всяка от $D^{(298)}[1, 2]$, $D^{(298)}[1, 3]$, ..., $D^{(298)}[1, 300]$ ползва тези стойности:

$$D^{(297)}[1, 1], D^{(297)}[1, 2], D^{(297)}[1, 3], \dots, D^{(297)}[1, 300]$$

Ясно е, че $D^{(297)}[1, 1]$ е 0, но останалите трябва да се сметнат. Това означава, че

- за да намерим $D^{(298)}[1, 2]$, ние трябва да имаме $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$,
- за да намерим $D^{(298)}[1, 3]$, ние трябва да имаме $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$,
- ...

[†]Ние вече се сблъскахме с този феномен на стр. 549 при пресмятане на числата на Fibonacci, което буквално следва рекурсията.

- за да намерим $D^{(298)}[1, 300]$, ние трябва да имаме $D^{(297)}[1, 2], \dots, D^{(297)}[1, 300]$,

Дори ако сме намерили $D^{(297)}[1, 2], D^{(297)}[1, 3], \dots, D^{(297)}[1, 300]$ и после сме сметнали $D^{(298)}[1, 2]$, когато започваме да пресмятаме $D^{(298)}[1, 3]$, ще пресметнем всяко от $D^{(297)}[1, 2], D^{(297)}[1, 3], \dots, D^{(297)}[1, 300]$ **отново**, ако следваме рекурсията в (12.11) буквално. Това разхитително пресмятане на едно и също нещо в различни върхове на дървото на рекурсията става на всяко ниво. Не е чудно, че сложността на алгоритъма се оказва $\Omega(n^{n-1})$.

Печелившата идея е, веднъж пресменати, стойностите да не се губят, а се пазят и при необходимост се вземат в $\Theta(1)$ време. Така можем да направим прилично бърз алгоритъм, който е базиран на (12.10) и (12.11), без обаче да ги имплементира буквално. Този алгоритъм не е рекурсивен, а итеративен, защото работи отдолу-нагоре: от $D^{(0)}$ намира $D^{(1)}$, от $D^{(1)}$ намира $D^{(2)}$, и така нататък, и накрая от $D^{(n-2)}$ намира $D^{(n-1)}$, което е решението.

SHORTEST PATH MM(W : матрица $n \times n$, представляваща тегловен ориентиран граф)

```

1  for i ← 1 to n
2      for j ← 1 to n
3          if i = j
4               $D^{(0)}[i, j] \leftarrow 0$ 
5          else
6               $D^{(0)}[i, j] \leftarrow \infty$ 
7  for k ← 1 to n - 1
8      for i ← 1 to n
9          for j ← 1 to n
10              $D^{(k)}[i, j] \leftarrow \infty$ 
11             for  $\ell \leftarrow 1$  to n
12                  $D^{(k)}[i, j] \leftarrow \min \{D^{(k)}[i, j], D^{(k-1)}[i, \ell] + W[\ell, j]\}$ 
13  return  $D^{(n-1)}$ 
```

Коректността му е очевидна за читател, който/която е убеден/убедена в коректността на (12.10) и (12.11), тъй като той пресмята ефикасно тази рекурсия. Сложността му по време очевидно е $\Theta(n^4)$. След малко ще видим как може да подобрим значително тази сложност. Сложността му по памет привидно е $\Theta(n^3)$, защото за всяко $i \in \{0, \dots, n-1\}$, матрицата $D^{(i)}$ е отделен обект. Всъщност, сложността по памет може лесно да се подобри до $\Theta(n^2)$, като използваме само една матрица и я презаписваме.

Сега да видим откъде идва “Matrix Multiplication”.

SHORTEST PATH MM реализира абстрактно матрично умножение. Да разгледаме отново кода на SHORTEST PATH MM, с намалени отстъпи в началото на редовете, и до него да разгледаме код, реализиращ повдигането на $n \times n$ числена матрица W на $(n-1)$ -ва степен, започвайки от матрицата-идентитет.

SHORTEST PATH MM(W)

```

for i ← 1 to n
  for j ← 1 to n
    if i = j
      D(0)[i, j] ← 0
    else
      D(0)[i, j] ← ∞
  for k ← 1 to n - 1
    for i ← 1 to n
      for j ← 1 to n
        D(k)[i, j] ← ∞
      for ℓ ← 1 to n
        D(k)[i, j] ← min(D(k)[i, j],
                        D(k-1)[i, ℓ] + W[ℓ, j])
  return D(n-1)

```

MATIX POWER(W)

```

for i ← 1 to n
  for j ← 1 to n
    if i = j
      A(0)[i, j] ← 1
    else
      A(0)[i, j] ← 0
  for k ← 1 to n - 1
    for i ← 1 to n
      for j ← 1 to n
        A(k)[i, j] ← 0
      for ℓ ← 1 to n
        A(k)[i, j] ← + (A(k)[i, j],
                       A(k-1)[i, ℓ] • W[ℓ, j])
  return A(n-1)

```

Приликата между двата алгоритъма не е повърхностна, а фундаментална. SHORTEST PATH MM наистина реализира $n - 1$ матрични умножения с входната матрица, започвайки от матрицата-идентитет, само че спрямо полупръстена[†] $\mathbb{R} \cup \{\infty\}$ с операции $\min$ (абстрактно събиране) и $+$ (абстрактно умножение), като абстрактната нула (идентитетът на абстрактното събиране) е ∞ , а абстрактната единица (идентитетът на абстрактното умножение) е 0. Съответствията са подчертани чрез еднакви цветове. Лесно може да проверим, че

1. абстрактното събиране $\min$ е асоциативно и комутативно с идентитет ∞ ;
2. абстрактното умножение $+$ е асоциативно и комутативно с идентитет 0;
3. абстрактното умножение е ляво и дясно дистрибутивно спрямо абстрактното събиране

$$\begin{aligned}
 a + \min(b, c) &= \min((a + b), (a + c)) \\
 \min(a, b) + c &= \min((a + c), (b + c))
 \end{aligned}$$

4. абстрактното умножение $+$ с абстрактната нула ∞ дава абстрактната нула:

$$a + \infty = \infty + a = \infty$$

така че това наистина е полупръстен. Повече информация за този вид полупръстени и техните свойства има в [76, Глава 35].

Префиксната нотация “ $+$ ($\dots$, $\dots$)” в алгоритъма MATIX POWER може да изглежда странно, понеже по правило събирането се записва с инфиксна нотация, примерно “ $a + b$ ”. Префиксната нотация тук е използвана, за да се подчертае аналогията с “ $\min(\dots, \dots)$ ” в алгоритъма SHORTEST PATH MM. Минимумът винаги се пише с префиксна нотация като “ $\min(a, b)$ ” и никога с инфиксна нотация като “ $a \min b$ ”.

[†]В алгебрата, полупръстен е структура, подобна на пръстен, но без изискването за обратен елемент на абстрактното събиране. В този случай абстрактното събиране е операцията $\min$, която наистина няма обратен елемент.

И така, SHORTEST PATH MM реализира матрично умножение, като

$$\begin{aligned} D^{(1)} &= W \\ D^{(2)} &= W^2 \\ &\dots \\ D^{(n-1)} &= W^{n-1} \end{aligned}$$

Подобряване на сложността по време до $\Theta(n^3 \lg n)$. Да кажем, че се налага да извършим умножения на матрица A със себе си, така че да получим A^m . Очевидният начин е да използваме $\Theta(m)$ матрични умножения ето така:

$$\begin{aligned} A^2 &\leftarrow A \cdot A \\ A^3 &\leftarrow A^2 \cdot A \\ A^4 &\leftarrow A^3 \cdot A \\ &\dots \\ A^m &\leftarrow A^{m-1} \cdot A \end{aligned}$$

Може обаче да постигнем същия резултат със само $\Theta(\lg m)$ матрични умножения:

$$\begin{aligned} A^2 &\leftarrow A \cdot A \\ A^4 &\leftarrow A^2 \cdot A^2 \\ A^8 &\leftarrow A^4 \cdot A^4 \\ &\dots \\ A^{m/2} &\leftarrow A^{m/4} \cdot A^{m/4} \\ A^m &\leftarrow A^{m/2} \cdot A^{m/2} \end{aligned}$$

Тази техника се нарича *повтаряне на повдигането на квадрат*, на английски, *repeated squaring*. На пръв поглед, това работи толкова чисто и елегантно само когато m е точна степен на двойката; в противен случай, предпоследната матрица ще има в степения показател число, по-малко от m , а последната, число, по-голямо от m .

Ключовото наблюдение е, че по отношение на повдигането на квадрат на матрицата, съдържаща дължините на най-къси пътища с горна граница върху броя на ребрата, където под “умножение на матрици” разбираме умножението от SHORTEST PATH MM, ако входната матрица (тоест, първата степен) няма отрицателни цикли, то след краен брой умножения със себе си, резултатът ще започне да се повтаря. Причината е, че W^m има смисъл именно на дължини на най-къси пътища с не повече от m ребра. При липса на отрицателни цикли, всеки най-къс път е прост (Наблюдение 69) и съдържа най-много $n - 1$ ребра. Ерго, може W^{n-2} да е равна или да не е равна на W^{n-1} , но със сигурност

$$W^{n-1} = W^n = W^{n+1} = W^{n+2} = \dots$$

Оттук следва, че ако искаме да получим W^{n-1} с повтаряне на повдигането на квадрат, няма да сбъркаме, ако получим някоя от W^n , W^{n+1} , W^{n+2} и така нататък, защото всички те са равни на W^{n-1} .

И така, търсим най-малката точна степен на двойката, по-голяма или равна на $n - 1$, за да я сложим в степения показател на последната матрица от серията. Тази степен на

двойката е $2^{\lceil \lg(n-1) \rceil}$. Сега ще обосновем това. Тръгвайки от очевидните неравенства

$$\begin{aligned} \lceil p \rceil &\geq p, & \forall p \in \mathbb{R} \\ \lceil p \rceil - 1 &< p, & \forall p \in \mathbb{R} \end{aligned}$$

имаме за $x > 0$

$$\begin{aligned} \lceil \lg x \rceil &\geq \lg x &\leftrightarrow & 2^{\lceil \lg x \rceil} \geq 2^{\lg x} = x &\leftrightarrow & 2^{\lceil \lg x \rceil} \geq x \\ \lceil \lg x \rceil - 1 &< \lg x &\leftrightarrow & 2^{\lceil \lg x \rceil - 1} < 2^{\lg x} = x &\leftrightarrow & 2^{\lceil \lg x \rceil - 1} < x \end{aligned}$$

Замествайки x с $n - 1$, получаваме

$$\begin{aligned} 2^{\lceil \lg(n-1) \rceil} &\geq n - 1 \\ 2^{\lceil \lg(n-1) \rceil - 1} &< n - 1 \end{aligned}$$

И така, търсената степен, на която е достатъчно да повдигнем W , за да получим желанния резултат, е $2^{\lceil \lg(n-1) \rceil}$.

Всичко това ни позволява да подобрим сложността на намирането на най-къси пътища чрез матрично умножение до $\Theta(n^3 \lg n)$. Разбира се, ако искаме ефикасна реализация, няма да започваме от единичната матрица, за да получим от нея W ; матрицата W е входът, така че ние разполагаме с нея. В псевдокода на SHORTEST PATH MM на стр. 560 започнахме от единичната матрица $D^{(0)}$ от съображения за елегантност. Тук започваме от W от съображения за ефикасност. Друга оптимизация, която можем да направим, е да не инициализираме $D^{(k)}[i, j]$ (която сега наричаме $\text{tmp}[i, j]$) с ∞ , а директно с $W[i, 1] + W[1, j]$, и да започнем най-вътрешния цикъл от $\ell = 2$, ако предпочитаме ефикасността пред теоретичната елегантност.

SHORTEST PATH MM, REPEATED SQUARING(W)

```

1  for k ← 1 to ⌈lg(n - 1)⌉
2      for i ← 1 to n
3          for j ← 1 to n
4              tmp[i, j] ← W[i, 1] + W[1, j]
5              for ℓ ← 2 to n
6                  tmp[i, j] ← min(tmp[i, j], W[i, ℓ] + W[ℓ, j])
7      W ← tmp
8  return W
```

Инвариант за най-външния цикъл е, всеки път, когато изпълнението е на ред 1, ако $\hat{W}$ означава входната матрица, то $W = \hat{W}^{2^{i-1}}$ за текущата W . Наистина, при първото достигане $i = 1$ и очевидно $W = \hat{W}^{2^{1-1}} = \hat{W}^{2^0} = \hat{W}^1$ в този момент; при всяко следващото достигане степенният показател очевидно се удвоява; при последното достигане $i = \lceil \lg(n - 1) \rceil + 1$, така че $W = \hat{W}^{2^{\lceil \lg(n-1) \rceil + 1 - 1}} = \hat{W}^{2^{\lceil \lg(n-1) \rceil}}$ в този момент, и алгоритъмът връща тази матрица.

Сложността по време е $\Theta(n^3 \lg n)$, понеже външният цикъл се изпълнява $\Theta(\lg n)$ пъти. Сложността по памет е $\Theta(n^2)$.

12.1.4.2 Алгоритъмът на Floyd-Warshall

Да характеризираме най-къс път $i \overset{p}{\rightsquigarrow} j$ по друг начин. Нека $\mathcal{U}$ е множеството от вътрешните върхове на p , като е възможно $\mathcal{U} = \emptyset$. Припомняме си, че в Подсекция 12.1.4, графът е

$G = (\{1, \dots, n\}, E)$. Тогава, ако p има вътрешни върхове, то p има максимален вътрешен връх, понеже всяко непразно $U \subset \{1, \dots, n\}$ има максимален елемент. А ако $U = \emptyset$, приемаме, че максималният вътрешен връх е 0.

И така, характеристиката сега е максималният вътрешен връх в пътя. Тази характеристика не е толкова очевидна, колкото характеристиката на стр. 558. Онази беше именно очевидна, защото пътят по дефиниция има ребра и техният брой е иманентен за пътя; броят на ребрата в пътя не зависи от номерацията на върховете на графа. От друга страна, какъв е максималният вътрешен връх в пътя зависи директно от номерацията на върховете. Това привидно внася неопределеност, тъй като дали даден връх е 1 или 100 зависи от “раздаването” на идентификатори на върховете. Ние обаче искаме алгоритъмът да връща един и същи резултат за всяка от $n!$ номерации на върховете на графа. Възможно ли е различни номерации на върховете на един и същи[†] граф да доведе до различни резултати[‡]? Отговорът е, че не може. Коректността на алгоритъма, който конструираме, не зависи от номерацията на върховете – той работи коректно при всяка номерация и сега ще се убедим в това.

Ето как можем да минем от характеристиката към рекурсивната декомпозиция. За всеки път p в G , нека “ $\text{intern}(p)$ ” означава множеството от вътрешните върхове на p .

Всеки най-къс път $i \rightsquigarrow^p j$ има множество от вътрешни върхове $\text{intern}(p)$, което може да е и празното множество. // това е тривиално ...

Нека $k = \max \text{intern}(p)$. Приемаме, че ако $\text{intern}(p) = \emptyset$, то $k = 0$, а **не** $k = -\infty$. Тук k се явява точната горна граница на $\text{intern}(p)$. // е, и?

Променяме леко формулировката: за някое $k \in \{0, \dots, n\}$ е изпълнено $\text{intern}(p) \subseteq \{1, \dots, k\}$. С други думи, k става горна граница, но не непременно точна, на $\text{intern}(p)$. // е, и?

Продължаваме: тогава или $k \in \text{intern}(p)$, или $k \notin \text{intern}(p)$. В първия случай p се състои от най-къс път $i \rightsquigarrow^q k$, чийто край k е “слепен” с началото k на най-къс път $k \rightsquigarrow^r j$; в този случай задължително е вярно, че $\text{intern}(q) \subseteq \{1, \dots, k-1\}$ и $\text{intern}(r) \subseteq \{1, \dots, k-1\}$. Във втория случай е вярно, че $\text{intern}(p) \subseteq \{1, \dots, k-1\}$. // аха, това е!

Ето подробна рекурсивна декомпозиция. Нека $i, j \in \{1, \dots, n\}$ са произволни. Разглеждаме $i \rightsquigarrow^p j$ с минимално тегло. Нека k е такова, че $\text{intern}(p) \subseteq \{1, \dots, k\}$.

- Ако $\text{intern}(p) = \emptyset$, то или $w(p) = 0$, ако $i = j$, или $w(p) = w((i, j))$, ако $i \neq j$ и $(i, j) \in E(G)$, или $w(p) = \infty$, ако $i \neq j$ и $(i, j) \notin E(G)$. С други думи, $w(p)$ е точно $W[i, j]$.
- Нека $\text{intern}(p) \neq \emptyset$.
 - ♦ Ако $k \in \text{intern}(p)$, то $w(p) = w(q) + w(r)$, където $i \rightsquigarrow^q k$ е път с минимално тегло, такъв че $\text{intern}(q) \subseteq \{1, \dots, k-1\}$ и $k \rightsquigarrow^r j$ е път с минимално тегло, такъв че $\text{intern}(r) \subseteq \{1, \dots, k-1\}$. В този случай казваме, че *връх k помага*.
 - ♦ Ако $k \notin \text{intern}(p)$, то $\text{intern}(p) \subseteq \{1, \dots, k-1\}$. С други думи, $w(p)$ е равно на теглото на най-къс път $i \rightsquigarrow j$, чийто вътрешни върхове са от $\{1, \dots, k-1\}$. В този случай казваме, че *връх k не помага*.

[†]Под “различни номерации на един и същи граф” имаме предвид следното. Представете си, че поначало върховете на графа са с някакви абсолютни идентификатори, примерно $v_1, \dots, v_n$, а после някаква биекция изобразява $\{v_1, \dots, v_n\}$ в $\{1, \dots, n\}$. Такава биекция е номерация. Очевидно има $n!$ различни номерации.

[‡]Става дума за алгоритъм, базиран на рекурсивна декомпозиция, която е базирана на характеристиката “максимален номер на вътрешен връх”.

Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)} \leftarrow W \quad (12.12)$$

$$D^{(k)}[i, j] \leftarrow \min \left\{ \underbrace{D^{(k-1)}[i, j]}_{k \text{ не помага}}, \underbrace{D^{(k-1)}[i, k] + D^{(k-1)}[k, j]}_{k \text{ помага}} \right\}, \text{ за } k > 0 \quad (12.13)$$

Забележете алгоритмичното предимство на (12.12) и (12.13) пред (12.10) и (12.11)! В (12.12) и (12.13) се пресмята минимумът на **2** числа, докато в (12.10) и (12.11) се пресмята минимумът на **n** числа.

Естествено, ако имплементираме (12.12) и (12.13) директно, ще получим рекурсивен алгоритъм с експоненциална сложност по време, защото при разклоненост 2 и височина $\Theta(n)$ на дървото на рекурсията, сложността по време е $\Omega(2^n)$. Но, също както при алгоритъма на стр. 560, ние няма да имплементираме (12.12) и (12.13) директно. За да получим решение, достатъчно е да пресметнем клетките на матриците $D^{(0)}, D^{(1)}, \dots, D^{(n)}$, всяка от които е $n \times n$, така че има $(n+1)n^2$ клетки за запълване. Очевидно правилният ред за пресмятане е този: $D^{(0)}$ е същата като W , $D^{(1)}$ се пресмята чрез $D^{(0)}$, $D^{(2)}$ се пресмята чрез $D^{(1)}$, и така нататък, $D^{(n)}$ се пресмята чрез $D^{(n-1)}$. Ерго, ще запълваме “стека от матрици” отдолу-нагоре, точно както диктува схемата **Динамично Програмиране**.

Всичко това води до следния кубичен алгоритъм за задачата APShP, който е публикуван от Robert Floyd [50], но се базира на теорема на Stephen Warshall [154].

FLOYD-WARSHALL(W : матрица $n \times n$, представяща тегловен ориентиран граф)

```

1   $D^{(0)} \leftarrow W$ 
2  for  $k \leftarrow 1$  to  $n$ 
3      for  $i \leftarrow 1$  to  $n$ 
4          for  $j \leftarrow 1$  to  $n$ 
5               $D^{(k)}[i, j] \leftarrow \min(D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j])$ 
6  return  $D^{(n)}$ 
```

Коректност и сложност по време и памет. Коректността на FLOYD-WARSHALL е очевидна, ако читателят е убеден в коректността на (12.12) и (12.13). Сложността му по време е $\Theta(n^3)$. Сложността му по памет, ако бъде реализиран по точно този начин, е $\Theta(n^3)$, но може лесно да бъде подобрена до $\Theta(n^2)$, избягвайки създаването на $n+1$ матрици и използвайки само две матрици, една, от която четем стойности (тя отговаря на $D^{(k-1)}$) и една, в която пишем (тя отговаря на $D^{(k)}$).

Заслужава да се отбележи, че всъщност може да минем със само една матрица, тоест, можем да пишем директно върху входа. Алгоритъмът ще продължи да работи коректно. Това е Задача 25.2-4 от [34, стр. 699]. Да видим защо е така. Да кажем, че използваме само една матрица D , която се инициализира с W . Същността на алгоритъма е

$$D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j]) \quad (12.14)$$

Ако $k \neq i$ и $k \neq j$, няма причина това присвояване да води до некоректно изчисление на новото $D[i, j]$. Да кажем БОО, че $k = i$. Тогава (12.14) става

$$D[i, j] \leftarrow \min(D[i, j], D[i, i] + D[i, j]) \quad (12.15)$$

Но $D[i, i]$ е задължително 0 при отсъствие на отрицателни цикли, така че (12.16) става

$$D[i, j] \leftarrow \min(D[i, j], D[i, j]) \quad (12.16)$$

което е същото като $D[i, j]$ да не се променя. Но това е коректно, защото дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от i (в момента сме допуснали, че $k = i$), е равна на дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от $i - 1$, защото пътищата са задължително прости (Наблюдение 69) и връх i вече се появява като начало на пътя, ерго, не може да е вътрешен връх. Но $D[i, j]$ преди изпълнението на (12.15) е именно дължината на най-къс път $i \rightsquigarrow j$, съдържащ вътрешен връх не по-голям от $i - 1$. С което решихме Задача 25.2-4 от [34, стр. 699].

И така, ако в началото на алгоритъма не копираме W в D , а работим **директно върху** W , ще реализираме алгоритъм с **константна сложност по памет**, иначе казано, in-place версия на **Floyd-Warshall**. Да си припомним (Подсекция 2.2.4), че при изследването на сложността по памет отчитаме само допълнителната памет, а входа игнорираме. Така че лесно може да направим версия на **Floyd-Warshall** със сложност по памет $\Theta(1)$, при условие, че сме готови да загубим входа, който ще се окаже презаписан.

Тежкият вариант. Разгледахме алгоритмите SHORTEST PATH MM и FLOYD-WARSHALL за APShP, но само за намиране на дължините на пътищата. Сега ще видим как да изчисляваме и самите пътища. Очевидно във варианта APShP няма да минем с $\Theta(n)$ памет за описание на пътищата, както беше във варианта SSShP, а ще ни трябва $\Theta(n^2)$ памет, за да опишем най-къси пътища по отношение на всички двойки върхове. А именно, ще използваме матрица $n \times n$, наречена “ Π ” в [34, стр. 695], със следния смисъл на елементите: $\Pi[i, j]$ съдържа предшественика на връх j по най-къс път от връх i . Тогава i -тият ред на Π описва дърво на най-къси пътища с корен връх i и можем да мислим за Π като за n масива на предшествие, каквито имахме в SSShP варианта, наредени един над друг.

Всъщност, алгоритъмът конструира редица от матрици $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$, по една за всяка стойност на k в FLOYD-WARSHALL, а Π е последната от тези матрици, тоест, $\Pi = \Pi^{(n)}$. Смисълът на елементите е следният: $\Pi^{(k)}[i, j]$ съдържа предшественика на връх j по най-къс път p с начало връх i , такъв че $\text{intern}(p) \subseteq \{1, \dots, k\}$. Ето рекурсията за тези матрици.

$$\Pi^{(0)}[i, j] = \begin{cases} \text{Nil}, & \text{ако } i = j \text{ или } W[i, j] = \infty \\ i, & \text{ако } i \neq j \text{ и } W[i, j] < \infty \end{cases}$$

$$\Pi^{(k)}[i, j] = \begin{cases} \Pi^{(k-1)}[i, j], & \text{ако } D^{(k-1)}[i, j] \leq D^{(k-1)}[i, k] + D^{(k-1)}[k, j] & // k \text{ не помага} \\ \Pi^{(k-1)}[k, j], & \text{ако } D^{(k-1)}[i, j] > D^{(k-1)}[i, k] + D^{(k-1)}[k, j] & // k \text{ помага} \end{cases}, \text{ за } k > 0$$

Следва псевдокод на FLOYD-WARSHALL, който пресмята и пътища.

FLOYD-WARSHALL PATHS(W : матрица $n \times n$, представяща тегловен ориентиран граф)

```

1  for i ← 1 to n
2    for j ← 1 to n
3       $D^{(0)}[i, j] \leftarrow W[i, j]$ 
4      if  $i = j$  or  $W[i, j] = \infty$ 
5         $\Pi^{(0)}[i, j] \leftarrow \text{Nil}$ 
6      else
7         $\Pi^{(0)}[i, j] \leftarrow i$ 
8  for k ← 1 to n
9    for i ← 1 to n
```



```

10      for j ← 1 to n
11          if  $D^{(k-1)}[i, j] \leq D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$ 
12               $D^{(k)}[i, j] \leftarrow D^{(k-1)}[i, j]$ 
13               $\Pi^{(k)}[i, j] \leftarrow \Pi^{(k-1)}[i, j]$ 
14          else
15               $D^{(k)}[i, j] \leftarrow D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$ 
16               $\Pi^{(k)}[i, j] \leftarrow \Pi^{(k-1)}[k, j]$ 
17  return  $(D^{(n)}, \Pi^{(n)})$ 

```

След като веднъж разполагаме с матрицата $\Pi = \Pi^{(n)}$, ето как може да получим най-късия път от i до j , който тя описва, ако има такъв, или информация, че път от i до j няма.

```

PRINT APSHP( $\Pi$ , i, j)
1  if i = j
2      print i
3  else
4      if  $\Pi[i, j] = \text{Nil}$ 
5          print no path
6      else
7          PRINT APSHP( $\Pi$ , i,  $\Pi[i, j]$ )
8          print j

```

Пример за работата на FLOYD-WARSHALL PATHS има в [34, стр. 696]. Там са показани всички матрици $D^{(0)}, \Pi^{(0)}, \dots, D^{(n)}, \Pi^{(n)}$, които се получават при работата на алгоритъма върху графа на стр. 690.

12.2 Фундамент

12.2.1 Задачи, подзадачи и подподзадачи

При динамичното програмиране често се говори за някаква *задача*, нейни *подзадачи* и техни *подподзадачи*. Това са директни преводи на съответно *problem*, *subproblems* и *subsubproblems*, термини, които се ползват широко в англоезичните учебници в обясненията за същността на схемата **Динамично Програмиране**. Ето цитат от учебника CLR [34, стр. 359]:

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ...

... dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.

Забележете, че това използване на термина *подзадача* е формално некоректно в контекста на Определение 1, според което изчислителна задача е двуместна релация $\Pi \subseteq \mathcal{I} \times \mathcal{S}$. Според Определение 1, Π е множество от наредени двойки. Щом Π е множество, нейна подзадача би трябвало да е някакво нейно подмножество. Ако сме приели Определение 1 и сме формални и педантични, трябва да мислим, че подзадача на Π е някакво множество $\Pi' \subseteq \Pi$.

Но всъщност се има предвид не подмножество, а нещо друго. Когато говорим за *задача* Π , нейни *подзадачи* и техни *подподзадачи* в Лекция 12, това е по отношение на някакъв екземпляр x на Π . “Задача” в този смисъл означава решаването на Π върху x . “Решаването”

е процесът[†], който генерира решение, за x . Нека $y_1, \dots, y_k$ са екземпляри, които се явяват *конституенти*[†] на x в някакъв смисъл. Тогава “подзадачите” са процесите, които решават P върху $y_1, \dots, y_k$. Нека $z_1, \dots, z_m$ са екземпляри, които се явяват конституенти, в някакъв смисъл, на екземплярите $y_1, \dots, y_k$. Тогава “подподзадачите” са процесите, които решават P върху $z_1, \dots, z_m$.

Приемайки тези определения, можем да кажем, че използваме техниката **Динамично Програмиране**, за да решим P , ако процесът, който генерира решение за x , ползва резултатите от процесите, които генерират решения за $y_1, \dots, y_k$. Процесите върху $y_1, \dots, y_k$ обаче не са независими, а ползват резултатите от процесите върху $z_1, \dots, z_m$. И така нататък, докато не се стигне до някакви начални условия. Близко е до ума, че за да бъде ефикасно цялото изчисление, тоталният брой на тези подзадачи–процеси, които възникват, трябва да е най-много полиномиален в размера на x .

Накратко: в тези лекционни записки *задача* е това, което трябва да се направи, а *алгоритъм* е механизъм, който го прави. Но в терминологията с *problems*, *subproblems* и *subsubproblems*, “problem” не е това, което трябва да се направи, а именно механизъм, който го прави. Авторът на записките съжалява за потенциалното объркване от това неконсистентно ползване на термините, но единствената смислена дефиниция на “изчислителна задача”, която вижда, е Определение 1, а учебникът CLR е написан според разбиранията на своите автори.

Забележете, че само свеждането на задачата до подзадачи ни дава алгоритъм по схемата **Разделяй-и-Владей**. Трябва подзадачите да имат общи подподзадачи, за да приложим схемата **Динамично Програмиране**. За приликите и разликите между **Разделяй-и-Владей** и **Динамично Програмиране** вижте Подсекция 12.2.5.

12.2.2 Конституенти и декомпозиция

Вече ползвахме понятието “конституенти”, но без дефиниция или добро обяснение. Ето какво имаме предвид. Дадена е изчислителна задача $P \subseteq \mathcal{I} \times \mathcal{S}$, за която искаме да конструираме ефикасен алгоритъм ALGX по схемата **Динамично Програмиране**. Нека $x \in \mathcal{I}$ е произволен екземпляр на задачата. *Непосредствените конституенти на x* ще наричаме екземпляри на задачата, от решенията за които можем да построим решение за x по ефикасен и непосредствен начин. Всеки от тях трябва да е строго по-малък от x и да се съдържа строго в x , в някакъв смисъл. Примерно, в задачата за опашката пред касата, непосредствените конституенти на $\langle h_1, \dots, h_n \rangle$ са $\langle h_1, \dots, h_{n-1} \rangle$ и $\langle h_1, \dots, h_{n-2} \rangle$, а в задачата VERTEX COVER върху дървета (Секция 12.8), непосредствените конституенти на кореновото дърво, което е екземплярът, са поддърветата, вкоренени в децата на корена.

В определението на “непосредствени конституенти” има доза условност, защото “ефикасно и непосредствено” може да означава какво ли не. В случая ползваме здрав разум, който ни казва, че в задачата за опашката пред касата, можем да построим по ефикасен и непосредствен начин решение за $\langle h_1, \dots, h_n \rangle$, ползвайки решения за $\langle h_1, \dots, h_{n-1} \rangle$ и $\langle h_1, \dots, h_{n-2} \rangle$, но само решение за $\langle h_1, \dots, h_{n-1} \rangle$ не е достатъчно, за да изчислим директно решение за $\langle h_1, \dots, h_n \rangle$. От друга страна, решение за $\langle h_1, \dots, h_{n-3} \rangle$ не ни трябва, ако имаме решения за $\langle h_1, \dots, h_{n-1} \rangle$ и $\langle h_1, \dots, h_{n-2} \rangle$. Ерго, наистина непосредствените конституенти са само $\langle h_1, \dots, h_{n-1} \rangle$ и $\langle h_1, \dots, h_{n-2} \rangle$.

[†]Въпросният процес не е същото като алгоритъм, решаващ P . Алгоритъм за P е нещо общо, което работи коректно върху всеки екземпляр на P , докато въпросният процес е работата (на алгоритъма) именно върху x .

[†]“Конституенти” означава “съставни части”. Обяснено е подробно в Подсекция 12.2.2.

Определение 94: Конституенти

В текущия контекст, нека $\mathcal{C}^0(\mathbf{x}) = \{\mathbf{x}\}$ и за $i > 0$,

$$\mathcal{C}^i(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{y} \text{ е непосредствен конституент на някой елемент на } \mathcal{C}^{i-1}(\mathbf{x})\}$$

Множеството от конституентите на $\mathbf{x}$ е $\mathcal{C} = \bigcup_{i=0}^t \mathcal{C}^i(\mathbf{x})$, където t е максималното число, за което множеството $\mathcal{C}^t(\mathbf{x})$ има смисъл.

Тъй като размерът на $\mathbf{x}$ е краен и елементите на $\mathcal{C}^i(\mathbf{x})$ са със строго по-малък размер от елементите на $\mathcal{C}^{i-1}(\mathbf{x})$, ясно е, че след краен брой итерации конституентите ще станат атомарни в смисъл, че нямат конституенти на свой ред.

Примерно, в задачата за опашката за билети пред касата, множеството от конституентите на $\langle h_1, \dots, h_n \rangle$ е

$$\mathcal{C} = \{\langle h_1, \dots, h_n \rangle, \langle h_1, \dots, h_{n-1} \rangle, \langle h_1, \dots, h_{n-2} \rangle, \dots, \langle h_1, h_2 \rangle, \langle h_1 \rangle\} \quad (12.17)$$

Очевидно конституентите, наредени по подходящ начин, представляват някаква *декомпозиция*. Като пример, алгоритъмът ALG QUEUE PROCESSING на стр. 545 ползва неявно множеството $\mathcal{C}$ от (12.17), наредено по очевидния начин, и по този начин реализира удачна декомпозиция.

В задачата с опашката пред касата множеството от конституентите има мощност $\Theta(n)$. В задачата MATRIX-CHAIN MULTIPLICATION (Задача 32), която предстои да разгледаме, множеството от конституентите има мощност $\Theta(n^2)$. А може ли за задачата с опашката пред касата множеството от конституентите да е с квадратична мощност, примерно

$$\{\langle h_i, \dots, h_j \rangle \mid 1 \leq i \leq j \leq n\}$$

Отговорът е, че може, но няма смисъл. Щом имаме линеен алгоритъм, базиран на декомпозиция с линеен размер, защо да въвеждаме декомпозиция с квадратичен размер, която би довела до (поне) квадратичен алгоритъм? Декомпозицията при MATRIX-CHAIN MULTIPLICATION е с квадратичен размер, защото няма, или поне не е известна, декомпозиция със субквадратичен размер, водеща до субквадратичен алгоритъм.

12.2.3 Състояния

Решението на задачата за опашката пред касата ползва декомпозиция, за която можем да кажем, че е основана на множеството от конституентите, наредено по подходящ начин. Сега ще се убедим, че в други задачи не трябва да мислим за множеството от конституентите при дизайна на декомпозицията, защото конституентите са необятно много.

Да си припомним APShP и ефикасните алгоритми за нея от Подсекция 12.1.4. Общият екземпляр на задачата във варианта APShP е $\mathbf{x} = \langle G, w \rangle$, където $G = (V, E)$ е ориентиран граф, а $w : E \rightarrow \mathbb{R}$ е тегловна функция. Кое е множеството $\mathcal{C}$ от конституентите на $\mathbf{x}$ съгласно Определение 94? Има смисъл да кажем, че конституентите са ориентираните пътища в G . За да се убедим в това, достатъчно е да разгледаме G като обединение на пътища; ако гледаме на пътищата като на графи и

$$\mathcal{P} = \{p \mid p \text{ е път в } G\}$$

то имаме право да кажем, че

$$G = \bigcup_{p \in \mathcal{P}} p$$

Тук няма опит да дефинираме G ; това би била порочно зациклена дефиниция. Казваме само, че можем да гледаме на G като на обединение на пътища. И така, $\mathcal{C}$ е $\mathcal{P}$.

Както знаем, в най-лошия случай ориентираните пътища са поне експоненциално много в броя на върховете. Имаме право да кажем, че множеството от конституентите $\mathcal{C}$ е необятно голямо.

Може да опитаме да намалим големината на $\mathcal{C}$, като за всеки два върха s и t , $\mathcal{C}$ съдържа само оптималните пътища от s до t . Но примерът от Допълнение 51 показва, че тези пътища и подпътища може да са експоненциално много. Следващата стъпка към получаване на множество, върху което да базираме декомпозицията и което да има мощност, не по-голяма от полиномиална в n , е Наблюдение 73.

Наблюдение 73

В текущия контекст, ако искаме само оптималните цени, за всеки $i, j \in V(G)$ може да помним/съхраняваме само оптималната цена на път от i до j .

Наблюдение 73 ни позволява да мислим за *състояния*, които са наредени двойки от върхове, наместо за конституенти-пътища. Но това не е достатъчно. За да имаме смислена декомпозиция, трябва да имаме мярка за големина на състояние, понеже изчисляваме цените на големите състояния от цените на малките състояния, грубо казано. Тъй като задачата е за пътища, най-естествената такава мярка за (i, j) , където $i, j \in V(G)$, е броят на ребрата в оптимален път $i \rightsquigarrow j$. При тази мярка, състоянията стават наредени тройки (i, j, k) , където $i, j \in V(G)$, а $k \in \{0, \dots, n-1\}$ е горна граница за броят на ребрата в оптимален път $i \rightsquigarrow j$. Но това е точно подходът от Подподсекция 12.1.4.1. Всяко състояние има цена, която е цената на кое да е оптимално решение (оптимален път), отговарящо на това състояние. Състоянията са само $\Theta(n^3)$ на брой, което е абсолютно съществено за конструирането на алгоритъма SHORTEST PATH MM на стр. 560.

Заслужава си да се повтори: конституентите в задачата за най-късите пътища са необятно много в общия случай, така че не можем ползваме тях за рекурсивна декомпозиция, от която да направим ефикасен алгоритъм; не и директно във всеки случай. Ключовото наблюдение е, че **ако минем от конституенти към състояния** от вида (i, j, k) , можем да ползваме състоянията, за да направим рекурсивна декомпозиция, която дава ефикасен алгоритъм, понеже състоянията са сравнително малко. Теорема 96 ни дава право да минем от конституенти-пътища към състояния (i, j, k) .

- Ако искаме да решим задачата в олекотения вариант, то при дадени два върха i и j и дадена горна граница k за броя на ребрата може изобщо да не записваме никой най-къс път $i \rightsquigarrow j$ с $\leq k$ ребра. Напълно е достатъчно да запишем само цената на такъв път. И наистина, $D^{(k)}[i, j]$ в (12.10) и (12.11) съдържа точно тази информация. Теорема 96 гарантира, че всеки оптимален път от този вид “върши работа”, така че самото “съдържание” на пътя може спокойно да се игнорира.
- Ако искаме да решим задачата в тежкия вариант, достатъчно е да запишем кой да е най-къс път $i \rightsquigarrow j$ с $\leq k$ ребра. Отново, Теорема 96 гарантира, че всеки оптимален път от този вид “върши работа”.

Забележете, че алгоритъмът SHORTEST PATH MM “обработва” състоянията в реда на нарастване на третите им елементи.

Наблюдение 74

Алгоритъмът SHORTEST PATH MM не обработва състоянията в реда на цените им.

Това не е така при всички алгоритми! Припомнете си Следствие 30, което казва, че алгоритъмът на Dijkstra за най-късите пътища обработва върховете по реда на претеглените дължини на оптималните пътища от началния връх; с други думи, в реда на цените им.

Алгоритъмът на Floyd-Warshall от Подподсекция 12.1.4.2 също ползва, поне концептуално, състояния, чийто брой е $\Theta(n^3)$. Те са наредени тройки (i, j, k) с различен смисъл: i и j пак са върхове, но $k \in \{0, \dots, n\}$ тук има смисъл на максимален вътрешен връх. И тук има смисъл да кажем, че алгоритъмът “обработва” състоянията в реда на нарастване на третите елементи на състоянията.

Заслужава си да се помисли върху следното. В задачата с опашката пред касата декомпозицията е директно чрез конституентите, а в APShP декомпозицията е чрез състоянията, които се явяват някакви абстракции на конституентите. Може ли и в задачата с опашката пред касата да въведем състояния? Отговорът е, че може, но няма смисъл. В задачата с опашката пред касата състоянията биха били числата $n, n-1, \dots, 2, 1$, които отговарят биективно съответно на конституентите $\langle h_1, \dots, h_n \rangle, \langle h_1, \dots, h_{n-1} \rangle, \dots, \langle h_1, h_2 \rangle, \langle h_1 \rangle$. Въвеждането на състояния не би подобрило с нищо ефикасността на декомпозицията и оттам на алгоритъма. Напълно аналогично, в задачата с боядисването на къщите конституентите са n на брой и състоянията са n на брой, така че е безсмислено да се въвежда понятието “състояние”. И при изчисляването на числата на Fibonaccі няма смисъл да се въвеждат състояния – там състоянията са самите числа на Fibonaccі.

APShP е качествено различна от задачата с опашката пред касата и от задачата за боядисването на къщите, защото APShP е задача върху общи графи. И задачата с опашката пред касата, и задачата за боядисването на къщите може да се формулират като графови задачи[†], но графът в тях е изключително прост: той е граф-път. Графът-път е най-простият възможен свързан граф: връх едно е съседен на връх две, връх две е съседен на върхове едно и три, връх три е съседен на върхове две и четири, и така нататък. Това, което прави APShP по-трудна (и по-интересна) от опашката пред касата и боядисването на къщите, е че в графа на APShP съседствата[‡] са произволни. Това е фундаменталната причина за наличието на необятно много конституенти в APShP, което на свой ред налага декомпозицията да се прави по абстракции на конституентите, които абстракции наричаме “състояния”.

12.2.4 Идентифициране на съвпадащи състояния – от дърво на рекурсията към даг на рекурсията

Същността на схемата Динамично Програмиране е ефикасно пресмятане на рекурсия. Както знаем, всяка рекурсия има свое кореново дърво. Да мислим за това дърво като за

[†]Ето как може да формулираме задачата с опашката пред касата като графова задача: даден е граф-път, в който и върховете, и ребрата имат тегла. Върховете са хората, ребрата са съседствата в опашката, теглата на върховете са времената за индивидуалните минавания, а теглата на ребрата са времената за комбинираните минавания. Търси се такова съчетание, че сумата от теглата на неговите ребра плюс сумата от теглата на несъчетаните върхове е минимална. “Съчетание” е дефинирано и обяснено подробно в Подподсекция 16.2.7.1.

[‡]Съседства, ако графът е неориентиран; в ориентираните графи на APShP не говорим за съседни върхове, а за родители и деца. Това обаче е без значение за основната мисъл тук.

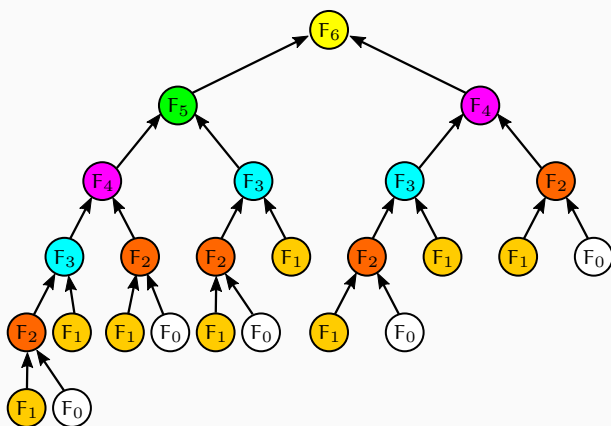
антиарборесценция. Всеки връх на дървото си има състояние – понятие, което въведохме в Подсекция 12.2.3; това е информация, която напълно идентифицира решението в този връх (тоест, изходът от това рекурсивно викане), позволявайки то (въпросното решение) да се ползва ефикасно от рекурсивни викания на по-високи нива. Когато имплементираме алгоритъма с таблица, от която вземаме наготово вече изчислени решения на подзадачи, състоянията са координатите в таблицата.

Нека отново разгледаме като пример задачата APSHP с рекурсивната декомпозиция (12.10), (12.11). Да си представим дървото на тази рекурсия. Неговите върхове са $\Omega(n^{n-1})$, което е “необятно много”. Но състоянията в рекурсивната декомпозиция са само n^3 . Всяко състояние се асоциира с цена, която е число. По-конкретно, състоянието (i, j, k) се асоциира с дължината на най-къс път с $\leq k$ ребра от i до j . Тези цени-дължини може да са произволно големи, защото зависят от теглата на ребрата, **но не може да са произволно много**. Броят на различните цени-дължини на пътища не е повече от n^3 . Очевидно, в който и връх на дървото на рекурсията да пресмятаме цената, съответна на (i, j, k) , тя ще е една и съща, понеже графът не се мени по време на изчисленията.

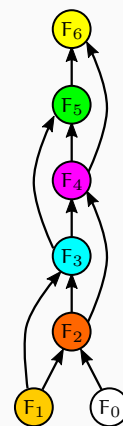
Това ни дава право да идентифицираме върховете с еднакви състояния в дървото на рекурсията. Заменяме всички върхове с еднакви състояния с един единствен връх и слагаме ребро (u, v) тстк в първоначалното дърво е имало ребро от връх в състояние u към връх в състояние v . С което получаваме даг, понеже рекурсията е такава: цената на състояние (i, j, k) , за $k > 0$, се пресмята чрез числата на $(\bullet, \bullet, k-1)$, а всяко състояние $(i, j, 0)$ е източник в дага.

Идентифициране на върхове с еднакви състояния в дървото на рекурсията, водещо до даг, се прави, поне на концептуално ниво, при конструирането на всеки алгоритъм по схемата **Динамично Програмиране**. Източниците на получения даг точно отговарят на върховете на дървото на рекурсията, съответни на началните условия. Фигура 12.1 илюстрира това идентифициране на върховете при пресмятането на числата на Fibonacci: дървото вляво е на рекурсията за пресмятането на F_6 от програмата на стр. 549, а дагът вдясно е на ефикасното пресмятане на F_6 от програмата на стр. 549.

Фигура 12.1 : Идентифициране на върхове в дърво на рекурсия.



Дървото на рекурсията за изчисляването на F_6 .



Идентифицираме върховете с еднакви състояния и получаваме даг.

Смисълът от това е очевиден. Дървото на рекурсията по правило е грамадно, с размер,

който е поне експоненциален в размера на входа. Поради това, ако изпълним рекурсията буквално, получаваме алгоритъм с поне експоненциална сложност. Във всички задачи, които решаваме ефикасно чрез **Динамично Програмиране**, броят на състоянията е най-много полиномиален[†] в размера на входа. Поради което въпросният даг има размер, който е само полиномиален в размера на входа. Ефикасността на алгоритмите по схемата **Динамично Програмиране** може да бъде обяснена с това, че основното в тяхната работа е, в някакъв смисъл, топо-сортиране на въпросния даг, последвано от обхождане в реда на топо-сортирането.

Наблюдение 75: Динамично Progr. и дагът, получен от дървото на рекурсията

Всеки алгоритъм, построен по схемата **Динамично Програмиране**, в някакъв абстрактен смисъл топо-сортира дага, получен от дървото на рекурсията след идентифицирането на върховете с еднакви състояния, и генерира решение, обхождайки този даг в реда на топо-сортирането.

Казваме “в някакъв абстрактен смисъл”, понеже нито строим дага експлицитно, нито го топо-сортираме експлицитно. Този даг е абстракция: мисловна конструкция, която въвеждаме само с учебна цел за по-добро осмисляне на предимствата на **Динамично Програмиране**.

За задачите, които са податливи на ефикасни алгоритмични решения по схемата **Динамично Програмиране**, е в сила следният принцип.

Определение 95: Принцип на оптималността

Дадена оптимизационна задача удовлетворява *принципа на оптималността*, ако по отношение на някаква рекурсивна декомпозиция, оптимално решение за даден екземпляр се изчислява от оптимални решения за подекземплярите, които са неговите конституенти във въпросната рекурсивна декомпозиция.

НАЙ-КЪСИ ПЪТИЩА удовлетворява принципа на оптималността по отношение на характеристиката брой на ребрата. Наистина, всеки оптимален път с не повече от k ребра се състои от оптимален подпът p с не повече от $k - 1$ плюс може би още едно ребро накрая. НАЙ-КЪСИ ПЪТИЩА удовлетворява принципа на оптималността и по отношение на характеристиката максимален вътрешен връх. Наистина, всеки оптимален път с вътрешен връх, не по-голям от k , се състои:

- или от един единствен оптимален път q с вътрешни върхове, не по-големи от $k - 1$,
- или от “слепването” на три обекта, а именно
 - ♦ оптимален път r с вътрешни върхове, не по-големи от $k - 1$,
 - ♦ връх k ,
 - ♦ и още един оптимален път s с вътрешни върхове, не по-големи от $k - 1$.

Интуитивно е ясно, че е така – и при двете характеристики само бихме загубили, ако използваме неоптимални пътища-конституенти p , q , r или s . Формалното доказателство е очевидно следствие от Теорема 96.

НАЙ-ДЪЛГИ ПЪТИЩА обаче не удовлетворява принципа на оптималността по отношение на характеристиката брой на ребрата, както става ясно от Допълнение 52.

[†]Или псевдополиномиален – вижте Определение 99.

12.2.5 За разликата между Разделяй-и-Владей и Динамично Програмиране

Има много задачи, за които можем да конструираме алгоритъм по схемата **Разделяй-и-Владей**, но не и по схемата **Динамично Програмиране**, защото подзадачите от фазата **Разделяй** нямат общи подподзадачи. Това е пълна сила MERGE SORT—типичен алгоритъм за схемата **Разделяй-и-Владей**—защото сортиранията на двата подмасива нямат нищо общо в смисъл, че сортирането на едната половина от масива няма общо със сортирането на другата половина от масива. Сортиранията на двата подмасива са едно и също действие, а именно сортиране, но в общия случай двете половини се състоят от различни числа и не можем да използваме едната от сортираните половини, за да си помогнем при сортирането на другата. Двете сортирания трябва да се случат поотделно и напълно независимо.

Друг пример е задачата за местенията на Ханойските кули, ако бъде формулирана по подходящ начин: при дадено n , да се генерира редицата от минимален брой ходове, която премества n диска от началния прът върху крайния прът с помощта на междинния прът. Известно е, че броят на тези ходове е $2^n - 1$. Може би трябва да се подчертае, че задачата не е да се изведе числото $2^n - 1$, а да се изведе **редицата от ходове**. Каква е сложността по време? Ако не отчитаме представянето на n , сложността по време не е дефинирана[†]. Ако n е представено в двоична позиционна бройна система, сложността е двойно експоненциална. Ако n е представено унарно, сложността е експоненциална. Да кажем, че n е представено унарно; примерно, дадени са самите дискове в началото – това на практика е същото като n да е записано в унарна бройна система. Знаем, че задачата за преместването на n диска се свежда до две премествания на $n - 1$ диска (плюс още едно местене на най-големия диск), но двете премествания на $n - 1$ диска трябва да се случат поотделно. Тези две премествания на $n - 1$ диска нямат обща част в смисъл, че те **и двете** трябва да се случат в своята пълнота и никаква част от първото преместване на $n - 1$ диска не може да замени част от второто преместване. Ерго, подзадачите нямат общи подподзадачи, в терминологията на Подсекция 12.2.1. И наистина, алгоритъмът не е по схемата **Динамично Програмиране**, а е чист **Разделяй-и-Владей** алгоритъм с експоненциална сложност по време, при казаните допускания.

По правило, алгоритмите по схемата **Разделяй-и-Владей** са по-лесни за конструране от алгоритмите по схемата **Динамично Програмиране**. Да бъде разделен входът на части, да се изпълни същият алгоритъм върху всяка от тях (стига да е достатъчно голяма) и от получените резултати да се намери търсеното решение е доста очевидна идея. Този подход е ползван още от древността и в никакъв случай не е достижение на модерната епоха. Да, ние видяхме както примери за доста неочевидни алгоритми по схемата **Разделяй-и-Владей** като алгоритъма за задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D от Подсекция 6.2.1 и алгоритъма SELECT от Подсекция 6.2.2, така и пример за напълно очевиден алгоритъм по схемата **Динамично Програмиране** като пресмятането на числа на Fibonacci с таблица. Но в общия случай, да се измисли алгоритъм по схемата **Динамично Програмиране** е доста по-трудно.

12.2.6 Какво казват експерти за Динамично Програмиране

Steve Skiena. Според [144, стр. 273–274]:

Once you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice. In fact, I find that dynamic programming algorithms

[†]Припомнете си Подсекция 2.2.5.

are often easier to reinvent than to try to look up in a book. That said, until you understand dynamic programming, it seems like magic. You must figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. The trick is seeing whether the naive recursive algorithm computes the same subproblems over and over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences. Dynamic programming is best learned by carefully studying examples until things start to click.

Последният параграф от цитата заслужава особено внимание. Skiena казва, че **Динамично Програмиране** е приложимо, когато има някакви обекти, подредени в линейна наредба, и тази наредба не се мени, а е фиксирана. Читателят ще забележи, че това е в сила за всички примери за удачно ползване на **Динамично Програмиране**, които ще разгледаме. Ако става дума за коренови дървета (за които пише и Skiena), линейната наредба е обхождането на върховете в postorder. Ако става дума за множество, което няма иманентна наредба, наредбата е произволна линейна наредба върху това множество (задачата KNAPSACK на стр. 631, примерно).

Обърнете внимание на последното изречение: иска се значителна практика, за да може дизайнерът на алгоритми да интернализира тази техника и да я прилага успешно.

Ето мнението на Steve Skiena за принципа на оптималността [144, стр. 302–303]:

Dynamic programming can be applied to any problem that observes the principle of optimality. Roughly stated, this means that partial solutions can be optimally extended with regard to the state after the partial solution, instead of the specifics of the partial solution itself.

...

Future decisions are made based on the consequences of previous decisions, not the actual decisions themselves.

Problems do not satisfy the principle of optimality when the specifics of the operations matter, as opposed to just the cost of the operations. . . . Properly formulated, however, many combinatorial problems respect the principle of optimality.

Формална дефиниция на “state” в този смисъл в [144] няма, но на стр. 284 става ясно какво има предвид Skiena под това понятие: става дума за описание на даден екземпляр, съдържащо необходимата и достатъчна информация за решение за него. Ерго, неговото “state” съвпада с нашето понятие “състояние” от Подсекция 12.2.3. Фразата “the specifics of the partial solution itself” може да се поясни с пример – при най-късите пътища, “the partial solution” е някакъв конституент-подпът, а “the specifics” е самият подпът като редица от върхове. “can be optimally extended with regard to the state after the partial solution, instead of the specifics . . .” ще рече, че можем да ползваме получената оптимална цена за състояние (i, j, k) , за да намерим оптималните цени за състоянията $(\bullet, \bullet, k + 1)$, без да знаем конкретни оптимални подпътища от i до j с $\leq k$ ребра – тези конкретики са **без значение** при намирането на оптималните цени за $(\bullet, \bullet, k + 1)$.

Jeff Erickson. Вижте мнението на още един експерт. Jeff Erickson е автор на отличен учебник с непретенциозното име “Algorithms” [45], който е публикуван онлайн с лиценз, позволяващ **безплатно и свободно сваляне**. Ето какво пише Jeff Erickson за същността на динамичното програмиране [45, стр. 105–106]:

In a nutshell, dynamic programming is recursion without repetition. Dynamic programming algorithms store the solutions of intermediate subproblems, often but not always in some kind of array or table. Many algorithms students (and instructors, and textbooks) make the mistake of focusing on the table—because tables are easy and familiar—instead of the much more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn’t really necessary, but if the recurrence is incorrect, we are well and truly hosed.

Dynamic programming is not about filling in tables. It’s about smart recursion!

И още [45, стр. 107–108]:

Warning: Greed is Stupid

If we’re incredibly lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a greedy algorithm. Like a backtracking algorithm, a greedy algorithm constructs a solution through a series of decisions, but it makes those decisions directly, without solving at any recursive subproblems. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are quite rare. Nevertheless, for many problems that should be solved by backtracking or dynamic programming, many students’ first intuition is to apply a greedy strategy.

...

Because the greedy approach is so incredibly tempting, but so rarely correct, I strongly advocate the following policy in any algorithms course, even (or perhaps especially) for courses that do not normally ask for proofs of correctness.

You will not receive any credit for any greedy algorithm, on any homework or exam, even if the algorithm is correct, without a formal proof of correctness.

Moreover, the vast majority of problems for which students are tempted to submit a greedy algorithm are actually best solved using dynamic programming. So I always offer the following advice to my algorithms students.

Whenever you write—or even think—the word “greeDY”, your subconscious is telling you to use DYnamic programming.

Even for problems that can be correctly solved by greedy algorithms, it’s usually more productive to develop a backtracking or dynamic programming algorithm first. First make it work, then make it fast.

Мнението, че алгоритмите по алчната схема са второ качество и използването им е лош навик, който трябва да се изкоренява, е екстремизъм, но основното в цитираните пасажии заслужава вниманието ни. Наистина, на биологическо ниво ние сме програмирани така, че търсим решението на цялата задачата в малки стъпки, на всяка от които търсим локално най-добро частично решение. Това е същността на алчната идея. Идеята на динамичното програмиране не ни “идва отвътре” и трябва да се култивира систематично, упорито и целенасочено. Когато конструираме алгоритъм по схемата **Динамично Програмиране**, първо и основно трябва да направим смислена рекурсивна декомпозиция; да започнем с таблицата е да сложим каруцата пред коня. Понятието *memoization* е обяснено в Секция 12.10.

Допълнение 57: За произхода на “Динамично Програмиране”

Интересен страничен въпрос е, защо тази схема се казва по този начин? Има ли статично програмиране, което да различаваме от динамичното? Какво изобщо означава “програмиране” тук? Отговорът е, че името е избрано от създателя на схемата Richard Bellman през 50-те години на 20 век заради привлекателността на “динамично” и прекалено общия смисъл. Bellman описва произхода на името със следния анекдот [12, стр. 59]:

An interesting question is, “Where did the name, dynamic programming, come from?” The 1950’s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defence, and he actually had a pathological fear and hatred of the word, research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place, I was interested in planning, in decision-making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying – I thought, let’s kill two birds with one stone. Let’s take a word which has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination which will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

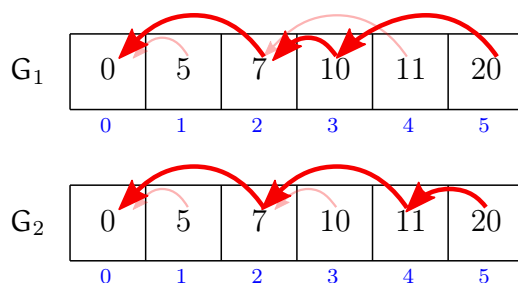
Статично програмиране, разбира се, няма, а “програмиране” през 50-те е означавало “систематично планиране или изпълнение”, за разлика от съвременния смисъл.

12.2.7 За конструирането на оптимално решение.

Най-често конструираме алгоритми по схемата **Динамично Програмиране**, решавайки оптимизационни задачи. Ако задачата е оптимизационна, тя има олекотен вариант, в който се иска само цената на оптимално решение, и тежък вариант, в който се иска и някое[†] оптимално решение. По правило първо конструираме ефикасен алгоритъм за олекотения вариант. След това лесно можем да модифицираме този алгоритъм, за да получим алгоритъм за тежкия вариант, който алгоритъм има същата сложност по време и памет. Тази модификация става по начин, аналогичен на модификацията на ALG QUEUE PROCESSING-1 в ALG QUEUE PROCESSING-2 в Подсекция 12.1.1.

1. Алгоритъмът за олекотения вариант неизбежно ползва таблица за съхранение на междинни резултати.
2. Във всяка клетка с на таблицата, отговаряща не на начално условие, слагаме указатели към клетките, които се ползват за непосредственото изчисление на c . По този начин конструираме ориентиран граф. Този граф е даг, защото наличието на цикъл би означавало рекурсията да зацikli. Този даг е *дагът на обратното проследяване*.
3. След намиране на оптималната цена, тя се съдържа в някоя клетка $\mathcal{O}$ на таблицата. Съвкупността от върховете на дага на обратното проследяване, достижими от клетка $\mathcal{O}$, дава едно оптимално решение.

Дагът на стр. 547 **не е** пример за даг на обратното проследяване, защото той е обединение на два дага на обратното проследяване, отговарящи на двете различни оптимални решения. Ето тези два дага; да ги наречем G_1 и G_2 . Във всеки от тях, ребрата от съответното оптимално решение са удебелени, а останалите ребра—които не са част от оптималното решение—са полупрозрачни.



Накратко, дагът на обратното проследяване трябва да съответства на едно единствено оптимално решение.

Всичко това е в сила само за оптимизационни задачи. Ако задачата е за търсене, примерно задачата за генериране на числата на Fibonacci или задачите в Секция 12.3, няма олекотен и тежък вариант и не се налага да генерираме оптимално решение с обратно проследяване.

[†]Както вече отбелязахме много пъти, множеството от всички оптимални решения в най-лошия случай е с мощност, поне експоненциална в размера на екземпляра, така че не можем да генерираме ефикасно всички решения.

12.3 Задачи за броене на комбинаторни структури

Много от фундаменталните комбинаторни структури като подмножества, разбивания и така нататък може да бъдат броени от алгоритми, основани на схемата **Динамично Програмиране**. Всеки такъв алгоритъм запълва таблица с числа съгласно някакви правила за рекурсивна декомпозиция и някакви начални условия.

Повечето задачи, които решаваме в Лекция 12, са оптимизационни. В контраст с това, задачите за броене на комбинаторни структури са задачи за търсене. При тях не говорим за олекотен вариант и тежък вариант. Освен това, те са и алгоритмично по-прости в смисъл, че състоянията (Подсекция 12.2.3) съвпадат с конституентите (Подсекция 12.2.2). При тези задачи цялата трудност е в комбинаториката – ако алгоритмичният дизайнер знае добре комбинаторната теория, конструирането на алгоритъм по схемата **Динамично Програмиране** е тривиално. В същината си това са комбинаторни задачи, макар и да са формулирани като задачи за съставяне на алгоритми.

Една забележка, отнасяща се до ефикасността на тези алгоритми. При тях входът се състои от две числа n и k , като $0 \leq k \leq n$. Както винаги, когато входът се състои от константен брой числа, възниква въпросът как дефинираме сложността. Очевидно не в големината на входа, която е $\Theta(1)$ [†]. Говорейки неформално, в този контекст “ефикасен алгоритъм” означава “алгоритъм, приемливо бърз на практика за разумно големи стойности на n и k ”. Това съдържа много субективизъм, но не е пълна безсмислица.

Формално прецизно би било да започнем да разглеждаме **размерите на кодирането на n и k** . Ако правим това и n и k са кодирани бинарно, алгоритмите биха били експоненциални. Ако обаче n и k са кодирани унарно, то алгоритмите, които предстои да разгледаме, са със сложност, не по-висока от квадратична.

12.3.1 Биномни коефициенти

Задача 59: Брой на подмножества

Конструирайте ефикасен алгоритъм, който по дадени n и k връща броя на k -елементните подмножества на n -елементно множество, където $k, n \in \mathbb{N}$.

Иска се да се сметне $\binom{n}{k}$. Нека $n \geq k$. Добре известно е, че $\binom{n}{k} = \binom{n}{n-k}$ и освен това

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k!} \quad (12.18)$$

$$\binom{n}{n-k} = \frac{n \cdot (n-1) \cdot \dots \cdot (k+1)}{(n-k)!} \quad (12.19)$$

Така че биномният коефициент може да бъде изчислен, като първо изчислим дали $k < n-k$ или не, и после го ползваме съответно (12.18) или (12.19).

Има обаче и друг начин да бъде изчислен биномен коефициент, който по същество е алго-

[†]Това прави сложността по време недефинирана или безкрайност; сравнете с Подсекция 2.2.5.

ритъм по схемата **Динамично Програмиране**. Следната формула е добре известна.

$$\binom{n}{k} = \begin{cases} 0, & \text{ако } k > n \\ 1, & \text{ако } k = 0 \text{ или } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{ако } 0 < k < n \end{cases} \quad (12.20)$$

Ето аргументация за коректност. Първо, тези възможности са очевидно изчерпателни и взаимно изключващи се. Началните условия се аргументират така.

- $k > n$ означава, че подмножеството има по-голяма мощност от тази на множеството; такова подмножество няма, затова там има нула;
- $k = 0$ означава празното множество като подмножество, а знаем, че празното множество е подмножество на всяко множество;
- $k = n$ означава, че вземаме като подмножество самото множество, а има един начин да сторим това.

Аргументацията за същинската рекурсия е, че при $0 < k < n$, фиксираме произволен елемент от n -елементното множество и разбиваме k -елементните подмножества на тези, които го съдържат, и тези, които не го съдържат; съответно $\binom{n-1}{k-1}$ и $\binom{n-1}{k}$ на брой.

(12.20) задава двумерна рекурсивна декомпозиция. Тя може да се ползва за bottom-up изчисление на биномните коефициенти, което запълва таблица, подобна на Таблица 12.1. И така, (12.20) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

$\begin{smallmatrix} k \\ n \end{smallmatrix}$	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0	0	0
5	1	5	10	10	5	1	0	0	0	0	0
6	1	6	15	20	15	6	1	0	0	0	0
7	1	7	21	35	35	21	7	1	0	0	0
8	1	8	28	56	70	56	28	8	1	0	0
9	1	9	36	84	126	126	84	36	9	1	0
10	1	10	45	120	210	252	210	120	45	10	1

Таблица 12.1: Таблица за изчисляване на биномни коефициенти.

Тази таблица е част от добре известния триъгълник на Pascal. Триъгълникът на Pascal е безкраен, а таблицата показва само “върха” му. Bottom-up означава[†], че се запълва ред по ред, като запълването на ред n става чрез изчисления, ползващи стойности от ред $n - 1$, за $n > 0$, точно както **Динамично Програмиране** изисква. Примерно, числото 84 на ред 9, колона 3, е равно на сумата над числото 56 над него и числото 28 над него вляво.

[†]По отношение на Таблица 12.1, изчислението, визуално говорейки, е отгоре надолу, но правилният термин е “bottom-up” въпреки това.


```

elif type(A, 'integer') and type(B, 'integer') then
  if 0 < A then
    if B < 0 or A < B then res := 0
    elif A < 2*B then res := binomial(a, a - b)
    elif 'binomial/a' = a and 'binomial/b' <= b then
      c := 'binomial/c';
      for i from 'binomial/b' + 1 to b do c := iquo((a - i + 1)*c, i) end do;
      'binomial/b' := b;
      'binomial/c' := c;
      res := 'binomial/c'
    elif 'binomial/a' = a then
      c := 'binomial/c';
      for i from 'binomial/b' by -1 to b + 1 do c := iquo(i*c, a - i + 1) end do;
      'binomial/b' := b;
      'binomial/c' := c;
      res := 'binomial/c'
    else
      c := 1;
      for i to b do c := iquo((a - i + 1)*c, i) end do;
      'binomial/a' := a;
      'binomial/b' := b;
      'binomial/c' := c;
      res := 'binomial/c'
    end if
  elif B < 0 and A < B then res := 0
  elif 0 < B then return NumericEvent('invalid_operation', (-1)^B*binomial(-AmB - 1, B))
  else return NumericEvent('invalid_operation', (-1)^AmB*binomial(-1 - B, -A - 1))
  end if
elif 'tools/type'(A, 'fraction') and 'tools/type'(B, 'integer') then
  if 'tools/type'(B < 0) then res := 0
  elif type(B, 'integer') then res := (AmB + 1)*binomial(A, B - 1)/B
  else res := 'binomial'(a, b)
  end if
elif type([A, B, AmB], ['fraction', 'fraction', 'integer']) then res := binomial(a, a - b)
elif 'tools/type'(A, 'negint') and type(B, 'complex(numeric)') then
  if type(B, 'integer') then
    ASSERT(not type(A, 'integer')); if B < 0 and 'tools/type'(AmB, 'negint') then res := 0 end if
  elif type(B, 'numeric') then
    return NumericEvent('real_to_complex', NumericEvent('division_by_zero', infinity + infinity*I))
  else return NumericEvent('division_by_zero', infinity + infinity*I)
  end if
elif type(2*B, 'odd') and type(A, 'integer') then
  s := A!/(GAMMA(B + 1)*GAMMA(AmB + 1));
  if has(s, 'GAMMA') then res := 'binomial'(a, b) else res := s end if
else
  s := signum(A);
  if member(s, [1, -1]) and 'tools/type'(AmB, 'negint') then
    i := signum(0, B, 0);
    if member(i, [1, -1]) then
      if s = 1 or s = i then res := 0
      else
        try if is(A, 'integer') and is(B, 'integer') then if i = -1 then res := 0 end if end if
        catch:
          end try
        end if
      end if
    elif s = 1 and 'tools/type'(B, 'negint') then res := 0
    end if
  end if
end if;
if not assigned(res) then
  if 'tools/type'(B, 'negint') then
    if coulditbe(A, 'negint') = false or coulditbe(AmB, 'nonnegint') = false then res := 0 end if
  elif 'tools/type'(AmB, 'negint') then
    if coulditbe(A, 'negint') = false or coulditbe(B, 'nonnegint') = false then res := 0 end if
  end if
end if;
if not assigned(res) then
  if 'tools/type'(A, 'negint') and coulditbe(B, 'integer') = false then
    if 'tools/type'(B, 'real') then
      return NumericEvent('real_to_complex', NumericEvent('division_by_zero', infinity + infinity*I))
    else return NumericEvent('division_by_zero', infinity + infinity*I)
  end if
end if;

```



```

        end if
    end if
end if;
if not assigned(res) then res := 'binomial(a, b)' end if;
if type(res, 'nonreal') and not hastype([args], 'nonreal') then NumericEvent('real_to_complex', res)
else res
end if
end proc

> quit
memory used=1.0MB, alloc=8.3MB, time=0.06

```

Редовете в червено са самото изчисление на $\binom{a}{b}$. Функцията `iquo`, от `integer quotient`, реализира целочислено деление: първият аргумент се дели на втория и функцията връща резултата. `for`-цикълът започва с $i = 1$, което е неявно, и след това извършва умножение на b частни. Това е по-икономично от първо намиране на числителя и знаменателя в (12.18), последвано от деление. Временният резултат $(a - i + 1) * c$ се дели на i винаги, така че няма загуба на прецизност заради целочисленото деление.

12.3.2 Числа на Stirling от втори род

Нотация 10: “ $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ ” означава числото на Stirling от втори род n -подмножество- k

Числото на Stirling от втори род n -подмножество- k бележим с $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$.

Задача 60: Брой на разбивания

Конструирайте ефикасен алгоритъм, който по дадени n и k връща броя на разбиванията на n -елементно множество на k дяла, където $k, n \in \mathbb{N}$.

Иска се да се сметне числото на Stirling от втори род $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$. За числата на Stirling от втори род е известна следната формула:

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = \frac{1}{k!} \sum_{j=0}^k (-1)^j \binom{k}{j} (k-j)^n \quad (12.21)$$

Аргументацията е проста: броят на разбиванията се получава от броя на сюрекциите от n -елементен домейн в k -елементен кодомейн чрез деление на $k!$; всяка сюрекция задава някакво разбиване на елементите на домейна на k дяла съобразно това, кои елементи от домейна върху кой елемент от кодомейна се изобразяват, а делим на $k!$, защото няма значение кой от тези дялове на точно кой елемент от кодомейна съответства.

Има и друг начин да се пресмятат числата на Stirling от втори род, който по същество е алгоритъм по схемата **Динамично Програмиране**. Следната формула е добре известна.

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}, & \text{ако } 0 < k < n \end{cases} \quad (12.22)$$

Ето аргументация за коректност. Първо, тези възможности са очевидно изчерпателни и взаимно изключващи се. Началните условия се аргументират така.

- $k = n$ означава, че дяловете са колкото елементите на множеството; броят на тези разбивания е 1, дори когато множеството е празното множество, а броят на дяловете е (съответно) нула;
- $k = 0, n > 0$ означава разбиване на нула дяла на непразно множество, което е невъзможно, така че има нула такива разбивания; $n < k$ означава повече (непразни) дялове, отколкото елементи, което също е невъзможно.

Аргументацията на същинската рекурсия е, че при $0 < k < n$, фиксираме произволен елемент от n -елементното множество и разбиваме разбиванията

- на тези, в които той е сам в дял, а те са $\{n-1\}$, защото те са колкото разбиванията на $(n-1)$ -елементно множество на $k-1$ дяла;
- и на тези, в които той бива “присъединен” към някой вече съществуващ дял на разбиване $(n-1)$ -елементно множество на k дяла, като броят на тези разбивания е $\{n-1\}_k$; има k възможности за това, къде да го “присъединим”.

(12.22) задава двумерна рекурсивна декомпозиция. Тя може да се ползва за bottom-up изчисление, което запълва таблица за изчисляване на числата на Stirling от втори род, подобна на Таблица 12.2. И така, (12.22) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

$\begin{smallmatrix} k \\ n \end{smallmatrix}$	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0	0	0
3	0	1	3	1	0	0	0	0	0	0	0
4	0	1	7	6	1	0	0	0	0	0	0
5	0	1	15	25	10	1	0	0	0	0	0
6	0	1	31	90	65	15	1	0	0	0	0
7	0	1	63	301	350	140	21	1	0	0	0
8	0	1	127	966	1701	1050	266	28	1	0	0
9	0	1	255	3025	7770	6951	2646	462	36	1	0
10	0	1	511	9330	34105	42525	22827	5880	750	45	1

Таблица 12.2: Таблица за изчисляване на числа на Stirling от втори род.

12.3.3 Числа на Stirling от първи род

Нотация 11: “ $\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right]$ ” означава числото на Stirling от първи род n -цикъл- k

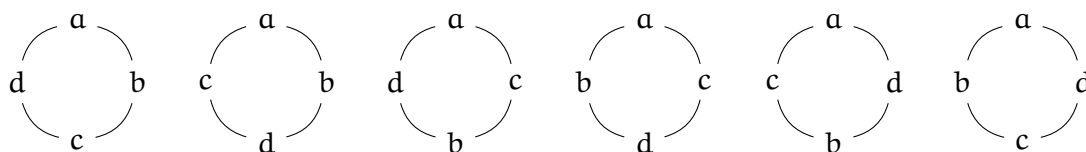
Числото на Stirling от първи род n -цикъл- k бележим с $\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right]$.

Задача 61: Брой на циклични наредби

Конструирайте ефикасен алгоритъм, който по дадени n и k връща броя на начините да бъдат разположени елементите на n -елементно множество в k цикъла, където $k, n \in \mathbb{N}$.

Да поясним какво означава да бъдат разположени елементите в цикли. Нека A е n -елементно множество. Става дума за разбиване на елементите на A на k дяла, като обаче след разбиването подреждаме елементите на всеки дял в кръгова наредба. Различните кръгови наредби на едни и същи елементи са тези, които не могат да се получат една от друга чрез ротация (но не и чрез рефлексия).

Като пример, да разгледаме $S = \{a, b, c, d\}$. Различните кръгови наредби на S са различните начини да бъдат подредени кръгово a, b, c и d . Те са точно шест на брой. Ето ги:

**Определение 96: Числа на Stirling от първи род**

Броят на начините да бъдат разположени елементите на n -елементно множество в точно k цикъла се нарича *число на Stirling от първи род n -цикъл- k* и се бележи с $\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right]$.

Ясно е, че в общия случай $\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right]$ е много по-голямо от $\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \}$. Примерно, $\{ \begin{smallmatrix} 4 \\ 1 \end{smallmatrix} \} = 1$, защото начините да разбием 4-елементно множество на един дял са само един (начин). Докато, както вече видяхме, $\left[\begin{smallmatrix} 4 \\ 1 \end{smallmatrix} \right] = 6$. Лесно се обобщава, че $\left[\begin{smallmatrix} n \\ 1 \end{smallmatrix} \right] = (n-1)!$ за всяко $n > 0$, докато $\{ \begin{smallmatrix} n \\ 1 \end{smallmatrix} \} = 1$ за всяко $n > 0$.

За числата на Stirling от първи род е известна формулата [32, стр. 51]:

$$\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right] = \sum_{0 \leq j \leq h \leq n-k} (-1)^{j+h} \binom{h}{j} \binom{n-1+h}{n-k+h} \binom{2n-k}{n-k-h} \frac{(h-j)^{n-k+h}}{h!} \quad (12.23)$$

Нейното извеждане обаче е много по-сложно от (12.18) или (12.21).

Има и друг начин да се пресмятат числата на Stirling от първи род, който по същество е алгоритъм по схемата **Динамично Програмиране**. Следната формула е добре известна.

$$\left[\begin{smallmatrix} n \\ k \end{smallmatrix} \right] = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ (n-1) \left[\begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right] + \left[\begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right], & \text{ако } 0 < k < n \end{cases} \quad (12.24)$$

Ето аргументация за коректност. Първо, тези възможности са очевидно изчерпателни и взаимно изключващи се. Началните условия се аргументират така.

- $k = n$ означава, че циклите са колкото елементите на множеството; броят на тези разполагания в цикли е 1, дори когато множеството е празното множество, а броят на циклите е (съответно) нула;

- $k = 0$, $n > 0$ означава разполагания в нула цикъла на непразно множество, което е невъзможно, така че има нула такива разбивания; $n < k$ означава повече цикли, отколкото елементи, което също е невъзможно.

Аргументацията на същинската рекурсия е, че при $0 < k < n$, фиксираме произволен елемент от n -елементното множество и разбиваме разполаганията

- на тези, в които той е сам в цикъл, а те са $\binom{n-1}{k-1}$, защото те са колкото разполаганията на $(n-1)$ -елементно множество в $k-1$ цикъла;
- и на тези, в които той бива “вмъкнат” в някой вече съществуващ цикъл на разполагане $(n-1)$ -елементно множество в k цикъла; щом елементите са $n-1$, то има точно $n-1$ места, в които той може да бъде вмъкнат, независимо от това, колко са циклите (тоест, колко е k), оттам идва и множителят $(n-1)$.

(12.24) задава двумерна рекурсивна декомпозиция. Тя може да се ползва за bottom-up изчисление на числата на Stirling от първи род, което запълва таблица, подобна на Таблица 12.3. И така, (12.24) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

$\begin{smallmatrix} k \\ n \end{smallmatrix}$	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0	0	0
3	0	2	3	1	0	0	0	0	0	0	0
4	0	6	11	6	1	0	0	0	0	0	0
5	0	24	50	35	10	1	0	0	0	0	0
6	0	120	274	225	85	15	1	0	0	0	0
7	0	720	1764	1624	735	175	21	1	0	0	0
8	0	5040	13068	13132	6769	1960	322	28	1	0	0
9	0	40320	109584	118124	67284	22449	4536	546	36	1	0
10	0	362880	1026576	1172700	723680	269325	63273	9450	870	45	1

Таблица 12.3: Таблица за изчисляване на числа на Stirling от първи род.

Има една съществена разлика между числата на Stirling от първи род, от една страна, и биномните коефициенти и числата на Stirling от втори род, от друга страна:

- докато биномните коефициенти и числата на Stirling от втори род се пресмятат по-бързо чрез формулите (12.18) и (12.21), съответно, отколкото чрез алгоритмите по схемата **Динамично Програмиране**, основани съответно на (12.20) и (12.22),
- то числата на Stirling от първи род се пресмятат по-бързо чрез алгоритъм по схемата **Динамично Програмиране**, основан на (12.24), отколкото чрез засуканата формула (12.23).

Можем да се убедим в това, ако разгледаме сорсовете на тези функции в Maple(TM). Идеята е, че разработчиците на Maple(TM) са достатъчно компетентни, за да имплементират по-бързия алгоритъм в своя продукт.

12.3.4 Брой на целочислени разбивания

Задача 62: Брой на целочислените разбивания

Конструирайте ефикасен алгоритъм, който по дадени n и k връща броя на целочислените разбивания на n на точно k части, където $k, n \in \mathbb{N}$.

Терминът “целочислено разбиване” въведохме в Определение 39. Частите на дадено целочислено разбиване на числото n са числата, които се сумират до n в него. Примерно, ако $n = 4$ и разбиването е $2 + 1 + 1$, частите са 2, 1 и 1.

Не трябва да се бърка “целочислено разбиване” с просто “разбиване”. Второто е по отношение на множества, първото е по отношение на естествени числа.

Традиционна нотация за броя на всички целочислени разбивания на n е “ $p(n)$ ”, а за броя на всички целочислени разбивания на n на точно k части е “ $p(n, k)$ ” или “ $p_k(n)$ ”. Тук ще ползваме нотацията, въведена от Knuth [95, стр. 399, (38)].

Нотация 12: “ $\left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right|$ ” означава броя на целочислените разбивания на n на k части

Броят на целочислените разбивания на n на точно k части бележим с $\left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right|$.

Както става ясно от примера на стр. 223, $\left| \begin{smallmatrix} 4 \\ 0 \end{smallmatrix} \right| = 0$, $\left| \begin{smallmatrix} 4 \\ 1 \end{smallmatrix} \right| = 1$, $\left| \begin{smallmatrix} 4 \\ 2 \end{smallmatrix} \right| = 2$, $\left| \begin{smallmatrix} 4 \\ 3 \end{smallmatrix} \right| = 1$ и $\left| \begin{smallmatrix} 4 \\ 4 \end{smallmatrix} \right| = 1$. Очевидно $\sum_{k=0}^n \left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right| = p(n)$.

За броя на целочислените разбивания е известна следната формула [95, стр. 399, (39)], която задава и алгоритъм по схемата **Динамично Програмиране** за тяхното изчисляване.

$$\left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right| = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ \left| \begin{smallmatrix} n-k \\ k \end{smallmatrix} \right| + \left| \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right|, & \text{ако } 0 < k < n \end{cases} \quad (12.25)$$

Ето аргументация за коректност. Първо, тези възможности са очевидно изчерпателни и взаимно изключващи се. Началните условия се аргументират така.

- $k = n$ означава, че частите са n на брой; има точно едно такова целочислено разбиване 1, дори когато $n = 0$ и $k = 0$;
- $k = 0$, $n > 0$ означава целочислено разбиване на положително число на нула части, което е невъзможно, така че има нула такива целочислени разбивания; $n < k$ означава повече части, отколкото е големината на числото, което също е невъзможно.

Аргументацията за същинската рекурсия е, че при $0 < k < n$, разбиваме целочислените разбивания на тези, които имат най-малка част 1, и тези, чиято най-малка част е по-голяма от 1.

- Броят на тези, които имат (поне една) част 1, е $\left| \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right|$, понеже съществува очевидна биекция между тях и всички целочислени разбивания на $n-1$ на точно $k-1$ части.
- Броят на тези, чиято най-малка част е поне 2, е $\left| \begin{smallmatrix} n-k \\ k \end{smallmatrix} \right|$. Причината е, че ако извадим единица от всяка част (което означава да извадим общо k единици), ще получим k

на брой части, всяка от които е поне 1, така че съществува очевидна биекция между целочислените разбивания на n на точно k части, всяка от които е поне 2, и всички целочислени разбивания на $n - k$ на точно k части.

Тъй като най-малката възможна част на всяко целочислено разбиване е единицата, можем да мислим за числото n като за множество от n на брой неразличими обекти, които можем да означаваме с точки. Тогава на всяко целочислено разбиване на n на точно k части съответства групиране на n точки в k групи, като за удобство тези групи са сортирани низходящо по големина. Това ни дава идеята за следното представяне на целочислени разбивания.

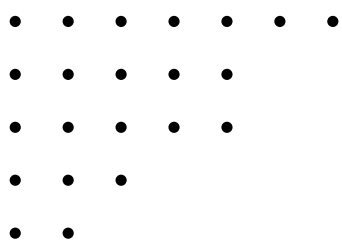
Определение 97: Диаграма на Ferrers

При дадени естествени n и k , такива че $1 \leq k \leq n$, на всяко целочислено разбиване $\alpha = \alpha_1 + \alpha_2 + \dots + \alpha_k$ на n съответства биективно диаграма, в която на първия ред има α_1 точки, на втория ред има α_2 точки, и така нататък, на k -ия ред има α_k точки, като във всеки ред разстоянията между точките са еднакви и редовете са подравнени вляво. Такава диаграма се нарича *диаграма на Ferrers*.

Ето пример за диаграма на Ferrers. Нека $n = 22$. Да разгледаме целочисленото разбиване:

$$\alpha = 7 + 5 + 5 + 3 + 2$$

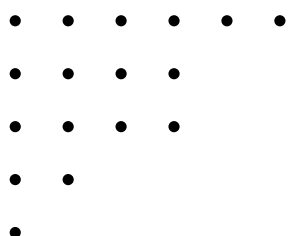
Съответната диаграма на Ferrers е



Диаграмите на Ferrers са много удобно средство за онагледяване на целочислени разбивания.

Да си припомним формулата $\left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right| = \left| \begin{smallmatrix} n-k \\ k \end{smallmatrix} \right| + \left| \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right|$ при $0 < k < n$. Както вече бе отбелязано,

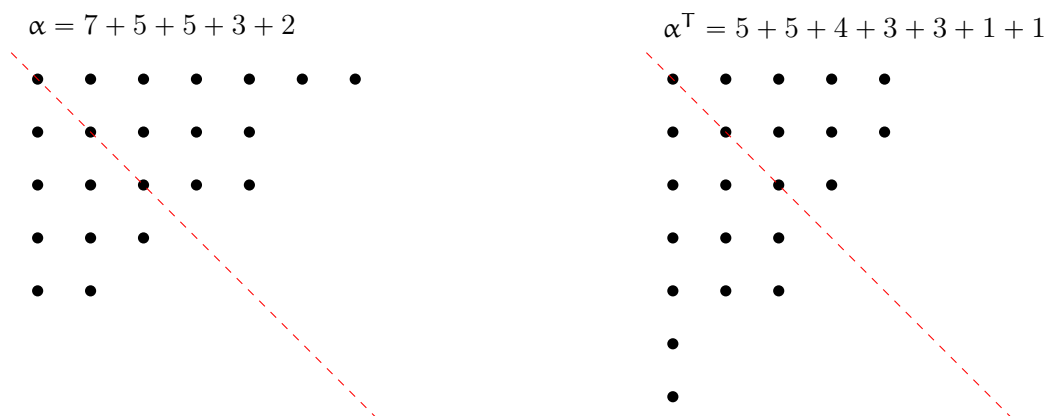
събираемостта $\left| \begin{smallmatrix} n-k \\ k \end{smallmatrix} \right|$ отговаря на разбиванията, чиято най-малка част е по-голяма от 1. Разбиването $\alpha = 7 + 5 + 5 + 3 + 2$ е точно такова. Тук $n = 22$ и $k = 5$. И наистина, ако разгледаме горната диаграма на Ferrers и изтрием по една точка от всеки ред, получаваме диаграма на съответното целочислено разбиване на $22 - 5 = 17$ на 5 части:



Броят на редовете очевидно се запазва, поради което долният индекс k остава непроменен, но k точки изчезват, поради което горният индекс става $n - k$.

Много полезно понятие е *спрегнатото целочислено разбиване*, на английски *conjugate integer partition* [95, стр. 394], което се илюстрира чудесно чрез диаграми. Спрегнатото разбиване

е това, чиято диаграма се получава от дадената чрез рефлексия по отношение на диагонала, минаващ през горния ляв ъгъл. Наместо “спрегнато”, можем да кажем “транспонирано” (сравнете с Определение 77). Като пример да разгледаме диаграмите на Ferrers на целочисленото разбиване $\alpha = 7 + 5 + 5 + 3 + 2$ на 22 и спрегнатото му целочислено разбиване $\alpha^T = 5 + 5 + 4 + 3 + 3 + 1 + 1$ (пак на 22, естествено).



Лесно се вижда, че най-голямата част на α , а именно 7, е равна на броя на частите на α^T . Конверсното също е вярно. Това очевидно може да се обобщи за всяко целочислено разбиване. Оттук заключаваме, че за всяко n и всяко k броят на целочислените разбивания на n на точно k части е равен на броя на целочислените разбивания на n , чиято най-голяма част е точно k .

Наблюдение 76

$\left| \begin{smallmatrix} n \\ k \end{smallmatrix} \right|$ може алтернативно да се дефинира като броят на целочислените разбивания на n , чиято най-голяма част е точно k .

(12.25) задава двумерна рекурсивна декомпозиция. Тя може да се ползва за bottom-up изчисление на броевете на целочислените разбивания, което запълва таблица, подобна на Таблица 12.4. И така, (12.25) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

$\begin{smallmatrix} k \\ n \end{smallmatrix}$	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0	0	0
3	0	1	1	1	0	0	0	0	0	0	0
4	0	1	2	1	1	0	0	0	0	0	0
5	0	1	2	2	1	1	0	0	0	0	0
6	0	1	3	3	2	1	1	0	0	0	0
7	0	1	3	4	3	2	1	1	0	0	0
8	0	1	4	5	5	3	2	1	1	0	0
9	0	1	4	7	6	5	3	2	1	1	0
10	0	1	5	8	9	7	5	3	2	1	1

Таблица 12.4: Таблица за изчисляване на броеве на целочислени разбивания.

Незначителна разлика между Таблица 12.1, Таблица 12.2 и Таблица 12.3 от една страна, и Таблица 12.4 от друга страна е, че

- при биномните коефициенти (уравнение (12.20) и числата на Stirling (уравнения 12.22 и 12.24) в същинската рекурсия, горният индекс вдясно винаги е $n - 1$, което означава, че в съответната таблица, ред n -ти се пресмята чрез ред $(n - 1)$ -ви,
- докато при броя на целочислените разбивания (уравнение 12.25), в същинската рекурсия, горният индекс вдясно е или $n - 1$, или $n - k$, откъдето следва, че ред n -ти се пресмята чрез ред $(n - 1)$ -ви и може би чрез други редове отгоре.

В Maple(TM) целочислените разбивания на числото n се намират с функцията `partition(n)`, която е част от пакета `combinat`. Функцията връща списък от самите разбивания, всяко от които на свой ред е сортиран списък от числата-части на разбиването. Ако искаме само броя на разбиванията, може да го получим с `numelems(partition(n))`.

Функцията `partition` може опционално да получи и втори аргумент, но, малко контраинтуитивно, `partition(n,k)` връща разбиванията, в които най-голямата част е **по-малка или равна** на k , а не е точно k . Ерго, ако искаме да изчислим $|P_k^n|$ с Maple, може да го направим с `numelems(partition(n,k)) - numelems(partition(n,k-1))`.

12.4 Задачи за оптимални скобувания на редици (Динамично Програмиране с триъгълна таблица)

В тази секция ще разгледаме задачи, във всяка от които присъства някаква линейна наредба[†], за която търсим оптимална декомпозиция, условно наречена “скобуване”. Наредбата не се мени по време на работата на алгоритъма.

Всички тези задачи се решават ефикасно с алгоритми по схемата **Динамично Програмиране**, ползващи двумерна квадратна таблица. В таблицата обаче се ползва (горе-долу) само половината над главния диагонал. Поради това понякога тези алгоритми се наричат и *алгоритми с триъгълна таблица*.

Същественото за тази секция е, че се иска оптимално скобуване. Този израз съдържа много условност, защото, примерно, при триангулацията на многоъгълниците, скобуване няма, поне не буквално. Винаги има обаче “главен срез” в редицата. Главният срез е или една “празнина” между два елемента на редицата, или един елемент от редицата, спрямо който се извършва последното действие (каквото и да е то). Скобуването се състои в това, че цялата редица се факторизира на две непразни подредици, те на свой ред се факторизират на по две непразни подредици, и така нататък, докато не бъдат достигнати началните условия.

[†]Да си припомним, че според цитата на Skiena на стр. 574, “Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order”.

12.4.1 Matrix-Chain Multiplication

Започваме с пример. Трябва да извършим матричното умножение $A \cdot B \cdot C \cdot D$, използвайки стандартния алгоритъм за умножение на матрици:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} 12 & -3 \\ -1 & 1 \\ 8 & 0 \\ -6 & 9 \\ 5 & 11 \end{bmatrix}}_B \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_C \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D \quad (12.26)$$

Един възможен начин да извършим това е:

$$\begin{aligned} A \cdot B \cdot C \cdot D &= \underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_C \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \\ &= \underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{((A \cdot B) \cdot C)} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{(((A \cdot B) \cdot C) \cdot D)} \end{aligned}$$

Да видим колко скалярни умножения са извършени при това матрично умножение. Ако умножим матрици $M_{x \times y} \cdot N_{y \times z}$, ние извършваме $x \cdot y \cdot z$ скалярни умножения. В случая, A е 2×5 , B е 5×2 , така че матричното умножение $(A \cdot B)$ “струва” $2 \cdot 5 \cdot 2 = 20$ скалярни умножения. $(A \cdot B)$ е 2×2 , а C е 2×6 , така че матричното умножение $((A \cdot B) \cdot C)$ “струва” $2 \cdot 2 \cdot 6 = 24$ скалярни умножения. $((A \cdot B) \cdot C)$ е 2×6 , а D е 6×2 , така че матричното умножение $((((A \cdot B) \cdot C) \cdot D))$ “струва” $2 \cdot 6 \cdot 2 = 24$ скалярни умножения. Като цяло, умножението на четирите матрици по този начин “струва” $20 + 24 + 24 = 68$ скалярни умножения.

Друг начин да умножим матриците е

$$\underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 30 & -328 \\ -56 & -44 \end{bmatrix}}_{(C \cdot D)} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{((A \cdot B) \cdot (C \cdot D))}$$

Крайният резултат е същият. Това не е изненада, понеже умножението на матрици е асоциативно. Но да видим колко скалярни умножения ни струва този начин на умножение на матриците. $(A \cdot B)$ струва, както вече видяхме, 20 скалярни умножения. $(C \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скалярни умножения. $((A \cdot B) \cdot (C \cdot D))$ струва $2 \cdot 2 \cdot 2 = 8$ скалярни умножения. Като цяло, броят на скалярните умножения е $20 + 24 + 8 = 52$. Кое е по-малко от 68.

Трети начин да умножим матриците е:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} 279 & -9 & -3 & -15 & -216 & 105 \\ -36 & -6 & 1 & 5 & 27 & -11 \\ 152 & -24 & 0 & 0 & -120 & 64 \\ -267 & -63 & 9 & 45 & 198 & -75 \\ -92 & -114 & 11 & 55 & 57 & 7 \end{bmatrix}}_{(B \cdot C)} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \\
 \underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{(A \cdot (B \cdot C))} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_D = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{((A \cdot (B \cdot C)) \cdot D)}$$

Крайният резултат отново е същият, но да видим колко скалярни умножения са извършени. В е 5×2 и С е 2×6 , така че $(B \cdot C)$ струва $5 \cdot 2 \cdot 6 = 60$ скалярни умножения. $(A \cdot (B \cdot C))$ струва $2 \cdot 5 \cdot 6 = 60$ скалярни умножения. $((A \cdot (B \cdot C)) \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скалярни умножения. Като цяло, броят на скалярните умножения е $60 + 60 + 24 = 144$.

Измерваме сложността на умножението на веригата от матрици с броя на скалярните умножения, понеже другата елементарна операция—събирането—е по-бърза. Заключаваме, че различните начини да бъде извършено умножението може да се различават значително като ефикасност, въпреки че крайният резултат е един и същи заради асоциативността на матричното умножение. Иначе казано, $((A \cdot B) \cdot C) \cdot D$, $((A \cdot B) \cdot (C \cdot D))$ и $((A \cdot (B \cdot C)) \cdot D)$ са математически неразличими като краен резултат, но алгоритмично, тоест, като сложност на изчислението, може да са доста различни. Причината е видима и в дадения пример: сложността на верижното умножение нараства, когато се появяват междинни матрици с големи размери, като $(B \cdot C)$ в примера. Този феномен се проявява силно, когато матриците-множители са далече от квадратни. Ако всички матрици-множители са квадратни, броят на скалярните умножения е един и същи при всеки начин на умножение и текущата задача се обезсмисля.

“Различните начини да бъде извършено умножението” е същото като “различните начини да бъде скобувана веригата”. При четири матрици има пет начина за това скобуване, като ние разгледахме само три тях в примера. Ето всичките пет начина:

- ① $((A \cdot B) \cdot C) \cdot D$
- ② $((A \cdot (B \cdot C)) \cdot D)$
- ③ $((A \cdot B) \cdot (C \cdot D))$
- ④ $(A \cdot ((B \cdot C) \cdot D))$
- ⑤ $(A \cdot (B \cdot (C \cdot D)))$

Броят на начините да бъде скобуван израз с n множители е C_{n-1} , където C_n означава *n-тото число на Catalan*. Първите няколко числа на Catalan са $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$. Числата на Catalan растат много бързо, като $C_n \asymp \frac{1}{n\sqrt{n}} 4^n$. Следователно, използването на алгоритъм с груба сила за задачата, която разглеждаме, е безсмислено.

Допълнение 59: За числата на Catalan

Да дефинираме n -тото число на Catalan C_n като броят на *балансираните изрази с $2n$ скоби*. Балансиран израз с $2n$ скоби е стринг, съдържащ точно n отварящи скоби “(” и точно n затварящи скоби “)”, като във всеки префикс, броят на отварящите скоби е по-голям или равен на броя на затварящите скоби.

Ето всички 5 балансирани изрази при $n = 3$:

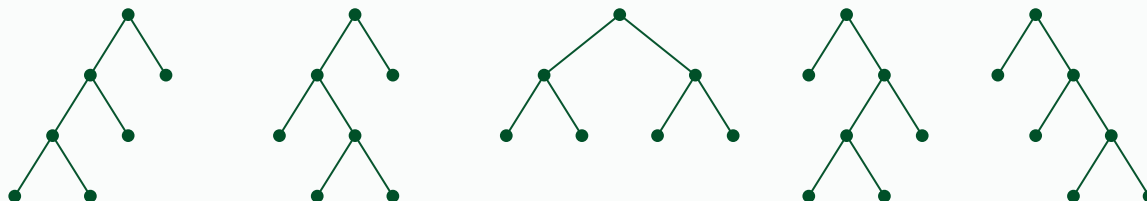
((())) ()(()) ()()() (())() ((()))

Забележете, че това не е същото като скобуване на израз с n множители. Тук единствените букви са скобите – множители няма.

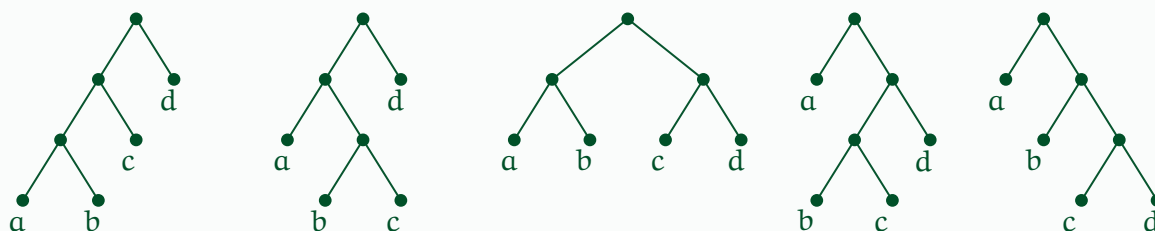
Лема 44

Съществува биекция между скобуваните изрази с $n + 1$ множители и балансираните изрази с $2n$ скоби.

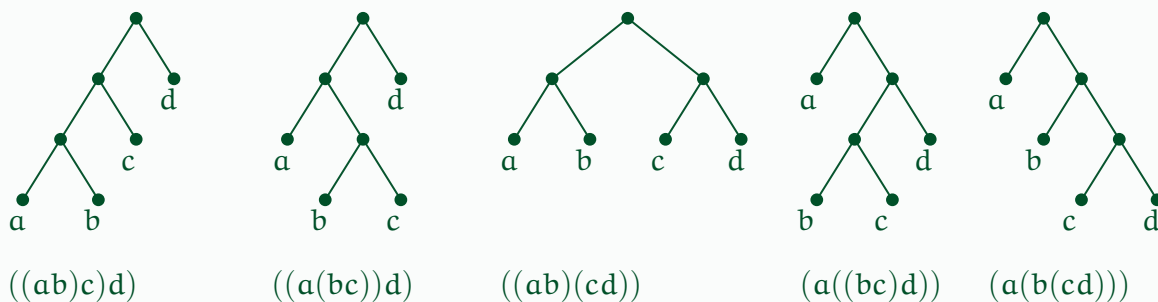
Доказателство: Първо ще покажем с пример, че съществува биекция между скобуваните изрази с $n + 1$ множители и пълните подредени двоични дървета с $n + 1$ листа. Нека $n = 3$. Тогава $n + 1 = 4$ и дърветата с анонимни върхове са тези:



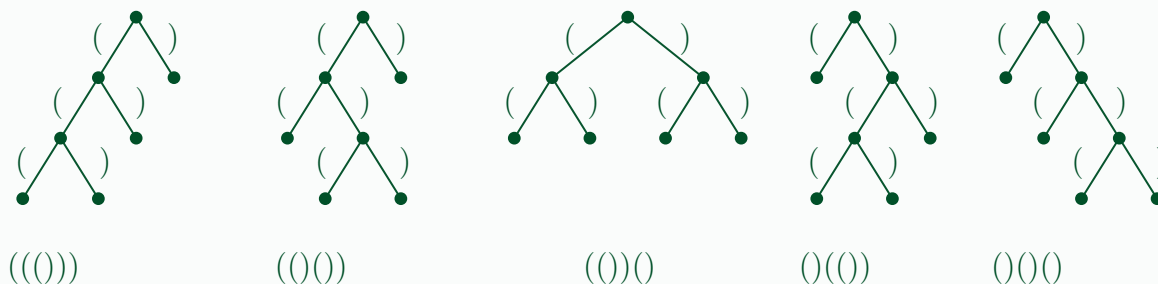
Да сложим имена на листата a, b, c и d , така че при обхождане в preorder (или postorder, няма значение), те се появяват в този ред:



Ето и съответните скобувани изрази:



Покажахме с пример биекция между скобуваните изрази с $n + 1$ множители и пълните подредени двоични дървета с $n + 1$ листа. Върху същия пример ще покажем биекция между пълните подредени двоични дървета с $n + 1$ листа и балансираните изрази с $2n$ скоби. Забележете, че при $n + 1$ листа, вътрешните върхове са n на брой, така че общо върховете са $2n + 1$, откъдето следва, че ребрата са точно $2n$. Ще маркираме всяко ребро или с “(”, ако е ребро към ляво дете, или с “)”, ако е ребро към дясно дете. Балансираните изрази са това, което би принтирал DFS, който обхожда дървото и принтира ребрата в реда на откриването, като във всеки вътрешен връх първо отива към лявото дете и след това, към дясното дете.



Това е краят на доказателството на Лема 44. □

Лема 45

Редицата от числата на Catalan е решението на следното рекурентно уравнение:

$$C_n = \begin{cases} 1, & \text{ако } n = 0, \\ \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i}, & \text{ако } n > 0 \end{cases} \quad (12.27)$$

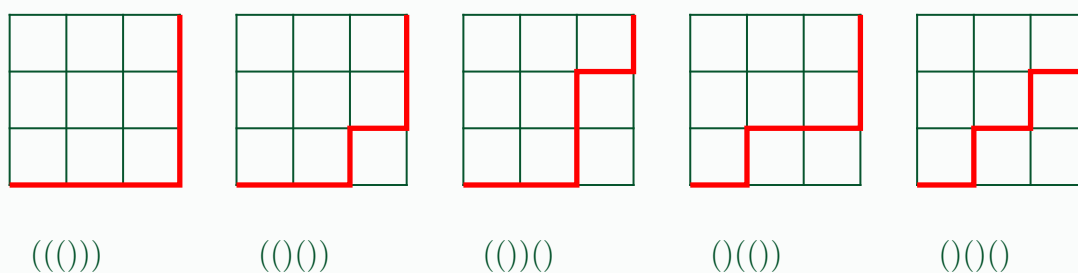
Доказателство: Ще покажем с индукция по n , че (12.27) брой скобуваните изрази с $n + 1$ множители. Но съгласно Лема 44 те са колкото балансираните изрази с $2n$ скоби, откъдето желаният резултат следва веднага.

Ако $n = 0$, (12.27) дава 1, а добре скобуваният израз с $0 + 1$ множителя е само един: (a) , ако игнорираме идентичността на множителя. ✓

Ако $n \geq 1$, става дума за $n + 1$ множители. В линейната наредба между тях има $(n + 1) - 1 = n$ “празнини”, във всяка от които може да попадне последното умножение. Същото нещо, казано в термините на пълните двоични подредени дървета, чиито листа са множителите, е: при фиксирана наредба отляво надясно на листата, има n възможности за това, кои листа са вляво от корена (което определя кои са вдясно от корена). Нека i е номерът на “празнината”, в която е последното умножение, като i взема такива стойности, че вляво от тази празнина има $i + 1$ множителя. Това означава, че вдясно има $(n + 1) - (i + 1) = n - i$ множителя. Очевидно минималната стойност на i е 0 (1 множител вляво), а максималната е $n - 1$ (n множителя вляво). За всяка позиция на празнината, умножаваме броя на скобуванията на подредицата вляво с броя на скобуванията на подредицата вдясно; съгласно индуктивното предположение, те са съответно C_i ($i + 1$ множителя вляво) и C_{n-1-i} ($n - i$ множителя вдясно). Накратко, за всяко i скобуванията са $C_i \cdot C_{n-1-i}$. Прилагаме комбинаторния принцип на разбиването, сумирайки по всички i , и получаваме израза от (12.27). $\square$

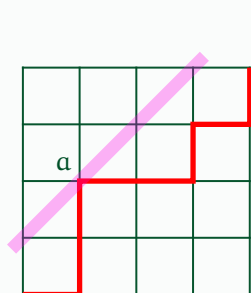
Рекурентното уравнение 12.27 не може да бъде решено нито с Мастър Теоремата, нито с метода с характеристичното уравнение. Ако имаме решение, можем да го докажем по индукция, но как да получим решение?

Ще изведем решение на уравнение 12.27 с елементарни комбинаторни съображения. Това уравнение брои балансираните изрази с $2n$ скоби. Ако отпадне изискването да са балансирани и остане само това, че са n скоби от вида “(” и още n скоби от вида “)”, ние знаем колко са тези изрази: $\binom{n+n}{n} = \binom{2n}{n}$. Знаем, че те отговарят биективно на разходки в правоъгълна мрежа $n \times n$, които разходки започват долу вляво и завършват горе вдясно. Нека скобата “(” е ход надясно, а “)” е ход нагоре. Сега връщаме ограничението изразите да са балансирани. Това означава в нито един префикс броят на “)” не е по-голям от броя на “(”. Иначе казано, никоя разходка не “излиза” над диагонала долу ляво–горе дясно. Ето петте такива разходки при $n = 3$ със съответните балансирани изрази:

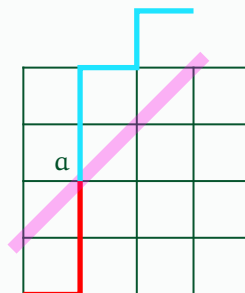


Търсим броя на разходките, които никога не излизат над диагонала долу ляво–горе дясно, като функция на n . Този брой ще получим, като от $\binom{2n}{n}$ извадим броя на разходките-нарушители – тези, които излизат над диагонала долу ляво–горе дясно. Следната фигура показва как броим нарушителите: всяка разходка-нарушител минава през поне една точка от диагонала (показан в лилаво), който е над диагонала долу ляво–горе дясно. От първата такава точка до края обръщаме разходката в смисъл, че всеки ход надясно става ход нагоре, и обратно, всеки ход нагоре става под надясно. Това ще наричаме “рефлексия след първата точка”. Ето пример: разходката вляво е

нарушител, a е първата нейна точка на пресичане със забранения диагонал (над диагонала долу ляво–горе дясно), а вдясно е показана същата разходка, в която частта след a е рефлектирана спрямо забранения диагонал.



разходка-нарушител

рефлексия след a

Очевидно рефлектираната разходка става разходка в правоъгълна мрежа $(n-1) \times (n+1)$. Не е трудно да се види, че има биекция между разходките-нарушители в оригиналната $n \times n$ мрежа и всички разходки в новата $(n-1) \times (n+1)$ мрежа. Оттук заключаваме, че броят на разходките, които не са нарушители, е

$$\binom{2n}{n} - \binom{n-1+n+1}{n-1} = \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!} = \frac{(2n)!}{n!(n-1)!} \left(\frac{1}{n} - \frac{1}{n+1} \right) = \frac{(2n)!}{n!(n-1)!} \left(\frac{n+1-n}{n(n+1)} \right) = \frac{(2n)!}{n!n!} \left(\frac{1}{n+1} \right) = \frac{1}{n+1} \binom{2n}{n}$$

Покажахме, че $C_n = \frac{1}{n+1} \binom{2n}{n}$. Това е решението на (12.27).

Да видим асимптотиката на C_n . От Теорема 35 следва, че $\binom{2n}{n} \asymp \frac{1}{\sqrt{2n}} 2^{2n}$, тоест $\binom{2n}{n} \asymp \frac{1}{\sqrt{n}} 4^n$. Тогава $C_n \asymp \frac{1}{n\sqrt{n}} 4^n$.

И така, нека е дадено верижно умножение на n матрици

$$A_1 \cdot A_2 \cdot \dots \cdot A_n$$

като A_1 е $d_0 \times d_1$, A_2 е $d_1 \times d_2$, ..., A_n е $d_{n-1} \times d_n$. Самите матрици не са дадени, защото тяхното съдържание не ни интересува в тази задача. Интересуваме се от техните размери и търсим начин да бъдат умножени, който минимизира общия брой на скаларните произведения. Задачата може да се формализира така.

Изн. Задача 32: MATRIX-CHAIN MULTIPLICATION

екземпляр: Редица $\langle d_0, d_1, \dots, d_n \rangle$ от цели положителни числа, която представлява редица от n матрици $\langle A_1, A_2, \dots, A_n \rangle$, като A_i е с размери $d_{i-1} \times d_i$, за $1 \leq i \leq n$. Матриците не са дадени.

решение: Скобуване на редицата $\langle A_1, A_2, \dots, A_n \rangle$, което минимизира броя на скаларните умножения за матричното умножение $A_1 \cdot A_2 \cdot \dots \cdot A_n$.

Олекотеният вариант. Ще опишем много подробно конструирането на алгоритъма от първи принципи.

Всяко решение, оптимално или не, има място, на което се извършва последното матрично умножение (в Допълнение 59 използвахме термина “празнина”). Формално, съществува $k \in \{1, \dots, n-1\}$, такова че последното матрично умножение е $B \times C$, където $B = A_1 \cdot A_2 \cdot \dots \cdot A_k$ и $C = A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$. // това е тривиално ...

Визуално е добре да си представим последното матрично умножение с нещо като разделител, поставен между A_k и A_{k+1} :

$$\underbrace{(A_1 \cdot A_2 \cdot \dots \cdot A_k)}_B \mid \underbrace{(A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n)}_C$$

При това положение, броят на скаларните умножения за цялото матрично умножение е сумата от следните три събираеми:

- ① броят на скаларните умножения в матричното произведение $A_1 \cdot A_2 \cdot \dots \cdot A_k$, тоест, за конструиране на матрицата B ;
- ② броят на скаларните умножения в матричното произведение $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_n$, тоест, за конструиране на матрицата C ;
- ③ броят на скаларните умножения за матричното умножение $B \cdot C$. // очевидно – е, и?

Но B е с размери $d_0 \times d_k$ и C е с размери $d_k \times d_n$. Следователно, броят на скаларните умножения в ③ е $d_0 d_k d_n$, **независимо** от броя на скаларните умножения в ① и ②; с други думи, независимо от начина, по който скобуваме редицата $\langle A_1, A_2, \dots, A_k \rangle$, за да получим B и независимо от начина по който скобуваме редицата $\langle A_{k+1}, A_{k+2}, \dots, A_n \rangle$, за да получим C . Нещо повече: броевете на скаларните умножения в ① и ② се минимизират независимо в смисъл, че скобуванията на $\langle A_1, A_2, \dots, A_k \rangle$ и $\langle A_{k+1}, A_{k+2}, \dots, A_n \rangle$ са взаимно независими. Този факт директно влече, че всяко оптимално умножение на цялата редица $\langle A_1, A_2, \dots, A_n \rangle$ с разделител между A_k и A_{k+1} се получава от **оптимално** умножение на $\langle A_1, A_2, \dots, A_k \rangle$ и **оптимално** умножение на $\langle A_{k+1}, A_{k+2}, \dots, A_n \rangle$. // аха, това е!

За да се убедим в последното, да допуснем БОО, че скобуването на $\langle A_1, A_2, \dots, A_k \rangle$, с което пресмятаме B , да не е оптимално. Тогава го заменяме с друго скобуване на $\langle A_1, A_2, \dots, A_k \rangle$, имащо по-малко скаларни умножения, и получаваме умножение на $\langle A_1, A_2, \dots, A_n \rangle$ с разделител между A_k и A_{k+1} , което се извършва с дори по-малко скаларни умножения, понеже броевете в ② и ③ не се променят от замяната. ⚡

Редиците $\langle A_1, A_2, \dots, A_k \rangle$ и $\langle A_{k+1}, A_{k+2}, \dots, A_n \rangle$ се явяват непосредствените конституенти на редицата $\langle A_1, A_2, \dots, A_n \rangle$, ако разделителят е между A_k и A_{k+1} . Тъй като разделителят трябва да е все някъде, решението за $\langle A_1, A_2, \dots, A_n \rangle$ е минимума по всички $k \in \{1, \dots, n-1\}$ от сумите ①, ② и ③, като множеството от непосредствените конституенти при това е

$$\{\langle A_1 \rangle, \langle A_1, A_2 \rangle, \dots, \langle A_1, \dots, A_{n-1} \rangle, \langle A_n \rangle, \langle A_{n-1}, A_n \rangle, \dots, \langle A_2, \dots, A_n \rangle\}$$

Вече се убедихме, че за всеки от непосредствените конституенти ни трябва оптимално решение. Заключаваме, че принципът на оптималността (Определение 95) е в сила.

Тъй като дължината на всеки непосредствен конституент е по-малка от n , това води към истинска рекурсивна декомпозиция. Дъното на рекурсията е редица с дължина едно: цената за нея е нула, понеже върху нея умножение на матрици не се извършва.

Нека осмислим добре рекурсивната декомпозиция, която получаваме. От написаното човек може да остане с впечатление, че при декомпозицията винаги получаваме екземпляри, които започват с A_1 или завършват с A_n . Това, разбира се, не е вярно. Да си представим още едно ниво на декомпозиция, означено на следната фигура със сини разделители:

$$\underbrace{\underbrace{((A_1 \cdot \dots \cdot A_\ell) \mid (A_{\ell+1} \cdot \dots \cdot A_k))}_{B_1} \mid \underbrace{((A_{k+1} \cdot \dots \cdot A_m) \mid (A_{m+1} \cdot \dots \cdot A_n))}_{C_2}}_B \quad \underbrace{\underbrace{((A_{k+1} \cdot \dots \cdot A_m) \mid (A_{m+1} \cdot \dots \cdot A_n))}_{C_1} \mid \underbrace{((A_{k+1} \cdot \dots \cdot A_m) \mid (A_{m+1} \cdot \dots \cdot A_n))}_{C_2}}_C$$

Става ясно, че всяка редица-конституент в декомпозицията започва с някаква матрица A_i и завършва с някаква матрица A_j , където $1 \leq i \leq j \leq n$. Тогава броят на конституентите, които възникват, е $\Theta(n^2)$. Тук няма смисъл да въвеждаме състояния като абстракции на конституентите. Може да кажем, че състоянията са наредените двойки (i, j) , където $1 \leq i \leq j \leq n$, но от това няма полза – тези двойки също са $\Theta(n^2)$ брой. Това се дължи на факта, че екземплярът в тази задача е линейна наредба.

Подробната рекурсивна декомпозиция е следната. Нека $M(i, j)$ означава минималния брой скалярни умножения за верижното матрично умножение $A_i \cdot \dots \cdot A_j$, където $1 \leq i \leq j \leq n$.

$$M(i, j) = \begin{cases} 0, & \text{ако } i = j, \\ \min \{M(i, k) + M(k+1, j) + d_{i-1}d_kd_j \mid i \leq k \leq j-1\}, & \text{ако } i < j \end{cases} \quad (12.28)$$

$d_{i-1}d_kd_j$ е броят на скалярните умножения в матричното умножение на матрицата-произведение $A_i \cdot \dots \cdot A_k$ (тя е с размери $(i-1) \times k$) и на матрицата-произведение $A_{k+1} \cdot \dots \cdot A_j$ (тя е с размери $k \times j$). Управляващият параметър на тази рекурсия е разликата $j-i$. Началното условие е за стойност 0 на $j-i$, а в рекурсивната част $M(i, j)$ се пресмята чрез някакви $M(i', j')$, където $j'-i' < j-i$.

Решението е $M(1, n)$.

Ето схемата за изчисление, базирана на рекурсивната декомпозиция (12.28). Нека M е квадратен масив $n \times n$. Изчислителното правило е пълнен аналог на (12.28):

$$M[i, j] = \begin{cases} 0, & \text{ако } i = j, \\ \min_{i \leq k \leq j-1} (M[i, k] + M[k+1, j] + d_{i-1}d_kd_j), & \text{ако } i < j \end{cases} \quad (12.29)$$

Да съобразим как работи изчислението. Базата на рекурсията е $M[i, i] = 0$, за $1 \leq i \leq n$. С други думи, слагаме нули върху главния диагонал. Въз основа на главния диагонал и изчислителното правило от (12.29) запълваме съседния вдясно диагонал на главния. Въз основа на тези два диагонала и изчислителното правило запълваме следващия диагонал вдясно. И така нататък, като последният диагонал в тази редица от диагонали има един елемент, а именно $M[1, n]$, който е и решението. Забележете, че клетките на масива под главния диагонал не се ползват.

Ето псевдокодът на алгоритъм, базиран на тази схема за изчисление.

ALG MATRIX-CHAIN MULTIPLICATION($\langle d_0, d_1, \dots, d_n \rangle$: цели положителни числа)

```

1  създай  $M[1 \dots n, 1 \dots n]$  от числа
2  for  $i \leftarrow 1$  to  $n$ 
3     $M[i, i] \leftarrow 0$ 
4  for  $\text{diag} \leftarrow 2$  to  $n$ 
```

```

5      for i ← 1 to n – diag + 1
6          j ← i + diag – 1
7          M[i, j] ← M[i, i] + M[i + 1, j] + di-1didj
8          for k ← i + 1 to j – 1
9              M[i, j] ← min (M[i, j], M[i, k] + M[k + 1, j] + di-1dkdj)
10     return M[1, n]

```

Има смисъл от малко технически разяснения за кода. $M[i, i]$ на ред 7, разбира се, е 0. Написано е по този начин, за да следва буквално (12.29) и да има прилика с ред 9.

for-цикълът на редове 4–9 е с управляваща променлива, наречена не случайно “diag”. Тя има смисъл на диагонал в направление горе-ляво – долу-дясно, чийто номер отговаря на номера на колоната, в която този диагонал пресича първия ред. Диагонал 1 е главният диагонал, който запълваме с нули на редове 2–3.

Клетките $M[i, j]$ в диагонал номер diag се отличават с това, че $j - i = \text{diag} - 1$, тоест, $j = i + \text{diag} - 1$. Примерно,

- за диагонал 2, разликата между j и i е 1: той се състои от $M[1, 2], M[2, 3], \dots, M[n-1, n]$;
- за диагонал 3, разликата между j и i е 2: той се състои от $M[1, 3], M[2, 4], \dots, M[n-2, n]$;
- и така нататък;
- за диагонал n , разликата между j и i е $n - 1$; той се състои от $M[1, n]$.

И наистина, на ред 6, на j се присвоява точно $i + \text{diag} - 1$.

Иначе казано, клетките $M[i, j]$, които принадлежат на диагонал номер diag, са тези:

$$M[1, \text{diag}], M[2, \text{diag} + 1], \dots, M[i, i + \text{diag} - 1], \dots, M[n - \text{diag} + 1, n - \text{diag} + 1 + \text{diag} - 1]$$

тоест,

$$M[1, \text{diag}], M[2, \text{diag} + 1], \dots, M[i, i + \text{diag} - 1], \dots, M[n - \text{diag} + 1, n]$$

За дадена стойност на diag, алгоритъмът първо пресмята $M[1, \text{diag}]$, после $M[2, \text{diag} + 1]$, после $M[3, \text{diag} + 2]$, и така нататък, и най-накрая пресмята $M[n - \text{diag} + 1, n]$, което означава, че всеки диагонал се запълва в посока от горе-ляво към долу-дясно.

Не е задължително да е така. За всяка клетка от даден диагонал, стойността се изчислява чрез стойности, които се вземат само от диагоналите с **по-малки номера** (както и от d стойностите от входа). Ерго, можем да запълваме клетките на даден диагонал в какъвто ред искаме. Тъй като сме избрали да запълваме всеки диагонал в посока от горе-ляво към долу-дясно, има смисъл да инициализираме i с 1 и да го инкрементираме, докато стане $n - (\text{diag} - 1) = n - \text{diag} + 1$, защото $\text{diag} - 1$ е броят на редовете под последния ред, съдържащ елемент на диагонал номер diag.

Да видим пример, взет от [34, стр. 376]. Нека $n = 6$ и $\langle d_0, \dots, d_6 \rangle = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$. Двумерният масив е 6×6 . Клетките под главния диагонал не се ползват. Слагаме нули върху главния диагонал.

Да видим колко е $M[1, 2]$. Съгласно (12.29),

$$\begin{aligned}
 M[1, 2] &= \min_{1 \leq k \leq 1} (M[1, k] + M[k + 1, 2] + d_{1-1}d_kd_2) \\
 &= M[1, 1] + M[2, 2] + d_0d_1d_2 = 0 + 0 + 30 \cdot 35 \cdot 15 = 15\,750
 \end{aligned}$$

	1	2	3	4	5	6
1	0	15 750				
2		0				
3			0			
4				0		
5					0	
6						0

Аналогично,

$$M[2, 3] = M[2, 2] + M[3, 3] + d_1 d_2 d_3 = 0 + 0 + 35 \cdot 15 \cdot 5 = 2\,625$$

$$M[3, 4] = M[3, 3] + M[4, 4] + d_2 d_3 d_4 = 0 + 0 + 15 \cdot 5 \cdot 10 = 750$$

$$M[4, 5] = M[4, 4] + M[5, 5] + d_3 d_4 d_5 = 0 + 0 + 5 \cdot 10 \cdot 20 = 1\,000$$

$$M[5, 6] = M[5, 5] + M[6, 6] + d_4 d_5 d_6 = 0 + 0 + 10 \cdot 20 \cdot 25 = 5\,000$$

Запълваме докрая диагонал 2:

	1	2	3	4	5	6
1	0	15 750				
2		0	2 625			
3			0	750		
4				0	1 000	
5					0	5 000
6						0

Да видим колко е $M[1, 3]$. Съгласно (12.29),

$$\begin{aligned}
 M[1, 3] &= \min_{1 \leq k \leq 2} (M[1, k] + M[k + 1, 3] + d_{1-k} d_k d_3) \\
 &= \min (M[1, 1] + M[2, 3] + d_0 d_1 d_3, M[1, 2] + M[3, 3] + d_0 d_2 d_3) \\
 &= \min (0 + 2\,625 + 5\,250, 15\,750 + 0 + 2\,250) \\
 &= \min (7\,875, 18\,000) = 7\,875
 \end{aligned}$$

Следната фигура показва запълнената клетка $M[1, 3]$, а с цветни линии е показано кои двой-

ки елементи от предишни диагонали участват в минимума.

	1	2	3	4	5	6
1	0	15 750	7 875			
2		0	2 625			
3			0	750		
4				0	1 000	
5					0	5 000
6						0

Да кажем, че главният диагонал е *първият диагонал*, този над него е *вторият диагонал*, а следващият—който запълваме в момента—е *третият диагонал*. Изобщо, номерът на диагонала е номерът на колоната, на която той “излиза” на първия ред. Ето запълването и на третия диагонал докрая.

$$\begin{aligned}
 M[2, 4] &= \min (M[2, 2] + M[3, 4] + d_1 d_2 d_4, M[2, 3] + M[4, 4] + d_1 d_3 d_4) \\
 &= \min (0 + 750 + 5\,250, 2\,625 + 0 + 1\,750) \\
 &= \min (6\,000, 4\,375) = 4\,375
 \end{aligned}$$

$$\begin{aligned}
 M[3, 5] &= \min (M[3, 3] + M[4, 5] + d_2 d_3 d_5, M[3, 4] + M[5, 5] + d_2 d_4 d_5) \\
 &= \min (0 + 1\,000 + 1\,500, 750 + 0 + 3\,000) \\
 &= \min (2\,500, 3\,750) = 2\,500
 \end{aligned}$$

$$\begin{aligned}
 M[4, 6] &= \min (M[4, 4] + M[5, 6] + d_3 d_4 d_6, M[4, 5] + M[6, 6] + d_3 d_5 d_6) \\
 &= \min (0 + 5\,000 + 1\,250, 1\,000 + 0 + 2\,500) \\
 &= \min (6\,250, 3\,500) = 3\,500
 \end{aligned}$$

	1	2	3	4	5	6
1	0	15 750	7 875			
2		0	2 625	4 375		
3			0	750	2 500	
4				0	1 000	3 500
5					0	5 000
6						0

Ето запълването на целия масив без шестия диагонал (състоящ се само от $M[1,6]$).

	1	2	3	4	5	6
1	0	15 750	7 875	9 375	11 875	
2		0	2 625	4 375	7 125	10 500
3			0	750	2 500	5 375
4				0	1 000	3 500
5					0	5 000
6						0

$M[1,6]$ се изчислява така:

$$\begin{aligned}
 M[1,6] &= \min(M[1,1] + M[2,6] + d_0 d_1 d_6, \\
 &\quad M[1,2] + M[3,6] + d_0 d_2 d_6, \\
 &\quad \mathbf{M[1,3] + M[4,6] + d_0 d_3 d_6}, \\
 &\quad M[1,4] + M[5,6] + d_0 d_4 d_6, \\
 &\quad M[1,5] + M[6,6] + d_0 d_5 d_6) \\
 &= \min(0 + 10\,500 + 26\,250, \\
 &\quad 15\,750 + 5\,375 + 11\,250, \\
 &\quad \mathbf{7\,875 + 3\,500 + 3\,750}, \\
 &\quad 9\,375 + 5\,000 + 7\,500, \\
 &\quad 11\,875 + 0 + 15\,000) = \min(36\,750, 32\,375, \mathbf{15\,125}, 21\,875, 26\,875) = \mathbf{15\,125}
 \end{aligned}$$

На следната фигура с цветни линии в пет различни цвята е показано кои двойки елементи

от предишни диагонали участват в пресмятането на минимума **15 125**.

	1	2	3	4	5	6
1	0	15 750	7 875	9 375	11 875	15 125
2		0	2 625	4 375	7 125	10 500
3			0	750	2 500	5 375
4				0	1 000	3 500
5					0	5 000
6						0

Това е и краят на примера.

Коректността на алгоритъма е очевидна за всеки, който е убеден в коректността на рекурсивната декомпозиция, основаната на нея схема за изчисление и на техническите подробности на кода. Сложността по време е $\Theta(n^3)$.

Тежкият вариант. Ако искаме и оптимално скобуване, една възможност е следната. Модифицираме ALG MATRIX-CHAIN MULTIPLICATION, като за всяка клетка $M[i, j]$, която не отговаря на начално условие, запомняме **двете** клетки, чиито стойности са събираеми в минималната сума, която дава стойността на $M[i, j]$. Ако действаме по този начин, за всяка клетка, която не отговаря на начално условие, пазим **два** индекса, които сочат назад в таблицата. След като изчислим цялата таблица, изпълняваме обратно проследяване от $M[1, n]$ чак до началните условия и това ни дава едно оптимално решение.

Ако действаме по този начин, обратното проследяване дава не ориентиран път—както беше в тежкия вариант на решението на задачата за опашката пред касата на стр. 545—а арборесценция с разклоненост две, защото от всяка вътрешна клетка биваме препратени към **две** други клетки. Ето частично попълнената таблица за обратно проследяване за примера, който видяхме преди малко. Таблицата е частично попълнена, понеже показва индексите-стрелки само на клетките на решението. Ако таблицата беше изцяло попълнена, трябваше от всяка клетка, която не отговаря на начално условие, да има две сочещи някъде назад (по отношение на рекурсията) стрелки, и фигурата щеше да е визуална каша.

	1	2	3	4	5	6
1	0	15 750	750	9 375	11 375	15 750
2		0	2 525	4 375	7 125	10 600
3			0	750	2 500	5 375
4				0	1 375	3 900
5					0	5 000
6						0

Решението, което съответства на тази фигура, е

$$((A_1(A_2A_3)))(((A_4A_5)A_6))$$

Има обаче и друг, по-икономичен и смислен начин да се получи оптимално решение, съответстващо на таблицата с оптималните цени. Ключовото наблюдение е, че адресите на двете клетки, към които сочат индексите за обратното проследяване, **не са независими**. В примера горе, клетка $M[1, 6]$ сочи към $M[1, 3]$ и $M[4, 6]$. Ако знаем, че едната е $M[1, 3]$, то другата не може да не е $M[4, 6]$, понеже $4 = 3 + 1$, а левият индекс 1 в $M[1, 3]$ и десният индекс 6 в $M[1, 6]$ са фиксирани от това, че разсъждаваме как е получена $M[1, 6]$. Ако обобщим това наблюдение, виждаме, че за да знаем как е получена дадена клетка $M[i, j]$, такава че $i < j$, е достатъчно да знаем за кое k е получен минимумът в рекурсивната декомпозиция.

С други думи, за всяка клетка $M[i, j]$, такава че $i < j$, стойността на $M[i, j]$ е изчислена като

$$M[i, j] \leftarrow \min_{i \leq k \leq j-1} \{M[i, k] + M[k+1, j] + d_{i-1}d_kd_j\}$$

Достатъчно е да запишем за кое k се реализира минимум на множеството вдясно. Ако за всички клетки, които не отговарят на начални условия, знаем тези стойности, не е проблем да се изчисли оптималното решение, съответно на масива с оптималните цени.

Ето псевдокод на алгоритъм, реализиращ тази идея. Нека всяко $M[i, j]$ да е наредена двойка $(M[i, j].val, M[i, j].prev)$, където $M[i, j].val$ е оптималната цена за умножението на $A_i \cdots A_j$, а $M[i, j].prev$ е 0, ако $i = j$, и

ALG MATRIX-CHAIN MULTIPLICATION-1($\langle d_0, d_1, \dots, d_n \rangle$: цели положителни числа)

```

1  създай  $M[1..n, 1..n]$  от наредени двойки от числа
2  for  $i \leftarrow 1$  to  $n$ 
3     $M[i, i].val \leftarrow 0$ 
4     $M[i, i].prev \leftarrow 0$ 
5  for  $diag \leftarrow 2$  to  $n$ 
6    for  $i \leftarrow 1$  to  $n - diag + 1$ 
7       $j \leftarrow i + diag - 1$ 
8       $M[i, j].val \leftarrow M[i, i].val + M[i+1, j].val + d_{i-1}d_jd_j$ 
```

```

9      M[i, j].prev ← i
10     for k ← i + 1 to j - 1
11         if M[i, j].val > M[i, k].val + M[k + 1, j].val + di-1dkdj
12             M[i, j].val ← M[i, k].val + M[k + 1, j].val + di-1dkdj
13             M[i, j].prev ← k
14     return M

```

Следният рекурсивен алгоритъм генерира описание на оптимално скобуване. Началното викане, разбира се, е SHOW ORDER(1, n).

SHOW-ORDER(двумерен масив $M[1 \dots n, 1 \dots n]$ от (val, prev), i, j)

```

1  if i = j
2      print "Ai"
3  else
4      k ← M[i, j].prev
5      print "("
6      SHOW-ORDER(M, i, k)
7      print "*"
8      SHOW-ORDER(M, k + 1, j)
9      print ")"

```

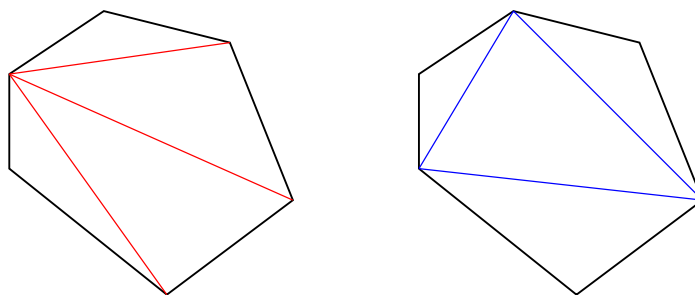
12.4.2 Minimum Weight Triangulation

Прост многоъгълник е множество от отсечки, поне три на брой, които не се пресичат две по две, освен може би в крайните точки, и чието обединение е проста затворена крива. Краищата на отсечките са *върховете на многоъгълника*. *Съседни върхове* са всеки два върха, които са краища на една от тези отсечки. Ако n е броят на отсечките (който е равен на броя на върховете), наричаме фигурата *прост n -ъгълник*.

Разглеждаме частен случай на прости многоъгълници, наречени *изпъкнали многоъгълници*. Изпъкнал многоъгълник е прост многоъгълник с вътрешни ъгли, по-малки от 180 градуса. Ако кажем само *n -ъгълник*, разбираме прост изпъкнал *n -ъгълник*.

Диагонал на прост многоъгълник е отсечка с краища върхове на многоъгълника, които не са съседни. При изпъкналите многоъгълници всички диагонали се съдържат във вътрешността на фигурата.

Триангулация на многоъгълник е множество негови диагонали, които не се пресичат по двойки, освен може би в крайните си точки, и които разбиват вътрешността на фигурата на триъгълници. Триангулирането на многоъгълници е най-важната декомпозиция на многоъгълници с големи приложения в Компютърната геометрия, CAD и други области на Компютърните науки. Ето пример със шестоъгълник и две негови различни триангулации.



Тривиално се доказва, че за всеки n -ъгълник, всяка триангулация има $n-3$ диагонала и $n-2$ триъгълника. Резултатът очевидно е в сила, ако $n = 3$. За целите на задачата и алгоритъма, който ще разгледаме, има смисъл да разглеждаме отсечките като *изродени многоъгълници*, двуъгълници в този случай, чиито триангулации имат 0 диагонала.

Тегло на триангулация е сумата от дължините на участващите в нея диагонали[†]. Примерно, на показаната фигура с двете триангулации, червената триангулация има тегло 127.61 мм, а синята, 116.86 мм (измерено в приложението, с което са генерирани).

Ако триангулацията се казва X , нейното тегло се бележи с $|X|$.

Изч. Задача 33: MINIMUM WEIGHT TRIANGULATION

екземпляр: Редица $\langle V_1, \dots, V_n \rangle$ от точки в равнината, задаваща изпъкнал n -ъгълник.

решение: Триангулация на n -ъгълника с минимално тегло.

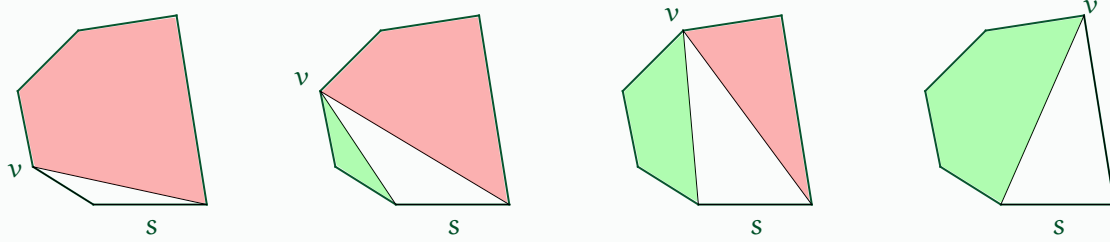
В тези записки ще разгледаме само олекотения вариант на задачата.

MINIMUM WEIGHT TRIANGULATION има доста общо с MATRIX-CHAIN MULTIPLICATION. Най-малкото, броят на триангулациите на изпъкнал n -ъгълник е C_{n-2} , тоест, $(n-2)$ -рото число на Catalan; да си припомним, че броят на скобуванията на израз с n множители е C_{n-1} .

Допълнение 60: Броят на триангулациите на изпъкнал n -ъгълник

Да си представим изпъкнал n -ъгълник за $n \geq 3$ и произволна негова триангулация. Да си представим произволна страна s на n -ъгълника. Очевидно s е страна на точно един триъгълник на триангулацията. Този триъгълник се определя еднозначно от s , която задава два от върховете му, и от третия си връх v , който не е връх от s . За v има точно $n-2$ възможности. На следната фигура, триъгълникът е в бяло за всеки избор на v .

[†]Има и други смислени дефиниции на тегло на триангулация. Може теглото да е минималното лице на триъгълник в триангулацията или минималния ъгъл на триъгълник в триангулацията – и двете имат отношение към по-лесното визуално възприемане. При тези дефиниции на тегло, задачата става максимизационна, но това е дребна разлика.



Всеки избор на трети връх v задава един k -ъгълник (в зелено на фигурата) и един m -ъгълник (в червено на фигурата), които, заедно с триъгълника, представляват разбиване на оригиналния n -ъгълник. Забележете, че $k + m = n + 1$, понеже k -ъгълникът и m -ъгълникът имат общ връх v . Минималната стойност на k е 2, при което k -ъгълникът е изроден, а m -ъгълникът е $(n - 1)$ -ъгълник. Максималната стойност на k е $n - 1$, при което m -ъгълникът е изроден.

За всеки избор на v , броят на триангулациите е произведението от броя на триангулациите на k -ъгълника и броя на триангулациите на m -ъгълника. Предвид това, че $m = n - k + 1$, броят T_n на триангулациите на n -ъгълника е

$$T_n = \begin{cases} 1, & \text{ако } n = 2, \\ \sum_{k=2}^{n-1} T_k \cdot T_{n-k+1}, & \text{ако } n > 2 \end{cases} \quad (12.30)$$

Да сравним (12.30) с (12.27). Веднага се вижда, че $T_n = C_{n-2}$.

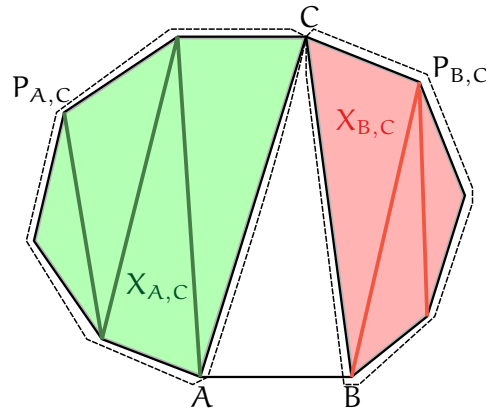
Наблюдение 77: Тегло на триангулация, декомпозиция

Нека е даден n -ъгълник P за $n \geq 3$ и нека отсечката AB е негова страна. Нека е дадена произволна триангулация X на P . Тогава AB е страна в точно един триъгълник спрямо X . Нека C е третият връх на този триъгълник. Нека многоъгълникът, състоящ се от редицата от отсечките от C до A , които не съдържат B , плюс отсечката AC , се нарича $P_{A,C}$. Нека многоъгълникът, състоящ се от редицата от отсечките от C до B , които не съдържат A , плюс отсечката BC , се нарича $P_{B,C}$. Нека $X_{A,C}$ е триангулацията X , ограничена до $P_{A,C}$. Нека $X_{B,C}$ е триангулацията X , ограничена до $P_{B,C}$.

Тогава

$$|X| = |AC| + |BC| + |X_{A,C}| + |X_{B,C}| \quad (12.31)$$

Това твърдение е прекалено тривиално, за да го доказваме строго. Ето илюстрация. Имайте предвид, че теглото на триангулация е сумата от теглата на диагоналите ѝ. Това означава, че отсечките на многоъгълника не участват в тази сума. В този пример, отсечката AC не участва в $|X_{A,C}|$ и, аналогично, BC не участва в $|X_{B,C}|$. Така че (12.31) наистина брои всички диагонали точно по един път.



Нека входът е редица от точки $\langle V_1, \dots, V_n \rangle$ в равнината, задаваща изпъкнал n -ъгълник P . Фиксираме посока в равнината, да кажем, по часовниковата стрелка. БОО, нека редицата точки $\langle V_1, \dots, V_n \rangle$ се среща в посока по часовниковата стрелка, ако обикаляме по периферията на многоъгълника. За всеки i и j , такива че $1 \leq i < j \leq n$, $P_{i,j}$ е многоъгълникът, състоящ се от отсечките $V_i V_{i+1}, V_{i+1} V_{i+2}, \dots, V_{j-1} V_j$ и отсечката $V_i V_j$, която е или страна, или диагонал на многоъгълника. Примерно, $P_{1,n} = P$, като $V_i V_j = V_1 V_n$ е страна. Като други примери: $P_{1,2}$ е изроден многоъгълник, съвпадащ със страната $V_1 V_2$, $P_{2,3}$ е изроден многоъгълник, съвпадащ със страната $V_2 V_3$, и така нататък, $P_{n-1,n}$ е изроден многоъгълник, съвпадащ със страната $V_{n-1} V_n$. Страната $V_1 V_n$ не може да бъде изразена по този начин, понеже, както казахме, $P_{1,n}$ е целият многоъгълник.

Рекурсивната декомпозиция, базирана на Наблюдение 77, е следната. При дадени i и j , такива че $i + 1 < j$, минималната триангулация на $P_{i,j}$ е минимума по всички точки $k \in \{i + 1, \dots, j - 1\}$ от сумата на $|V_i V_k|$, $|V_k V_j|$ и теглата на минимални триангулации на $P_{i,k}$ и $P_{k,j}$. Базата на рекурсията е $j = i + 1$ [†]: отсечките $P_{i,i+1}$ са изродени многоъгълници и техните триангулации с тегло нула дават началните условия.

Схемата за изчисление е следната. Нека $T[i, j]$ е теглото на минимална триангулация на $P_{i,j}$, където $1 \leq i < j \leq n$.

$$T[i, j] = \begin{cases} 0, & \text{ако } j = i + 1 \\ \min_{i+1 \leq k \leq j-1} (T[i, k] + T[k, j] + |V_i V_k| + |V_k V_j|), & \text{ако } j > i + 1 \end{cases} \quad (12.32)$$

Има една особеност: разстоянието между точки, които са съседни (иначе казано, които са краища на някоя от отсечките, дефиниращи многоъгълника), трябва да е 0. Не се твърди, че Евклидовото разстояние между тези точки е 0, а че за целите на нашето изчисление функцията, която връща разстоянието между две точки, които са краища на страна на многоъгълника, трябва да връща 0, ако те са съседни. Причината е, че за теглото на триангулацията ни трябва само дължини на диагонали, а съседни точки не са краища на диагонал.

Алгоритъмът за олекотения вариант на задачата е много подобен на ALG MATRIX-CHAIN MULTIPLICATION. Дребна разлика е, че тук главният диагонал не се ползва, а следващите два диагонала вдясно от него съдържат само нули: първият заради началните условия (страните на многоъгълника), а вторият поради това, че триъгълниците имат триангулации с нулево тегло, нямайки диагонали.

[†]А не $j = i$. $P_{i,i}$ е точка, а чак толкова изродени многоъгълници не разглеждаме.

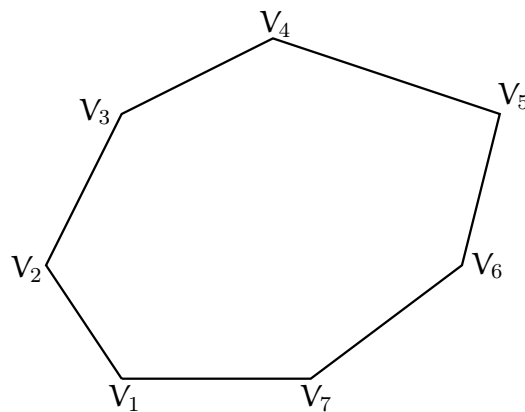
ALG MINIMUM WEIGHT TRIANGULATION($\langle V_1, V_2, \dots, V_n \rangle$: редица от точки, образуваща изпъкнал многоъгълник)

```

1  for  $i \leftarrow 1$  to  $n - 1$ 
2       $T[i, i + 1] \leftarrow 0$ 
3  for  $gap \leftarrow 2$  to  $n - 1$ 
4      for  $i \leftarrow 1$  to  $n - gap$ 
5           $j \leftarrow i + gap$ 
6           $T[i, j] \leftarrow T[i, i + 1] + T[i + 1, j] + \text{dist}(V_i, V_{i+1}) + \text{dist}(V_{i+1}, V_j)$ 
7          for  $k \leftarrow i + 2$  to  $j - 1$ 
8               $T[i, j] \leftarrow \min(T[i, j], T[i, k] + T[k, j] + \text{dist}(V_i, V_k) + \text{dist}(V_k, V_j))$ 
9  return  $T[1, n]$ 

```

Да разгледаме малък пример със седмоъгълник:



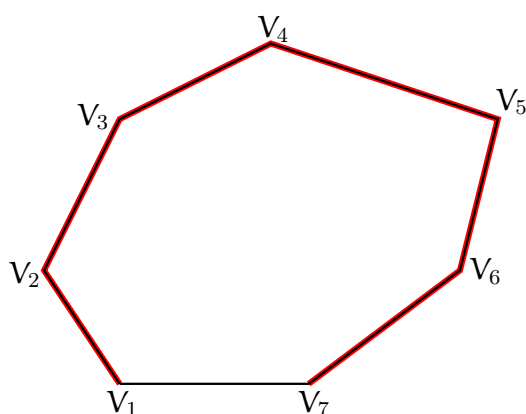
Така изглежда масивът T в началото на алгоритъма. Затъмнени са клетките, които няма да се ползват.

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

След изпълнението на **for**-цикъла на редове 1–2, T изглежда така:

	1	2	3	4	5	6	7
1		0					
2			0				
3				0			
4					0		
5						0	
6							0
7							

Поставянето на червените нули (това са началните условия) отговаря на триангулациите на изродените многоъгълници; тоест, страните на седмоъгълника без V_1V_7 .



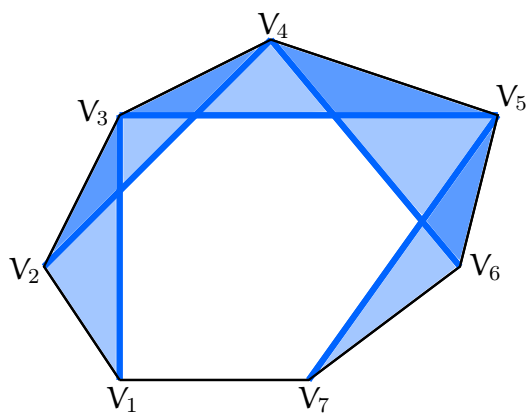
Да разгледаме запълването на следващия диагонал. Имаме $\text{gap} = 2$ от ред 3, $i = 1$ от ред 4 и $j = 3$ от ред 5. На ред 6:

$$T[1, 3] \leftarrow \underbrace{T[1, 2]}_0 + \underbrace{T[2, 3]}_0 + \underbrace{\text{dist}(V_1, V_2)}_0 + \underbrace{\text{dist}(V_2, V_3)}_0$$

така че $T[1, 3]$ става 0. Аналогично, целият диагонал се запълва с нули (в синьо):

	1	2	3	4	5	6	7
1		0	0				
2			0	0			
3				0	0		
4					0	0	
5						0	0
6							0
7							

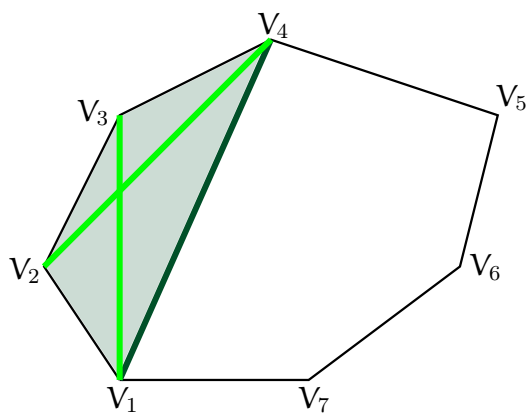
Поставянето на сините нули отговаря на триангулациите на ето тези триъгълници:



Да разгледаме запълването на следващия диагонал. Имаме $\text{gap} = 3$ от ред 3, $i = 1$ от ред 4 и $j = 4$ от ред 5. Редове 6–8 реализират това:

$$T[1, 4] \leftarrow \min \left(\underbrace{T[1, 2]}_0 + \underbrace{T[2, 4]}_0 + \underbrace{\text{dist}(V_1, V_2)}_0 + \underbrace{\text{dist}(V_2, V_4)}_{42.43 \text{ mm}}, \right. \\ \left. \underbrace{T[1, 3]}_0 + \underbrace{T[3, 4]}_0 + \underbrace{\text{dist}(V_1, V_3)}_{35 \text{ mm}} + \underbrace{\text{dist}(V_3, V_4)}_0 \right)$$

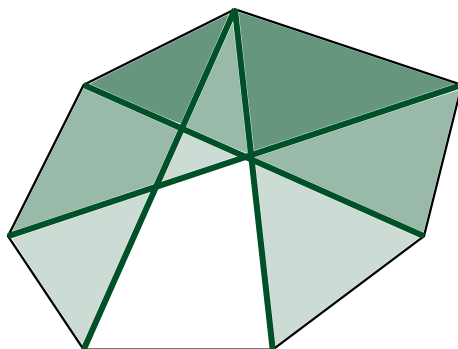
така че $T[1, 4]$ става 35 mm. Във фигурата, това изчисление отговаря на минимална триангулация на зеления четириъгълник $P_{1,4}$, която е по-малкият от двата диагонала V_2V_4 и V_1V_3 .



В масива записваме 35 в клетка $[1, 4]$:

	1	2	3	4	5	6	7
1		0	0	35			
2			0	0			
3				0	0		
4					0	0	
5						0	0
6							0
7							

За да запълним този диагонал, пресмятаме минималните триангулации (тоест, по-малкият от двата диагонала) на тези четири четириъгълника:



След запълването на зеления диагонал, масивът изглежда така:

	1	2	3	4	5	6	7
1		0	0	35			
2			0	0	42.43		
3				0	0	39.05	
4					0	0	43.01
5						0	0
6							0
7							

С това подробната симулация на алгоритъма върху примера спира. Когато гар стане 4, търсим теглото на минимална триангулация на петоъгълници, три на брой; за всеки петоъгълник вземаме минимума от три стойности, отговарящи на трите междинни върха. Когато гар стане 5, търсим теглото на минимална триангулация на два шестоъгълника; за всеки от тях вземаме минимума на четири стойности. И накрая, при гар = 6, намираме теглото на минимална триангулация на целия седмоъгълник като минимума от пет стойности.

Коректността на алгоритъма е очевидна. Сложността му по време е $\Theta(n^3)$.

Алгоритъм за тежкия вариант на задачата се конструира по начин, аналогичен на конструирането на алгоритъм за тежкия вариант на MATRIX-CHAIN MULTIPLICATION на стр. 603.

12.4.3 Неасоциативна операция върху стринг

Дадена е азбука Σ и стринг $x = x_1x_2 \cdots x_n$ над Σ . Нека $|\Sigma| = k$. Дадена е бинарна операция $*$: $\Sigma^2 \rightarrow \Sigma$. Дадена е и буква $\sigma \in \Sigma$. Пита се: има ли скобуване на x , такова че, ако $*$ бъде приложена върху x в реда на това скобуване, резултатът да бъде σ ?

Операцията $*$ да бъде приложена върху x в реда на дадено скобуване на x означава тя да бъде приложена първо върху някои най-вътрешни две букви в скобуването " $(x_i x_{i+1})$ ", резултатът от което е буквата $x_i * x_{i+1}$, и така нататък докрая .

Наместо строга индуктивна дефиниция ще поясним това с пример. Нека $\Sigma = \{a, b, c\}$. Нека $x = abb$, а следната таблица дефинира $*$:

	a	b	c
a	a	c	c
b	a	a	b
c	b	c	a

x има две скобувания: $((ab)b)$ и $(a(bb))$. Ако приложим $*$ съгласно първото скобуване, получаваме $((a * b) * b) = (c * b) = c$. Ако приложим $*$ съгласно второто скобуване, получаваме $(a * (b * b)) = (a * a) = a$. Причината да има разлика е, че $*$ не е асоциативна.

Ако $*$ е асоциативна, задачата се обезсмисля, защото резултатът е един и същи при всяко скобуване. Но при неасоциативна операция, както видяхме, резултатът от прилагането върху различни скобувания може да е различен.

Името на задачата не е универсално прието. Авторът я е срещал като “NON-ASSOCIATIVE MULTIPLICATION”, но това име не е подходящо за тези лекционни записки, понеже може да предизвика объркване със задачата за матричното умножение.

Изч. Задача 34: NON-ASSOCIATIVE OPERATION

екземпляр: Алфавит Σ и стринг $x = x_1x_2 \cdots x_n$ над Σ . Бинарна операция $*$: $\Sigma^2 \rightarrow \Sigma$. Дадена е и буква $\sigma \in \Sigma$.

въпрос: Дали съществува скобуване на x , такова че, ако $*$ бъде приложена съгласно него, резултатът е σ ?

Това е олекотеният вариант на задачата. Тежкият вариант, който няма да разгледаме, е да се изчисли едно скобуване, което дава σ .

NON-ASSOCIATIVE OPERATION прилича много на MATRIX-CHAIN MULTIPLICATION. В задачата MATRIX-CHAIN MULTIPLICATION матричното умножение е асоциативно, но това няма значение, защото там гледаме не каква е матрицата-резултат, а с цената на колко скалярни умножения я постигаме, което вече не е асоциативно. Съществената разлика между NON-ASSOCIATIVE OPERATION, от една страна, и MATRIX-CHAIN MULTIPLICATION и MINIMUM WEIGHT TRIANGULATION, от друга, е че в NON-ASSOCIATIVE OPERATION отговорът не е число и стойностите в таблицата на алгоритъма не са числа. NON-ASSOCIATIVE OPERATION се решава ефикасно от алгоритъм с триъгълна таблица, която се запълва от главния диагонал навън, но сега клетките на таблицата съдържат множества от букви. Ако таблицата се казва T , то $T[i, j]$ има следният смисъл:

$$T[i, j] = \begin{cases} \{x_i\}, & \text{ако } i = j \\ \bigcup_{k=i}^{j-1} \{a * b \mid a \in T[i, k], b \in T[k + 1, j]\}, & \text{ако } i < j \end{cases}$$

Следният алгоритъм изчислява ефикасно отговора съгласно посочената рекурсивна декомпозиция.

ALG NON-ASSOCIATIVE OPERATION($\Sigma, * : \Sigma^2 \rightarrow \Sigma, x = x_1 \cdots x_n \in \Sigma^+, \sigma \in \Sigma$)

```

1  for i ← 1 to n
2    T[i, i] ← {xi}
3  for diag ← 2 to n
4    for i ← 1 to n - diag + 1
5      j ← i + diag - 1
6      T[i, j] ← ∅
7      for k ← i to j - 1
```

```

8      foreach a ∈ T[i, k]
9          foreach b ∈ T[k + 1, j]
10             T[i, j] ← T[i, j] ∪ {a * b}
11  if σ ∈ T[1, n]
12      return 1
13  else
14      return 0

```

Коректността на алгоритъма е очевидна предвид това, че той реализира коректна рекурсивна декомпозиция. Сложността му по време е $\Theta(n^3)$, ако смятаме, че $k = \Theta(1)$. В противен случай трябва да изразим сложността като функция на n и k по този начин: $\Theta(k^2 n^3)$.

Ето малък пример. Нека $\Sigma = \{1, 2, 3, 4\}$, $x = 123412$, $*$ да е дефинирана чрез таблицата

	1	2	3	4
1	1	3	3	1
2	1	1	4	2
3	2	3	1	2
4	1	4	1	4

и нека $\sigma = 4$.

Ето програма на Maple(TM), имплементираща на алгоритъма без редове 11–14, и пускането на програмата върху примера. Очевидно 4 не се съдържа в $T[1, 6]$, така че отговорът е НЕ.

```

% "C:\Program Files\Maple 2018\bin.X86_64_WINDOWS\cmaple"
  ||~|/|      Maple 2018 (X86 64 WINDOWS)
._|||  |/_|. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  / All rights reserved. Maple is a trademark of
 <____> Waterloo Maple Inc.
  |      Type ? for help.
> T := Array(1 .. 6, 1 .. 6, fill = {}):
> x := [1, 2, 3, 4, 1, 2]:
> myoperation := [[1, 3, 3, 1], [1, 1, 4, 2], [2, 3, 1, 2], [1, 4, 1, 4]]:
> for i to 6 do T[i, i] := {x[i]} end do:
> for diag from 2 to 6 do
>   for i from 1 to 6 - diag + 1 do
>     j := i+diag-1:
>     T[i, j] := {}:
>     for k from i to j-1 do
>       for a in op(T[i,k]) do
>         for b in op(T[k+1,j]) do
>           T[i, j] := T[i, j] union {myoperation[a, b]}:
>         end do:
>       end do:
>     end do:
>   end do:
> end do:
> T;

```

```

[ {1}    {3}    {1}    {1, 3}    {1, 2, 3}    {1, 2, 3} ]
[                                     ]

```

[{}	{2}	{4}	{1, 4}	{1}	{1, 2, 3, 4}]
[					]
[{}	{}	{3}	{2}	{1, 2}	{1, 2, 3, 4}]
[					]
[{}	{}	{}	{4}	{1}	{1, 3}]
[					]
[{}	{}	{}	{}	{1}	{3}]
[					]
[{}	{}	{}	{}	{}	{2}]

```
> quit;
```

```
memory used=2.6MB, alloc=8.3MB, time=0.08
```

Забележете, че $T[3, 6] = T[2, 6] = \{1, 2, 3, 4\}$. Това означава, че подстринговете 23412 и 3412 имат скобувания, върху които операцията дава 4. Но това, че 4 може да бъде получен от подстринг на 123412, не означава, че може да бъде получен от целия стринг 123412.

Алгоритъм за тежкия вариант на задачата се конструира по начин, аналогичен на конструирането на алгоритъм за тежкия вариант на MATRIX-CHAIN MULTIPLICATION на стр. 603.

12.4.4 Деривация в контекстно-свободна граматика

Контекстно-свободна граматика (КСГ) е наредена четворка $\Gamma = \langle \Sigma, N, S, R \rangle$, където Σ е крайно множество от терминали (Σ е азбуката), N е крайно множество от нетерминали, такова че $N \cap \Sigma = \emptyset$, $S \in N$ е фиксиран нетерминал, наречен начален нетерминал, и R е множеството от правила (още се наричат продукции) от вида $A \rightarrow X$, където A е нетерминал, а $X \in (\Sigma \cup N)^*$. Формално, R е бинарна релация $R \subseteq N \times (\Sigma \cup N)^*$. Дефинираме бинарната релация $\Rightarrow \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$ така:

$$x \Rightarrow z \stackrel{\text{def}}{\iff} \exists A \in N \exists x_1, x_2 \in (\Sigma \cup N)^* (x = x_1 A x_2 \text{ и има правило } A \rightarrow \alpha \text{ и } z = x_1 \alpha x_2)$$

Релацията $\Rightarrow^*$ е рефлексивното и транзитивно затваряне на $\Rightarrow$. Ако $x \Rightarrow^* y$, казваме, че има *деривация от x до y в Γ* . За всеки $y \in \Sigma^*$ казваме, че y има *деривация в Γ* , ако $S \Rightarrow^* y$. Езикът на граматиката е $L(\Gamma) = \{y \in \Sigma^* \mid S \Rightarrow^* y\}$.

Нека $\epsilon \notin L(\Gamma)$. КСГ е в Нормална Форма на Чомски (пишем кратко КСГНФЧ), ако всички правила са от вида $A \rightarrow BC$ или $A \rightarrow a$, където A, B и C са нетерминали и a е терминал.

Изн. Задача 35: CONTEXT-FREE GRAMMAR DERIVATION, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: КСГНФЧ $\Gamma = \langle \Sigma, N, S, R \rangle$, $y \in \Sigma^+$.

въпрос: Дали y има деривация в Γ ?

Добре известен е следният алгоритъм, изграден по схемата **Динамично Програмиране**. Този алгоритъм се нарича СΥΚ на имената на създателите си Cocke, Younger и Kasami. Алгоритъмът връща или 1, ако $S \Rightarrow^* y$, или 0, в противен случай. За анализа на сложността приемаме, че размерът на Γ е $O(1)$.

Нека $y = y_1 y_2 \cdots y_n$. БОО, нека $n \geq 1$. Ако за всеки $y_i \cdots y_j$, където $1 \leq i \leq j \leq n$, намерим множеството $X_{i,j} = \{A \in N \mid A \Rightarrow^* y_i \cdots y_j\}$, то ние сме готови! Отговорът е 1 тстк $S \in X_{1,n}$. Това, което позволява да направим ефикасен алгоритъм е, че при $i < j$ можем да пресмятаме ефикасно всяко $X_{i,j}$ от множествата $X_{i',j'}$, където $i \leq i' \leq j' \leq j$ и $j' - i' < j - i$.

Базата на рекурсията са множествата $X_{i,i}$ за $1 \leq i \leq n$:

$$\forall i \in \{1, 2, \dots, n\} : X_{i,i} = \{A \in N \mid \text{има правило } A \rightarrow y_i\}$$

защото в КСГНФЧ, ако по време на деривацията в сентенциалната форма се появи нетерминал, той никога не изчезва в смисъл, че в крайния стринг ще има поне един съответстващ на него терминал; ерго, правилата от вида $A \rightarrow BC$ никога не могат да участват в деривация на стринг с една буква.

При $i < j$, $X_{i,j}$ е обединението, по всички нетривиални факторизации $(y_i \cdots y_k) \cdot (y_{k+1} \cdots y_j)$ на стринга $y_i \cdots y_j$ (тоест, по всички $k \in \{i, \dots, j-1\}$) на тези нетерминали, които са левите страни в продукции от вида $A \rightarrow BC$, такива че $B \xRightarrow{*} y_i \cdots y_k$ и $C \xRightarrow{*} y_{k+1} \cdots y_j$. Но тези нетерминали B и C са елементите на множествата $X_{i,k}$ и $X_{k+1,j}$. При изчисляването на $X_{i,j}$ множествата $X_{i,k}$ и $X_{k+1,j}$ са вече известни.

Самият алгоритъм може да се реализира по начин, аналогичен на реализацията на алгоритъма MATRIX-CHAIN MULTIPLICATION.

DERIVATION($\Gamma = \langle \Sigma, N, S, R \rangle$, $y \in \Sigma^*$)

```

1   $n \leftarrow |y|$ 
2  for  $i \leftarrow 1$  to  $n$ 
3       $M[i, i] \leftarrow \{A \mid (A \rightarrow y_i) \in R\}$ 
4  for  $\text{diag} \leftarrow 2$  to  $n$ 
5      for  $i \leftarrow 1$  to  $n - \text{diag} + 1$ 
6           $j \leftarrow i + \text{diag} - 1$ 
7           $M[i, j] \leftarrow \emptyset$ 
8          for  $k \leftarrow i$  to  $j - 1$ 
9              for  $B \in M[i, k]$ 
10                 for  $C \in M[k + 1, j]$ 
11                     for  $P \in R$ 
12                         if  $P = (A \rightarrow BC)$ 
13                              $M[i, j] \leftarrow M[i, j] \cup \{A\}$ 
14 if  $S \in M[1, n]$ 
15     return 1
16 else
17     return 0

```

Обосновката на коректността следва от обосновката на рекурсивната декомпозиция. Сложността по време е $\Theta(n^3)$, ако граматиката има константна големина – тогава както множествата $M[a, b]$, така и множеството от продукции, имат големина, ограничени от константа, от което следва, че и трите най-вътрешни цикъла “въртят” само константен брой пъти.

12.5 Задачи за оптимални подмножества

12.5.1 2-Partition

Дадено е мултимножество A от цели положителни числа. Във варианта на задача за разпознаване се пита дали A има подмултимножество, такова че сумата от неговите числа е равна на сумата на останалите числа. Във варианта на оптимизационна задача се търси такова подмултимножество, че абсолютната стойност на разликата между сумата на неговите

числа и сумата на останалите числата е минимална.

Тази задача е частен случай на задачата KNAPSACK (вижте Подсекция 12.5.3), но решението ѝ си заслужава да бъде разгледано отделно. Ето я, формално написана, но не чрез мултимножества, а чрез обикновени множества, чиито елементи имат размери (*sizes*, оттам и “s” за името на функцията). Функцията на размерите не е непременно инекция, откъдето и възможността за **мултимножество**.

Изч. Задача 36: 2-PARTITION, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Крайно непразно множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

въпрос: Дали съществува $X \subset A$, такова че $\sum_{a \in X} s(a) = \sum_{a \in A \setminus X} s(a)$?

Оттук насетне в Подсекция 12.5.1, “A” и “s” имат смисъла от дефиницията на Задача 36.

Определение 98: Сумата на подмножество на A

Ако $Y \subseteq A$, то “сумата на Y” означава $\sum_{a \in Y} s(a)$. За краткост може да я бележим със “ $\sum Y$ ”.

В тези записки разглеждаме задачата във версията за разпознаване. На практика по-полезната версия е оптимизационната (Задача 37). Ползата от “окастрената” версия за разпознаване е теоретична и ще стане очевидна в Подсекция 16.2.9, където ще покажем нейната алгоритмична неподатливост. Доказателствата за неподатливост са толкова по-прости и лесни за възприемане, колкото по-“окастрена” е задачата, стига “кастренето” да не премахне неподатливостта. А щом версията за разпознаване е неподатлива, няма как оптимизационната версия да е податлива.

Изч. Задача 37: 2-PARTITION, ОПТИМИЗАЦИОННА ВЕРСИЯ

екземпляр: Крайно непразно множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

решение: Множество $X \subset A$, такова че $|\sum X - \sum A \setminus X|$ е минимална.

Пълното име на задачата е SET 2-PARTITION. Както знаем, има и други видове разбивания, примерно целочислено разбиване (Определение 39), поради което има смисъл да се уточни, че става дума за разбиване на множество. Но тук за краткост ще изпускаме “SET”.

Подчертаваме, че терминът “PARTITION” в този контекст няма **нищо общо** нито с алгоритъма PARTITION-SELECT (Алгоритъм 10), който се вика от SELECT (Подсекция 6.2.2), нито с алгоритъма PARTITION, реализиран като PARTITION-Hoare или PARTITION-Lomuto, който се вика от QUICKSORT (Секция 6.4).

Задачата във версия за разпознаване. Да разгледаме сумата на A. Очевидно е, че ако тя е нечетна, отговорът е НЕ. Нека сумата на A е четна и нека B е полусумата на A. Очевидно е, че ако има елемент с размер, надхвърлящ B, отговорът е НЕ. Нека за всяко $i \in \{1, \dots, n\}$, $s(a_i) \leq B$.

Разглеждаме множеството A като наредено с линейна наредба, а именно тази, която идва от имената на елементите: a_1 , после a_2 , ..., после a_n . Това може да изглежда странно, защото тази наредба е абсолютно произволна, а ние ще конструираме решение по отношение на нея. Всъщност няма нищо странно. Всяка линейна наредба на A е еднакво добра за конструиране на решение. Искане се да има **някаква** линейна наредба – точно както казва Skiena на стр. 575. За всяко $i \in \{0, \dots, n\}$, “ A_i ” означава множеството $\{a_1, \dots, a_i\}$. Очевидно $A_0 = \emptyset$ и $A_n = A$.

Разглеждаме произволен ДА-екземпляр (A, s) на задачата. Съществува непразно $X \subset A$, такава, че сумата на X е точно B . X има поне един елемент. // очевидно – е, и?

Продължаваме: X се състои от множество, може би празно, със сума $B - p$, и още един елемент с размер p . // това вече е нещо.

Продължаваме: или $a_n \in X$, в който случай $X \setminus \{a_n\}$ е подмножество на A_{n-1} със сума $B - s(a_n)$, или $a_n \notin X$, в който случай X е подмножество на A_{n-1} със сума B . // аха, това е!

Преди да преминем към рекурсивната декомпозиция, да видим кои са конституентите и кои са състоянията. От най-общи съображения има смисъл да кажем, че конституентите са непразните подмножества на A . Но техният брой е $2^n - 1$, така че е немислимо да получим ефикасно решение, ако то включва изчисления върху (значителна част от) всички тях. Може да подобрим драстично нещата, ако започнем да разсъждаваме в термините на състояния, както беше в решението на задачата APShP (Подсекция 12.2.3). Сега състоянията са наредените двойки (i, j) , където $i \in \{1, \dots, n\}$ и $j \in \{0, \dots, B\}$. Смисълът е този: i представлява A_i , а j е число, което потенциално[†] може да бъде сума на размерите на елементи от A_i . Както и при оптимизационните задачи, състоянията имат цени, но тук възможните цени са две: или ДА, или НЕ. А именно, цената на (i, j) е ДА тстк A_i има подмножество със сума j .

Как да се досетим да използваме такива състояния? Няма рецепта за това какви състояния да използваме в общия случай, ако опитваме да конструираме алгоритъм по схемата **Динамично Програмиране** за някаква нова за нас изчислителна задача. Въпреки че нямаме рецепта, можем да ползваме следните правила. Първо, състоянията не трябва да са прекалено много. Второ, често се налага да си “разширим кръгозора” и да започнем да решаваме по-обща задача; такава, на която оригиналната се явява частен случай. 2-PARTITION е добър пример за последното: оригиналната задача е дали мултимножество от числа се разбива на две подмножества с еднаква сума, а ние решаваме по-общата задача дали мултимножество от числа има подмножество с дадена сума j . Ясно е, че оригиналната задача е частен случай на последната, защото се получава от нея, ако фиксираме j да е полусумата на мултимножеството. Причината да минем към по-общата задача е, че тя е решима с доста очевиден алгоритъм по схемата **Динамично Програмиране**, докато за оригиналната задача едва ли ще измислим добър алгоритъм, ако не мислим за по-общата задача. Това си заслужава да се обясни подробно. Ако мислим **само** за оригиналната задача дали мултимножество от числа се разбива на две подмножества с еднаква сума, **няма** да направим смислена рекурсивна декомпозиция, дори да се досетим да разглеждаме множествата $A_0, A_1, \dots, A_n$ в този ред. Отговорът ДА/НЕ за A_n не може да се изчисли ефикасно само от отговорите ДА/НЕ за $A_{n-1}, \dots, A_1, A_0$. Ако човек си поблъска главата, ще види, че състоянията за $A_{n-1}, \dots, A_1, A_0$ трябва да съдържат и някаква числена компонента. В решението, което строим, числената компонента е j , имащо смисъл на сума на елементи.

Да кажем, че някак сме се досетили да въведем такива състояния (i, j) . Трябва да имаме и мярка за големина на състояние, за да пресмятаме големите състояния от малките, грубо казано. Има смисъл тази мярка да е i . Тогава базата на изчислението е $i = 1$, а след това ДА/НЕ за състоянията с по-големи i се пресмятат от вече изчислени ДА/НЕ за състояния с по-малки i .

И в края на това отклонение с общи разсъждения, нека помислим дали принципът на оптималността (Определение 95) е релевантен за решението на тази задача. Задачата е за разпознаване, но въпреки това принципът на оптималността е релевантен, в някакъв “хлабав” смисъл: ако под “оптимално решение за някакво мултимножество” разбираме “ДА върху това

[†]Казваме “потенциално”, защото за дадено i , не всяко $j \in \{0, \dots, B\}$ е сума от размерите на елементи от A_i . Примерно, ако $i = 1$, има само две числа от $\{0, \dots, B\}$, които са такива суми, а именно 0 и $s(a_1)$.

мултимножество[†], наистина решението за дадено състояние се изчислява от оптималните решения за по-малки състояния.

Да се върнем към рекурсивна декомпозиция. Въвеждаме множество от предикати

$$\mathcal{P} = \{P(i, j) \mid 1 \leq i \leq n, 0 \leq j \leq B\}$$

със следния смисъл:

$$P(i, j) \stackrel{\text{def}}{=} \exists X \subseteq A_i : \sum X = j$$

Очевидно търсеният отговор е ДА тстк $P(n, B) = T$.

Следната рекурсия ни позволява да пресмятаме $\mathcal{P}$ ефикасно.

- Разглеждаме $P(1, j)$, за $0 \leq j \leq B$. Има точно две подмножества на A_1 , сумата от теглата на чиито елементи е от $\{0, \dots, B\}$. А именно,
 - ♦ празното множество $\emptyset$ със сума 0 и
 - ♦ множеството $\{a_1\}$ със сума $s(a_1)$.

Ерго, $P(1, j)$ е истина тстк $j = 0$ или $j = s(a_1)$.

- Разглеждаме $P(i, j)$ при $i > 1$, за $0 \leq j \leq B$. Подмножествата на A_i , сумата от теглата на чиито елементи е от $\{0, \dots, B\}$, се разбиват на:
 - ♦ подмножествата на A_{i-1} , сумата от теглата на чиито елементи е някой елемент от $\{0, \dots, B\}$, и
 - ♦ множествата от вида $X \cup \{a_i\}$, където $X \subseteq A_{i-1}$ и сумата от теглата на елементите на X е от $\{0, \dots, B - s(a_i)\}$.

Ерго, $P(i, j)$ е истина тстк $P(i-1, j)$ е истина **или** $s(a_i) \leq j$ и $P(i-1, j - s(a_i))$ е истина. Това **или** е включващо.

Ето схема за изчисление, базирана на тази рекурсивна декомпозиция.

$$P[1, j] = \begin{cases} T, & \text{ако } j = 0 \text{ или } j = s(a_1) \\ F, & \text{в противен случай} \end{cases} \quad (12.33)$$

$$P[i, j] = \begin{cases} T, & \text{ако } P[i-1, j] \text{ или } (s(a_i) \leq j \text{ и } P[i-1, j - s(a_i)]) \\ F, & \text{в противен случай} \end{cases}, \text{ ако } i > 1 \quad (12.34)$$

Ето алгоритъм по схемата **Динамично Програмиране**, който извършва тези изчисления.

ALG 2-PARTITION($A = \{a_1, \dots, a_n\}, s : A \rightarrow \mathbb{N}^+$)

```

1  if isodd ( $\sum_{i=1}^n s(a_i)$ )
2    return F
3   $B \leftarrow \frac{1}{2} \sum_{i=1}^n s(a_i)$ 
4  if  $\max \{s(a_i) \mid 1 \leq i \leq n\} > B$ 
```

[†]Всяка задача за разпознаване тривиално е максимизационна задача, ако ДА се интерпретира като 1, а НЕ като 0.

```

5   return F
6   създай булев масив P[1 .. n, 0 .. B]
7   инициализирай P с F
8   P[1, 0] ← T, P[1, s(a1)] ← T
9   for i ← 2 to n
10    for j ← 0 to B
11      P[i, j] ← P[i - 1, j] or (s(ai) ≤ j and P[i - 1, j - s(ai)])
12  return P[n, B]

```

Ето пример за работата на алгоритъма. За простота показваме само T, а знаем, че празните клетки съдържат F. Нека $n = 5$ и $s(a_1) = 1$, $s(a_2) = 9$, $s(a_3) = 5$, $s(a_4) = 3$ и $s(a_5) = 8$. Тогава сумата от всички размери е $1 + 9 + 5 + 3 + 8 = 26$. Това е четно число и половината от него е 13. Нито един от размерите не надхвърля 13, така че продължаваме с алгоритъма. Таблицата е с 5 реда и 14 колони, индексирани от 0 до 13. Тя се попълва отгоре надолу, ред по ред. Първият ред отговаря на началните условия: в него записваме точно две T, едно на позиция 0 заради празното множество, а другото на позиция 1, понеже $s(a_1) = 1$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2														
3														
4														
5														

Във втория ред записваме четири T.

- Двете T от P[1, 0] и P[1, 1] се преписват съответно в P[2, 0] и P[2, 1].
- Двете T от P[1, 0] и P[1, 1] “пораждат” T съответно в P[2, 9] и P[2, 10], понеже $s(a_2) = 9$; това е показано с червени стрелки.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2	T	T								T	T			
3														
4														
5														

Да разгледаме третия ред. В него записваме шест—а не осем!“T”.

- T от P[2, 0], P[2, 1], P[2, 9] и P[2, 10] се преписват съответно в P[3, 0], P[3, 1], P[3, 9] и P[3, 10].

- Двете Т от $P[2, 0]$ и $P[2, 1]$ “пораждат” съответно Т в $P[3, 5]$ и $P[3, 6]$, понеже $s(a_3) = 5$. Обаче Т от $P[2, 9]$ и $P[2, 10]$ не “пораждат” Т на третия ред, тъй като $9 + 5 > 13$ и $10 + 5 > 13$; това е причината на третия ред Т да са само шест. С червени стрелки илюстрираме поражданията налясно (с отместване 5).

Р

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Т	Т												
2	Т	Т								Т	Т			
3	Т	Т				Т	Т			Т	Т			
4														
5														

Да разгледаме четвъртия ред. В него записваме единадесет “Т”.

- Т от $P[3, 0]$, $P[3, 1]$, $P[3, 5]$, $P[3, 6]$, $P[3, 9]$ и $P[3, 10]$ се преписват съответно в $P[4, 0]$, $P[4, 1]$, $P[4, 5]$, $P[4, 6]$, $P[4, 9]$ и $P[4, 10]$.
- Двете Т от $P[3, 0]$ и $P[3, 1]$ “пораждат” съответно Т в $P[4, 3]$ и $P[4, 4]$, понеже $s(a_4) = 3$. Т от $P[3, 5]$ поражда Т в $P[4, 8]$ по същата причина. За Т от $P[3, 6]$ не може да кажем, че поражда Т в $P[4, 9]$, защото вече сме записали Т в $P[4, 9]$. Двете Т в $P[3, 9]$ и $P[3, 10]$ пораждат Т съответно в $P[4, 12]$ и $P[4, 13]$. Петте червени стрелки показват поражданията с отместване 3 налясно на елементи Т, които не се получават с преписване от ред 3 надолу. Синята стрелка показва пораждане на елемент Т, който вече е получен с преписване от ред 3 надолу.

Р

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Т	Т												
2	Т	Т								Т	Т			
3	Т	Т				Т	Т			Т	Т			
4	Т	Т		Т	Т	Т	Т		Т	Т	Т		Т	Т
5														

Ето как изглежда окончателната таблица в нашия пример. Петият ред се запълва аналогично. Отместването сега е 8, понеже $s(a_5) = 8$, и има само едно пораждане на Т налясно – Т в $P[4, 3]$ поражда Т в $P[5, 11]$ (червената стрелка), а останалите Т в петия ред са получени с преписване надолу от четвъртия ред (сините стрелки).

Р

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2	T	T								T	T			
3	T	T				T	T			T	T			
4	T	T		T	T	T	T		T	T	T		T	T
5	T	T		T	T	T	T		T	T	T	T	T	T

Екземплярът е Да-екземпляр, тъй като $P[5, 13] = T$ след приключване на алгоритъма.

Коректността на алгоритъма е очевидна предвид (12.33) и (12.34). Интерес представлява изследването на сложността му по време. Тя е $\Theta(nB)$. Множителят n е големината на входа, при допускането, че всяко от числата $s(a_i)$ има големина на представянето $\Theta(1)$. Но B какво е? B е полусумата от **големините** на числата. Тези големина са **произволни** по отношение на n . С други думи, не са ограничени отгоре от каквато и да е функция на n . Ерго, B не е ограничено отгоре. Но тогава сложността не е дефинирана. С други думи, сложността е безкрайност. Разбира се, този парадоксален извод се появява само при допускането на нашия изчислителен модел, че числата във входа имат кодирания с константни дължини за всички големина; с други думи, когато не отчитаме кодиранията им (вижте Подсекция 2.2.5). В по-реалистичен модел на изчисление, който отчита и кодиранията на числата, сложността $\Theta(nB)$ е експоненциална в големината на входа.

Накратко, в изчислителния модел, който типично използваме, ALG 2-PARTITION има недефинирана (безкрайна) сложност, а в по-реалистичен модел има експоненциална сложност. Това не прави алгоритъма безполезен. Ако числата са малки, в смисъл, че имат кодирания, чиито големина можем да смятаме за константи, алгоритъмът е полезен и смислен. Забележете, че ALG 2-PARTITION винаги има “истинска” полиномиална сложност по време, ако числата $s(a_i)$ са кодирани в унарна бройна система! Това не прави алгоритъмът по-бърз, дори напротив, но кодирането в унарна бройна система изкуствено “раздува” големината на входа, а ние мерим сложността като функция именно на големината на входа. Следното определение се основава на този факт.

Определение 99: Псевдополиномиална сложност

Казваме, че алгоритъм A има *псевдополиномиална сложност* по време, ако A има полиномиална сложност по време при условие, че числата във входа са кодирани в унарна бройна система.

И така, ALG 2-PARTITION има псевдополиномиална сложност по време. Не е известен алгоритъм с полиномиална сложност за 2-PARTITION и най-вероятно такъв не съществува. Задачата е **NP**-пълна, както ще видим в Подсекция 16.2.9. При песимистичната хипотеза, че за **NP**-пълните задачи няма полиномиални алгоритми, няма надежда за алгоритъм, драстично по-бърз от ALG 2-PARTITION в най-лошия случай за 2-PARTITION.

Допълнение 61: Още за псевдополиномиалната сложност

Да разгледаме следния фрагмент от алгоритъм.

```

1  for i ← 1 to n
2    { някакъв код, работещ в константно време }

```

Каква е сложността на този фрагмент като функция на n ? Отговорът е $\Theta(n)$, защото има n итерации и всяка от тях се извършва в $\Theta(1)$. А каква е сложността на целия алгоритъм, ако допуснем, че основната работа се върши от този **for**-цикъл, а извън него се върши само константна допълнителна работа? Човек може да се изкуши да каже, че пак е $\Theta(n)$, но сега това не е непременно верният отговор. Сложността на алгоритъма е **функция от големината на входа**. А какъв е входът? Не е ясно, никъде не е показано. Докато не се знае какъв е входът, въпросът е безсмислен.

Ако входът е масив $A[1..n]$ и алгоритъмът е този:

ALG1($A[1..n]$: цели числа)

```

1  s ← 0
2  for i ← 1 to n
3    s ← s + A[i]
4  return s

```

то сложността му е $\Theta(n)$. Ако обаче входът е числото n и алгоритъмът е този:

ALG2($n \in \mathbb{N}^+$)

```

1  s ← 0
2  for i ← 1 to n
3    s ← s++
4  return s

```

то въпросът за сложността на алгоритъма става триков. Дискусиите в Подсекция 2.2.1 Подсекция 2.2.5 са напълно релевантни и тук.

- Ако не отчитаме дължината на кодирането на n , размерът на входа е единица, така че сложността на ALG2 е безкрайност.
- Ако отчитаме дължината на кодирането на n и n е кодирано в двоична позиционна бройна система, или в която и да е друга “смислена” бройна система, то ALG2 е експоненциален алгоритъм.
- Ако отчитаме дължината на кодирането на n и n е кодирано в унарна позиционна бройна система, то ALG2 е линеен алгоритъм.

Забележете, че броят стъпки, които изпълнява ALG2(n), по никакъв начин не намалява, ако n е записано в унарна бройна система, спрямо случая, в който n е записано в бинарна бройна система. Така че квалификациите “експоненциален” и “линеен” може да са подвеждащи. Разликата между тези два случая е **аршинът**, с който мерим! – а не броят на стъпките, които се изпълняват. Ако n е записано унарно, то аршинът, с който мерим, е огромен, и, измерено с него, броят на стъпките е линеен. Ако n е записано бинарно, то аршинът е малък и, ако мерим с него, броят на стъпките е експоненциален. Забележете също така, че при вход $A[1..n]$ можем да мислим, че n -те елементи на масива записват n в унарна бройна система. Поради това, този алгоритъм е линеен


```

ALG3(A: масив от цели числа)
1  s ← 0
2  for i ← 1 to length(A)
3      s ← s++
4  return s

```

Тук елементите на входа не се ползват изобщо, така че цялата роля на масива-вход е да бъде запис на число в унарна бройна система.

Задачата във версия за търсене. Сега вече се иска или някое множество X , за което $\sum_{a \in X} s(a) = \sum_{a \in A \setminus X} s(a)$, или индикация, че такова множество не съществува. Нереалистично е да се опитаме да генерираме всички решения, понеже в най-лошия случай броят им е експоненциален в големината на входа[†]. Следният алгоритъм, който е тривиална модификация на ALG 2-PARTITION, връща и таблица Q , върху която можем да изпълним обратно проследяване с друг алгоритъм.

```

ALG 2-PARTITION, SEARCH VERSION( $A = \{a_1, \dots, a_n\}$ ,  $s : A \rightarrow \mathbb{N}^+$ )
1  if isodd( $\sum_{i=1}^n s(a_i)$ )
2      return "няма решение"
3   $B \leftarrow \frac{1}{2} \sum_{i=1}^n s(a_i)$ 
4  if  $\max\{s(a_i) \mid 1 \leq i \leq n\} > B$ 
5      return "няма решение"
6  създай булев масив  $P[1..n, 0..B]$  и масив от индекси  $Q[1..n, 0..B]$ 
7  инициализирай  $P$  с F и  $Q$  с Nil
8   $P[1, 0] \leftarrow T$ ,  $P[1, s(a_1)] \leftarrow T$ 
9  for i ← 2 to n
10     for j ← 0 to B
11         if  $P[i-1, j]$ 
12              $P[i, j] \leftarrow T$ ,  $Q[i, j] \leftarrow j$ 
13         else if  $s(a_i) \leq j$  and  $P[i-1, j-s(a_i)]$ 
14              $P[i, j] \leftarrow T$ ,  $Q[i, j] \leftarrow j-s(a_i)$ 
15  return  $\langle P, Q \rangle$ 

```

Ето алгоритъм, който връща решение, генерирано с обратно проследяване, или индикира, че решение няма.

```

GENERATE 2-PARTITION, SEARCH VERSION( $n, B \in \mathbb{N}^+$ ,  $A = \{a_1, \dots, a_n\}$ ,  $s : A \rightarrow \mathbb{N}^+$ ,  $P[1..n, 0..B]$ :
булев масив,  $Q[1..n, 0..B]$ : масив от индекси)
1  (*  $P$  и  $Q$  са генерирани от ALG 2-PARTITION, SOLUTION *)
2  if  $P[n, B] = F$ 
3      return "няма решение"
4  else
5      създай празно множество res

```

[†]Представете си, че $s(a) = 1$ за всяко $a \in A$ и $|A|$ е четно. Тогава всяко подмножество X на A , такова че $|X| = \frac{1}{2}|A|$, е решение. Броят на тези подмножества е $\binom{n}{n/2}$. Както знаем от Теорема 35, $\binom{n}{n/2} \asymp \frac{2^n}{\sqrt{n}}$.

```

6      j ← B
7      for i ← n downto 2
8          if Q[i, j] ≠ j
9              res ← res ∪ s(ai)
10             j ← Q[i, j]
11      if Q[2, j] ≠ 0
12          res ← res ∪ s(a1)
13      return res

```

Коректността е напълно очевидна, може би с едно изключение. Третираме първия ред на таблицата по различен начин (11–12) от останалите редове и може да не е ясно защо е така. Причината е, че таблицата P (съответно и Q) няма нулев ред.

Теоретично по-елегантно би било динамичното да използва таблица $P[0..n, 0..B]$, като нулевият ред съответства на A_0 , което е празното множество. Ако бяхме подхождали така, в нулевия ред щяхме да запишем само едно T , а именно в $P[0, 0]$, което би било единственото начално условие, а останалите редове биха се запълвали по начина, който знаем: T от $P[0, 0]$ генерира T в $P[1, 0]$ и $P[1, s(a_1)]$, и така нататък. При този подход би имало $n + 1$ реда, ерго, n на брой прехода от ред към горния ред, които преходи биха съответствали точно $a_n, a_{n-1}, \dots, a_2, a_1$. Поради това **for**-цикълът на редове 7–10 би бил **for i ← n downto 1** и **if**-ът на редове 11–12 не би присъствал.

Но от практическа гледна точка не е добра идея да добавяме такъв нулев ред, понеже това би бил чист преразход на памет. Затова P започва с ред едно и има точно n реда, поради което преходите от ред към горния ред съответстват на $a_n, a_{n-1}, \dots, a_2$. Ерго, a_1 си няма съответен преход от ред към горния ред. Ерго, това дали решението ползва a_1 или не, се определя по начин, който е различен от начина за $a_n, a_{n-1}, \dots, a_2$.

- Дали решението ползва a_i , или не ползва a_i , за $i = n, \dots, 2$, се определя от това, дали съответно $Q[i, j] \neq j - 1$ или $Q[i, j] = j - 1$.
- Дали решението ползва a_1 , или не ползва a_1 , се определя от това, дали съответно $Q[2, j] \neq 0$ или $Q[2, j] = 0$. Не е същото нещо!

Ето как изглежда таблицата Q за примера, който видяхме. N означава Nil.

Q

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	N	N	N	N	N	N	N	N	N	N	N	N	N	N
2	0	1	N	N	N	N	N	N	N	0	1	N	N	N
3	0	1	N	N	N	0	1	N	N	9	10	N	N	N
4	0	1	N	0	1	5	6	N	5	9	10	N	9	10
5	0	1	N	3	4	5	6	N	8	9	10	3	12	13

Показаната с червени стрелки част от обратното проследяване съответства на следните добавяния или не-добавяния към решението:

- a_5 със $s(a_5) = 5$ не се добавя, понеже $Q[5, 13] = 13$,
- a_4 със $s(a_4) = 3$ се добавя, понеже $Q[4, 13] \neq 13$,

- a_3 със $s(a_3) = 5$ не се добавя, понеже $Q[3, 10] = 10$,
- a_2 със $s(a_2) = 9$ се добавя, понеже $Q[2, 10] \neq 10$.

След това добавяме и a_1 със $s(a_1) = 1$, понеже $Q[2, 10] \neq 0$. Крайното решение е $\{a_1, a_2, a_4\}$. С други думи, $\{1, 9, 3\}_M$, като сумата от тези числа наистина е 13.

12.5.2 2 Equal Sum Subsets (2-ESS)

Тази задача е обобщение на 2-PARTITION. Сега се пита дали A има две различни непразни подмножества с еднакви суми. БОО, тези две подмножества нямат общи елементи, защото наличието на общи елементи в тях е без значение за въпроса, на който трябва да се отговори. Накратко ще наричаме задачата “2-ESS”. Забележете, че ако тези две множества представляват разбиване на A , то задачата става 2-PARTITION, поради което казваме, че 2-ESS обобщава 2-PARTITION.

Изч. Задача 38: 2-ESS, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

въпрос: Дали съществуват непразни $X, Y \subset A$, такива че $X \cap Y = \emptyset$ и $\sum X = \sum Y$?

Изч. Задача 39: 2-ESS, ОПТИМИЗАЦИОННА ВЕРСИЯ

екземпляр: Множество $A = \{a_1, a_2, \dots, a_n\}$. Функция $s : A \rightarrow \mathbb{N}^+$.

решение: Непразни $X, Y \subset A$, такива че $X \cap Y = \emptyset$ и $|\sum X - \sum Y|$ е минимална.

Тук разглеждаме само версията за разпознаване. Задачата е **NP**-пълна [158].

Наблюдение 78

Ако $\sum A < 2^n - 1$, то отговорът е тривиално ДА по принципа на Dirichlet: броят на непразните подмножества на A е точно $2^n - 1$ и ако възможните стойности за сума на подмножество са по-малко, то съществуват различни $C, D \subseteq A$, такива че $\sum C = \sum D$. Ако $C \cap D \neq \emptyset$, то $C' = C \setminus D$ и $D' = D \setminus C$ са непразни подмножества на A с празно сечение и $\sum C' = \sum D'$.

Това обаче е неконструктивен резултат. Ние знаем, че има такива подмножества C и D , но нямаме представа кои са или как да ги намерим ефикасно.

Решение на 2-ESS чрез модификация на ALG 2-PARTITION. Следният факт бе посочен на автора на записките от **Мартин Георгиев Георгиев**. Разглеждаме алгоритъма ALG 2-PARTITION на стр. 620. Съществуват две различни непразни $X, Y \subset A$ с празно сечение, такива че $\sum X = \sum Y$, тогава и само тогава, когато съществуват $i \in \{2, \dots, n\}$ и $j \in \{1, \dots, B\}$, такива че $P[i-1, j] = T$ и $s(a_i) \leq j$ и $P[i-1, j-s(a_i)]$. Казано по друг начин, множества X и Y , каквито се искат в задачата 2-ESS, съществуват, ако и само ако, по време на изпълнението на ALG 2-PARTITION(A, s) при поне едно достигане на ред 11 са в сила

- и $P[i-1, j]$,
- и $s(a_i) \leq j$ и $P[i-1, j-s(a_i)]$.

Казано още по-просто, такива X и Y има тстк в таблицата P поне едно T се получава в някоя клетка $P[i, j]$ по двата възможни начина: с преписване от горния ред в същата колона и чрез отнемстване $s(a_i)$ от горния ред, колона $j - s(a_i)$.

Да докажем това формално. В едната посока, нека има непразни $X, Y \subset A$, такива че $X \cap Y = \emptyset$ и $\sum X = \sum Y$. Да кажем, че $X = \{a_{p_1}, \dots, a_{p_q}\}$ и $Y = \{a_{m_1}, \dots, a_{m_t}\}$. Очевидно е, че $\{p_1, \dots, p_q\} \subset \{1, \dots, n\}$, $\{m_1, \dots, m_t\} \subset \{1, \dots, n\}$ и $\{p_1, \dots, p_q\} \cap \{m_1, \dots, m_t\} = \emptyset$. БОО, нека $p_1 < \dots < p_q$ и $m_1 < \dots < m_t$. БОО, нека $p_q < m_t$. Тогава $X \subset A_{m_t}$ и $Y \subset A_{m_t}$. Нека $\sum X = \ell$. Да разгледаме изпълнението на $ALG\ 2-PARTITION(A, s)$, когато $i = m_t$, $j = \ell$ и изпълнението достигне ред 11. Алгоритъмът е коректен, ние вече знаем това, така че в този момент,

- $P[m_t - 1, \ell] = T$, понеже A_{m_t-1} съдържа подмножество—а именно X —с сума ℓ ,
- $P[m_t - 1, \ell - s(a_{m_t})] = T$, понеже A_{m_t-1} съдържа подмножество—а именно $Y \setminus \{a_{m_t}\}$ —с сума $\ell - s(a_{m_t})$.

В другата посока, нека при изпълнението на $ALG\ 2-PARTITION(A, s)$, при някое достигане на ред 11 е вярно, че $i \geq 2$ и $P[i - 1, j] = T$ и $j \geq s(a_i)$ и $P[i - 1, j - s(a_i)] = T$. Тъй като алгоритъмът е коректен, това означава, че

- A_{i-1} съдържа едно множество, да го наречем Y , със сума j , а
- A_{i-1} едно множество, да го наречем X , със сума $j - s(a_i)$.

Очевидно $X \neq Y$, тъй като имат различни суми. Нека X' е множеството $X \oplus \{a_i\}$ (Нотация 13). Тогава $\sum X' = \sum Y = j$, но $X' \neq Y$, понеже $a_i \in X'$ и $a_i \notin Y$. Нямаме гаранция, че $X' \cap Y = \emptyset$. Да разгледаме $X'' = X' \setminus Y$ и $Y'' = Y \setminus X'$. Веднага забелязваме, че $\sum X'' = \sum X' - \sum(X' \cap Y)$ и $\sum Y'' = \sum Y - \sum(X' \cap Y)$. Но тогава $\sum X'' = \sum Y''$, понеже $\sum X' = \sum Y$. Освен това X'' е непразно, понеже съдържа a_i (който е в X' , но не е в Y), така че $\sum X'' > 0$. Щом X'' е непразно и $\sum Y'' = \sum X''$, и Y'' е непразно. Докажахме, че в A има две непразни множества с празно сечение и една и съща сума, а именно X'' и Y'' .

Сложността по време на това решение е $\Theta(nB)$ – точно колкото на $ALG\ 2-PARTITION$.

За илюстрация на това решение може да ползваме фигурата на стр. 622, която е част от илюстрацията на работата на $ALG\ 2-PARTITION$. Да си припомним, че разглежданият екземпляр на $2-PARTITION$ е с мултимножеството $\{1, 9, 5, 3, 8\}_M$. Фигурата на стр. 622 има T на ред 4, колона 9, което се получава по двата споменати начина – преписване отгоре и отгоре ляво. И наистина, в мултимножеството $\{1, 9, 5, 3\}_M$ има непразни подмултимножества с празно сечение и една и съща сума, а именно $\{1, 5, 3\}_M$ и $\{9\}_M$. От друга страна, в мултимножеството $\{1, 9, 5\}_M$ няма такива две подмултимножества, на което отговаря факта, че в предните три показани таблици няма T , което се получава по двата начина.

Решение на 2-ESS, което не се основава на решение на 2-PARTITION. Въпреки че 2-ESS е обобщение на 2-PARTITION, алгоритъмът за 2-ESS, който ще конструираме тук, не решава 2-PARTITION. Този алгоритъм е по-бавен от $ALG\ 2-PARTITION$, тъй като в израза за сложността има един множител B вповече, но авторът смята, че рекурсивната декомпозиция е интересна и заслужава да се разгледа.

Нека $B = \lfloor \frac{1}{2} \sum_{a \in A} s(a) \rfloor$. За разлика от 2-PARTITION, сега не държим $\sum_{a \in A} s(a)$ да е четно. Също като в 2-PARTITION, нека $A_i = \{a_1, \dots, a_i\}$, за $0 \leq i \leq n$. Нека $Q(t, i, j)$, за $1 \leq t \leq n$ и

$0 \leq i, j \leq B$, е предикат със следния смисъл:

$$Q(t, i, j) \stackrel{\text{def}}{=} \exists (X, Y) \subseteq A_t \times A_t : \left(X \cap Y = \emptyset \wedge \sum_{a \in X} s(a) = i \wedge \sum_{a \in Y} s(a) = j \right)$$

Отговорът е ДА тстк съществува $k \in \{1, \dots, B\}$, такова че $Q(n, k, k) = T$.

Може да изчислим ефикасно Q така.

- Разглеждаме $Q(1, i, j)$, за $0 \leq i, j \leq B$. Има точно три наредени двойки $(X, Y) \subseteq A_1 \times A_1$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \dots, B\} \times \{0, \dots, B\}$. А именно,
 - $(\emptyset, \emptyset)$, за която наредената двойка от сумите е $(0, 0)$,
 - $(\{a_1\}, \emptyset)$, за която наредената двойка от сумите е $(s(a_1), 0)$, и
 - $(\emptyset, \{a_1\})$, за която наредената двойка от сумите е $(0, s(a_1))$.

Ерго, $Q(1, i, j)$ е истина тстк $(i, j) = (0, 0)$ или $(i, j) = (s(a_1), 0)$ или $(i, j) = (0, s(a_1))$.

- Разглеждаме $Q(t, i, j)$ при $t > 1$, за $0 \leq i, j \leq B$. Наредените двойки $(X, Y) \in A_t \times A_t$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \dots, B\} \times \{0, \dots, B\}$, се разбиват на:
 - наредените двойки $(X, Y) \in A_{t-1} \times A_{t-1}$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \dots, B\} \times \{0, \dots, B\}$, и
 - наредените двойки $(X, Y) \in A_t \times A_{t-1}$, такива че $X = Z \cup \{a_t\}$, където $Z \subseteq A_{t-1}$, $Z \cap Y = \emptyset$ и $(\sum_{a \in Z} s(a), \sum_{a \in Y} s(a)) \in \{0, \dots, B - s(a_t)\} \times \{0, \dots, B\}$, и
 - наредените двойки $(X, Y) \in A_{t-1} \times A_t$, такива че $Y = Z \cup \{a_t\}$, където $Z \subseteq A_{t-1}$, $X \cap Z = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Z} s(a)) \in \{0, \dots, B\} \times \{0, \dots, B - s(a_t)\}$.

Ерго, $Q(t, i, j)$ е истина тстк $Q(t-1, i, j)$ е истина **или** $(s(a_t) \leq i$ и $Q(t-1, i - s(a_t), j)$ е истина) **или** $(s(a_t) \leq j$ и $Q(t-1, i, j - s(a_t))$ е истина). Това **или** е включващо.

Ето схема за изчисление, базирана на тази рекурсивна декомпозиция. За краткост, наместо " $Q[t, i, j]$ " пишем " $Q^{(t)}[i, j]$ ".

$$Q^{(1)}[i, j] = \begin{cases} T, & \text{ако } i = j = 0 \vee (i = s(a_1) \wedge j = 0) \vee (i = 0 \wedge j = s(a_1)) \\ F, & \text{в противен случай} \end{cases} \quad (12.35)$$

$$Q^{(t)}[i, j] = \begin{cases} T, & \text{ако } Q^{(t-1)}[i, j] \vee (i \geq s(a_t) \wedge Q^{(t-1)}[i - s(a_t), j]) \vee (j \geq s(a_t) \wedge Q^{(t-1)}[i, j - s(a_t)]) \\ F, & \text{в противен случай} \end{cases}, \text{ ако } t > 1 \quad (12.36)$$

Би трябвало да е ясно как да конструираме алгоритъм от (12.35) и (12.36). Коректността е очевидна предвид коректността на рекурсивната декомпозиция, а сложността е $\Theta(nB^2)$.

Алгоритъмът връща ДА или НЕ, понеже задачата е задача за разпознаване. Нещо важно: след като изгради цялата тримерна таблица $Q^{(t)}[i, j]$, алгоритъмът **не връща** просто съдържанието на дадена клетка на $Q^{(n)}[i, j]$! Алгоритъмът връща ДА, ако за поне едно $i \in \{1, \dots, n\}$ е изпълнено $Q^{(n)}[i, i] = T$; в противен случай връща НЕ. С други думи, ако гледаме на $Q^{(n)}[i, j]$ като на двумерна, $(B+1) \times (B+1)$ таблица, алгоритъмът преглежда главния ѝ диагонал—без $Q^{(n)}[0, 0]$, която клетка задължително съдържа Т, отговарящо на празното решение, което не ни интересува—за да реши дали да върне Т или F.

Да видим пример. Разглеждаме екземпляр на 2-ESS, в който $n = 4$, $s(a_1) = 2$, $s(a_2) = 3$, $s(a_3) = 4$ и $s(a_4) = 5$. Тогава $B = 7$. Ще си представим $Q^{(1)}, \dots, Q^{(4)}$ като четири отделни таблици, като $Q^{(1)}$ е базата (12.35), а всяка от следващите, да кажем $Q^{(t)}$, се попълва съгласно (12.36), използвайки предната таблица и съответното $s(a_t)$.

За да получим $Q^{(1)}$, слагаме Т в клетка $[0, 0]$ и, тъй като $s(a_1) = 2$, още по едно Т в $[0, 2]$ и $[2, 0]$. Тогава $Q^{(1)}$ е:

	0	1	2	3	4	5	6	7
0	T		T					
1								
2	T							
3								
4								
5								
6								
7								

За да получим $Q^{(2)}$, преписваме единиците от $Q^{(1)}$. После, тъй като $s(a_2) = 3$, слагаме по едно Т на разстояние 3, по хоризонтал и вертикал, от досега сложените Т. Тогава $Q^{(2)}$ е (новите Т са в синьо):

	0	1	2	3	4	5	6	7
0	T		T	T		T		
1								
2	T			T				
3	T		T					
4								
5	T							
6								
7								

За да получим $Q^{(3)}$, преписваме Т от $Q^{(2)}$. После, тъй като $s(a_3) = 4$, слагаме по едно Т на разстояние 4, по хоризонтал и вертикал, от досега сложените Т. Тогава $Q^{(3)}$ е (новите единици са в синьо):

	0	1	2	3	4	5	6	7
0	T		T	T	T	T	T	T
1								
2	T			T	T			T
3	T		T		T		T	
4	T		T	T		T		
5	T				T			
6	T			T				
7	T		T					

Аналогично получаваме $Q^{(4)}$, като имаме предвид, че $s(a_4) = 5$:

	0	1	2	3	4	5	6	7
0	T		T	T	T	T	T	T
1								
2	T			T	T	T		T
3	T		T		T	T	T	T
4	T		T	T		T		T
5	T		T		T	T	T	T
6	T			T		T		
7	T		T	T	T	T		T

С червен фон са посочени единиците върху главния диагонал. Както знаем, отговорът е ДА тук има поне едно Т върху главния диагонал. В случая това е изпълнено, така че за нашия пример алгоритъмът връща ДА.

Забележете, че Т в $Q^{(4)}[5, 5]$ отговаря на решението $\{\{2, 3\}, \{5\}\}$, а Т в $Q^{(4)}[7, 7]$ отговаря на решението $\{\{2, 5\}, \{3, 4\}\}$.

12.5.3 Задачата за раницата (Knapsack)

Тази задача е много стара и се разисква в статия от 1957 г. [36]. Съвременната формулировка често е следната. Крадец иска да открадне предмети, като ги сложи в раница. Всеки предмет има положителни тегло и цена. Теглото и цената са напълно независими. Крадецът иска да задигне предмети с максимална сумарна цена. Но той не може просто да грабне всички предмети, защото има ключово ограничение: раницата се къса, ако бъде претоварена. По-точно казано, дадено е максимално сумарно тегло за предметите в раницата, което е *капацитетът* на раницата. Крадецът трябва да вземе предмети с максимална сумарна цена, но със сумарно тегло, ненадхвърлящо капацитета на раницата.

Тази задача има огромни и разнообразни практически приложения, далеч отвъд нелегалната дейност, описана току-що. Едно доста очевидно приложение е алокирането на пари при ограничение върху общата сума: имаме списък с неща за купуване, като всяко нещо има цена и освен това важност за нас, която важност се измерва с някакво положително число. Имаме финансов лимит за покупките (иначе бихме купили всичко), в който трябва да се вмести. При тази формулировка, цените отговарят на теглата от предната формулировка, важностите отговарят на цените от предната формулировка, а лимитът отговаря на капацитета от предната формулировка. Задачата става следната: да намерим подмножество от неща със сумарна цена, ненадхвърляща лимита, но с максимална важност, която е сумата от индивидуалните важности на нещата.

Fractional Knapsack. Ако предметите не са атомарни, а са неогранично делими, можем да кажем с известна доза условност, че е дадена *континуална* версия на задачата, която често се нарича *Fractional Knapsack*. Примерно, нещата за крадене са златен прах, сребърен прах и бронзов прах. Континуалната версия на задачата е решима тривиално от алчен алгоритъм, който първо пресмята относителните цени на делимите предмети—да кажем, че килограм златен прах е по-скъп от килограм сребърен прах, който на свой ред е по-скъп от килограм бронзов прах—и после пълни раницата с относително най-скъпото нещо (златен прах в нашия пример), докато не достигне капацитета ѝ или това нещо не се изчерпи. Ако относително най-скъпото нещо първо се изчерпи, алчният алгоритъм продължава да пълни раницата с второто относително най-скъпо нещо, и така нататък. Очевидно този алгоритъм е ефикасен и намира оптимално решение.

Discrete Knapsack. Ако предметите са атомарни, ще казваме, че е дадена *дискретна* версия на задачата. В дискретната версия алчният подход може да доведе до решение, което не е оптимално – вземайки предмет с най-високо отношение цена/тегло, може да се лишим от възможност да сложим други предмети, които биха били по-изгодни като цяло, въпреки че поотделно имат по-лошо отношение цена/тегло. С други думи, локалната, или късогледата, алчност сега не е печеливша стратегия; сега трябва да сме “глобално алчни”. Ето пример за това, че локално алчната идея не работи. Нека са дадени три предмета с цени 7, 5 и 5 и съответни тегла 6, 5 и 5, а капацитетът на раницата е 10. Най-добро отношение цена към тегло има първият предмет: $7/6 \approx 1.17$. За втория и третия предмет отношението цена към тегло е $5/5 = 1$. Ерго, алчният алгоритъм слага a_1 в раницата, в раницата остава свободен капацитет $10 - 6 = 4$ и това е крайт, понеже вторият и третият имат тегло 5 всеки и нито един от тях не може да бъде взет. Печалбата на крадеца е 7. А ако беше взел втория и третия предмет, които заедно се побират в раницата, щеше да е с печалба $5 + 5 = 10$.

Алчната идея обаче директно води до ефикасен апроксимиращ алгоритъм [8, стр. 42–43], който връща решение със сумарна цена V , такава че $V \geq \frac{1}{2}V_{\text{opt}}$, където V_{opt} е оптималната сумарна цена за дадения вход. В тези записки дискутираме апроксимиращи алгоритми в Лекция 17.

Skiena [144, стр. 429] посочва проста евристика, която често дава добри резултати – *масшабиране*, в оригинал *scaling*. Избираме някаква прагова стойност за капацитета на раницата, която е във възможностите на нашата изчислителна техника. Да наречем този прагов капацитет $\hat{C}$. Ако ни е даден екземпляр с капацитет C , такъв че $C > \hat{C}$, умножаваме теглата с коефициент $\frac{\hat{C}}{C}$, закръглявайки до най-близкото цяло число (теглата **трябва** да са целочислени!), и решаваме за този екземпляр, използвайки **Динамично Програмиране**. Полученото решение директно ни дава решение за оригиналния екземпляр. Skiena казва, че тази евристика работи добре на практика.

Дискретната версия на задачата е широко известна в поне три варианта, които ще разгледаме сега.

12.5.3.1 Unbounded Knapsack

Изч. Задача 40: UNBOUNDED KNAPSACK

екземпляр: Множество от видове предмети $A = \{a_1, \dots, a_n\}$, като има неограничено много предмети от всеки вид. За всеки вид a_i , $1 \leq i \leq n$, са дадени неговата цена $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е и капацитет $C \in \mathbb{N}^+$.

решение: $(x_1, \dots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^n x_i w(a_i) \leq C \quad (12.37)$$

$$\sum_{i=1}^n x_i v(a_i) \text{ е максимална} \quad (12.38)$$

Лесно можем да преведем това формално описание в терминологията на крадеца с раницата. C е капацитетът на раницата. Предметите за крадене са от n вида, като видовете са $a_1, \dots, a_n$, а от всеки вид има неограничено много предмети за крадене. Под “неограничено много” нямаме предвид безброй много, а нещо много по-реалистично: за всеки вид предмети, крадецът може да напълни раницата догоре с предмети от само този вид. С други думи,

никога няма да се изчерпят предметите от даден вид, запълвайки раницата. Предметите от всеки вид a_i са неразличими помежду си и имат една и съща цена $v(a_i)$ и едно и също тегло $w(a_i)$. Наредената n -орка от естествени числа $(x_1, \dots, x_n)$ ни казва по колко броя от всеки вид ще бъдат сложени в раницата. (12.37) и (12.38) казват, че трябва да се вземат по толкова предмети от всеки вид, че общото тегло да не надхвърли капацитета, като тази подборка трябва да има максимална сумарна цена.

Олекотеният вариант. Да направим характеристика на някое оптимално решение X .

X е непразно и се побира в раница с капацитет C .

// дотук е тривиално ...

Продължаваме: решението X се получава от някое подрешение Y , като Y може и да е празно, чрез добавяне на един предмет от някой вид a_i .

// е, и?

Продължаваме: Y се побира в раница с капацитет $C - w(a_i)$.

// това вече е нещо.

Продължаваме: оптималното решение X е мултимножество от предмети. То се получава от някое подмултимножество Y с добавяне на един предмет от някой вид a_i , като при това сумарното тегло не надхвърля C . Очевидно Y е оптимално решение за раница с капацитет $C - w(a_i)$.

// аха, това е!

Преди да преминем към рекурсивната декомпозиция, да видим кои са конституентите и кои са състоянията. Също както при 2-PARTITION на стр. 618, конституентите са непразните подмножества на A , техният брой е $2^n - 1$ и е немислимо да получим ефикасно решение, ако то включва изчисления върху (значителна част от) всички тях. Също както при 2-PARTITION, подобряваме нещата, въвеждайки състояния. За разлика от решението на 2-PARTITION обаче, състоянията сега са всички възможни “частични капацитети” $0, 1, \dots, C - 1, C$, а не наредените двойки (i, j) от индекс i и частичен капацитет j , както беше при 2-PARTITION.

Да помислим защо състоянията в решението на UNBOUNDED KNAPSACK са по-прости тези в решението на 2-PARTITION, откъдето и таблицата на алгоритъма за UNBOUNDED KNAPSACK е n пъти по-малка от таблицата на алгоритъма за 2-PARTITION. Причината е, че тези състояния-частични капацитети са достатъчни за адекватно описание на конституентите в контекста на UNBOUNDED KNAPSACK. Имайки оптимални решения за състояния $0, 1, \dots, C - 1$, бързо—тоест, в линейно време, опитвайки всеки вид предмет—можем да изчислим оптимално решение за състояние C .

В контраст с това, при 2-PARTITION в оптималното решение не може да има повтаряне на елементи, така че не може просто да опитаме да добавим елемент от всеки вид към оптималните решения за състояния $0, 1, \dots, C - 1$, за да видим дали има подобрение при минаване към състояние (капацитет) C . При 2-PARTITION се налага да знаем за всеки частичен капацитет измежду $0, 1, \dots, C - 1$ и кои елементи участват в съответното оптимално решение. Идеята да увеличим състоянията при 2-PARTITION с един индекс i , превръщайки ги в наредени двойки (индекс, частичен капацитет), се оказа печеливша за 2-PARTITION.

Накратко, няма смисъл да имаме компонента i в състоянията, когато решаваме UNBOUNDED KNAPSACK. Не че не бихме могли да го сторим, но това би добавило ненужно един множител n в израза за сложността на алгоритъма по памет. Поради това състоянията са частичните капацитети. Големината на състояние е самото число. Изграждаме решение за големите състояния от решения за малките състояния.

Принципът на оптималността (Определение 95) е спазен, понеже оптимално решение за капацитет C се получава от оптимално решение за капацитет C' , където $C' < C$.

Да помислим за рекурсивна декомпозиция. Да кажем, че числата $0, 1, \dots, C - 1$ са *редуцираните капацитети*. Ако за всеки редуциран капацитет j знаем цената на оптимално решение,

което се избира в раница с капацитет j , то можем да намерим оптимално решение за пълния капацитет C , като за всяко j и всеки вид предмет a_i опитаем да добавим предмет от вид a_i към раница с капацитет j . При това игнорираме решения със сумарно тегло над C , а от останалите вземаме решение с максимална цена.

Следната рекурсивна декомпозиция е добре известна. Нека $M[j]$ е максималната цена на предмети, които можем да сложим в раница с капацитет j , за всички $j \in \{0, \dots, C\}$.

$$M[j] = \begin{cases} 0, & \text{ако } j = 0 \\ \max(M[j-1], \max_{i \in \{1, \dots, n\} \text{ и } w(a_i) \leq j} (M[j - w(a_i)] + v(a_i))), & \text{ако } j > 0 \end{cases}$$

Търсеното решение е $M[C]$.

Идеята е ясна. Ако раницата има нулев капацитет, в нея не можем да сложим нищо. Ако позволим капацитет $j > 0$, в нея можем да сложим най-много:

- или това, което можем да сложим при по-малкия капацитет $j - 1$; с други думи не се възползваме от целия капацитет j ;
- или, възползвайки от целия капацитет j , опитваме последователно да сложим един предмет от всеки вид, чието тегло не надхвърля j . В крайна сметка слагаме предмет a_i , който максимизира сумарната цена. Но при положение, че капацитетът е j , за да сложим предмет от вид a_i , трябва да има “свободно място” поне $j - w(a_i)$. Максимумът, който можем да постигнем, слагайки предмет от вид a_i , е максималната цена за раница с капацитет $j - w(a_i)$ плюс цената $v(a_i)$.

Следният алгоритъм пресмята тази рекурсия ефикасно.

ALG UNBOUNDED KNAPSACK($A = \{a_1, \dots, a_n\}$, $v : A \rightarrow \mathbb{R}^+$, $w : A \rightarrow \mathbb{N}^+$, $C \in \mathbb{N}^+$)

```

1  създай масив  $M[0..C]$  от числа
2   $M[0] \leftarrow 0$ 
3  for  $j \leftarrow 1$  to  $C$ 
4       $M[j] \leftarrow M[j-1]$ 
5      for  $i \leftarrow 1$  to  $n$ 
6          if  $w(a_i) > j$ 
7              continue
8          else
9               $M[j] \leftarrow \max(M[j], M[j - w(a_i)] + v(a_i))$ 
10 return  $M[C]$ 
```

Сложността по време е $\Theta(nC)$, а по памет е $\Theta(C)$. Това е поредният псевдополиномиален алгоритъм, който разглеждаме. Ако числата $w(a_i)$ са малки, масивът M е малък и алгоритъмът е ефикасен, в практическия смисъл.

Трябва да имаме предвид, че UNBOUNDED KNAPSACK е **NP**-пълна задача във версията за разпознаване (Подсекция 16.2.11), така че, при широко приетите допускания, няма драстично по-бърз алгоритъм от ALG UNBOUNDED KNAPSACK.

За частния случай $n = 2$ има истински полиномиален алгоритъм за UNBOUNDED KNAPSACK, но този алгоритъм изобщо не е очевиден и е предмет на научни статии [72], [88]. Както се

посочва в [113, стр. 92], наивният алгоритъм при $n = 2$ е тестване за $j \in \left\{0, \dots, \left\lfloor \frac{C}{w(a_1)} \right\rfloor\right\}$ за $x_1 = j$ и $x_2 = \left\lfloor \frac{C - j \cdot w(a_1)}{w(a_2)} \right\rfloor$ и това е псевдополиномиален, но не и полиномиален, алгоритъм.

Тежкият вариант. Следният алгоритъм решава задачата в тежкия вариант. Сега $M[j]$ не е число, а наредена тройка $(val, prev, obj)$, където $val \in \mathbb{R}$, $prev \in \{-1, 0, \dots, C - 1\}$ и $obj \in \{0, \dots, n\}$, със следния смисъл:

- $M[j].val$ е максималната цена, постижима при капацитет j ,
- $M[j].prev$ ни дава информация за това, как сме изчислили $M[j].val$.
 - ♦ Ако $M[j].val$ е получено от присвояването на ред 4, то увеличаването на капацитета от $j - 1$ на j не ни дава възможност за увеличаване на цената; в този случай $M[j].prev = j - 1$.
 - ♦ Ако $M[j].val$ е получено от присвояване на ред 10, то увеличаването на капацитета от $j - 1$ на j ни дава възможност за увеличаване на цената; в този случай $M[j].prev = j - w(a_i)$ за стойност на i , за която се постига максимум на $M[j].val$ след сравненията на ред 9; формално,

$$M[j].prev = j - w(a_{\arg\max \{M[j - w(a_i)] + v(a_i) \mid i \in \{1, \dots, n\}\}})$$

- $M[j].obj$ ни дава информация за това, кой вид обект да добавим към обектите със сумарно тегло $M[j].prev$, за да получим оптималната цена $M[j].val$.
 - ♦ Ако $M[j].obj = 0$, то не добавяме нищо (обект от вид 0 няма). Това се случва точно тогава, когато $M[j].val$ е получено от присвояването на ред 4.
 - ♦ В противен случай, $M[j].obj$ е видът на обект, който максимизира сумарната цена на предметите в раницата, бивайки добавен към раница с капацитет $M[j].prev$; ; формално,

$$M[j].obj = \arg\max \{M[j - w(a_i)] + v(a_i) \mid i \in \{1, \dots, n\}\}$$

ALG UNBOUNDED KNAPSACK, SOLUTION($A = \{a_1, \dots, a_n\}$, $v : A \rightarrow \mathbb{R}^+$, $w : A \rightarrow \mathbb{N}^+$, $C \in \mathbb{N}^+$)

```

1  създай масив  $M[0..C]$  от наредени тройки от числа  $(val, prev, obj)$ 
2   $M[0].val \leftarrow 0$ ,  $M[0].prev \leftarrow -1$ ,  $M[0].obj \leftarrow 0$ 
3  for  $j \leftarrow 1$  to  $C$ 
4     $M[j].val \leftarrow M[j - 1].val$ ,  $M[j].prev \leftarrow M[j - 1].prev$ ,  $M[j].obj \leftarrow 0$ 
5    for  $i \leftarrow 1$  to  $n$ 
6      if  $w(a_i) > j$ 
7        continue
8      else
9        if  $M[j].val < M[j - w(a_i)].val + v(a_i)$ 
10          $M[j].val \leftarrow M[j - w(a_i)].val + v(a_i)$ 
11          $M[j].prev \leftarrow j - w(a_i)$ 
12          $M[j].obj \leftarrow i$ 
13  return  $M$ 
```

Забележете, че $M[j].prev = M[j-1].prev$ **не е същото** като $M[j].val = M[j-1].val$. В примера за работата на алгоритъма, който ще разгледаме след малко (Таблица 12.5), виждаме, че и решението $(1, 0, 1)$ на редове 11 и 12, и решението $(0, 1, 1)$ на ред 13 се получава от решението $(0, 0, 1)$ на ред 8, с добавяне на съответно елемент от първия вид и от втория вид. Поради това $M[11].prev = M[12].prev = M[13].prev = 8$, но $M[11].val = M[12].val = 16.19$, а $M[13].val = 18.89$.

Би трябвало да е очевидно, че **ALG UNBOUNDED KNAPSACK, SOLUTION** връща масив M , от който можем да генерираме оптимално решение чрез обратно проследяване във време $O(C)$.

Сложността по време е $\Theta(nC)$ и сложността по памет е $\Theta(C)$, точно както при **ALG UNBOUNDED KNAPSACK**.

При този подход, оптимално решение се намира чрез обратно проследяване в изходната таблица M , проследявайки $M[j].prev$ стойностите от $M[C].prev$ назад до $M[0].prev$. Алтернативен подход е да използваме вектори $(x_1, \dots, x_n) \in \mathbb{N}^n$ за описание на решенията за частичните капацитети. Забележете, че в Задача 40 решението е точно такова: x_i е броят на предметите от i -ия вид, за $1 \leq i \leq n$. Но ако записваме решенията за частичните капацитети с n -вектори, сложността по памет ще нарастне до $\Theta(nC)$.

Пример за работата на ALG UNBOUNDED KNAPSACK, SOLUTION. Нека $n = 3$, $C = 30$, а теглата, цените и относителните цени са следните:

i	1	2	3
$v(a_i)$	3.3	6.0	12.89
$w(a_i)$	3	5	8
$v(a_i)/w(a_i)$	1.1	1.2	1.61125

Таблица 12.5 показва $M[j]$, за $j \in \{0, \dots, 30\}$, и съответните състояния на вектора (x_1, x_2, x_3) . Забележете, че алгоритъмът не генерира вектори $(x_1, x_2, \dots, x_n)$ никъде. В Таблица 12.5, векторите (x_1, x_2, x_3) са за удобство на читателя.

j	M[j].val	M[j].prev	M[j].obj	(x_1, x_2, x_3)
0	0.0	-1	0	(0, 0, 0)
1	0.0	-1	0	(0, 0, 0)
2	0.0	-1	0	(0, 0, 0)
3	3.3	0	1	(1, 0, 0)
4	3.3	0	0	(1, 0, 0)
5	6	0	2	(0, 1, 0)
6	6.6	3	1	(2, 0, 0)
7	6.6	3	0	(2, 0, 0)
8	12.89	0	3	(0, 0, 1)
9	12.89	0	0	(0, 0, 1)
10	12.89	0	0	(0, 0, 1)
11	16.19	8	1	(1, 0, 1)
12	16.19	8	0	(1, 0, 1)
13	18.89	8	2	(0, 1, 1)
14	19.49	11	1	(2, 0, 1)
15	19.49	11	0	(2, 0, 1)
16	25.78	8	3	(0, 0, 2)
17	25.78	8	0	(0, 0, 2)
18	25.78	8	0	(0, 0, 2)
19	29.08	11	3	(1, 0, 2)
20	29.08	11	0	(1, 0, 2)
21	31.78	16	2	(0, 1, 2)
22	32.38	19	1	(2, 0, 2)
23	32.38	19	0	(2, 0, 2)
24	38.67	16	3	(0, 0, 3)
25	38.67	16	0	(0, 0, 3)
26	32.38	16	0	(0, 0, 3)
27	41.97	24	1	(1, 0, 3)
28	41.97	24	0	(1, 0, 3)
29	44.67	24	2	(0, 1, 3)
30	45.27	27	1	(2, 0, 3)

Таблица 12.5: Пример за работата на ALG UNBOUNDED KNAPSACK, SOLUTION.

12.5.3.2 0-1 Knapsack

В този вариант на задачата има само един предмет от всеки вид, така че няма смисъл да говорим за видове предмети. Задачата на крадеца е дали да вземе даден предмет или не, а не колко броя от даден вид да вземе, както беше в UNBOUNDED KNAPSACK.

Изч. Задача 41: 0-1 КНАПСАК

екземпляр: Множество от предмети $A = \{a_1, \dots, a_n\}$. За всяко a_i , $1 \leq i \leq n$, са дадени неговата цена $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е и капацитет $C \in \mathbb{N}^+$.

решение: Подмножество $X \subseteq A$, такова че:

$$\sum_{a \in X} w(a) \leq C$$

$$\sum_{a \in X} v(a) \text{ е максимална}$$

Бихме могли да дефинираме и този вариант на задачата чрез вектор $(x_1, \dots, x_n)$, само че булев, а не от естествени числа. Това би бил характеристичен вектор, а знаем, че характеристичните вектори определят подмножества. Така че дори с такава формулировка, това, което се има предвид, е подмножество с максимална сумарна цена и сумарно тегло, не по-голямо от капацитета.

Тази задача е интересна, ако капацитетът не е прекалено голям. Прекалено голям капацитет е капацитет, поне равен на сумата от теглата на предметите. Ако капацитетът е прекалено голям в този смисъл, задачата става тривиална – крадецът слага в раницата всички предмети и постига максимална печалба. Нещата стават нетривиални, ако не всички предмети влизат в раницата и крадецът трябва да изчисли какво да вземе и какво да остави. Забележете разликата с UNBOUNDED KNAPSACK, при която задачата не се тривиализира, колкото и да е голям капацитетът.

Наистина, 0-1 KNAPSACK прилича повече на 2-PARTITION (Подсекция 12.5.1), отколкото на UNBOUNDED KNAPSACK. Също както в решението на 2-PARTITION, и тук не можем да правим рекурсивна декомпозиция само по подмножествата от първите i елемента на A . Ето пример за това. Нека $n = 4$, $C = 15$, а теглата, цените и относителните цени са следните:

i	1	2	3	4
$v(a_i)$	19	20	17	43
$w(a_i)$	8	7	5	10
$v(a_i)/w(a_i)$	2.375	2.857	3.4	4.3

Очевидно оптималното решение е $\{a_3, a_4\}$ със сумарно тегло $5 + 10 = 15$ и сумарна цена $17 + 43 = 60$. Оптималното решение за $\{a_1, a_2, a_3\}$ обаче е $\{a_1, a_2\}$ със сумарно тегло $8 + 7 = 15$ и сумарна цена $19 + 20 = 39$. Ясно е, че оптималното решение за целия екземпляр не “ползва” оптималното решение за $\{a_1, a_2, a_3\}$; иначе казано, принципът на оптималността не е в сила, ако гледаме на целия екземпляр като състоящ се от $\{a_1, a_2, a_3\}$ и $\{a_4\}$.

Да конструираме рекурсивната декомпозиция подробно. Ето съображения, водещи към нея. Разглеждаме произволен екземпляр $(A = \{a_1, \dots, a_n\}, v, w, C)$ на задачата. Нека и тук, за всяко $i \in \{0, \dots, n\}$, “ A_i ” означава $\{a_1, \dots, a_i\}$, като $A_0 = \emptyset$ и $A_n = A$. Да направим характеристикация на някое оптимално решение $X \subset A$.

X е непразно и се побира в раница с капацитет C .

// дотук е тривиално ...

Продължаваме: X се състои от множество Y и още един елемент, като Y , което може и да е празно, не съдържа въпросния елемент.

// е, и?

Продължаваме: $X = \{a_{i_1}, \dots, a_{i_k}\}$ за някакво $\{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$, където $k \geq 1$ и освен това $i_1 < \dots < i_k$. Тогава множеството X се състои от множеството $Y = \{a_{i_1}, \dots, a_{i_{k-1}}\}$ и елемента a_{i_k} .

// това вече е нещо.

Продължаваме: Y се побира в раница с капацитет $C - w(a_i)$.

// така е.

Продължаваме: Y се побира в раница с капацитет $C - w(a_i)$ и освен това Y е подмножество на $\{a_{i_1}, \dots, a_{i_k} - 1\}$.

// аха, това е!

За конституентите и състоянията в тази задача са в сила съображенията на стр. 618, касаещи 2-PARTITION. Конституентите са непразните подмножества на A и са необятно много, а състоянията са наредените двойки (i, j) , където $i \in \{0, \dots, n\}$ и $j \in \{0, \dots, C\}$ със следния смисъл: i представлява A_i , а j е редуциран капацитет.

Ще направим двумерна рекурсивна декомпозиция, далечно подобна на рекурсивната декомпозиция за решението на 2-PARTITION на стр. 619. Нека $M[i, j]$ е цената на оптимално решение за раница с капацитет j , ползващо предмети измежду $a_1, \dots, a_i$, където $0 \leq i \leq n$ и $0 \leq j \leq C$.

$$M[i, 0] = 0, \text{ за } 1 \leq i \leq n \quad // \text{ нулев капацитет}$$

$$M[0, j] = 0, \text{ за } 1 \leq j \leq C \quad // \text{ няма предмети за слагане}$$

$$M[i, j] = M[i - 1, j], \text{ ако } i > 0 \text{ и } j > 0 \text{ и } w(a_i) > j \quad // \text{ } a_i \text{ не се побира при капацитет } j$$

$$M[i, j] = \max \left(\underbrace{M[i - 1, j]}_{\text{не вземаме } a_i}, \underbrace{M[i - 1, j - w(a_i)] + v(a_i)}_{\text{вземаме } a_i} \right), \text{ ако } i, j > 0 \text{ и } w(a_i) \leq j \quad (12.39)$$

Решението за олекотения вариант е $M[n, C]$.

Случай (12.39) е основата на декомпозицията и може би единственият, чиято обосновка не е напълно очевидна. Имаме две възможности.

- Да не вземем предмета a_i . Цената на такова решение ни е известна и тя е $M[i - 1, j]$.
- Да вземем предмета a_i . Помним, че всичко това е по отношение на раница с капацитет j . Щом изобщо разглеждаме възможността да сложим a_i в раница с капацитет j , необходимо е да е изпълнено $w(a_i) \leq j$. Това, разбира се, не е достатъчно. Ние ще вземем a_i само ако сумата от тези двете:

- ♦ максималната сумарна цена за предмети измежду $a_1, \dots, a_{i-1}$ при капацитет $w(a_i) - j$, и
- ♦ цената на a_i .

е по-голяма от максималната сумарна цена за предмети измежду $a_1, \dots, a_{i-1}$ при капацитет j .

Следователно, оптималната сумарна цена, която може да получим от a_i и някакви други предмети измежду $a_1, \dots, a_{i-1}$ при капацитет j е $M[i - 1, j - w(a_i)] + v(a_i)$. Това е тривиално следствие от принципа на оптималността, който е в пълна сила: би било грешка да вземем не-оптимално решение за капацитет $j - w(a_i)$. Поради това и изчисляваме максимума на $M[i - 1, j]$ и $M[i - 1, j - w(a_i)] + v(a_i)$.

Олекотеният вариант. Алгоритъм, който реализира рекурсията (12.39) и връща само оптималната цена, е следният.

0-1 KNAPSACK($A = \{a_1, \dots, a_n\}$, $v : A \rightarrow \mathbb{R}^+$, $w : A \rightarrow \mathbb{N}^+$, $C \in \mathbb{N}^+$)

```

1  for i ← 0 to n
2      M[i, 0] ← 0
3  for j ← 0 to C
```

```

4   M[0, j] ← 0
5   for i ← 1 to n
6     for j ← 1 to C
7       if w(ai) > j
8         M[i, j] ← M[i - 1, j]
9       else
10        M[i, j] ← max(M[i - 1, j], M[i - 1, j - w(ai)] + v(ai))
11  return M[n, C]

```

Коректността на алгоритъма е очевидна предвид коректността на (12.39). Сложността както по време, така и по памет, е $\Theta(nC)$.

Тежкият вариант. Ако искаме да получим и някое оптимално решение, може да подходим по следния начин. За всяка клетка $M[i, j]$ записваме две неща:

1. оптималната цена на решение x за капацитет j при използване на предмети от A_i ;
2. дали решението x ползва a_i или не.

Това може да се реализира със структура, представляваща подмножество на A , примерно булев вектор, но тук подходиме другояче – по начин, аналогичен на ALG 2-PARTITION, SEARCH VERSION на стр. 624. Клетките на M сега не са числа, а наредени двойки. За $0 \leq i \leq n$ и $0 \leq j \leq C$, $M[i, j].val$ е оптималната стойност на решение x за капацитет j при използване на предмети от A_i , а $M[i, j].prev$ показва дали x ползва a_i :

- ако $i \geq 1$ и $j \geq 1$, то

$$M[i, j].prev = \begin{cases} j - w(a_i), & \text{ако } a_i \text{ участва в решението} \\ j, & \text{в противен случай} \end{cases}$$

- ако $i = 0$ или $j = 0$, то $M[i, j].prev = \text{Nil}$.

0-1 KNAPSACK, SOLUTION($A = \{a_1, \dots, a_n\}$, $v: A \rightarrow \mathbb{R}^+$, $w: A \rightarrow \mathbb{N}^+$, $C \in \mathbb{N}^+$)

```

1   for i ← 0 to n
2     M[i, 0].val ← 0
3     M[i, 0].prev ← Nil
4   for j ← 0 to C
5     M[0, j].val ← 0
6     M[0, j].prev ← Nil
7   for i ← 1 to n
8     for j ← 1 to C
9       if w(ai) > j
10        M[i, j].val ← M[i - 1, j].val
11        M[i, j].prev ← j
12      else
13        if M[i - 1, j - w(ai)] + v(ai) > M[i - 1, j].val
14          M[i, j].val ← M[i - 1, j - w(ai)] + v(ai)
15          M[i, j].prev ← j - w(ai)
16      else

```



```

17         M[i, j].val ← M[i - 1, j].val
18         M[i, j].prev ← j
19 return M

```

От изходната таблица лесно може да се генерира едно оптимално решение чрез обратно проследяване: стартирайки от клетка $M[n, C]$, проследяваме поинтърите $M[i, j].prev$, докато не достигнем Nil, като:

- ако $M[i, j].prev = M[i - 1, j].prev$, то a_i не се добавя към решението,
- ако $M[i, j].prev \neq M[i - 1, j].prev$, то a_i се добавя към решението.

Ето малък пример за работата на алгоритъма. Ще ползваме примера, който вече разгледахме, а именно $n = 4$, $C = 15$, $v = \langle 19, 20, 17, 43 \rangle$ и $w = \langle 8, 7, 5, 10 \rangle$. Това е таблицата, която алгоритъмът връща. Поинтърите са записани с червени стрелки за по-лесно възприемане, като стойностите Nil не са записани.

			редуцирани капацитети															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
предмети	0	v w	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	19 8	0	0	0	0	0	0	0	0	19	19	19	19	19	19	19	19
	2	20 7	0	0	0	0	0	0	0	20	20	20	20	20	20	20	20	39
	3	17 5	0	0	0	0	0	17	17	20	20	20	20	20	37	37	37	39
	4	43 10	0	0	0	0	0	17	17	20	20	20	43	43	43	43	43	60

След като разполагаме с тази таблица, тривиално е да направим обратно проследяване от клетка $[4, 15]$ до клетка, чийто поинтър е Nil, и да намерим оптимално решение $\{a_3, a_4\}$.

			редуцирани капацитети															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
предмети	0	v w	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	19 8	0	0	0	0	0	0	0	0	19	19	19	19	19	19	19	19
	2	20 7	0	0	0	0	0	0	0	20	20	20	20	20	20	20	20	39
	3	17 5	0	0	0	0	0	17	17	20	20	20	20	20	37	37	37	39
	4	43 10	0	0	0	0	0	17	17	20	20	20	43	43	43	43	43	60

12.5.3.3 Bounded Knapsack

Сега са дадени определен брой копия от всеки предмет. Тази версия на задачата е обобщение на 0-1 KNAPSACK, където беше дадено точно по един предмет от всеки вид. Тук ще разгледаме решение от книгата на Martello и Toth [113].

Изч. Задача 42: BOUNDED KNAPSACK

екземпляр: Множество от видове предмети $A = \{a_1, \dots, a_n\}$. За всеки вид a_i , $1 \leq i \leq n$, са дадени неговата цена $v(a_i) \in \mathbb{R}^+$, неговото тегло $w(a_i) \in \mathbb{N}^+$ и броят предмети от него $b_i \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.

решение: $(x_1, \dots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^n x_i w(a_i) \leq C \quad (12.40)$$

$$0 \leq x_i \leq b_i, \text{ за } 1 \leq i \leq n \quad (12.41)$$

$$\sum_{i=1}^n x_i v(a_i) \text{ е максимална} \quad (12.42)$$

БОО, нека $\sum_{i=1}^n b_i w(a_i) > C$, защото в противен случай имаме тривиално решение $x_i = b_i$ за $1 \leq i \leq n$.

Задачата се свежда до 0-1 KNAPSACK със следната трансформация. Нека е даден екземпляр Π_B на BOUNDED KNAPSACK:

$$\langle \{a_1, \dots, a_n\}, (v_1, \dots, v_n), (w_1, \dots, w_n), (b_1, \dots, b_n), C \rangle$$

На него съответства екземпляр $\Pi_{0,1}$ на 0-1 KNAPSACK:

$$\langle \{\hat{a}_1, \dots, \hat{a}_m\}, (\hat{v}_1, \dots, \hat{v}_m), (\hat{w}_1, \dots, \hat{w}_m), C \rangle$$

За всеки вид a_i , $1 \leq i \leq n$, конструираме $\lceil \log_2(b_i + 1) \rceil$ предмети в $\Pi_{0,1}$ така:

- първо, $\lfloor \log_2 b_i \rfloor$ предмети, чиито наредени двойки (цена, тегло) са

$$(v_i, w_i), (2v_i, 2w_i), (4v_i, 4w_i), \dots, (\lfloor \log_2 b_i \rfloor v_i, \lfloor \log_2 b_i \rfloor w_i)$$

- второ, ако b_i не е число от вида “точна степен на двойката минус единица”, още един предмет с цена

$$b_i v_i - (v_i + 2v_i + 4v_i + \dots + \lfloor \log_2 b_i \rfloor v_i)$$

и размер

$$b_i w_i - (w_i + 2w_i + 4w_i + \dots + \lfloor \log_2 b_i \rfloor w_i)$$

Очевидно броят на предметите в $\Pi_{0,1}$, съответни на a_i , е $\lceil \log_2(b_i + 1) \rceil$, като сумата от цените им е $b_i v_i$ и сумата от размерите им е $b_i w_i$. Общо броят на предметите в $\Pi_{0,1}$ е $\sum_{i=1}^n \lceil \log_2(b_i + 1) \rceil$. Накратко ще записваме това число като m

m е и броят на булевите променливи в решението на $\Pi_{0,1}$. Ерго, на всяка променлива-естествено число x_i от решението Π_B съответстват булеви променливи $\hat{x}_{i,1}, \dots, \hat{x}_{i, \lceil \log_2(b_i + 1) \rceil}$ от решението на $\Pi_{0,1}$. Нещо повече, за $1 \leq h \leq \lceil \log_2(b_i + 1) \rceil$, булевата променлива $\hat{x}_{i,h}$ “дава” n_h предмети от Π_B , където

$$n_h = \begin{cases} 2^{h-1}, & \text{ако } h < \lceil \log_2(b_i + 1) \rceil \\ b_i - \sum_{j=1}^{\lceil \log_2(b_i + 1) \rceil - 1} 2^{j-1}, & \text{ако } h = \lceil \log_2(b_i + 1) \rceil \end{cases}$$

Следователно $x_i = \sum_{h=1}^{\lceil \log_2(b_i+1) \rceil} n_h \hat{x}_{i,h}$ може да приеме всяка стойност от $\{0, \dots, b_i\}$.

Сложността на алгоритъма е сложността на трансформацията плюс сложността на 0-1 KNAPSACK върху конструирания екземпляр. Сложността на трансформацията е $\Theta(m)$, а след това 0-1 KNAPSACK работи във време $\Theta(mC)$, така че общата сложност е $\Theta(mC)$. Да се направи подробно формално доказателство на коректността би било доста дълго, така че ще разгледаме примера за трансформацията от [113, стр. 83].

Като пример, да разгледаме следния екземпляр на BOUNDED KNAPSACK, в който $n = 3$ и $C = 10$.

	a_1	a_2	a_3
v	10	15	11
w	1	3	5
b	6	4	2

На око се вижда, че оптималното решение е $(x_1, x_2, x_3) = (6, 1, 0)$ с обща цена

$$6 \cdot 10 + 1 \cdot 15 + 0 \cdot 11 = 75$$

и общо тегло

$$6 \cdot 1 + 1 \cdot 3 + 0 \cdot 5 = 9$$

което очевидно не нарушава капацитета 10.

Да приложим трансформацията. Имаме

$$\lceil \log_2(b_1 + 1) \rceil = \lceil \log_2(6 + 1) \rceil = 3$$

$$\lceil \log_2(b_2 + 1) \rceil = \lceil \log_2(4 + 1) \rceil = 3$$

$$\lceil \log_2(b_3 + 1) \rceil = \lceil \log_2(2 + 1) \rceil = 2$$

Ерго, $m = 3 + 3 + 2 = 8$. Нека трите предмета, съответстващи на a_1 , са a'_1 , a''_1 и a'''_1 , трите предмета, съответстващи на a_2 , са a'_2 , a''_2 и a'''_2 , и двата предмета, съответстващи на a_3 , са a'_3 и a''_3 . Това са осемте предмета в екземпляра на 0-1 KNAPSACK, който строим. Цените и теглата са следните.

	a'_1	a''_1	a'''_1	a'_2	a''_2	a'''_2	a'_3	a''_3
v	10	20	30	15	30	15	11	11
w	1	2	3	3	6	3	5	5

Новият капацитет съвпада с оригиналния и е 10.

$a'_1 = 1 \cdot 10 = 10$, $a''_1 = 2 \cdot 10 = 20$, а $a'''_1 = b(a_1) \cdot v(a_1) - (a'_1 + a''_1) = 60 - 30 = 30$. Останалите цени и тегла се получават аналогично. Ако пуснем този екземпляр на 0-1 KNAPSACK в knapsack solver с $C = 10$, получаваме оптимално решение с цена 75 и тегло 9 при

$$(x'_1, x''_1, x'''_1, x'_2, x''_2, x'''_2, x'_3, x''_3) = (11110000)$$

Имената на променливите x съответстват по очевиден начин на предметите. Решението (11110000) означава вземане на следните предмети и само тях.

- Предмети a'_1 , a''_1 и a'''_1 . Тоест, $1 + 2 + 3 = 6$ предмети от вида a_1 .
- Предмет a'_2 . Тоест, 1 предмет от вида a_2 .

Точно както в решението на око.

12.6 Задачи за планиране (Scheduling)

При тези задачи е дадено множество от някакви дейности, всяка от които се дефинира чрез начало във времето, край във времето и може би тегло, и се търси оптимално подмножество или оптимално разбиване.

12.6.1 Задачата за безработния актьор (Interval Scheduling)

Тази задача е известна още с имена като “ACTIVITY-SELECTION” [34, стр. 415] или INTERVAL SCHEDULING. Дадено е непразно множество $A = \{a_1, \dots, a_n\}$ от отворени интервали, като $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leq i \leq n$. Ако мислим за интервалите като за дейности във времето, дейност a_i се дефинира чрез своето начало s_i и своя край f_i .

Определение 100: Съвместими интервали, безконфликтно множество от интервали

Казваме, че два интервала са *съвместими*, ако имат празно сечение. В противен случай те са *несъвместими*. Казваме, че множество от интервали е *безконфликтно*, ако всеки два различни интервала от него са съвместими.

Забележете, че a_i и a_j имат празно сечение тстк $f_i < s_j$ или $f_j < s_i$. Ако $f_i = s_j$, то a_i и a_j имат празно сечение, тъй като са отворени. Също така забележете, че напълно съвпадащи—и като начално, и като крайно време—интервали не може да има, защото идентифицираме всеки интервал с наредената двойка от началното му и крайното му време, а A е множество.

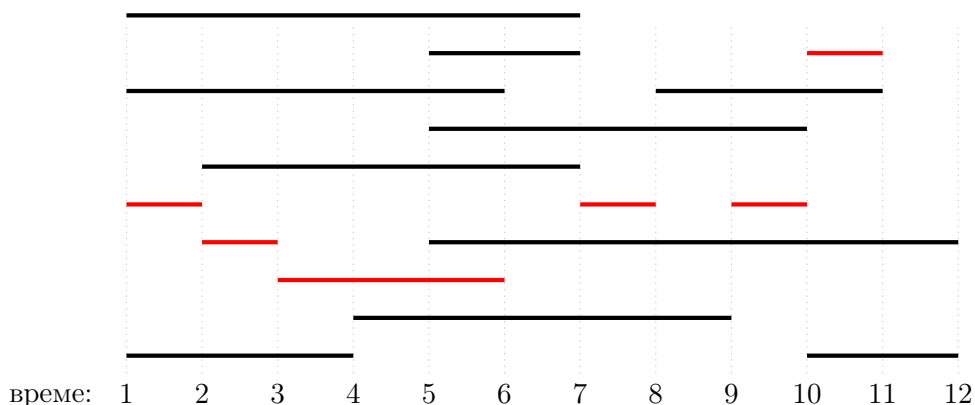
Изч. Задача 43: INTERVAL SCHEDULING

екземпляр: Множество от отворени интервали A .

решение: Безконфликтно $A' \subseteq A$ с максимална мощност.

Тази задача още се нарича “задачата за безработния актьор”, защото може да мислим, че има актьор, който си търси работа и иска да се снима в клипове, като клиповете са интервали във времето – снимането на всеки клип има начало и край, които задават съответния интервал. Актьорът не може просто да бъде ангажиран във всички клипове, защото в общия случай клиповете се застъпват взаимно във времето, а в даден момент той не може да е ангажиран в повече от един клип. Ако всички клипове дават едно и също възнаграждение, актьорът би искал да намери колкото може повече взаимно незастъпващи се клипове. Ако всеки клип си има свое възнаграждение, актьорът би искал да намери подмножество клипове, взаимно незастъпващи се, за които сумарното възнаграждение е максимално. Това са съответно нетегловната и тегловната версия на задачата.

Ето пример с шестнадесет интервала. Оптимално решение е подмножеството от шестте интервала, оцветени в червено. Очевидно то е безконфликтно. Ако не е очевидно, че няма по-голямо безконфликтно множество, от изложението в Подсекция 12.6.1.1 ще стане ясно, че това наистина е така (че няма по-голямо безконфликтно множество).



12.6.1.1 Алчно решение

Ще разгледаме четири идеи, всяка от които може да е в основата на алчен алгоритъм.

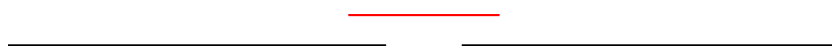
- ① ПЪРВИЯТ НАЛИЧЕН ИНТЕРВАЛ. Актьорът взема клипа, чието начало е най-рано. След това всички клипове, които са несъвместими с този клип, отпадат. Измежду останалите клипове отново взема този с най-ранно начално време, и така нататък.

Тази идея не работи в смисъл, че има примери, върху които дава неоптимални решения. Ето контрапример: актьорът взема първата започваща работа (червения интервал) и изтървава четири други.



- ② ПЪРВО НАЙ-КЪСИЯТ ИНТЕРВАЛ. Актьорът взема клипа, който е най-къс. След това всички клипове, които са несъвместими с този клип, отпадат. Измежду останалите клипове отново взема този, който е най-къс, и така нататък.

Тази идея също не работи. Ето контрапример: актьорът взема най-късата във времето работа (червения интервал) и изтървава две други.



- ③ ПЪРВО ИНТЕРВАЛЪТ С НАЙ-МАЛКО КОНФЛИКТИ. Актьорът взема клипа, за който има най-малко други клипове, с които е несъвместим. След това всички клипове, които са несъвместими с избрания клип, отпадат. Измежду останалите клипове отново взема този, който е нъвместим с най-малко други клипове, и така нататък.

Тази идея също не работи. Ето контрапример: актьорът взема работата с най-малко конфликти (червения интервал, който е в конфликт само с два други интервала) и губи възможността да вземе оптималното решение от четири клипа (в синьо).



- ④ **НАЙ-РАННО ПРИКЛЮЧВАНЕ.** Актьорът взема клипа, който приключва най-рано. След това всички клипове, които са несъвместими с избрания клип, отпадат. Измежду останалите клипове отново взема този, който приключва най-рано, и така нататък.

Тази идея работи. Читателят лесно може да се убеди, че, следвайки тази идея, в примера от ① актьорът ще вземе четири клипа, в примера от ② актьорът ще вземе два клипа и в примера от ③ актьорът ще вземе четири клипа, като всички тези решения са оптимални. Но това, че тази идея работи, трябва да се докаже формално.

Теорема 101: НАЙ-РАННО ПРИКЛЮЧВАНЕ е оптимално за INTERVAL SCHEDULING

За всеки екземпляр на INTERVAL SCHEDULING, подмножеството, конструирано съгласно стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ, е максимално по мощност безконфликтно множество от интервали.

Доказателство: Това, че стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ конструира безконфликтно подмножество от интервали, е очевидно. Ще докажем, че конструираното множество е максимално по мощност.

Да допуснем противното. Нека $A = \{a_1, \dots, a_n\}$ е контрапример с минимална мощност. Да кажем, че стратегията НАЙ-РАННО ПРИКЛЮЧВАНЕ върху A конструира $A' = \{a_{i_1}, \dots, a_{i_k}\}$. Както казахме, очевидно A' е безконфликтно. Щом е безконфликтно, върху него съществува естествена линейна наредба. БОО, нека наредбата е такава, че

$$\begin{aligned} f_{i_1} &\leq s_{i_2} \\ f_{i_2} &\leq s_{i_3} \\ &\dots \\ f_{i_{k-2}} &\leq s_{i_{k-1}} \\ f_{i_{k-1}} &\leq s_{i_k} \end{aligned}$$

Тогава можем да мислим за A' като за редица $A' = (a_{i_1}, \dots, a_{i_k})$.

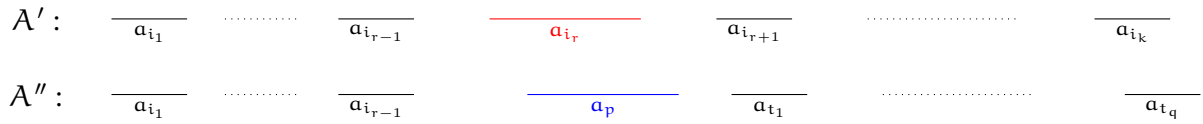
Допускането, че A е контрапример с минимална мощност, е важно. Това означава, че за всяко множество интервали B , такова че $|B| < |A|$, НАЙ-РАННО ПРИКЛЮЧВАНЕ връща оптимално решение.

Щом A е контрапример, съществува безконфликтно $A'' \subseteq A$, такова че $|A''| > |A'|$. Да разгледаме A' и A'' . Ще докажем, че е невъзможно $A' \subset A''$. Да допуснем, че $A' \subset A''$. Тогава A'' се състои от интервалите в A' и още поне един интервал a_m , който не е в A' . Този a_m е или вляво от най-левия интервал на A' , или между два интервала на A' , или вдясно от най-десния интервал на A' . И трите случая са невъзможни – лесно се вижда, че НАЙ-РАННО ПРИКЛЮЧВАНЕ би сложила и a_m в A' . Провергахме допускането, че $A' \subset A''$.

И така, съществува поне един интервал в A' , който не се съдържа в A'' . Нека

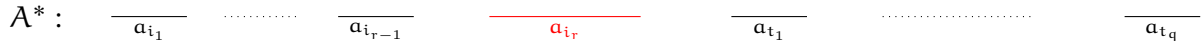
$$r = \min \{j \in \{1, \dots, k\} \mid a_{i_j} \notin A''\}$$

Това означава, че, ако гледаме на A' и на A'' като на редици, редицата $(a_{i_1}, \dots, a_{i_{r-1}})$ е максималното общо начало на A' и A'' . Забележете, че тя е празна, ако $r = 1$. Нека a_p е интервалът в A'' , който е непосредствено след $a_{i_{r-1}}$, ако $r > 1$, или е първият интервал в A'' , ако $r = 1$. Такъв a_p съществува, иначе няма да е изпълнено $|A''| > |A'|$. Нека подредицата на A'' вдясно от a_p е $(a_{t_1}, \dots, a_{t_q})$. Очевидно $|A''| = r + q$, така че $r + q > k$. Ето как изглеждат A' и A'' в общия случай.



Ключовото наблюдение е, че $f_{i_r} \leq f_p$, точно както е показано на илюстрацията, защото a_{i_r} и a_p са различни, а A' е конструирана съгласно евристиката НАЙ-РАННО ПРИКЛЮЧВАНЕ. Дали $s_{i_r} < s_p$, както е показано на илюстрацията, или не, не е съществено. Важното е, че $f_{i_r} \leq f_p$.

Нека $A^* = (A'' \setminus \{a_p\}) \cup \{a_{i_r}\}$. Веднага се вижда, че A^* е безконфликтно множество от интервали, подмножество на A , и $|A^*| = |A''|$.



Нека

$$\begin{aligned}
 \overline{A^r} &= \{a \in A \mid (\exists j \in \{1, \dots, r\} : a \text{ е несъвместим с } a_{i_j})\} \\
 \tilde{A} &= A \setminus (\overline{A^r} \cup \{a_{i_1}, \dots, a_{i_r}\})
 \end{aligned}$$

Но тогава $(a_{i_{r+1}}, \dots, a_{i_k})$ се явява безконфликтно множество интервали в $\tilde{A}$ с мощност $k - r$, генерирано от евристиката НАЙ-РАННО ПРИКЛЮЧВАНЕ, а $(a_{t_1}, \dots, a_{t_q})$ се явява безконфликтно множество интервали в $\tilde{A}$ с мощност q . Но вече отбелязахме, че $r + q > k$, което е същото като $q > k - r$. Но тогава $\tilde{A}$ се явява контрапример с мощност, по-малка от $|A|$, в противоречие с допускането, че A е контрапример с минимална мощност.

Полученото противоречие показва, че такъв контрапример A не съществува. $\square$

Допълнение 62: Алчността на алчните алгоритми: глупава или умна

Нека е дадено някакво крайно непразно множество S . Знаем разликата между “максимално по мощност подмножество на S ” и “максимално по включване подмножество на S спрямо свойство π ”^a. В контекста на задачата INTERVAL SCHEDULING, свойството е безконфликтност. На английски съответните термини са “maximum” и “maximal”.

Ако за всяко $S' \subseteq S$, изчисляването на предиката $\pi(S')$ е ефикасно, то има очевиден ефикасен алчен алгоритъм за намиране на максимално по включване подмножество U . Изграждаме решението U итеративно. В началото $U \leftarrow \emptyset$. Докато можем, правим следното: за всеки елемент от $x \in S \setminus U$, в произволен ред, пресмятаме дали $\pi(U \cup \{x\})$ е истина.

- За първия x в този ред, за който $\pi(U \cup \{x\})$ е истина, прекъсваме текущата итерация, правим $U \leftarrow U \cup \{x\}$, и итериране отново спрямо новото U .
- Ако за никой x не получим истина, спираме итериранията.

Този процес спира след краен брой стъпки и точно едно от тези две е вярно:

- Текущото U е същинско подмножество на S , тоест, $S \setminus U$ е непразно, обаче няма елемент $x \in S \setminus U$, за който $\pi(U \cup \{x\})$ е истина.
- Текущото U съвпада с S .

И в двата случая, $\mathcal{U}$ е максимално по включване подмножество на S спрямо свойството π и алгоритъмът връща $\mathcal{U}$.

За задачата INTERVAL SCHEDULING бихме могли да подходим по подобен начин, но полученото решение, макар и максимално по включване, в общия случай не било максимално по мощност. Подходът, за който става дума тук, е алчен, но е **алчен по глупав начин**: той избира нов интервал за добавяне напълно произволно. Алгоритъмът INTERVAL SCHEDULING, който решава задачата, е основан на евристиката НАЙ-РАННО ПРИКЛЮЧВАНЕ. Той също е алчен, но е **алчен по интелигентен начин**: критерият му за избор на нов интервал за добавяне е основан на Теорема 101.

И така, ако искаме да получим решение, което е максимално по включване—тоест, не може да бъде подобрено—можем да използваме глупаво алчния подход. За да получим решение, което е максимално по мощност, този подход (най-често) не работи и, ако има алчно решение, то е алчно по умен начин.

^aДа си припомним, че π е свойство на подмножества, а не на елементи, на S . Ерго, ако $S' \subseteq S$, то $\pi(S')$ е или лъжа, или истина, но ако $s \in S$, то $\pi(s)$ не е нито лъжа, нито истина, а е невалиден израз.

Следният алгоритъм решава Задача 43, базирайки се на НАЙ-РАННО ПРИКЛЮЧВАНЕ.

INTERVAL SCHEDULING, GENERAL($A = \{a_1, \dots, a_n\}$, където $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leq i \leq n$)

```

1  сортирай A по f-стойностите, резултатът е  $\langle a'_1, \dots, a'_n \rangle$ 
2   $A' \leftarrow \emptyset$ 
3  for i  $\leftarrow$  1 to n
4      ако няма несъвместимост между  $a'_i$  и елемент от  $A'$ , добави  $a'_i$  към  $A'$ 
5  return  $A'$ 
```

Ключово за ефикасността на алгоритъма е ефикасността на проверката на ред 4. Няма нужда да проверяваме за съвместимост между a_i , от една страна, и всеки интервал в текущия A' , от друга. Напълно достатъчно е да проверяваме само дали a_i е съвместим с интервала, който е добавен последно в A' . Ако a_i е съвместим с него, то той е съвместим с всеки интервал в A' заради транзитивността на релацията $<$. Ако a_i не е съвместим с него, просто го прескачаме. И така, достатъчно е да помним в някаква променлива крайното време f^* на интервала, последно сложен в A' , и да сравняваме s_i с f^* . Ето подробностите.

INTERVAL SCHEDULING, DETAILED($A = \{a_1, \dots, a_n\}$, където $n \geq 1$, $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leq i \leq n$)

```

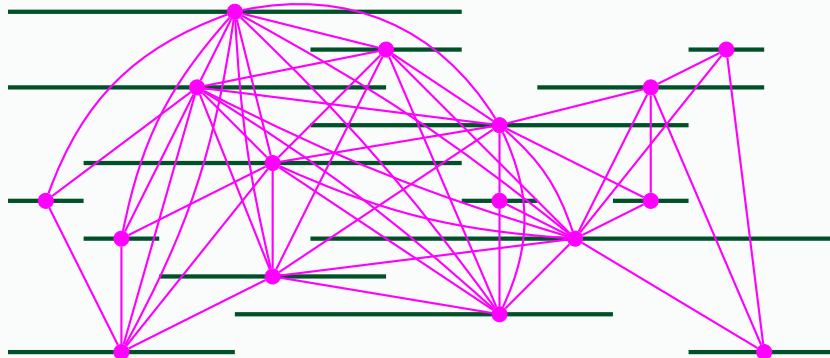
1  сортирай A по f-стойностите, резултатът е  $\langle a'_1, \dots, a'_n \rangle$ 
2   $A' \leftarrow \{a'_1\}$ 
3   $f^* \leftarrow f'_1$ 
4  for i  $\leftarrow$  2 to n
5      if  $f^* \leq s'_i$ 
6           $A' \leftarrow A' \cup \{a'_i\}$ 
7           $f^* \leftarrow f'_i$ 
8  return  $A'$ 
```

Коректността се обосновава с Теорема 101. Сложността по време е $\Theta(n \lg n)$ заради сортирането в началото, а сложността по памет е $\Theta(n)$. Сложността $\Theta(n \lg n)$ е оптимална, което

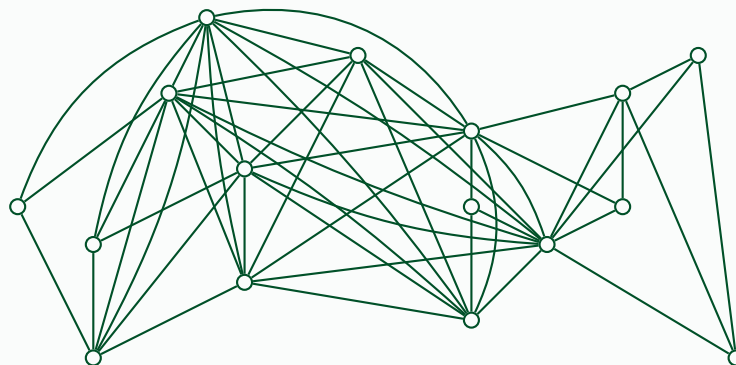
ще покажем в Подсекция 13.3.4.

Допълнение 63: МАКСИМАЛНА АНТИКЛИКА върху интервални графи

Задачата МАКСИМАЛНА АНТИКЛИКА е дефинирана на стр. 704. Тук ще покажем, че алгоритъмът INTERVAL SCHEDULING в някакъв смисъл решава МАКСИМАЛНА АНТИКЛИКА. Нека е дадено множество интервали $A = \{a_1, \dots, a_n\}$, точно както в INTERVAL SCHEDULING. На всеки интервал a_i съпоставяме връх v_i , и слагаме ребро (v_i, v_j) тстк $i \neq j$ и $a_i \cap a_j \neq \emptyset$. По този начин изграждаме граф, който съответства на A . Като пример да вземем множеството от интервалите на стр. 645 и да си представим въпросния граф, нарисуван върху рисунката на интервалите.



Този граф съответства точно на застъпванията на интервалите. Нека го разгледаме самостоятелно.



Графите, които определят по такъв начин застъпванията в множество от интервали, се наричат интервални графи.

Определение 101: Интервален граф

Граф $G = (V, E)$ се нарича *интервален граф*, ако съществува множество от интервали $\mathcal{I} = \{i_1, \dots, i_n\}$, такова че съществува биекция $\phi : V \rightarrow \mathcal{I}$, такава че за всеки два различни $u, v \in V$, $(u, v) \in E$ тогава и само тогава, когато $\phi(u) \cap \phi(v) \neq \emptyset$.

На английски понятието е *interval graph*. Интервалните графи не са самоцелно дефинирани. Едно от първите им приложения е в моделирането на задачи от биологията през 1959 г. [60, стр. 171].

Много общо, грубо и непрецизно казано, свързаните интервални графи се състоят от клики, които са “залепени” в линейна наредба; съседни в тази наредба, максимални по включване, клики имат общи подклики. Кликите идват оттам, че, говорейки нагледно, ако вертикална права пресича k интервала, то всеки два от тях се пресичат взаимно и в графа те задават k -клика. В нашия пример има 8 интервала, които се пресичат взаимно, и наистина в графа има съответна 8-клика.

Казано по друг начин: граф е интервален тогава и само тогава, когато съществува линейна наредба (редица) на максималните по включване клики, такава че в нея всеки връх се появява в непрекъсната подредица от клики [60, Theorem 8.1, стр. 172].

Далече не всеки граф е интервален. Прост пример за граф, който не е интервален, е всеки граф-цикъл с дължина поне 4. Неговите максимални по включване клики са 2-клики и те не са подредени линейно, а кръгово. Всяко дърво с поне един връх от степен поне 3, от който “излизат” поне 3 пътя с дължина поне 2 всеки, също не е интервален. Известни са линейни алгоритми, които разпознават интервалните графи [25].

Да се разгледаме алгоритъма INTERVAL SCHEDULING в светлината на интервалните графи. Веднага се вижда, че, без да искаме, сме конструирали ефикасен алгоритъм за задачата МАКСИМАЛНА АНТИКЛИКА (дефиницията е на стр. 704); наистина, всяко безконфликтно множество от интервали директно отговаря на антиклика в съответния интервален граф. Добре известно е, че МАКСИМАЛНА АНТИКЛИКА е **NP**-трудна задача (Лекция 14). На пръв поглед това, че разполагаме с много ефикасен алгоритъм за задача, която е **NP**-трудна, е парадоксално. Но всъщност парадокс няма. МАКСИМАЛНА АНТИКЛИКА е **NP**-трудна върху графи **по принцип**. А ние имаме ефикасен алгоритъм за МАКСИМАЛНА АНТИКЛИКА върху **ограничен клас графи**. Структурата на интервалните графи е доста по-ограничена и в някакъв смисъл по-бедна от структура на общите графи. По-бедната структура на интервалните графи е това, което позволява за тях да има ефикасен алгоритъм. Както ще видим в Подсекция 12.8.1, има ефикасен алгоритъм за МАКСИМАЛНА АНТИКЛИКА и върху дърветата.

И още нещо. Ние построихме алгоритъм със сложност $\Theta(n \lg n)$ за МАКСИМАЛНА АНТИКЛИКА върху интервални графи, който ползва представяне на тези графи само чрез множествата от интервали. Очевидно е, че множеството от интервалите представя графа недвусмислено. В Лекция 8 ние видяхме основните, най-често ползвани представяния на графи: списъци на съседство и матрици на съседство. Представянето чрез интервали е друг вид представяне; той очевидно е приложим само за графи, които са интервални. Ние бихме могли да конструираме списъците на съседство от интервалите. Обаче това би искало, в най-лошия случай, време, което е квадратично в броя на интервалите. За да се убедите в това, разгледайте n интервала, всеки два от които се пресичат. Съответният интервален граф е K_n , който има $\Theta(n^2)$ ребра. Само конструирането му като списъци иска време $\Omega(n^2)$. Същото е в сила и ако искаме да конструираме матрицата на съседство.

И така, това, че имаме субквадратичен алгоритъм е поради факта, че не се опитваме да строим представяне чрез списъци или матрица, а работим директно върху интервалите. Виждаме пример за това, че графов алгоритъм с оптимална ефикасност върху ограничен клас графи е възможен само ако графът е представен по някакъв начин, специфичен за този клас; започнем ли да строим “канонично представяне”, сложността по време нараства.

12.6.1.2 Решение по схемата Дин. Пр. за тегловния вариант

Ще усложним задачата. Сега всеки интервал a_i е наредена тройка (s_i, f_i, w_i) , където s_i е началото, f_i е крайт, а w_i е теглото на a_i , като $w_i \in \mathbb{R}^+$. В частност е възможно $w_i = f_i - s_i$, но в общия случай w_i и $f_i - s_i$ са независими. Решението е подмножество от интервали, два по два съвместими, които имат максимално сумарно тегло. Следва формалното описание на задачата, което е очевидно обобщение на Задача 43.

Изч. Задача 44: INTERVAL SCHEDULINGQ, WEIGHTED

екземпляр: Множество от отворени интервали A , тегловна функция $w : A \rightarrow \mathbb{R}^+$.

решение: Безконфликтно $A' \subseteq A$, такова че $\sum_{a \in A'} w(a)$ е максимално.

Задача 44 не може да се реши ефикасно с леко обобщение на алчния алгоритъм от предишната подсекция. Ще видим решение, конструирано по схемата **Динамично Програмиране**, което има същата сложност по време $\Theta(n \lg n)$ като алчното решение, но е несравнимо по-изтънчено. Естествено, Задача 44 се превръща в Задача 43, ако дадем едно и също тегло на всеки интервал, така че решението по схемата **Динамично Програмиране** решава и Задача 43.

Нека множеството от интервалите е $A = \{a_1, \dots, a_n\}$, а сортираната им редица по f -стойности е $S = (a'_1, \dots, a'_n)$. Това означава, че

$$f'_1 \leq f'_2 \leq \dots \leq f'_n$$

За ефикасно решение с динамично програмиране ни е необходимо да изчисляваме бързо за всеки интервал a'_i в S , интервала с най-голям номер в S , който е вляво от a'_i и е съвместим с a'_i . Да наречем функцията, реализираща това изображение, ϕ , като $\phi : \{1, \dots, n\} \rightarrow \{0, \dots, n-1\}$. За всяко $i \in \{1, \dots, n\}$, нека $C(i)$ е множеството от интервалите в S , които са вляво от i и са съвместими с i . Формалната дефиниция на ϕ е

$$\forall i \in \{1, \dots, n\} : \phi(i) \stackrel{\text{def}}{=} \begin{cases} \max \{j \mid a'_j \in C(i)\}, & \text{ако } C(i) \neq \emptyset \\ 0, & \text{ако } C(i) = \emptyset \end{cases}$$

Рекурсивната декомпозиция е по наредбата в S . Оптималното решение за първите i елемента на S или ползва a'_i , или не го ползва. Ако не го ползва, то е същото като за първите $i-1$ елемента. Ако го ползва, стойността му е сумата от w'_i и оптималната стойност за първите $\phi(i)$ елемента.

Да допуснем, че имаме всички стойности на ϕ в масив $\phi[1 \dots n]$. Схемата за изчисление е:

$$\begin{aligned} \text{opt}[0] &= 0 \\ \text{opt}[i] &= \max(\text{opt}[i-1], w'_i + \text{opt}[\phi[i]]), \text{ ако } i > 0 \end{aligned}$$

Следният итеративен алгоритъм решава ефикасно задачата.

WEIGHTED INTERVAL SCHEDULING($A = \{a_1, \dots, a_n\}$, където $n \geq 1$, $a_i = (s_i, f_i)$, $s_i < f_i$ и a_i има тегло w_i , за $1 \leq i \leq n$)

- 1 сортирай A по f -стойностите, резултатът е $\langle a'_1, \dots, a'_n \rangle$
- 2 $\text{opt}[0] \leftarrow 0$
- 3 **for** $i \leftarrow 1$ **to** n

```

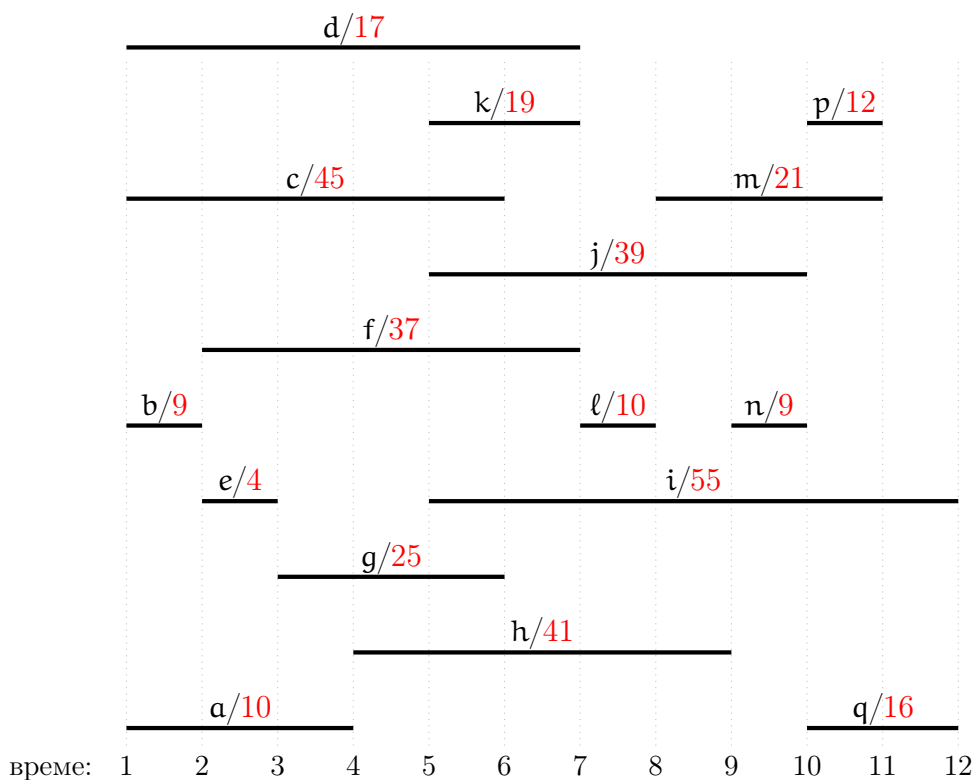
4   opt[i] ← max(opt[i - 1], w'_i + opt[φ[i]])
5   return opt[n]

```

Коректността е очевидна, ако сме убедени в коректността на рекурсивната декомпозиция и схемата за изчисление. Да разгледаме сложността. Ако изчисляваме стойностите на масива $\phi[1 \dots n]$ по наивния начин, като за всяко $\phi[i]$ се връщаме назад, може би чак до началото, за да видим кой е най-десният интервал в S , съвместим с i -ия, ще получим квадратичен алгоритъм само за пресмянето на $\phi[1 \dots n]$ и оттам, квадратична сложност на целия алгоритъм. За щастие, стойностите на масива $\phi[1 \dots n]$ може да се пресметнат във време $O(n \lg n)$ така: за всяко a'_i , с двоично търсене намираме максималното j , такова че $f'_j \leq s'_i$. Щом четем в момента a'_i , ние разполагаме с s'_i , а това f'_j ще намерим във време $O(\lg i)$, защото разполагаме със сортираната по f -стойности редица от интервалите.

Щом можем да построим $\phi[1 \dots n]$ във време $O(n \lg n)$, очевидно имаме алгоритъм със сложност по време $\Theta(n \lg n)$.

Да разгледаме пример. Интервалите са именувани $a, \dots, q$ (без име “o”), а теглата са написани до имената в червено.



Можете ли да откриете оптимално решение на око? Авторът на записките признава, че не може. Да подходим алгоритмично. Ето редицата от интервалите, сортирана по крайни времена, с индексите на интервалите в нея, написани отдолу:

b	e	a	g	c	f	k	d	ℓ	h	n	j	m	p	q	i
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Ето масивът $\phi[1 \dots 16]$:

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\phi(x)$	0	1	0	2	0	1	3	0	8	3	10	3	9	12	12	3

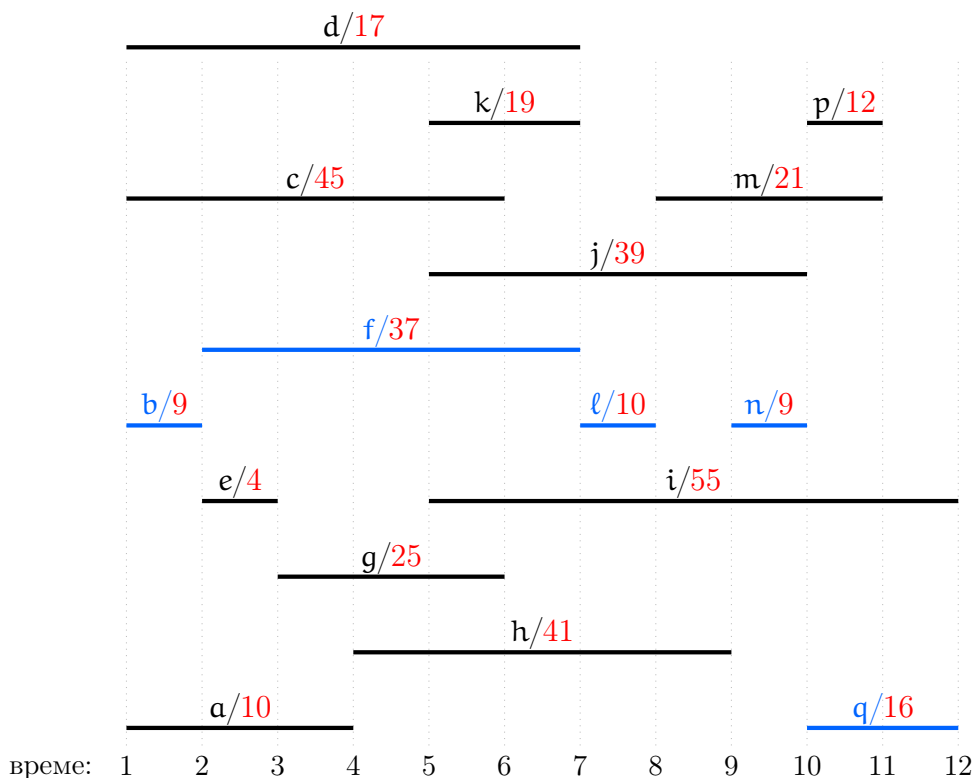
Ето и изчисленията за $\text{opt}[16]$:

$\text{opt}[1] = \max(0, 9 + \text{opt}[0]) = 9.$ ✓
 $\text{opt}[2] = \max(9, 4 + \text{opt}[1]) = 13.$ ✓
 $\text{opt}[3] = \max(13, 10 + \text{opt}[0]) = 13.$
 $\text{opt}[4] = \max(13, 25 + \text{opt}[2]) = 38.$ ✓
 $\text{opt}[5] = \max(38, 45 + \text{opt}[0]) = 45.$ ✓
 $\text{opt}[6] = \max(45, 37 + \text{opt}[1]) = 46.$ ✓
 $\text{opt}[7] = \max(46, 19 + \text{opt}[3]) = 46.$
 $\text{opt}[8] = \max(46, 17 + \text{opt}[0]) = 46.$
 $\text{opt}[9] = \max(46, 10 + \text{opt}[8]) = 56.$ ✓
 $\text{opt}[10] = \max(56, 41 + \text{opt}[3]) = 56.$
 $\text{opt}[11] = \max(56, 9 + \text{opt}[10]) = 65.$ ✓
 $\text{opt}[12] = \max(65, 39 + \text{opt}[3]) = 65.$
 $\text{opt}[13] = \max(65, 21 + \text{opt}[9]) = 77.$ ✓
 $\text{opt}[14] = \max(77, 12 + \text{opt}[12]) = 77.$
 $\text{opt}[15] = \max(77, 16 + \text{opt}[12]) = 81.$ ✓
 $\text{opt}[16] = \max(81, 55 + \text{opt}[3]) = 81.$

И така, оптималното решение има стойност 81.

Ако искаме да получим и самото оптимално решение, а не само стойността му, можем да постъпим така. В изчисленията за $\text{opt}[16]$, с червен тик е отбелязан всеки ред, в който стойността е получена от втория аргумент на $\max$, а именно $w'_i + \text{opt}[\phi[i]]$ на ред 4. Иначе казано, това са редовете, в които има увеличение спрямо предишния ред. Тръгвайки от последния ред, връщаме се назад до първия тик, който е на $\text{opt}[15]$. Виждаме, че q участва в оптималното решение и освен това $\text{opt}[15]$ е получен като $16 + \text{opt}[12] = 81$. Отиваме на реда на $\text{opt}[12]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на $\text{opt}[11]$. Тогава n участва в оптималното решение и освен това $\text{opt}[11]$ е получен като $9 + \text{opt}[10] = 65$. Отиваме на реда на $\text{opt}[10]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на $\text{opt}[9]$. Тогава ℓ участва в оптималното решение и освен това $\text{opt}[9]$ е получен като $10 + \text{opt}[8] = 56$. Отиваме на реда на $\text{opt}[8]$. Той няма тик, така че се връщаме до първия ред назад с тик, който е редът на $\text{opt}[6]$. Тогава f участва в оптималното решение и освен това $\text{opt}[6]$ е получен като $37 + \text{opt}[1] = 46$. Отиваме на реда на $\text{opt}[1]$, добавяме b към решението и сме готови. Решението се състои от q , n , ℓ , f и b , сумата от теглата на които наистина е $16 + 9 + 10 + 37 + 9 = 81$. Читателят лесно може да съобрази как да кодира тази идея в алгоритъма.

Ето и визуализация на оптималното решение. Интервалите от оптималното решение са в синьо.



12.7 Задачи върху редици и стрингове

Както обикновено, редиците са крайни (Конвенция 4). *Стринг* е редица, чиито елементи са от някакво крайно множество, наречено *азбука*. По правило, ако кажем само “редица”, имаме предвид елемент на $\mathbb{Z}^n$; тоест, целочислена редица.

12.7.1 Оптимално разбиване на редица

Тази задача е от [144, Глава 8.5, стр. 294–297]. Екип от няколко човека трябва да прегледа щателно всички книги, намиращи се на един рафт в една библиотека. Задачата е трудоемка: всяка книга трябва да бъде прочетена напълно, от началото до края. Защо – не се казва, може би се търси нещо, което е записано в една от книгите; в най-лошия случай, нещото ще бъде открито най-накрая. Екипът си поделва работата, като всяка книга се пада на точно един човек от екипа. С други думи, книгите са атомарни в смисъл, че не може повече от един човек да чете една и съща книга. Има смисъл да кажем, че множеството на книгите се разбива на толкова дялове, колкото са хората. Искане се подялбата на работата да е възможно най-справедлива. Това ще рече, че трябва да се минимизира максималният брой страници, които се падат за четене на човек от екипа. В термините на разбиването, ако книгите се идентифицират с броя на страниците си, дадено е мултимножество от цели положителни числа и се търси негово разбиване на толкова дяла, колкото са хората, такова че максималната сума от числата в дял се минимизира.

Досега описаната задача е **NP**-пълна; това е задачата MULTIPROCESSOR SCHEDULING от Подсекция 16.2.14. Но има още едно ограничение, което позволява да се конструира ефикасен алгоритъм: книгите, които кой да е човек от екипа взема за четене, са в **непрекъснатата редица** върху рафта. Това означава, че ако екипът е от k човека, решение на задачата може да се опише чрез слагане на $k - 1$ разделителя между книги на рафта.

Ето и формалното описание.

Задача 63: Оптимално разбиване на редица от числа

Дадена е непразна редица от цели положителни числа $\alpha = \langle a_1, a_2, \dots, a_n \rangle$. Дадено е и цяло число $k \in \{1, \dots, n\}$. Нека $i_0, i_1, i_2, \dots, i_{k-1}, i_k$ са числа, такива че

$$\begin{aligned} i_0 &= 1, \\ i_0 &< i_1, \\ i_1 &< i_2, \\ &\dots \\ i_{k-1} &< i_k, \\ i_k &= n + 1 \end{aligned}$$

От тях i_0 и i_k са фиксирани, а останалите наричаме *индексите*. Нека I е множеството от индексите. Цената на I е

$$c(I) = \max_{0 \leq j \leq k-1} \sum_{s=i_j}^{i_{j+1}-1} a_s$$

Предложете ефикасен алгоритъм, който намира такива индекси, че $c(I)$ е минимална. Достатъчно е Вашият алгоритъм да намира само цената, а не самите индекси. Аргументирайте коректността на Вашия алгоритъм и изследвайте сложността му по време.

Очевидно n е броят на книгите, $a_1, \dots, a_n$ са броевете на страниците, k е броят на хората в екипа, който ще чете, а индексите казват кой кои книги ще чете по отношение на наредбата върху рафта: първият човек ще чете книги $1, \dots, i_1 - 1$, вторият ще чете книги $i_1, \dots, i_2 - 1$, и така нататък, последният ще чете книги $i_{k-1}, \dots, i_k - 1 = n$.

Решение: На прост български, дадена е редица от n положителни числа, примерно

$$\alpha = \langle 100, 50, 40, 25, 200, 55, 95 \rangle$$

и цяло положително $k \leq n$. Искане се α да бъде разбита на k непрекъснати подредици, такива че максималната сума от елементи на подредица да е минимална. Забележете, че това има смисъл дори за $k = 1$. Съгласно формалното описание, тогава $I = \emptyset$, като цената не е нула, а е

$$c(\emptyset) = \sum_{s=i_0}^{i_{0+1}-1} a_s = \sum_{s=1}^{i_1-1} a_s = \sum_{s=1}^n a_s$$

тоест, сумата от елементите на α . Кое е смислено, ако разбиването има само един дял.

Ако $k = 2$, задачата е елементарна. За примера с $\alpha = \langle 100, 50, 40, 25, 200, 55, 95 \rangle$ на око се вижда, че оптималното разбиване е $\{\{100, 50, 40, 25\}, \{200, 55, 95\}\}$, като първият дял има сума 215, вторият дял има сума 350, така че цената на това решение е 350.

За произволно $n \geq 2$ и $k = 2$, решението е минимумът от следните $n - 1$ числа:

$$\begin{aligned} & \max \{a_1, a_2 + a_3 + a_4 + \dots + a_{n-1} + a_n\} \\ & \max \{a_1 + a_2, a_3 + a_4 + \dots + a_{n-1} + a_n\} \\ & \max \{a_1 + a_2 + a_3, a_4 + \dots + a_{n-1} + a_n\} \\ & \dots \\ & \max \{a_1 + a_2 + a_3 + \dots + a_{n-1}, a_n\} \end{aligned}$$

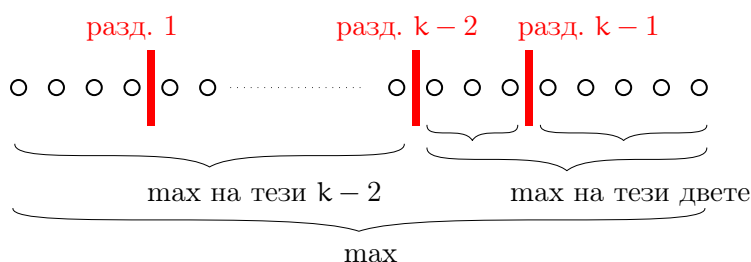
Всяко от тези числа се получава след нещо като слагане на един разделител в някоя “празнина” между две числа в редицата, като празнините са $n - 1$ на брой, и вземане на максимума от двете суми спрямо разделителя.

За големи стойности на n и k обаче решенията с груба сила стават неефективни. Всяко решение се определя еднозначно от разполагането на $k - 1$ разделители, отговарящи на индексите, между елементите на редицата, като обаче не може да има два разделителя един до друг. Накракто, има $n - 1$ позиции и от тях се избират $k - 1$, на които се разполагат разделители. Това прави $\binom{n-1}{k-1}$ начина. Както знаем, при горен индекс n , средният биномен коефициент има асимптотика, близка до $\Theta(2^n)$, така че решение с груба сила е безполезно на практика при $k \approx n/2$.

Разсъждаваме така. Нека $k \geq 2$. Последната редица започва веднага след най-десния разделител. Тогава характеристика на всяко решение, оптимално или не, е мястото на най-десния разделител. Да си представим **фиксиран разделител**, които са сложени последователно отляво надясно. Тогава най-десният разделител е сложен последен. Преди неговото слагане е имало $k - 1$ редици, разделени чрез $k - 2$ разделителя. Цената на решението, съдържащо най-десния разделител, е максимумът на следните:

- максималната сума на редица измежду първите $k - 2$ редици и
- максималната сума на някоя от последните две редици, получени от бившата последна редица чрез слагане на най-десния разделител някъде в нея.

Образно:

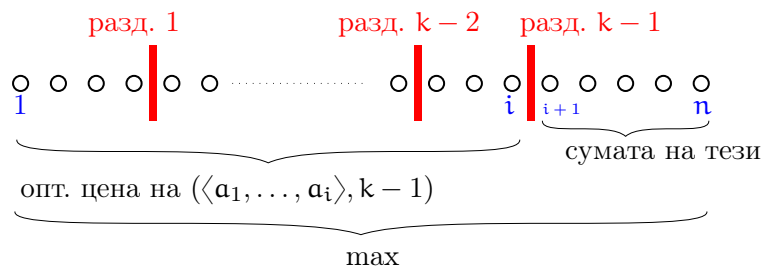


Да помислим за рекурсивна декомпозиция. Екземпляр на задачата е наредена двойка от числена редица и брой на разделителите. Подредицата вляво от най-десния разделител и числото $k - 1$ (броят на подредиците в нея) е екземпляр на задачата с по-къса числена редица и по-малко число от оригинала, така че може да направим рекурсивна декомпозиция. За фиксирана позиция на най-десния разделител—да кажем, че той е между i -ия и $(i + 1)$ -вия елемент—оптималната цена на решение е максимумът от

- оптималната цена на екземпляра вляво и

- сумата $a_{i+1} + \dots + a_n$.

Принципът на оптималността е в сила: няма смисъл да разглеждаме не-оптимално решение на екземпляра вляво. Образно:



Това i може да е най-малко $k-1$ и най-много $n-1$. Тогава оптималната цена на оригиналния екземпляр е

$$M(n, k) = \min \left\{ \max \left\{ M(i, k-1), \sum_{j=i+1}^n a_j \right\} : i \in \{k-1, \dots, n-1\} \right\}$$

Да видим началните условия. Те са

$$M(m, 1) = \sum_{s=1}^m a_s$$

$$M(m, m) = \max \{a_1, \dots, a_m\}$$

за $m \in \{1, \dots, n\}$.

Изчислението става с “трапецовидна” таблица. Нека масивът е $M[1 \dots n, 1 \dots k]$. Ползва се само частта под и включително главния диагонал, откъде и името “трапецовидна”.

В колона 1, отгоре надолу се слагат $a_1, a_1 + a_2, \dots, a_1 + \dots + a_n$. В главния диагонал, в посока от горе-ляво към долу-дясно, се слагат $a_1, \max \{a_1, a_2\}, \dots, \max \{a_1, a_2, \dots, a_n\}$. Това са началните условия.

Отговорът, който ни трябва, е $M[n, k]$. Той се пресмята като минимум на $n-1-(k-1)+1 = n-k+1$ числа, всяко от които е максимум от две числа, едното от които вече е записано в клетка от $M[n-1, k-1], M[n-2, k-1], \dots, M[k-1, k-1]$ (общо $n-k+1$ числа), а другото е една сума.

В общия случай, за $3 \leq p \leq n$ и $2 \leq q \leq p-1$:

$$M[p, q] \leftarrow \min_{q-1 \leq i \leq p-1} \max \{M[i, q-1], a_{i+1} + \dots + a_n\}$$

Редът на изчислението може да е подобен на този, който приехме за триъгълника на Паскал: по редове отгоре надолу, в рамките на един ред до главния диагонал – отляво надясно.

Ето илюстрация на изчислението на $M[n, k]$ в конкретен пример. Червената клетка $M[10, 6]$ се изчислява чрез сините клетки $M[9, 5], M[8, 5], M[7, 5], M[6, 5]$ и $M[5, 5]$.

n \ k	k					
	1	2	3	4	5	6
1	a_1					
2	$a_1 + a_2$	$\max_{a_1, a_2}$				
3	$\text{sum } a_1 \dots a_3$		$\max_{a_1 \dots a_3}$			
4	$\text{sum } a_1 \dots a_4$			$\max_{a_1 \dots a_4}$		
5	$\text{sum } a_1 \dots a_5$				$\max_{a_1 \dots a_5}$	
6	$\text{sum } a_1 \dots a_6$					$\max_{a_1 \dots a_6}$
7	$\text{sum } a_1 \dots a_7$					
8	$\text{sum } a_1 \dots a_8$					
9	$\text{sum } a_1 \dots a_9$					
10	$\text{sum } a_1 \dots a_{10}$					

На свой ред те ползват $M[8, 4]$, $M[7, 4]$, $M[6, 4]$, $M[5, 4]$ и $M[4, 4]$, и така нататък.

n \ k		1	2	3	4	5	6
1		a_1					
2		$a_1 + a_2$	$\max_{a_1, a_2}$				
3		$\text{sum } a_1 \dots a_3$		$\max_{a_1 \dots a_3}$			
4		$\text{sum } a_1 \dots a_4$			$\max_{a_1 \dots a_4}$		
5		$\text{sum } a_1 \dots a_5$				$\max_{a_1 \dots a_5}$	
6		$\text{sum } a_1 \dots a_6$					$\max_{a_1 \dots a_6}$
7		$\text{sum } a_1 \dots a_7$					
8		$\text{sum } a_1 \dots a_8$					
9		$\text{sum } a_1 \dots a_9$					
10		$\text{sum } a_1 \dots a_{10}$					

Клетките под петия диагонал реално не се ползват. Дали ще ги пресметнем или не, зависи от това, доколко искаме да оптимизираме изчислението. Асимптотиката в най-лошия случай не се променя (дори да запълним цялата правоъгълна таблица и над главния диагонал, асимптотиката на най-лошия случай не се променя).

Ето алгоритмът. Записът не е съвсем детайлен, но е ясно как да се имплементира.

РАЗБИВАНЕ НА ПОДРЕДИЦИ ($\alpha = \langle a_1, \dots, a_n \rangle, k \in \{1, \dots, n\}$)

```

1  създай  $M[1 \dots n, 1 \dots k]$ 
2  for  $i \leftarrow 1$  to  $n$ 
3       $M[i, 1] \leftarrow a_1 + \dots + a_i$ 
4  for  $i \leftarrow 2$  to  $n$ 
5       $M[i, i] \leftarrow \max(a_1, \dots, a_i)$ 
6  for  $p \leftarrow 3$  to  $n$ 
7      for  $q \leftarrow 2$  to  $p - 1$ 
8           $M[p, q] \leftarrow \min_{q-1 \leq i \leq p-1} (\max(M[i, q-1], a_{i+1} + \dots + a_n))$ 
9  return  $M[n, k]$ 
```

Коректността следва директно от коректността на рекурсивната декомпозиция. Сложността по време, ако се имплементира буквално, е $\Theta(kn \times n^2)$, което е $\Theta(kn^3)$. Причината е, че клетките са $\Theta(kn)$ и времето за попълване на клетка $M[p, q]$ е $\Theta((p - q) \times (n - p + q))$, което е $\Theta(n^2)$ в най-лошия случай.

Сложността може да бъде подобрена до $\Theta(kn^2)$, ако сумите $a_{i+1} + \dots + a_n$ не се изчисляват буквално по този начин, а чрез предварително пресмятане на стойностите на суфиксите и

съхраняването им в едномерен масив (друг алгоритъм по схемата **Динамично Програмиране**.)

12.7.2 Longest Increasing Subsequence

Определение 102: Поредица и растяща поредица

Нека е дадена редица $S = (a_1, a_2, \dots, a_n)$. *Поредица в S* е всяка редица $(a_{i_1}, a_{i_2}, \dots, a_{i_k})$, такава че $1 \leq i_1 < i_2 < \dots < i_k \leq n$. *Растяща поредица в S* е всяка поредица $(a_{i_1}, a_{i_2}, \dots, a_{i_k})$, такава че $a_{i_1} < a_{i_2} < \dots < a_{i_k}$. *Максимална растяща поредица в S* е растяща поредица в S с максимална дължина.

Изч. Задача 45: LONGEST INCREASING SEQUENCE

екземпляр: Редица $S = (a_1, a_2, \dots, a_n)$.

решение: Максимална растяща поредица в S .

В олекотения вариант, задачата е само да се намери дължината на максимална растяща поредица.

Забележете, че “поредица в S ” е редица от елементи, които **не са непременно съседни** в S . Поредица, чиито елементи са съседни—това е частен случай на поредица—наричаме *непрекъсната*, на английски *contiguous*; ако цялата редица е стринг, непрекъсната подредица в нея е всеки непразен подстринг. Разговорно казано, елементите на поредицата в общия случай са пръснати в S . Ето пример.

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Поредица в S е, примерно, $(9, -5, 3, 19, 10)$. Ето я като част от S :

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Непрекъсната поредица в S е, примерно, $(11, -2, 0, 3, -4)$. Ето я като част от S :

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Растяща поредица в S , максимална при това, е $(-5, -2, 0, 3, 6, 8, 10)$. Ето я като част от S :

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Максимална непрекъсната растяща в S е $(-2, 0, 3)$. Ето я като част от S :

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Намирането на максимална непрекъсната растяща поредица е тривиална задача. Има очевидно решение в линейно време с едно пребитане, но дори решението с груба сила не е невъзможно бавно, понеже непрекъснатите поредици са $\Theta(n^2)$ на брой. Намирането на максимална растяща поредица обаче е нетривиална задача. Поредиците без ограничението за непрекъснатост са много повече—а именно, $2^n - 1$, ако не броим празната поредица—и подходът с груба сила е безнадежден. В примера със синята поредица, когато започнем с -5 и отидем надясно, как да разберем, че трябва да прескочим 11 , но да не прескочим -2 ?

12.7.2.1 Дин. Прогр. със сложност $\Theta(n^2)$

Първо решаваме задачата в олекотения вариант, в който се иска само стойността на оптимално решение. Изглежда смислено да характеризираме решение-поредица чрез най-десния елемент. Това е a_k за някое $k \in \{1, \dots, n\}$.

Нека да мислим за рекурсивна декомпозиция за тази характеристика. Нека

$$S^1 = (a_1)$$

$$S^2 = (a_1, a_2)$$

$$S^3 = (a_1, a_2, a_3)$$

...

$$S^n = (a_1, a_2, \dots, a_n) = S$$

Нека ℓ_k е дължината на максимална растяща поредица в S^k , завършваща на a_k , за $1 \leq k \leq n$. Ако имаме тези ℓ_k , то ние сме готови! Тогава отговорът е $\max \{\ell_k \mid 1 \leq k \leq n\}$, тъй като всяка максимална растяща поредица завършва на някой елемент.

Как да получим ℓ_k от $\ell_{k-1}, \dots, \ell_1$? Само числата $\ell_{k-1}, \dots, \ell_1$ не са достатъчни, но двойките $(\ell_{k-1}, a_{k-1}), \dots, (\ell_1, a_1)$ са достатъчни. Интересуват ни само тези a_j , ако има такива, за които $a_j < a_k$.

- Ако има такива, то за всяко такова a_j е вярно, че a_k разширява растяща поредица, завършваща на a_j . За всяко такова a_j има смисъл да вземем максимална растяща поредица, завършваща на a_j , така че принципа на оптималността е в сила. Добавяйки a_k в десния край, ние разширяваме с точно един елемент. Ерго, ℓ_k е максимумът от ℓ -стойностите на въпросните a_j , плюс единица.
- Ако няма такива, то a_k не разширява поредица, а започва нова поредица, така че ℓ_k е единица.

Нека

$$\forall k \in \{1, \dots, n\} : R_k = \{j \in \{1, \dots, k-1\} \mid a_j < a_k\}$$

Тогава, за $1 \leq k \leq n$:

$$\ell_k = \begin{cases} \max \{\ell_j \mid j \in R_k\} + 1, & \text{ако } R_k \neq \emptyset \\ 1, & \text{ако } R_k = \emptyset \end{cases}$$

Превръщането на рекурсивната декомпозиция в алгоритъм е тривиално. Забележете, че $R_1 = \emptyset$, така че $\ell_1 = 1$ като начално условие.

Сега да помислим за по-тежкия вариант на задачата, в който се иска не просто дължината на максимална растяща поредица, а и някоя максимална растяща поредица. За всяка позиция $k \in \{1, \dots, n\}$ записваме позицията π_k на елемента—ако има такъв—който предшества a_k в максимална растяща поредица, завършваща на a_k . Ако няма такъв, нека $\pi_k = \text{nil}$. Ето алгоритъм, който реализира тази идея.

LONGEST INCREASING SUBSEQUENCE($(a_1, a_2, \dots, a_n)$)

- 1 $\ell[1] \leftarrow 1$
- 2 $\pi[1] \leftarrow \text{nil}$
- 3 **for** $k \leftarrow 2$ **to** n

```

4    $\ell[k] \leftarrow 1$ 
5    $\pi[k] \leftarrow \text{nil}$ 
6   for  $j \leftarrow 1$  to  $k - 1$ 
7       if  $\ell[k] < \ell[j] + 1$  and  $a[k] > a[j]$ 
8            $\ell[k] \leftarrow \ell[j] + 1$ 
9            $\pi[k] \leftarrow j$ 
10  return  $(\max_{1 \leq k \leq n} \{\ell[k]\}, \pi)$ 

```

Коректността е очевидна. Сложността е $\Theta(n^2)$. Както ще видим, има по-ефикасен алгоритъм за тази задача, но квадратичната сложност по време не е непоносима. Ето примерна работа на алгоритъма върху $S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[k]$	14	9	-5	11	-2	0	3	-4	12	19	6	8	22	10
$\ell[k]$	1	1	1	2	2	3	4	2	5	6	5	6	7	7
$\pi[k]$	nil	nil	nil	3	3	5	6	3	7	9	7	11	10	12

Едно оптимално решение е маркирано в синьо.

12.7.2.2 Дин. Прогр. със сложност $\Theta(n \lg n)$

Оригиналната статия с решение на задачата LONGEST INCREASING SUBSEQUENCE на Fredman от 1972 г. [54] предлага алгоритъм със сложност по време $\Theta(n \lg n)$, но анализът на алгоритъма е само скициран.

Има множество свободно достъпни решения със сложност $\Theta(n \lg n)$ в Интернет, но повечето имат много малко обосновка или обосновката не е съгласно схемата **Динамично Програмиране** или авторите директно казват, че този алгоритъм не е изграден по схемата **Динамично Програмиране**. Представеното тук решение се базира на [онлайн ресурс на професор Geppino Rucci](#) от Университета на Padova, Италия. Авторът на тези лекционни записки е на мнение, че има дребна формална грешка в рекурсивната декомпозиция на професор Руси, поради което изложението тук се различава от неговото. Също така, тук се ползва различна нотация.

В Подподсекция 12.7.2.1 характеризирахме оптималното решение като поредица, завършваща с някакъв елемент a_k . Сега ще характеризираме по друг начин. Нека S^k има смисълът от Подподсекция 12.7.2.1. За всяко $k \in \{1, \dots, n\}$, измежду всички максимални растящи поредици в S^k , разглеждаме такава, която завършва с минимален елемент. Подчертаваме разликата:

- преди искахме поредицата в S^k да завършва непременно на a_k ,
- а сега отпускате това ограничение и искаме поредицата да завършва на **някакъв** $a_j \in S^k$, който е минимален измежду всички завършеци на максимални растящи поредици в S^k .

Защо искаме завършващият елемент да е минимален? Следното тривиално наблюдение дава отговора.

Наблюдение 79

Измежду всички нарастващи поредици с дадена дължина, тази с минималния завършек има най-добър шанс да бъде увеличена чрез “залепване” на елемент вдясно.

За краткост ползваме “МРПМЗ” вместо “максимална растяща поредица с минимален завършек”. Като пример да разгледаме $S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$. Нека $k = 5$. Тогава $S^k = S^5 = (14, 9, -5, 11, -2)$. Максималните растящи поредици в S^5 са $(9, 11)$, $(-5, 11)$ и $(-5, -2)$. МРПМЗ в S^5 е $(-5, -2)$, понеже $-2 < 11$.

Ако знаем както МРПМЗ за S^k , така и a_{k+1} , дали знаем МРПМЗ за S^{k+1} ? Не непременно. Нека МРПМЗ за S^k е $(a_{i_1}, \dots, a_{i_j})$.

- Ако $a_{i_j} < a_{k+1}$, със сигурност $(a_{i_1}, \dots, a_{i_j}, a_{k+1})$ е МРПМЗ за S^{k+1} , понеже
 - ♦ има дължина $j + 1$, а в S^k няма растяща поредица с дължина $j + 1$, ерго, в S^{k+1} няма растяща поредица с дължина $> j + 1$;
 - ♦ завършва на a_{k+1} , който елемент очевидно е най-малкият възможен завършек на растяща поредица с дължина $j + 1$, защото е единственият завършек на растяща поредица с дължина $j + 1$ в S^{k+1} .
- Ако обаче $a_{i_j} > a_{k+1}$, то $(a_{i_1}, \dots, a_{i_j}, a_{k+1})$ не е растяща поредица. В този случай със сигурност в S^{k+1} няма растяща поредица с дължина $> j$, но в S^k може да има растяща поредица $(a_{t_1}, \dots, a_{t_{j-1}})$, такава че $a_{t_{j-1}} < a_{k+1}$. Забележете, че $(a_{t_1}, \dots, a_{t_{j-1}})$ не е максимална растяща поредица в S^k , имайки дължина само $j - 1$, но в S^{k+1} , $(a_{t_1}, \dots, a_{t_{j-1}}, a_{k+1})$ е растяща поредица с дължина j . Тогава и $(a_{i_1}, \dots, a_{i_j})$, и $(a_{t_1}, \dots, a_{t_{j-1}}, a_{k+1})$ са максимални растящи поредици в S^{k+1} , но $(a_{t_1}, \dots, a_{t_{j-1}}, a_{k+1})$ е МРПМЗ, а $(a_{i_1}, \dots, a_{i_j})$ не е МРПМЗ.

И така, убедихме се, че дори да знаем както МРПМЗ за S^k , така и a_{k+1} , това не е достатъчно, за да знаем МРПМЗ за S^{k+1} . Ако обаче за **всяка** дължина j знаем растяща поредица с минимален завършек в S^k с дължина j , то знаем МРПМЗ за S^{k+1} .

Да дефинираме, че за всеки $j, k \in \{1, \dots, n\}$, L_j^k е растяща поредица с дължина j и минимален завършек в S^k , а ℓ_j^k е нейният завършек; ако няма такава, то приемаме, че $L_j^k = ()$ (празната поредица), а $\ell_j^k = \infty$ (идентитетът на минимума).

Лема 46

В текущия контекст, нека за някое j , L_j^k и L_{j+1}^k са непразни. Нека $L_j^k = (a_{i_1}, \dots, a_{i_j})$ и $L_{j+1}^k = (a_{t_1}, \dots, a_{t_j}, a_{t_{j+1}})$, като $a_{i_j} = \ell_j^k$ и $a_{t_{j+1}} = \ell_{j+1}^k$. Тогава $\ell_j^k < \ell_{j+1}^k$.

Доказателство: Да допуснем, че $\ell_{j+1}^k < \ell_j^k$, тоест $a_{t_{j+1}} < a_{i_j}$. Но тогава $a_{t_j} < a_{i_j}$, понеже $a_{t_j} < a_{t_{j+1}}$. Следователно, поредицата $(a_{t_1}, \dots, a_{t_j})$ е растяща поредица в S^k с дължина j и завършек, по-малък от ℓ_j^k . Но това е невъзможно, понеже по конструкция L_j^k , която има завършек ℓ_j^k , е растяща поредица в S^k с дължина j и с **минимален** завършек. ⚡ □

И така, доказахме, че във всяко S^k , завършеците на непразните растящи поредици с минимални завършеци, растат с растенето на дължините на поредиците.

Да направим рекурсивна декомпозиция. Началните условия са следните:

$$L_1^k = (\min(S^k)), \quad \text{за } 1 \leq k \leq n \quad (12.43)$$

$$L_j^k = (), \quad \text{за } 1 \leq k < j \leq n \quad (12.44)$$

Коректността им е очевидна. Самата рекурсия изглежда така. При $2 \leq k \leq n$ и $2 \leq j \leq k$:

$$L_j^k = L_j^{k-1}, \quad \text{ако } \ell_j^{k-1} < a_k \quad (12.45)$$

$$L_j^k = L_j^{k-1}, \quad \text{ако } a_k < \ell_j^{k-1} \text{ и } a_k < \ell_{j-1}^{k-1} \quad (12.46)$$

$$L_j^k = \text{concat}(L_{j-1}^{k-1}, a_k), \quad \text{ако } \ell_{j-1}^{k-1} < a_k < \ell_j^{k-1} \quad (12.47)$$

Ето обосновката.

- В (12.45), ℓ_j^{k-1} , завършекът на L_j^{k-1} , е по-малък от a_k . Тогава “долепянето” на a_k като завършек не би дало растяща поредица с дължина j и по-малък завършек, така че L_j^{k-1} е растяща поредица с дължина j и минимален завършек както в S^{k-1} , така и в S^k . В (12.45), L_j^{k-1} е непразна, иначе ℓ_j^{k-1} би бил ∞ , при което $\ell_j^{k-1} < a_k$ би било невъзможно.
- Да допуснем, че $a_k < \ell_j^{k-1}$. Сега е възможно L_j^{k-1} да е празна, което влече $\ell_j^{k-1} = \infty$.
 - ♦ В (12.46), $a_k < \ell_{j-1}^{k-1}$.
 - * Да допуснем, че L_{j-1}^{k-1} е непразна, което влече $\ell_{j-1}^{k-1} < \infty$. В този случай, всяка растяща поредица с дължина $j-1$ в S^{k-1} има завършек-число, по-голямо от a_k . Ерго, a_k не може да се използва за “удължаване” до растяща поредица с дължина j . Ерго, $L_j^k = L_j^{k-1}$.
 - * Да допуснем, че L_{j-1}^{k-1} е празна, което влече $\ell_{j-1}^{k-1} = \infty$. В този подслучай наистина $a_k < \ell_{j-1}^{k-1}$, но конкатенацията на празната L_{j-1}^{k-1} с a_k дава растящата поредица (a_k) . От друга страна обаче, тъй като $j > 1$, поредицата (a_k) не може да има дължина j . Ерго, отново a_k не може да се използва за “удължаване” до растяща поредица с дължина j . Ерго, $L_j^k = L_j^{k-1}$.
 - ♦ Ако обаче $\ell_{j-1}^{k-1} < a_k < \ell_j^{k-1}$ (12.47), то a_k може да се използва за “удължаване” на растяща поредица с дължина $j-1$ до растяща поредица с дължина j , като при това новият завършек a_k е по-малък от предишния завършек ℓ_j^{k-1} на растяща поредица с дължина j . Забележете, че щом $\ell_{j-1}^{k-1} < a_k$, то L_{j-1}^{k-1} не е празна.

Нека пак разгледаме примера на стр. 660.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[k]	14	9	-5	11	-2	0	3	-4	12	19	6	8	22	10

Ако следваме буквално (12.43), (12.44), (12.45), (12.46) и (12.47), получаваме следните поредици (само непразните поредици са показани):

$$\begin{array}{lll}
L_1^1 = (14) & L_1^9 = (-5) & L_1^{12} = (-5) \\
L_1^2 = (9) & L_2^9 = (-5, -4) & L_2^{12} = (-5, -4) \\
L_1^3 = (-5) & L_3^9 = (-5, -2, 0) & L_4^{12} = (-5, -2, 0, 3) \\
L_1^4 = (-5) & L_4^9 = (-5, -2, 0, 3) & L_5^{12} = (-5, -2, 0, 3, 6) \\
L_2^4 = (-5, 11) & L_5^9 = (-5, -2, 0, 3, 12) & L_6^{12} = (-5, -2, 0, 3, 6, 8) \\
L_1^5 = (-5) & L_1^{10} = (-5) & L_1^{13} = (-5) \\
L_2^5 = (-5, -2) & L_2^{10} = (-5, -4) & L_2^{13} = (-5, -4) \\
L_1^6 = (-5) & L_3^{10} = (-5, -2, 0) & L_3^{13} = (-5, -2, 0) \\
L_2^6 = (-5, -2) & L_4^{10} = (-5, -2, 0, 3) & L_4^{13} = (-5, -2, 0, 3) \\
L_3^6 = (-5, -2, 0) & L_5^{10} = (-5, -2, 0, 3, 12) & L_5^{13} = (-5, -2, 0, 3, 6) \\
L_1^7 = (-5) & L_6^{10} = (-5, -2, 0, 3, 12, 19) & L_6^{13} = (-5, -2, 0, 3, 6, 8) \\
L_2^7 = (-5, -2) & L_1^{11} = (-5) & L_7^{13} = (-5, -2, 0, 3, 6, 8, 22) \\
L_3^7 = (-5, -2, 0) & L_2^{11} = (-5, -4) & L_1^{14} = (-5) \\
L_4^7 = (-5, -2, 0, 3) & L_3^{11} = (-5, -2, 0) & L_2^{14} = (-5, -4) \\
L_1^8 = (-5) & L_4^{11} = (-5, -2, 0, 3) & L_3^{14} = (-5, -2, 0) \\
L_2^8 = (-5, -4) & L_5^{11} = (-5, -2, 0, 3, 6) & L_4^{14} = (-5, -2, 0, 3) \\
L_3^8 = (-5, -2, 0) & L_6^{11} = (-5, -2, 0, 3, 12, 19) & L_5^{14} = (-5, -2, 0, 3, 6) \\
L_4^8 = (-5, -2, 0, 3) & & L_6^{14} = (-5, -2, 0, 3, 6, 8) \\
& & L_7^{14} = (-5, -2, 0, 3, 6, 8, 10)
\end{array}$$

Виждаме добра илюстрация на Лема 46: по отношение на фиксирано k , завършеците на поредиците L_j^k нарастват с нарастването на j . Примерно, при $k = 10$, завършеците са, в този ред:

$$-5, -4, 0, 3, 12, 19$$

Сега да си представим за всяко $k \in \{1, \dots, 14\}$, за всяко $j \in \{1, \dots, 14\}$, **всички** поредици L_j^k в нашия пример. Това са 196 поредици и няма да ги записваме явно, но да си представим

нещата за различните k :

$$\begin{aligned}
 k = 1: & \quad L_1^1 = (14), L_2^1 = (), L_3^1 = (), L_4^1 = (), \dots, L_{14}^1 = () \\
 k = 2: & \quad L_1^2 = (9), L_2^2 = (), L_3^2 = (), L_4^2 = (), \dots, L_{14}^2 = () \\
 k = 3: & \quad L_1^3 = (-5), L_2^3 = (), L_3^3 = (), L_4^3 = (), \dots, L_{14}^3 = () \\
 k = 4: & \quad L_1^4 = (-5), L_2^4 = (-5, 11), L_3^4 = (), L_4^4 = (), \dots, L_{14}^4 = () \\
 k = 5: & \quad L_1^5 = (-5), L_2^5 = (-5, -2), L_3^5 = (), L_4^5 = (), \dots, L_{14}^5 = () \\
 k = 6: & \quad L_1^6 = (-5), L_2^6 = (-5, -2), L_3^6 = (-5, -2, 0), L_4^6 = (), \dots, L_{14}^6 = () \\
 & \quad \dots \\
 k = 13: & \quad L_1^{13} = (-5), \dots, L_7^{13} = (-5, -2, 0, 3, 6, 8, 22), L_8^{13} = (), \dots, L_{14}^{13} = () \\
 k = 14: & \quad L_1^{14} = (-5), \dots, L_7^{14} = (-5, -2, 0, 3, 6, 8, 10), L_8^{14} = (), \dots, L_{14}^{14} = ()
 \end{aligned}$$

Забележе, че за всяко релевантно k , при прехода от $k - 1$ към k , точно за едно j поредицата става различна, а за останалите j поредиците са едни и същи. Примерно:

- при прехода от $k = 1$ към $k = 2$ е вярно, $L_1^1 = (14)$ и $L_1^2 = (9)$ се различават, а за $j \in \{2, \dots, 14\}$ имаме $L_j^1 = L_j^2 = ()$,
- при прехода от $k = 13$ към $k = 14$ е вярно, че L_7^{13} и L_7^{14} се различават, а за $j \in \{1, \dots, 6, 8, \dots, 14\}$ имаме $L_j^{13} = L_j^{14}$.

Това не е случайно. В [ресурса на професор Gerppino Rusci](#), на страница 4 се казва (написано в текущата терминология):

... тъй като завършеците са в нарастващ ред съгласно Лема 46, то когато превключваме от един префикс S^{k-1} към S^k , само за едно единствено j , подзадачата за това j в контекста на $k - 1$ може да има различно решение от подзадачата за това j в контекста на k .

При това терминът “може” е слаб: не просто може, а задължително има (точно едно) j , за което решението е различно.

Сега ще докажем това твърдение строго.

Теорема 102: Обосновка на бързия алг. за LONGEST INCREASING SEQUENCE

За всяко изпълнение на рекурсивния алгоритъм, определен от (12.43), (12.44), (12.45), (12.46) и (12.47), за всяко $k \in \{2, \dots, n\}$ съществува точно едно $j \in \{1, \dots, n\}$, такова че $L_j^{k-1} \neq L_j^k$.

Доказателство: Да разгледаме отново (12.45), (12.46) и (12.47). Тъй като редицата

$$(\ell_1^{k-1}, \ell_2^{k-1}, \dots, \ell_k^{k-1})$$

е сортирана съгласно Лема 46 и освен това числата в S са две по две различни, наистина има точно едно j , такова че a_k попада между ℓ_{j-1}^{k-1} и ℓ_j^{k-1} . Това е в сила

- и когато $\ell_j^{k-1} < \infty$, тоест, a_k е по-малък от поне един завършек
-

когато $\ell_j^{k-1} = \infty$, тоест, когато a_k е по-голямо число от всички крайни завършеци ℓ_t^{k-1} , където $t < j$. За това j , съгласно (12.47), завършекът на L_j^k става a_k , а a_k е различен (по-малък, ако трябва да сме точни) от завършека на L_j^{k-1} , който е именно ℓ_j^{k-1} . За всички $t \neq j$ е вярно, че $L_t^k = L_t^{k-1}$ съгласно (12.45) или (12.46). $\square$

Например да разгледаме списъка от непразните поредици на стр. 665.

Това, че съществува такова j , лесно следва от факта, че числата в S са две по две различни. А че това j не е повече от едно—което е същността на наблюдението—следва лесно от факта, че новото число S_{k+1} попада на едно единствено място в досегашната сортирана редица от завършеци. “Попада на място в редицата” означава, че в тази (сортирана) редица има една-единствена позиция, чийто елемент е най-малкият елемент, по-голям от новото S_{k+1} , и S_{k+1} го заменя; ако новото S_{k+1} е по-голямо от всеки в редицата, то елементът, който бива заменен, е най-лявата ∞ .

За нашия пример, при $k = 1$, редицата от завършеците е

$$14, \infty, \dots, \infty$$

Минавайки към $k = 2$, имаме $S_2 = 9$, което попада върху 14, и редицата от завършеците е

$$9, \infty, \dots, \infty$$

Минавайки към $k = 3$, имаме $S_3 = -5$, което попада върху 9, и редицата от завършеците е

$$-5, \infty, \dots, \infty$$

Минавайки към $k = 4$, имаме $S_4 = 11$, което попада върху най-лявата ∞ , и редицата от завършеците е

$$-5, 11, \infty, \dots, \infty$$

Минавайки към $k = 5$, имаме $S_5 = -2$, което попада върху 11, и редицата от завършеците е

$$-5, -2, \infty, \dots, \infty$$

Минавайки към $k = 6$, имаме $S_6 = 0$, което попада върху най-лявата ∞ , и редицата от завършеците е

$$-5, -2, 0, \infty, \dots, \infty$$

И така нататък.

не е довършено...

12.7.2.3 PATIENCESORT

Статията на Aldous и Diaconis [5] съдържа много подробна информация за теоретичната основа на тази сортиране, както и сложни и нетривиални резултати за асимптотиките на максималните растящи поредици.

12.7.2.4 Building Bridges

12.7.3 Longest Common Subsequence

Сега разглеждаме редици над някаква крайна азбука Σ . Както се разбрахме в началото на Подсекция 12.7, такива редици се наричат стрингове.

Изч. Задача 46: LONGEST COMMON SUBSEQUENCE

екземпляр: Стрингове $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$ над азбука Σ .

решение: Максимална поредица, обща за X и Y .

Олекотеният вариант. Ето идея за ефикасно решение по схемата **Динамично Програмиране**, което връща цената на оптимално решение. Разглеждаме всички префикси на $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$, включително и празните. Нека $M(i, j)$, за $0 \leq i \leq n$ и $0 \leq j \leq m$, означава максималната дължина на обща поредица в префиксите $x_1 \cdots x_i$ и $y_1 \cdots y_j$. Очевидно численото решение е $M(n, m)$.

Ако поне единият префикс е празен, дължината на максимална обща поредица е нула. Да допуснем, че и двата префикса са непразни. Нека

$$X^{i-1} = x_1 \cdots x_{i-1}, \quad X^i = x_1 \cdots x_i, \quad Y^{j-1} = y_1 \cdots y_{j-1}, \quad Y^j = y_1 \cdots y_j$$

за някакво $i \in \{1, \dots, n\}$ и някакво $j \in \{1, \dots, m\}$. Нека Z е максимална обща поредица за X^{i-1} и Y^{j-1} ; ерго, $M(i-1, j-1) = |Z|$. Разглеждаме следните две възможности, взаимно изключващи се и изчерпателни.

- ① Нека $x_i = y_j$ и тази буква е $a \in \Sigma$. Тогава $X^i = X^{i-1}a$, $Y^j = Y^{j-1}a$, и очевидно Za се явява максимална обща поредица за X^i и Y^j , като $|Za| = |Z| + 1$. Ерго, $M(i, j) = M(i-1, j-1) + 1$ в този случай.
- ② Нека $x_i \neq y_j$. Дали Z е максимална обща поредица за X^i и Y^j ? Не непременно. Ето пример:

$$X^i = c d e f$$

$$Y^j = g h i c$$

Очевидно $X^{i-1} = c d e$ и $Y^{j-1} = g h i$, така че $Z = \epsilon$. А максималната обща поредица за X^i и Y^j е c , като c се явява максимална обща поредица за X^{i-1} и Y^j .

Същественото е това, че не може максимална обща поредица за X^i и Y^j да има повече от една буква от максимална обща поредица за X^{i-1} и Y^j , понеже разликата между X^{i-1} и X^i е една буква. От съображения за симетрия, не може максимална обща поредица за X^i и Y^j да има повече от една буква от максимална обща поредица за X^i и Y^{j-1} . Следователно, ако $x_i \neq y_j$, $M(i, j) = \max(M(i-1, j), M(i, j-1))$.

Нещо повече. Не може максимална обща поредица за X^{i-1} и Y^j да има повече от една буква от максимална обща поредица за X^{i-1} и Y^{j-1} . От съображения за симетрия, не може максимална обща поредица за X^i и Y^{j-1} да има повече от една буква от максимална обща поредица за X^{i-1} и Y^{j-1} . Следователно, $\max(M(i-1, j), M(i, j-1))$ в случай ② не може да надхвърля $M(i-1, j-1) + 1$ от случай ①.

И така,

$$M(i, j) = \begin{cases} 0, & \text{ако } i = 0 \text{ или } j = 0, \\ M(i-1, j-1) + 1, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i = y_j \\ \max\{M(i, j-1), M(i-1, j)\}, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i \neq y_j \end{cases} \quad (12.48)$$

Таблицата $M[0..n, 0..m]$ е правоъгълна с размери $(n+1) \times (m+1)$. Съгласно началните условия, колона 0 и ред 0 са запълнени с нули. После попълваме отгоре надолу и отляво надясно, като $M[i, j]$ се изчислява чрез $M[i-1, j-1]$, $M[i-1, j]$ и $M[i, j-1]$. Търсеният числен отговор е $M[n, m]$.

Тежкият вариант. Ако искаме и някоя максимална поредица, достатъчно е за всяка клетка, която не е в нулевия ред и не е в нулевата колона, да запишем откъде сме “влезли” в нея. По-формално казано, всяка клетка $M[i, j]$ за $0 \leq i \leq n$ и $0 \leq j \leq m$ съдържа две неща

- $M[i, j].val$: число, което е дължината на максимална обща поредица на префиксите $x_1 \cdots x_i$ и $y_1 \cdots y_j$;
- $M[i, j].dir$: елемент на $\{up, left, diag, Nil\}$, който указва предната клетка – тази, от която сме “влезли” в $[i, j]$. Имената на елементите би трябвало да е с ясен смисъл: **diag** означава, че сме дошли от $[i-1, j-1]$, **left** означава, че сме дошли от $[i, j-1]$, а **up** означава, че сме дошли от $[i-1, j]$. Правилото, по което $M[i, j].dir$ получава стойността си, е следното.
 - ♦ ако $i = 0$ или $j = 0$, то $M[i, j].dir = Nil$,
 - ♦ ако $i \geq 1$ и $j \geq 1$ и $x_i = y_j$, то $M[i, j].dir = diag$,
 - ♦ ако $i \geq 1$ и $j \geq 1$ и $x_i \neq y_j$ и $M[i, j-1].val \geq M[i-1, j].val$, то $M[i, j].dir = left$,
 - ♦ ако $i \geq 1$ и $j \geq 1$ и $x_i \neq y_j$ и $M[i, j-1].val < M[i-1, j].val$, то $M[i, j].dir = up$.

ALG LCS(Σ : азбука, $n, m \in \mathbb{N}^+$, $X[1..n]$ и $Y[1..m]$: стрингове над Σ)

```

1  създай  $M[0..n, 0..m]$ 
2  for  $i \leftarrow 0$  to  $n$ 
3     $M[i, 0].val \leftarrow 0$ 
4     $M[i, 0].dir \leftarrow Nil$ 
5  for  $j \leftarrow 1$  to  $m$ 
6     $M[0, j].val \leftarrow 0$ 
7     $M[0, j].dir \leftarrow Nil$ 
8  for  $i \leftarrow 1$  to  $n$ 
9    for  $j \leftarrow 1$  to  $m$ 
10     if  $X[i] = Y[j]$ 
11        $M[i, j].val \leftarrow M[i-1, j-1].val + 1$ 
12        $M[i, j].dir \leftarrow diag$ 
13     else if  $M[i-1, j].val \geq M[i, j-1].val$ 
14        $M[i, j].val \leftarrow M[i-1, j].val$ 
15        $M[i, j].dir \leftarrow up$ 
16     else
17        $M[i, j].val \leftarrow M[i, j-1].val$ 
18        $M[i, j].dir \leftarrow left$ 
19  return  $M$ 
```

Ето малък пример. Нека азбуката е $\{A, B, C\}$ и

$X = CCBACCB$

$Y = ABCABCBACB$

М

Y

	ε	A	B	C	A	B	C	A	B	C	A	B	C
	0	1	2	3	4	5	6	7	8	9	10	11	12
ε 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0										
C 2	0												
B 3	0												
A 4	0												
C 5	0												
C 6	0												
B 7	0												

X

Да видим $M[1, 3]$ какво е. Разглеждаме $i = j = 3$. Тъй като $x_1 = C$ и $y_3 = C$, условието на ред 10 е истина и се изпълняват редове 11 и 12. Поради това $M[1, 3].val$ става $M[0, 2].val + 1$, което е 1. А $M[1, 3].prev$ става $diag$.

М

Y

	ε	A	B	C	A	B	C	A	B	C	A	B	C
	0	1	2	3	4	5	6	7	8	9	10	11	12
ε 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0	1									
C 2	0												
B 3	0												
A 4	0												
C 5	0												
C 6	0												
B 7	0												

X

По аналогичен начин запълваме цялата таблица.

Y

M

	ε	A	B	C	A	B	C	A	B	C	A	B	C
	0	1	2	3	4	5	6	7	8	9	10	11	12
ε 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0	1	1	1	1	1	1	1	1	1	1
C 2	0	0	0	1	1	1	2	2	2	2	2	2	2
B 3	0	0	1	1	1	2	2	2	3	3	3	3	3
A 4	0	1	1	1	2	2	2	3	3	3	4	4	4
C 5	0	1	1	2	2	2	3	3	3	4	4	4	5
C 6	0	1	1	2	2	2	3	3	3	4	4	4	5
B 7	0	1	2	2	2	3	3	3	4	4	4	5	5

X

Разполагайки с пълната таблица, лесно може да генерираме решение с обратно проследяване. Ще следваме указателите `prev`, изобразени с червени стрелки, тръгвайки от клетка $M[7, 12]$, когато не срещнем клетка, чийто указател е `Nil` (без изображение на рисунката). Ето клетките, през които се минава, показани в зелено или циан. Циановите клетки съответстват точно на указателите тип `diag` и те дават решението, понеже съответстват на буква, която е обща за X и Y . От друга страна, зелените клетки съответстват точно на указателите тип `left` и `up`; те не дават букви в решението, понеже всеки от тях съответства на буква, която се намира или само в X (`up`), или само в Y (`left`).

Y

M

	ε	A	B	C	A	B	C	A	B	C	A	B	C
	0	1	2	3	4	5	6	7	8	9	10	11	12
ε 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0	1	1	1	1	1	1	1	1	1	1
C 2	0	0	0	1	1	1	2	2	2	2	2	2	2
B 3	0	0	1	1	1	2	2	2	3	3	3	3	3
A 4	0	1	1	1	2	2	2	3	3	3	4	4	4
C 5	0	1	1	2	2	2	3	3	3	4	4	4	5
C 6	0	1	1	2	2	2	3	3	3	4	4	4	5
B 7	0	1	2	2	2	3	3	3	4	4	4	5	5

X

Оптималното решение, което получихме, е поредицата **CCBAC**. Ето я в контекста на двата

стринга:

X = C C B A C C B

Y = A B C A B C A B C A B C

12.7.4 Sequence Alignment

Става дума за добре известна задача от областта на изчислителната биология. Дадени са две ДНК или две РНК или два протеина. Искане се да се намери подобие между тези ДНК (или РНК или протеини). Както ДНК, така и РНК, така и протеините, грубо казано, имат едномерна линейна структура[†] и може да мислим за тях като за стрингове над някаква малка азбука:

- при ДНК азбуката се състои от четирите бази на ДНК аденин, цитозин, гуанин и тимин,
- при РНК азбуката е от четирите бази на РНК аденин, цитозин, гуанин и урацил,
- а при протеините азбуката е от около двадесет аминокиселини.

И така, двете дадени молекули са два стринга над някаква азбука, като дължината на стринговете е много голяма. Искане се да се установи степента на подобие между тези стрингове като те бъдат “наложени” един върху друг по начин, който максимизира общи подстрингове, които се оказват точно един срещу друг, но допуска появи на “празнини” в стринговете, които пък трябва да са минимални. Значението на задачата е огромно, понеже ДНК, РНК и протеините са градивни елементи на живота. По думите на Henikoff и Henikoff [71]:

Among the most useful computer-based tools in modern biology are those that involve sequence alignments of proteins, since these alignments often provide important insights into gene and protein function.

Далече не всеки стринг над съответната азбука представлява валидна ДНК, РНК или протеинова молекула. Но това вече е предмет на биологията. За целите на тази лекция всички стрингове са валидни.

Да разгледаме малък пример със стрингове, които представляват ДНК молекули. Азбуката е {A, C, G, T}. Нека двата стринга, в които търсим общи подстрингове, са X и Y:

```

X = C C A A C C G G A T G C T A
Y = A A C C G G A T G G C T A

```

Примерът е достатъчно малък и недвусмислен, така че общите подстрингове се идентифицират на око веднага, като няма съмнение кои са те:

```

X = C C A A C C G G A T G C T A
Y = A A C C G G A T G G C T A

```

Общите подстрингове могат да се видят лесно и по друг начин: не чрез оцветявания в различни цветове, а чрез подходящи подравнявания на подстрингове, при което може да се появят празнини:

```

C C A A C C G G A T G   C T A
      A A C C G G A T G G C T A

```

[†]Това не е точно така. Както ДНК, така и протеините всъщност имат тримерна структура.

Тези подравнявания всъщност са вмъквания на буква шпация '␣', поради което има смисъл азбуката да бъде $\{A, C, G, T, \text{␣}\}$. Ето как изглеждат X и Y с вмъкнати на подходящи места шпации, което ги превръща в съответно X' и Y' :

$$\begin{array}{rcccccccccccccccc} X' & = & C & C & A & A & C & C & G & G & A & T & G & \text{␣} & C & T & A \\ Y' & = & \text{␣} & \text{␣} & A & A & C & C & G & G & A & T & G & G & C & T & A \end{array}$$

Шпации може да се вмъкват в левия край или десния край или вътрешността на всеки от първоначално дадените стрингове. Абсолютно безсмислено обаче е на една и съща позиция в модифицираните стрингове да има шпация. Ето пример за такава **безсмислица**:

$$\begin{array}{rcccccccccccccccc} X'' & = & C & C & \text{␣} & A & A & C & C & G & G & A & \text{␣} & T & G & \text{␣} & C & T & A \\ Y'' & = & \text{␣} & \text{␣} & \text{␣} & A & A & C & C & G & G & A & \text{␣} & T & G & G & C & T & A \end{array}$$

Оттук и правилото, че в двата модифицирани стринга не може да има шпация на една и съща позиция.

Да формализираме изложението. Дадена е азбука Σ , съдържаща буквата '␣', и два непразни стринга $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$ над $\Sigma \setminus \{\text{␣}\}$. *Попълване* на всеки от X или Y е всеки стринг, който се получава от X или Y след поставяне на нула или повече шпации на колкото искаме места, било в началото, в края или във вътрешността. Например, ако $X = \text{ACCT}$, следните стрингове са попълвания на X : A␣C␣C␣T , ␣␣AC␣␣␣CT␣ и ␣ACCT .

Подравняване на X и Y е всяко попълване на X и Y с по такъв начин, че дължините на двата попълнени стринга са равни. Разглеждаме само такива подравнявания, в които на всяка позиция във всеки от двата попълнени стринга, поне единият символ **не е** ␣.

Да разгледаме произволно подравняване на X и Y , като X' е стрингът, получен от X , а Y' е стрингът, получен от Y . Нека $N = |X'| = |Y'|$. Тогава, за всяко $i \in \{1, \dots, N\}$, *теглото на позиция i* се дефинира по следния начин.

- Ако на i -та позиция в X' и Y' се намира една и съща буква—която не може да е шпация—съгласно правилата за подравняването—теглото на тази позиция е $+2$. Това е “награда за съвпадение на букви”.
- Ако на i -та позиция в X' и Y' се намират различни букви, като и двете са различни от шпация, теглото на тази позиция е -1 . Това е “наказание за несъвпадение на букви”.
- Ако на i -та позиция в X' и Y' се намират различни символи, точно единият от които е шпация, теглото на тази позиция е -2 . Това е “наказание за разминаване”.

Теглото на подравняването е сумата от теглата по позиции.

Да разгледаме пример. Нека $X = \text{AGTCGTC}$ и $Y = \text{GACTCAGG}$.

- Следното подравняване има тегло -6 . Под всяка позиция в попълнените стрингове е написано нейното тегло.

A	G	␣	T	␣	C	␣	G	T	C
␣	G	A	C	T	C	A	G	␣	G
-2	+2	-2	-1	-2	+2	-2	+2	-2	-1

Наистина, $-2 + 2 - 2 - 1 - 2 + 2 - 2 + 2 - 2 - 1 = -6$.

- Това подравняване на X и Y има тегло -3 :

A	G	□	□	T	C	□	G	T	C
□	G	A	C	T	C	A	G	□	G
-2	+2	-2	-2	+2	+2	-2	+2	-2	-1

Наистина, $-2 + 2 - 2 - 2 + 2 + 2 - 2 + 2 - 2 - 1 = -3$.

- А следното подравняване на X и Y има тегло 0. Нека читателят сам/сама се убеди, че това подравняване е с максимално тегло.

□	A	G	T	C	G	T	C
G	A	C	T	C	A	G	G
-2	+2	-1	+2	+2	-1	-1	-1

Наистина, $-2 + 2 - 1 + 2 + 2 - 1 - 1 - 1 = 0$.

Идеята е, че колкото по-голямо е теглото на подравняването, толкова по-добро съвпадение на общи подстрингове ни дава това подравняване. Очевидно, ако $X = Y$, то директно имаме подравняване с максималното възможно тегло $2|X| = 2|Y|$, в което попълненият X съвпада с X и попълненият Y съвпада с Y .

Изч. Задача 47: SEQUENCE ALIGNMENT

екземпляр: Стрингове $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$ над азбука Σ .

решение: Подравняване на X и Y с максимално тегло.

Олекотеният вариант. Ето идея за ефикасно решение по схемата **Динамично Програмиране**, което връща цената на оптимално решение. Разглеждаме всички префикси на $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$, включително и празните. Нека $M(i, j)$, за $0 \leq i \leq n$ и $0 \leq j \leq m$, означава максималното тегло на подравняване на префиксите $x_1 \cdots x_i$ и $y_1 \cdots y_j$. Очевидно численото решение е $M(n, m)$.

Ако и двата префикса са празни, теглото е нула, което дава началното условие $M(0, 0) = 0$. Но $M(i, 0)$, за $1 \leq i \leq n$, не са нули, и $M(0, j)$, за $1 \leq j \leq m$, не са нули. Да разсъждаваме за $M(i, 0)$ за $i \geq 1$. Тъй като подравняваме (под)стрингове с **еднаква** дължина, $M(i, 0)$ е теглото на подравняване на два стринга с дължина i , а **не** на стринг с дължина i и на празния стринг. По определение, $M(i, 0)$ е мярка за префикса $x_1 \cdots x_i$ на X и празния префикс на Y . Това означава, че погълваме празния префикс на Y с шпации, i на брой, така че да стане обект, който може да бъде съпоставян с $x_1 \cdots x_i$, позиция по позиция. Очевидно теглото е $-2i$. Тогава $M(i, 0) = -2i$ за $1 \leq i \leq n$. Аналогично, $M(0, j) = -2j$ за $1 \leq j \leq m$.

Ако нито един от префиксите не е празен, $M(i, j)$ е максимумът на следните три числа.

- Ако съпоставяме x_i с y_j , те може да съвпадат или да са различни. Ако съвпадат, теглото на тази позиция в подравнените префикси е $+2$. Ако не съвпадат, теглото на тази позиция в подравнените префикси е -1 . И в двата случая, теглото на въпросната позиция се сумира с $M(i-1, j-1)$.
- Ако съпоставяме x_i с шпация (в другия префикс), теглото на позицията е -2 и това се сумира с $M(i-1, j)$.
- Ако съпоставяме y_j с шпация (в другия префикс), теглото на позицията е -2 и това се сумира с $M(i, j-1)$.

И така:

$$M(i, j) = \begin{cases} 0, & \text{ако } i = 0 \text{ и } j = 0, \\ -2i, & \text{ако } i \geq 1 \text{ и } j = 0, \\ -2j, & \text{ако } i = 0 \text{ и } j \geq 1, \\ \max \{M(i-1, j-1) + \delta, M(i-1, j) - 2, M(i, j-1) - 2\}, & \text{ако } i \geq 1 \text{ и } j \geq 1. \end{cases} \quad (12.49)$$

където δ е $+2$, ако $x_i = y_j$, или -1 , в противен случай.

Таблицата е правоъгълна с размери $(n+1) \times (m+1)$. От началните условия имаме колона 0 запълнена, отгоре надолу, с $0, -2, -4, \dots, -2n$, и ред 0, запълнен, отляво надясно, с $0, -2, -4, \dots, -2m$. После попълваме отгоре надолу и отляво надясно, като $M[i, j]$ се изчислява чрез $M[i-1, j-1]$, $M[i-1, j]$ и $M[i, j-1]$. Търсеният числен отговор е $M[n, m]$.

Тежкият вариант. Ако искаме и някое оптимално подравняване, достатъчно е за всяка клетка без $M[0, 0]$ да запишем откъде сме “влезли” в нея. По-формално казано, всяка клетка $M[i, j]$ съдържа две неща

- $M[i, j].val$: число, което е максималното тегло на подравняване на префиксите $x_1 \dots x_i$ и $y_1 \dots y_j$;
- $M[i, j].dir$: елемент на $\{up, left, diag, Nil\}$, който указва предната клетка – тази, от която сме “влезли” в $[i, j]$. Имената на елементите би трябвало да са с ясен смисъл: **diag** означава, че сме дошли от $[i-1, j-1]$, **left** означава, че сме дошли от $[i, j-1]$, а **up** означава, че сме дошли от $[i-1, j]$. Правилото, по което $M[i, j].dir$ получава стойността си, е следното.
 - ♦ ако $(i, j) = (0, 0)$, то $M[i, j].dir = Nil$,
 - ♦ ако $i = 0$ и $j \geq 1$, то $M[i, j].dir = left$,
 - ♦ ако $i \geq 1$ и $j = 0$, то $M[i, j].dir = up$,
 - ♦ ако $i \geq 1$ и $j \geq 1$, то
 - * ако $\max(M[i-1, j-1] + \delta, M[i-1, j] - 2, M[i, j-1] - 2) = M[i-1, j-1] + \delta$, то $M[i, j].dir = diag$, без оглед на това дали δ е $+2$ или -1 .
 - * ако $\max(M[i-1, j-1] + \delta, M[i-1, j] - 2, M[i, j-1] - 2) = M[i, j-1] - 2$, то $M[i, j].dir = left$,
 - * ако $\max(M[i-1, j-1] + \delta, M[i-1, j] - 2, M[i, j-1] - 2) = M[i-1, j] - 2$, то $M[i, j].dir = up$,

Ето псевдокод, генериращ таблицата.

ALG SEQUENCE ALIGNMENT(Σ : азбука, $n, m \in \mathbb{N}^+$, $X[1..n]$ и $Y[1..m]$: стрингове над Σ)

```

1  създай  $M[0..n, 0..m]$ 
2   $M[0, 0].val \leftarrow 0$ 
3   $M[0, 0].dir \leftarrow Nil$ 
4  for  $i \leftarrow 1$  to  $n$ 
5       $M[i, 0].val \leftarrow -2i$ 
6       $M[i, 0].dir \leftarrow up$ 
7  for  $j \leftarrow 1$  to  $m$ 
8       $M[0, j].val \leftarrow -2j$ 
9       $M[0, j].dir \leftarrow left$ 
10 for  $i \leftarrow 1$  to  $n$ 
11     for  $j \leftarrow 1$  to  $m$ 
12         if  $X[i] = Y[j]$ 
13              $\delta \leftarrow 2$ 
14         else
15              $\delta \leftarrow -1$ 
16          $themax \leftarrow \max(M[i-1, j-1].val + \delta, M[i-1, j].val - 2, M[i, j-1].val - 2)$ 
17         if  $themax = M[i-1, j-1].val + \delta$ 
18              $M[i, j].val \leftarrow M[i-1, j-1].val + \delta$ 
19              $M[i, j].dir \leftarrow diag$ 
20         else if  $themax = M[i-1, j].val - 2$ 
21              $M[i, j].val \leftarrow M[i-1, j].val - 2$ 
22              $M[i, j].dir \leftarrow up$ 
23         else
24              $M[i, j].val \leftarrow M[i, j-1].val - 2$ 
25              $M[i, j].dir \leftarrow left$ 
26 return  $M$ 

```

Теглото на оптималното подравняване е $M[n, m].val$. Ако искаме и някое оптимално подравняване, ето код, който го генерира.

GENERATE OPTIMUM SEQ ALIGNMENT(Σ : азбука, $n, m \in \mathbb{N}^+$, $X[1..n]$ и $Y[1..m]$: стрингове над Σ , двумерен масив $M[0..n, 0..m]$ от (val, dir))

```

1  създай празни стрингове buffX и buffY
2   $p \leftarrow n, q \leftarrow m$ 
3  while TRUE do {
4      if  $p = 0$  and  $q = 0$ 
5          break
6      switch
7          case  $M[p, q].dir = \text{diag}$ 
8              prepend(buffX,  $X[p]$ )
9              prepend(buffY,  $Y[q]$ )
10              $p--, q--$ 
11             break
12         case  $M[p, q].dir = \text{left}$ 
13             prepend(buffX,  $\sqcup$ )
14             prepend(buffY,  $Y[q]$ )
15              $q--$ 
16             break
17         case  $M[p, q].dir = \text{up}$ 
18             prepend(buffX,  $X[p]$ )
19             prepend(buffY,  $\sqcup$ )
20              $p--$ 
21             break
22     }
23 return buffX, buffY

```

Ето малък пример. Нека азбуката е $\{A, B\}$ и

$X = \text{BAVABVVBABA}$

$Y = \text{ABABVBAVAB}$

Таблицата е 11×10 , индексирана от 0 по редове и колони. Началните условия са:

- що се отнася до числата, от 0 до -18 със стъпка 2 в нулевия ред и от 0 до -20 със стъпка 2 в нулевата колона;
- що се отнася до посоките, във всеки елемент без нулевия в нулевия ред посоката е наляво и за всеки елемент без нулевия в нулевата колона посоката е нагоре. Посоките рисуваме със стрелки – така е много по-прегледно. Стрелката-зануляване на елемента $[0, 0]$ не е показана.

Y

M	ϵ	A	B	A	B	B	A	B	A	B
	0	1	2	3	4	5	6	7	8	9
ϵ 0	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
B 1	-2									
A 2	-4									
B 3	-6									
A 4	-8									
B 5	-10									
B 6	-12									
B 7	-14									
A 8	-16									
B 9	-18									
A 10	-20									

X

Да видим $M[1, 1]$ какво е. Тъй като $X[1] = B$ и $Y[1] = A$, то $\delta = -1$ и съгласно (12.49),

$$M[1, 1] = \max \{M[0, 0] - 1, M[1, 0] - 2, M[0, 1] - 2\}$$

Имайки предвид, че $M[0, 0] = 0$, $M[1, 0] = -2$ и $M[0, 1] = -2$, това става

$$M[1, 1] = \max \{0 - 1, -2 - 2, -2 - 2\} = \max \{-1, -4, -4\} = -1$$

Тогава $M[1, 1].val = -1$, а $M[1, 1].dir = \text{diag}$.

Y

M	ε	A	B	A	B	B	A	B	A	B
	0	1	2	3	4	5	6	7	8	9
ε 0	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
B 1	-2	-1								
A 2	-4									
B 3	-6									
A 4	-8									
B 5	-10									
B 6	-12									
B 7	-14									
A 8	-16									
B 9	-18									
A 10	-20									

X

Да видим $M[1, 2]$ какво е. Тъй като $X[1] = B$ и $Y[2] = B$, то $\delta = 2$ и съгласно (12.49),

$$M[1, 2] = \max \{M[0, 1] + 2, M[0, 2] - 2, M[1, 1] - 2\}$$

Имайки предвид, че $M[0, 1] = -2$, $M[0, 2] = -4$ и $M[1, 1] = -1$, това става

$$M[1, 2] = \max \{-2 + 2, -4 - 2, -1 - 2\} = \max \{0, -6, -3\} = 0$$

Тогава $M[1, 2].val = 0$, а $M[1, 1].dir = \text{diag}$.

Y

M

	ϵ	A	B	A	B	B	A	B	A	B
	0	1	2	3	4	5	6	7	8	9
ϵ 0	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
B 1	-2	-1	0							
A 2	-4									
B 3	-6									
A 4	-8									
B 5	-10									
B 6	-12									
B 7	-14									
A 8	-16									
B 9	-18									
A 10	-20									

X

Попълваме таблицата до края и получаваме:

M

Y

	ϵ 0	A 1	B 2	A 3	B 4	B 5	A 6	B 7	A 8	B 9
ϵ 0	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
B 1	-2	-1	0	-2	-4	-6	-8	-10	-12	-14
A 2	-4	0	-2	2	0	-2	-4	-6	-8	-10
B 3	-6	-2	2	0	4	2	0	-2	-4	-6
A 4	-8	-4	0	4	2	3	4	2	0	-2
B 5	-10	-6	-2	2	6	4	2	6	4	2
B 6	-12	-8	-4	0	4	8	6	4	5	6
B 7	-14	-10	-6	-2	2	6	7	8	6	7
A 8	-16	-12	-8	-4	0	4	8	6	10	8
B 9	-18	-14	-10	-6	-2	2	6	10	8	12
A 10	-20	-16	-12	-8	-4	0	4	8	12	10

X

Разполагайки с пълната таблица M, GENERATE OPTIMUM SEQ ALIGNMENT генерира оптимално подравняване с тегло 10. Този алгоритъм просто следва указателите, тръгвайки от $M[10, 9]$ и вървейки назад до $M[0, 0]$. Стринговете в подравняването са buffX, съответен на X и buffY, съответен на Y, които се запълват отдясно наляво съгласно следното правило:

- ако ходът е диагонален, в buffX и buffY се слагат съответно буква от X и буква от Y;
- ако ходът е нагоре, в buffX се слага буква от X, а в buffY се слага шпация;
- ако ходът е наляво, в buffY се слага буква от Y, а в buffX се слага шпация.

Ето клетките, през които се минава, показани в зелено.

Y

M

	ε	A	B	A	B	B	A	B	A	B
	0	1	2	3	4	5	6	7	8	9
ε	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
B	-2	-1	0	-2	-4	-6	-8	-10	-12	-14
A	-4	0	-2	2	0	-2	-4	-6	-8	-10
B	-6	-2	2	0	4	2	0	-2	-4	-6
A	-8	-4	0	4	2	3	4	2	0	-2
B	-10	-6	-2	2	6	4	2	6	4	2
B	-12	-8	-4	0	4	8	6	4	5	6
B	-14	-10	-6	-2	2	6	7	8	6	7
A	-16	-12	-8	-4	0	4	8	6	10	8
B	-18	-14	-10	-6	-2	2	6	10	8	12
A	-20	-16	-12	-8	-4	0	4	8	12	10

X

Ето и самото оптимално подравняване. Стринговете в него ще наречем X' и Y' . Те са с дължина 11, а броят зелените клетки на таблицата е 12. Причината е ясна: буквите в X' и Y' съответстват не на зелените клетки, а на **преходите** (на английски бихме казали *gaps*) между тях.

$$\begin{array}{rcl}
 X' & = & \text{B} \quad \text{A} \quad \text{B} \quad \text{A} \quad \text{B} \quad \text{B} \quad \square \quad \text{B} \quad \text{A} \quad \text{B} \quad \text{A} \\
 Y' & = & \square \quad \text{A} \quad \text{B} \quad \text{A} \quad \text{B} \quad \text{B} \quad \text{A} \quad \text{B} \quad \text{A} \quad \text{B} \quad \square \\
 & & \begin{array}{cccccccccccc}
 -2 & +2 & +2 & +2 & +2 & +2 & -2 & +2 & +2 & +2 & -2
 \end{array}
 \end{array}$$

Наистина, $-2 + 2 + 2 + 2 + 2 + 2 - 2 + 2 + 2 + 2 - 2 = 10$.

Допълнение 64: За алгоритъма на Needleman и Wunsch

Алгоритъмът за SEQUENCE ALIGNMENT, който разгледахме, се нарича “алгоритъмът на Needleman-Wunsch”. Доколкото е известно на автора, това е най-старият ефикасен алгоритъм за SEQUENCE ALIGNMENT. Авторите са Saul B. Needleman and Christian D. Wunsch от 1970 г. [120].

Стойностите $+2$ (съвпадение), -1 (несъвпадение) и -2 (разминаване), които въведохме горе, не са в оригиналния алгоритъм. Тези стойности са избрани субективно, защото звучат смислено от общи съображения. На практика обаче е възможно да се получат по-добри резултати с други стойности. По думите на Needleman и Wunsch [120, стр. 2]:

In the simplest method, MAT_{ij} is assigned the value, one, if A_j is the same kind of amino acid as B_i ; if they are different amino acids, MAT_{ij} is assigned the

value, zero. The sophistication of the comparison is increased if, instead of zero or one, each cell value is made a function of the composition of the proteins, the genetic code triplets representing the amino acids, the neighboring cells in the array, or any theory concerned with the significance of a pair of amino acids. A penalty factor, a number subtracted for every gap made, may be assessed as a barrier to allowing the gap. The penalty factor could be a function of the size and/or direction of the gap.

В [120] стринговете са протеини, не ДНК, и затова буквите са аминокиселини. Двата протеина са A и B, като A_j и B_i са съответно j -тата и i -тата аминокиселина. MAT е това, което ние наричаме “таблицата на динамичното”, а MAT_{ij} описва резултата от сравнението на префикс на A, съдържащ A_j , и префикс на B, съдържащ B_i .

12.7.5 Edit Distance

Задачата е да се изчисли разстоянието между два дадени стринга. Мярката за близост между тях е минималният брой операции, с които единият може да бъде трансформиран в другия. Най-често тези операции са три: замяна на една буква с друга, вмъкване на буква и изтриване на буква. Тези три операции ще наричаме *трансформиращите операции*.

Доколкото е известно на автора, първоначално трансформиращите операции са дефинирани в статия на съветския математик Левенштейн [106]. Левенштейн разглежда булеви стрингове, които се предават на голямо разстояние в среда, в която може да възникнат грешки в предаването: може да се обърнат битове, да изчезне бит или да се вмъкне бит. При изчисляването на разстоянието между стринговете тези три операции са с еднаква тежест.

Може да бъдат разгледани и други трансформиращи операции, примерно транспозиция, което е размяна на съседни букви, като операция със същата тежест като другите. Това има очевидно приложение при обработка на текст, въведен от човек с клавиатура, понеже този вид грешка на ръката е често срещана на практика, но в това изложение няма да разглеждаме транспозиции.

Задачата EDIT DISTANCE има големи практически приложения, изброени в [119]. Освен вече посоченото приложение в коригирането на грешки при предаване на информация, тя се появява като подзадача в обработката на сигнали, OCR, обработка на текст с възможни грешки при набирането, intrusion detection, компресиране на изображения и други. Не на последно място в списъка с приложенията на EDIT DISTANCE е изчислителната биология, в която се иска да се изчисли колко са близки две ДНК или РНК или протеинови молекули.

Ето два прости примера, но не с булеви стрингове, а с думи от българския език.

1. Да кажем, “сланина” в “планина”. Очевидно разстоянието е единица, защото не е нула (не са една и съща дума), а втората се получава от първата чрез превръщането на ‘с’ в ‘п’ (една трансформираща операция).
2. Да кажем, “приспособление” в “разпространение”. Ето една възможна редица от осем трансформиращи операции:

приспособление	// изтриване на ‘п’
ри с пособление	// замяна на ‘и’ с ‘а’
ра с пособление	// замяна на ‘с’ със ‘з’
разп р особление	// вмъкване на ‘р’

разпрос~~У~~обление // вмъкване на ‘т’
 разпрост~~о~~обление // замяна на ‘о’ с ‘р’
 разпростр~~б~~ление // замяна на ‘б’ с ‘а’
 разпростра~~л~~ение // замяна на ‘л’ с ‘н’
 разпространение

Както предстои да видим, разстоянието между тези два стринга е осем. Ерго, тази редица е оптимална.

Това описание на трансформацията е доста многословно. Трансформацията на X в Y може да се опише компактно и недвусмислено чрез *prepus* (на английски е *transcript*, вижте [64, стр. 216]), който е стринг над азбуката $\{s, m, d, i\}$ и който описва трансформацията на X в Y , като s означава заместване (substitute), m означава съвпадение (match), d означава изтриване на буква (delete) и i означава вмъкване на буква (insert). Ако четем X отляво надясно и записваме трансформациите по този начин, ще получим въпросния препис. Като пример да видим пак трансформацията на “приспособление” в “разпространение”:

п	р	и	с	п	□	о	с	□	о	б	л	е	н	и	е
□	р	а	з	п	р	о	с	т	р	а	н	е	н	и	е
d	m	s	s	m	i	m	m	i	s	s	s	m	m	m	m

Наистина, преписът `dmssmimmisssmmm` описва едно решение–редица от прилагане на трансформиращи операции. Теглото на решението е броят на буквите, различни от ‘m’; в случая е осем. Това описание е недвусмислено само ако знаем и X , и Y ; ако знаем само X , нямаме представа коя буква се вмъква при i и коя е заместващата буква при s .

Допълнение 65: EDIT DISTANCE срещу SEQUENCE ALIGNMENT

Ние вече видяхме в Подсекция 12.7.4 задачата SEQUENCE ALIGNMENT, която търси максимално съвпадение между стрингове и поради това прилича на EDIT DISTANCE, която търси превръщане на стринг в друг стринг с минимален брой операции. Оказва се, че SEQUENCE ALIGNMENT може да се разглежда като частен случай на EDIT DISTANCE. Както посочва Gusfield [64, стр. 217], от математическа гледна точка

1. оптимално подравняване на два стринга X и Y , от една страна,
2. и оптимално трансформиране на X в Y чрез трансформиращите операции, от друга страна,

са еквивалентни начини да се даде количествена оценка за разликата между X и Y . “Еквивалентни” не означава, че теглото на оптимално подравняване на X и Y е непременно равно на разстоянието между X и Y ; идеята е, че тези два подхода са еднакво адекватни за описание на разликата между X и Y . От гледна точка на моделирането обаче, тези подходи са доста различни! Трансформирането в EDIT DISTANCE описва **процеса** на превръщането на X в Y , докато подравняването в SEQUENCE ALIGNMENT описва **резултата** от този процес.

В примера с превръщането на “приспособление” в “разпространение”, оптималното подравняване между тези стрингове е с тегло пет. Ето едно такова подравняване, което точно съответства на горепосоченото прилагане на осем трансформиращи операции:

п	р	и	с	п	□	о	с	□	о	б	л	е	н	и	е
□	р	а	з	п	р	о	с	т	р	а	н	е	н	и	е
-2	+2	-1	-1	+2	-2	+2	+2	-2	-1	-1	-1	+2	+2	+2	+2

Теглото на това подравняване е $-2+2-1-1+2-2+2+2-2-1-1-1+2+2+2+2 = 5$. Забележе, че теглото на оптималното подравняване може да се получи от преписа като сума, в която **d** участва като -2 , **i** участва като -2 , **s** участва като -1 и **m** участва като 2 . В примера с трансформацията на “приспособление” в “разпространение”, преписът **dmssmimmissmmmm** дава точно 5 :

$$-2+2-1-1+2-2+2+2-1-1-1-2+2+2+2+2 = 5.$$

За да осмислим още по-добре разликата между EDIT DISTANCE и SEQUENCE ALIGNMENT, да си представим сравняване на ДНК на човек и ДНК на шимпанзе, от една страна, и morphing (превръщане) на образ на човек в образ на шимпанзе, от друга страна.

Сравняване на ДНК. Общите части на човешкото ДНК и ДНК на шимпанзе са поне 95%, като някои източници казват, че са почти 99%. Колкото и да са, когато сравняваме ДНК-тата, подходиме като в SEQUENCE ALIGNMENT: намираме максимални общи части.

Тук за превръщане не може да става дума, защото никой не се опитва да прави човешко ДНК от ДНК на шимпанзе (или обратното). Хората и шимпанзетата имат общ прародител, от когото са дивергирали преди милиони години и това дивергиране е окончателно – те няма да се слоят отново никога.

Morphing на образи. Ако обаче превръщаме образ на човешко лице в образ на лице на шимпанзе и искаме това да изглежда най-естествено, подходиме като в EDIT DISTANCE: намираме минимален брой промени в матрицата от пикселите, които превръщат единия образ в другия.

При превръщане (morphing) на образи се интересуваме именно от междинните етапи, които трябва да са колкото може по-малко, за да изглежда превръщането колкото може по-естествено. Превръщането не се състои в намиране на общи области между двата образа.

Skiena [144, стр. 291–294] илюстрира практическата полза от EDIT DISTANCE с morphing на образ на омар в образ на човешко лице. Доколко това превръщането е убедително е друг въпрос. Нека читателят разгледа [144, стр. 293, Figure 8.7] и реши сам(а).

Ето и формалното описание на задачата.

Изч. Задача 48: EDIT DISTANCE

екземпляр: Стрингове $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$ над азбука Σ .

решение: Редица от минимален брой прилагания на трансформирани операции, превръщащи X в Y . Трансформирани операции са изтриване на буква, добавяне на буква и превръщане на буква в друга буква.

Забележете, че ако единствената разрешена операция е замяна на буква с друга буква, задачата става задачата на намиране на *разстоянието по Hamming* между X и Y ; това има смисъл само ако $m = n$. От друга страна, ако единствените разрешени операции са изтриване на буква и добавяне на буква, задачата става LONGEST COMMON SUBSEQUENCE от Подсекция 12.7.3.

Олекотеният вариант на задачата е да се изчисли само разстоянието между X и Y . Да разгледаме ефикасно решение за него. За всяко $i \in \{0, \dots, n\}$ и $j \in \{0, \dots, m\}$, нека $D(i, j)$ е разстоянието между $x_1 \dots x_i$ и $y_1 \dots y_j$. Ние търсим $D(n, m)$, но за да го изчислим, ще изчислим всички $D(i, j)$.

Очевидно $D(i, 0) = i$, за $0 \leq i \leq n$, защото става дума за превръщането на $x_1 \dots x_i$ в празния стринг по най-икономичен начин, който начин трябва да е i на брой изтривания на букви, а именно на $x_1, \dots, x_i$. Аналогично, $D(0, j) = j$, за $0 \leq j \leq m$.

Нека $i \in \{1, \dots, n\}$ и $j \in \{1, \dots, m\}$. Да помислим как префиксът $x_1 \dots x_i$ може да се превърне в префикса $y_1 \dots y_j$ чрез манипулиране на едната или двете крайни букви x_i и y_j .

- Може x_i да бъде изтрита, което е една трансформираща операция, и оставащият стринг $x_1 \dots x_{i-1}$ да бъде превърнат в $y_1 \dots y_j$ по най-икономичен начин. Цената е оптималната цена на трансформацията на $x_1 \dots x_{i-1}$ в $y_1 \dots y_j$ плюс единица.
- Може y_j да бъде добавена към $y_1 \dots y_{j-1}$, което е една трансформираща операция, ако преди това $x_1 \dots x_i$ е бил превърнат в $y_1 \dots y_{j-1}$ по най-икономичен начин. Цената е оптималната цена на трансформацията на $x_1 \dots x_i$ в $y_1 \dots y_{j-1}$ плюс единица.
- Оставащата възможност е $x_1 \dots x_{i-1}$ да бъде превърнат в $y_1 \dots y_{j-1}$ по най-икономичен начин, след което:
 - ♦ ако $x_i = y_j$, няма какво повече да се прави – имаме трансформация на $x_1 \dots x_i$ в $y_1 \dots y_j$ на цената на трансформацията на $x_1 \dots x_{i-1}$ в $y_1 \dots y_{j-1}$;
 - ♦ ако $x_i \neq y_j$, превръщаме x_i в y_j и по този начин имаме трансформация на $x_1 \dots x_i$ в $y_1 \dots y_j$ на цената на трансформацията на $x_1 \dots x_{i-1}$ в $y_1 \dots y_{j-1}$ плюс единица.

И така:

$$D(i, j) = \begin{cases} i, & \text{ако } j = 0, \\ j, & \text{ако } i = 0 \\ \min \{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta\}, & \text{ако } i \geq 1 \text{ и } j \geq 1. \end{cases} \quad (12.50)$$

където δ е 0, ако $x_i = y_j$, или 1, в противен случай.

Таблицата е правоъгълна с размери $(n+1) \times (m+1)$. От началните условия имаме колона 0 запълнена, отгоре надолу, с 0, 1, 2, ..., n , и ред 0, запълнен, отляво надясно, с 0, 1, 2, ..., m . После попълваме отгоре надолу и отляво надясно, като $D[i, j]$ се изчислява чрез $D[i-1, j-1]$, $D[i-1, j]$ и $D[i, j-1]$. Търсеният числен отговор е $D[n, m]$.

Алгоритъм за олекотения вариант на задачата се конструира много лесно от тази рекурсивна декомпозиция. Да конструираме ефикасен алгоритъм за тежкия вариант на задачата. Всяка клетка $D[i, j]$ съдържа $D[i, j].val$ и $D[i, j].dir$, имащи следния смисъл. $D[i, j].val$ е минималният брой операции, превръщащи $x_1 \dots x_i$ в $y_1 \dots y_j$. $D[i, j].dir$, който е елемент на $\{\text{up}, \text{left}, \text{diag}, \text{Nil}\}$, показва от коя клетка сме “влезли” в $D[i, j]$ при изчисляването на оптималните стойности. Правилото, по което $D[i, j].dir$ получава стойността си, е следното.

- ако $(i, j) = (0, 0)$, то $D[i, j].dir = \text{Nil}$,
- ако $i = 0$ и $j \geq 1$, то $D[i, j].dir = \text{left}$,
- ако $i \geq 1$ и $j = 0$, то $D[i, j].dir = \text{up}$,
- ако $i \geq 1$ и $j \geq 1$, то

- ♦ ако $\min \{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta\} = D[i-1, j] + 1$, то $D[i, j].dir = \text{left}$,
- ♦ ако $\min \{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta\} = D[i, j-1] + 1$, то $D[i, j].dir = \text{up}$,
- ♦ ако $\min \{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta\} = D[i-1, j-1] + \delta$, то $D[i, j].dir = \text{diag}$, без оглед на това дали δ е 0 или 1.

Ето псевдокод, генериращ таблицата.

```

ALG EDIT_DISTANCE( $\Sigma$ : азбука,  $n, m \in \mathbb{N}^+$ ,  $X[1..n]$  и  $Y[1..m]$ : стрингове над  $\Sigma$ )
1  създай  $D[0..n, 0..m]$ 
2   $D[0, 0].val \leftarrow 0$ 
3   $D[0, 0].dir \leftarrow Nil$ 
4  for  $i \leftarrow 1$  to  $n$ 
5       $D[i, 0].val \leftarrow i$ 
6       $D[i, 0].dir \leftarrow up$ 
7  for  $j \leftarrow 1$  to  $m$ 
8       $D[0, j].val \leftarrow j$ 
9       $D[0, j].dir \leftarrow left$ 
10 for  $i \leftarrow 1$  to  $n$ 
11     for  $j \leftarrow 1$  to  $m$ 
12         if  $X[i] = Y[j]$ 
13              $\delta \leftarrow 0$ 
14         else
15              $\delta \leftarrow 1$ 
16          $themin \leftarrow \min(D[i-1, j-1].val + \delta, D[i-1, j].val + 1, D[i, j-1].val + 1)$ 
17         if  $themin = D[i-1, j-1].val + \delta$ 
18              $D[i, j].val \leftarrow D[i-1, j-1].val + \delta$ 
19              $D[i, j].dir \leftarrow diag$ 
20         else if  $themin = D[i-1, j].val + 1$ 
21              $D[i, j].val \leftarrow D[i-1, j].val + 1$ 
22              $D[i, j].dir \leftarrow up$ 
23         else
24              $D[i, j].val \leftarrow D[i, j-1].val + 1$ 
25              $D[i, j].dir \leftarrow left$ 
26 return  $D$ 

```

Теглото на оптимално решение е $D[n, m].val$. Ако искаме и някое оптимално решение, записано като препис, ето код, който го генерира.

GENERATE TRANSCRIPT(Σ : азбука, $n, m \in \mathbb{N}^+$, $X[1 \dots n]$ и $Y[1 \dots m]$: стрингове над Σ , двумерен масив $D[0 \dots n, 0 \dots m]$ от (val, dir))

```

1  t ← max(n, m)
2  създай празен стринг result
3  p ← n, q ← m
4  while TRUE do {
5      if p = 0 and q = 0
6          break
7      switch
8          case D[p, q].dir = diag
9              if X[p] = Y[q]
10                 prepend(result, m)
11             else
12                 prepend(result, s)
13             p--, q--
14             break
15         case D[p, q].dir = left
16             prepend(result, i)
17             q--
18             break
19         case D[p, q].dir = up
20             prepend(result, d)
21             p--
22             break
23     }
24  return result

```

Да видим как работят алгоритмите върху примера с трансформацията на “приспособление” в “разпространение”. Таблицата е 14×15 , индексирана от 0 по редове и колони. Началните условия са:

- що се отнася до числата, от 0 до 15 със стъпка 1 в нулевия ред и от 0 до 14 със стъпка 1 в нулевата колона;
- що се отнася до посоките, във всеки елемент без нулевия в нулевия ред посоката е наляво и за всеки елемент без нулевия в нулевата колона посоката е нагоре. Посоките рисуваме със стрелки – така е много по-прегледно. Стрелката-зануляване на елемента $[0, 0]$ не е показана.

	€	Р	А	З	П	Р	О	С	Т	Р	А	Н	Е	Н	И	Е
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
€ 0	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9	← 10	← 11	← 12	← 13	← 14	← 15
П 1	↑ 1															
Р 2	↑ 2															
И 3	↑ 3															
С 4	↑ 4															
П 5	↑ 5															
О 6	↑ 6															
С 7	↑ 7															
О 8	↑ 8															
Б 9	↑ 9															
Л 10	↑ 10															
Е 11	↑ 11															
Н 12	↑ 12															
И 13	↑ 13															
Е 14	↑ 14															

Крайната таблица е следната. Решението е акцентирано със зелен фон.

	ε	Р	А	З	П	Р	О	С	Т	Р	А	Н	Е	Н	И	Е
ε	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
П	1	1	2	3	3	4	5	6	7	8	9	10	11	12	13	14
Р	2	1	2	3	4	3	4	5	6	7	8	9	10	11	12	13
И	3	2	2	3	4	4	4	5	6	7	8	9	10	11	11	12
С	4	3	3	3	4	5	5	4	5	6	7	8	9	10	11	12
П	5	4	4	4	3	4	5	5	5	6	7	8	9	10	11	12
О	6	5	5	5	4	4	4	5	6	6	7	8	9	10	11	12
С	7	6	6	6	5	5	5	4	5	6	7	8	9	10	11	12
О	8	7	7	7	6	6	5	5	5	6	7	8	9	10	11	12
Б	9	8	8	8	7	7	6	6	6	6	7	8	9	10	11	12
Л	10	9	9	9	8	8	7	7	7	7	7	8	9	10	11	12
Е	11	10	10	10	9	9	8	8	8	8	8	8	8	9	10	12
Н	12	11	11	11	10	10	9	9	9	9	9	8	9	8	9	10
И	13	12	12	12	11	11	10	10	10	10	10	9	9	9	8	9
Е	14	13	13	13	12	12	11	11	11	11	11	10	9	10	9	8

Нека читателят се убеди сам(а), че това решение дава преписа **dmssmimmissssmmmm**.

12.7.6 Факторизация на стринг на минимален брой палиндромии

Нека Σ е азбука. За всеки стринг $x \in \Sigma^+$, нека $\text{rev}(x)$ означава стринга, който е обратната пермутация на x . Примерно, ако $\Sigma = \{a, b\}$ и $x = aabbb$, то $\text{rev}(x) = bbbba$. *Палиндром над Σ* е всеки $x \in \Sigma^+$, такъв че $x = \text{rev}(x)$. Неформално, палиндром е стринг, който се чете по същия начин отляво надясно и отдясно наляво. Примери за палиндромии над $\{a, b\}$ са $aabbaa$, b и bab . Примери за не-палиндромии са ab , aab и $bbbbba$. *Факторизация* на непразен стринг x е редица от непразни стрингове $\langle y_1, y_2, \dots, y_k \rangle$, такава че x се явява тяхната конкатенация в този ред:

$$x = y_1 y_2 \cdots y_k$$

По очевидни причини, $y_1, \dots, y_k$ се наричат *факторите*.

Примерно, ако $x = aabbb$, то една факторизация на x е $\langle aab, bb \rangle$, друга факторизация на x е $\langle a, a, b, b, b \rangle$, а трета факторизация на x е $\langle aabbb \rangle$. Очевидно всяка факторизация на

x се определя еднозначно от слагане на разделители между букви на x ; в примера току-що, тези разделители са запетайте. Тъй като празнините между буквите на x са $|x| - 1$, броят на факторизациите на x е $2^{|x|-1}$.

Тук се интересуваме от факторизация, чиито фактори са палиндромы. Всеки стринг има тривиална факторизация на еднобуквени фактори. Трудно и интересно е да се намери факторизация на минимален брой фактори. Очевидно, този минимален брой е 1 тстк самият стринг е палиндром.

Изч. Задача 49: PALINDROME PARTITIONING

екземпляр: Азбука Σ и $x \in \Sigma^+$, като $x = x_1 \cdots x_n$.

решение: Минималното число $k \in \{1, \dots, n\}$, такова че x има факторизация на k на брой фактори-палиндромы.

Терминът “PARTITIONING” не е особено удачен, защото той предполага, че факторизацията е множество от подстрингове, докато в действителност тя е редица от подстрингове, но изглежда, че това име на задачата се е наложило.

Задачата далечно прилича на MATRIX-CHAIN MULTIPLICATION от Подсекция 12.4.1. Нека $M(i, j)$ е минималният брой палиндромы-фактори на подстринга $x_i \cdots x_j$, за $1 \leq i \leq j \leq n$. Очевидно $M(i, i) = 1$, а отговорът е $M(1, n)$. Ето една рекурсивна декомпозиция:

$$M(i, i) = 1, \text{ ако } 1 \leq i \leq n$$

$$M(i, j) = 1, \text{ ако } 1 \leq i < j \leq n \text{ и } x_i \cdots x_j \text{ е палиндром}$$

$$M(i, j) = \min \{M(i, k) + M(k + 1, j)\} \mid 1 \leq k \leq j - 1\}, \text{ ако } 1 \leq i < j \leq n \text{ и } x_i \cdots x_j \text{ не е палиндром}$$

Коректността е очевидна: наистина, ако $x_i \cdots x_j$ не е палиндром, той винаги има факторизация на палиндромы, ако ще да са еднобуквени, а оптималната му факторизация на палиндромы може да се представи като $y'y''$, като y' и y'' са непразни негови фактори, а оптималната цена за x е просто сумата от техните оптимални цени.

Тази рекурсивна декомпозиция може лесно да се преведе в псевдокод на алгоритъм по схемата **Динамично Програмиране**, който ползва триъгълна таблица. Псевдокодът е почти същият като на ALG MATRIX-CHAIN MULTIPLICATION на стр. 598, с незначителните разлики, че

1. първо се проверява дали $x_i \cdots x_j$ е палиндром, което става тривиално във време $O(|x_i \cdots x_j|)$
2. нищо не се добавя към $M(i, k) + M(k + 1, j)$.

Алгоритъмът връща $M[1, n]$.

Сложността по време е $\Theta(n^3)$ със същите съображения като при матричното умножение. Проверката дали $x_i \cdots x_j$ е палиндром не добавя нищо към общата асимптотична сложност, понеже така или иначе намирането на минимума става във време $\Theta(|x_i \cdots x_j|)$. Сложността по памет е $\Theta(n^2)$ със същите съображения като при матричното умножение.

12.7.7 Разполагане на текст в параграф

Тази задача е от [34, Задача 15-4]. Става дума за опростен вариант на известна типографска задача. *Типографията*[†] е техниката за разполагане на писмен текст върху страници, хартиени или екранни, по начин, който прави четенето максимално лесно и приятно и който

[†]Някои твърдят, че типографията е изкуство.

е естетично издържан. Основна единица на писмения текст е *параграфът* (още се нарича *абзац*), състоящ се от няколко изречения, свързани с общ смисъл, различен от смисъла на текста преди и след него. По правило, всеки добре оформен текст е разположен в параграфи, които са подравнени вляво[†] и са с една и съща ширина[‡], което ги прави подравнени и вдясно. Обикновено текстът на един параграф не се побира на един ред, откъдето естествено възниква следната задача: при дадено съдържание на един параграф, дадена ширина на реда, даден шрифт и дадени правила за пренасяне на думи на нов ред, включително и дали и доколко е допустимо текст да излиза надясно от дясната граница[§], какви да са разстоянията между буквите в думите, между думите и между изреченията и къде да са прекъсванията в редовете в рамките на даден параграф, така че полученият графичен резултат да е максимално удобен и приятен за четене. Примерно, *б*иш думи или разляти думи или големи разстояния между думи са визуални дразнители, пречещи на четенето.

На практика, тази задача е доста трудна, като част от трудността идва от субективността на това, какво е красиво и приятно за четене. Тук разглеждаме силно опростен вариант на задачата. *Символ* е всяка буква, цифра или препинателен знак, а също и шпацията; тоест, всеки атомарен типографски обект, който може да се появи в текста. Шрифтът е *monospace*, което означава, че всички символи имат една и съща ширина. Monospace шрифтовете не са особено красиви, понеже символите си имат естествено различни ширини, но в епохата преди масовите лазерни принтери и персонални компютри, когато се пишеше масово с пишещи машини, те бяха много често срещани. Дадена е максимална ширина M на един ред от текста; тоест, никой ред от параграфа не може да има повече от M символа. Параграфът трябва да е подравнен вляво, но не и вдясно. Празни редове не може да има. Пренасяне на думи на нов ред не е разрешено, откъдето следва, че всяка дума има дължина, не по-голяма от M , иначе не би имало решение. Между думите има само една шпация. От изредените дотук изисквания следва, че един параграф може да изглежда ето така при $M = 50$ и следния текст “Dixit algorizmi: laudes deo rectori nostro atque defensori dicamus dignas, que et debitum ei reddant, et augendo multiplicent laudem,”[¶]:

```
Dixit algorizmi: laudes deo rectori nostro atque
defensori
dicamus dignas, que et debitum ei
reddant, et augendo multiplicent laudem,
```

За удобство можем да мислим, че всеки ред има дължина точно M символа, попълвайки с шпации вдясно. Ето как изглежда същият пример с видими шпации и попълване:

```
Dixit_algorizmi:_laudes_deo_rectori_nostro_atque_
defensori_
dicamus_dignas,_que_et_debitum_ei_
reddant,_et_augendo_multiplicent_laudem,
```

И накрая да видим каква цената на дадено решение. Тъй като е фиксирано, че между две думи има точно една шпация, единственото, което можем да варираме по отношение на даден екземпляр-текст, е на кои места да сложим новите редове. Определяйки къде да са минаванията на нов ред, ние определяме и краищата на редовете. Искаме в краищата на

[†]Може би с изключение на първото изречение, което има някакъв отстъп.

[‡]Примерно, в тези лекционни записки тази ширина е 17.5 см.

[§]В терминологията на L^AT_EX това е “\rightmargin”.

[¶]Това е началото на текста на латински от Фигура 1.1. Преводът, според google translate, е “Algorizmi said: let us give praise to God our ruler and defender worthy, that they may repay the debt to him, and multiply the praise by increasing it”.

редовете да няма дълги попълвания с шпации. И по-точно, за всеки ред **без последния** вземаме броя на попълващите шпации в края и повдигаме този брой на трета степен, после сумираме тези числа и това е цената на решението. Задачата е минимизационна.

За примера горе, цената е $2^3 + 33^3 + 17^3 = 40\,858$. Както ще видим, това е далече от оптимума.

Ето формално описание на задачата. За целите на задачата, за всяко $n \in \mathbb{N}^+$ и за всяка редица от цели положителни числа $\alpha = \langle \ell_1, \ell_2, \dots, \ell_n \rangle$ дефинираме $w(\alpha) = (\sum_{k=1}^n \ell_k) + n - 1$. Това е точно броят на символите в текст, който се състои от думи с дължини $\ell_1, \dots, \ell_n$, отчитайки и шпациите между думите.

Изч. Задача 50: PRETTY TYPESETTING

екземпляр: Редица α от цели положителни числа и цяло положително число M .

решение: Разбиване на α на непразни подредици $\alpha_1, \alpha_2, \dots, \alpha_m$, такива че $w(\alpha_k) \leq M$ за $1 \leq k \leq m$ и сумата $\sum_{k=1}^{m-1} (M - w(\alpha_k))^3$ е минимална.

“Разбиване на α на подредици $\alpha_1, \alpha_2, \dots, \alpha_m$ ” означава, че α е конкатенацията на тези редици в този ред.

Забележете, че $M - w(\alpha_k)$ е точно броят на попълващите шпации в k -ия ред, ако на него е разположен текст, който е записан с думи с дължини, съответни на α_k .

Следният алчен алгоритъм е първото, което би хрумнало на човек, който трябва да разположи текст с **monospace** шрифт в параграф с горна граница M върху броя символи в един ред. Самият текст е без значение, важна е само редицата от дължините на думите.

GREEDY TYPESETTING(α : редица от цели положителни числа; M : цяло положително число)

- 1 (* α има смисъл на редицата от дължините на думите на някаков текст *)
- 2 (* M има смисъл на максимален брой символи на ред *)
- 3 (* всяко число от α е не по-голямо от M *)
- 4 направи нов ред и това е последният ред
- 5 докато α е непразна, прави следното
- 6 нека β е максималният префикс на α , побиращ се на последния ред
- 7 сложи β на последния ред
- 8 премахни β от α
- 9 ако α стане празна, то край
- 10 в противен случай, направи нов ред и това е последният ред
- 11 върни полученото разполагане на текста

Този алгоритъм щеше да решава задачата, ако ценовата функция беше $\sum_{k=1}^{m-1} (M - w(\alpha_k))$. Тоест, ако бройките на попълващите шпации не се повдигаха на трета степен. Ето горе-долу прецизно доказателство на това твърдение.

Теорема 103: GREEDY TYPESETTING дава опт. решение, ако ценовата функция е линейна

За всеки екземпляр на PRETTY TYPESETTING, алгоритъмът GREEDY TYPESETTING дава оптимално решение, ако ценовата функция е $\sum_{k=1}^{m-1} (M - w(\alpha_k))$.

Доказателство: Да си представим друг алгоритъм за същата задача, наречен ОПТИМАЛНА ТИПОГРАФИЯ, такъв че съществува поне една редица α и съществува поне една горна

граница M , такива че изходът на $\text{OPTIMAL TYPESETTING}(\alpha, M)$, да го наречем σ , има по-малка цена от изхода на $\text{GREEDY TYPESETTING}(\alpha, M)$, който ще наречем τ . Да кажем, че $\sigma = \langle \sigma_1, \dots, \sigma_s \rangle$ и $\tau = \langle \tau_1, \dots, \tau_t \rangle$. По дефиниция, и σ , и τ са разбивания на α , тоест редици от редици.

Щом σ и τ имат различни цени, те се различават поне в един елемент. Нека j е най-малкият номер на елемент, който е различен за тях. Това означава, че $\sigma_1 = \tau_1, \sigma_2 = \tau_2, \dots, \sigma_{j-1} = \tau_{j-1}$, но $\sigma_j \neq \tau_j$. Първото ключово наблюдение е, че първият елемент-число на σ_j съвпада с първия елемент-число на τ_j . Щом $\langle \sigma_1, \dots, \sigma_{j-1} \rangle = \langle \tau_1, \dots, \tau_{j-1} \rangle$, това е неизбежно вярно. Оттук и второто ключово наблюдение: σ_j се явява същински префикс на τ_j , понеже τ_j е подредица на α , генерирана от алчна стратегия. Нека $\tau_j = \langle x_1, \dots, x_p \rangle$, където $\langle x_1, \dots, x_p \rangle$ е някаква непразна подредица на α . Но σ_j е същински префикс на τ_j , от което заключаваме, че $\sigma_j = \langle x_1, \dots, x_q \rangle$ за някое $q < p$. Следва, че σ_{j+1} съществува и тя започва с числото x_{q+1} , което е елемент на τ_j .

Модифицираме σ така: прехвърляме x_{q+1} от σ_{j+1} в края на σ_j . По отношение на текста в параграфа, това има смисъл на прехвърлянето на една дума (с дължина x_{q+1}) от началото на $(j+1)$ -ия ред в края на j -ия ред. Нека така модифицираната σ се казва σ' . Лесно се вижда, че σ' е валидно решение, понеже $\langle x_1, \dots, x_q, x_{q+1} \rangle$ е префикс, може би не същински, на τ_j , а $w(\tau_j) \leq M$. Да видим каква е цената на σ' по отношение на цената на σ .

- Ако x_{q+1} не е единственият елемент на σ_{j+1} , тоест $\sigma_{j+1} = \langle x_{q+1}, x_{q+2}, \dots, x_z \rangle$ за някакво z , то след прехвърлянето цената на σ' остава равна на цената на σ . За да се убедим в това, забелязваме, че единствената разлика е нарастването на цената на j -тата редица с $x_{q+1} + 1$ и намаляването на цената на $(j+1)$ -тата редица с $x_{q+1} + 1$.
- Ако x_{q+1} е единственият елемент на σ_{j+1} , тоест $\sigma_{j+1} = \langle x_{q+1} \rangle$, след прехвърлянето редицата σ_{j+1} става празна (което отговаря на факта, че ако на ред $j+1$ преди прехвърлянето е имало само една дума, след прехвърлянето той става празен). Правим още една модификация: изтриваме получената празна подредица от σ' (което отговаря на изтриване на получения празен ред от параграфа). Тогава цената на σ' е с $x_{q+1} + 1$ по-малка от цената на σ .

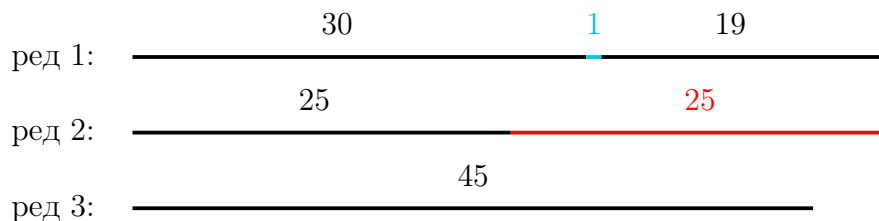
Ясно е, че прехвърлянето не може да увеличи цената.

Лесно се вижда, че ако продължим да модифицираме σ по аналогичен начин, прехвърляйки елементи в предишната подредица (което отговаря на прехвърляне на думи от началото на даден ред в края на горния ред, щом има място), в крайна сметка σ ще стане същата като τ , като при това цената само може да намалее, но не и да нарастне. Това противоречи на допускането, че цената на σ е била по-малка от цената на τ поначало. $\square$

Теорема 104: $\text{GREEDY TYPESETTING}$ не дава винаги опт. решение, ако ценовата ф-я е кубична

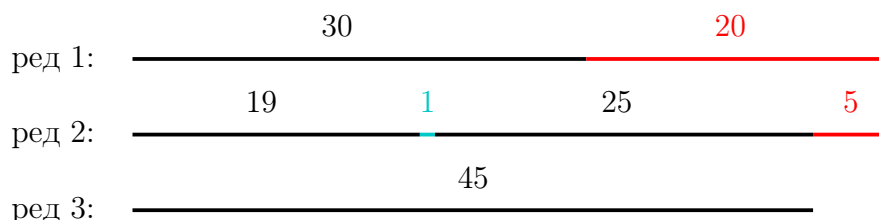
Съществува екземпляр на $\text{PRETTY TYPESETTING}$, за който алгоритъмът $\text{GREEDY TYPESETTING}$ не дава оптимално решение, ако ценовата функция е $\sum_{k=1}^{m-1} (M - w(\alpha_k))^3$.

Доказателство: Нека $\alpha = \langle 30, 19, 25, 45 \rangle$ и $M = 50$. Алчният алгоритъм слага думите с дължини 30 и 19 на първия ред, където не остава празно място, думата с дължина 25 отива на втория ред, където остава празно място от 25 шпации, а думата с дължина 45 отива на третия ред:



Последният ред е без значение за цената. Цената на алчното решение е $25^3 = 15\,625$.

Има обаче решение и решение с цена $20^3 + 5^3 = 8\,125$:



□

Повдигането на степен, по-голяма от едно, е “суперлинейно наказание” за дълги редици от попълващи шпации. Това, че е избрано повдигане на точно трета степен, е произволно[†]. Можеше да е степен две или степен четири. Можеше да степен две и половина или степен π . И така нататък. Последният ред не влиза в сметката, понеже не е визуално дразнещо, ако е къс, важното е останалите редове да са колкото се може по-равномерно подравнени.

Решение по схемата Динамично Програмиране. Решение с груба сила на тази задача е неефекасно. Броят на всевъзможните разполагания на текст от n думи на m реда с поне една дума на ред, включително и разполаганията, които нарушават ограничението за дължина на ред, е $\binom{n-1}{m}$. Това се получава директно от резултат от курса Дискретни Структури, казващ, че $\binom{p-1}{q}$ е броят на разполаганията на p неразличими топки в q различни кутии. Тук топките са думите и ги смятаме за неразличими, понеже редът им е фиксиран, а кутиите са редовете и те наистина имат идентичности. Да няма празни кутии е същото като да няма празни редове. Съгласно Теорема 35, $\binom{n-1}{m}$ има експоненциална асимптотика, ако $m \approx \frac{n}{2}$. Налага се да направим нещо по-умно.

Следното решение е фолклор. Нека M е цяло положително число и нека $\alpha = \langle \ell_1, \dots, \ell_n \rangle$ е редица от цели положителни числа, такива че $\ell_k \leq M$ за $1 \leq k \leq n$. Нека α отговаря на текст T , който е редица от думи и шпации между тях:

$$T = w_1 \sqcup w_2 \sqcup \dots \sqcup w_{n-1} \sqcup w_n$$

като $|w_k| = \ell_k$ за $1 \leq k \leq n$. Трябва да разположим T в параграф на редове, всеки от които е не по-дълъг от M символа, минимизирайки сумата от кубовете на броевете на шпациите в края на редовете без последния ред.

[†]В [34, Задача 15-4] “наказанието” е именно трета степен.

С какви конституенти можем да постигнем ефикасен алгоритъм? Да опитаем с тези:

$$\begin{aligned} w_1 \\ w_1 \sqcup w_2 \\ w_1 \sqcup w_2 \sqcup w_3 \\ \dots \\ w_1 \sqcup w_2 \sqcup \dots \sqcup w_{n-1} \end{aligned}$$

Ако това “проработи”, ще имаме решение с $\Theta(n)$ конституенти. Но как да получим оптимално разполагане на текста $w_1 \sqcup \dots \sqcup w_j$, ако разполагаме с оптимални разполагания на текстовете $\boxed{w_1}$, $\boxed{w_1 \sqcup w_2}$, $\dots$, $\boxed{w_1 \sqcup \dots \sqcup w_{j-1}}$? Не изглежда да има начин да направим това директно, без да имаме допълнителна информация.

Да допуснем, че отнякъде знаем, че последният ред на оптималното разполагане на $w_1 \sqcup \dots \sqcup w_j$ е $w_i \sqcup \dots \sqcup w_j$, което означава, че предпоследният ред завършва с w_{i-1} . Тогава предишните редове съдържат разполагане на $w_1 \sqcup \dots \sqcup w_{i-1}$. И то не какво да е, а оптимално разполагане на $w_1 \sqcup \dots \sqcup w_{i-1}$. Принципът на оптималността е в сила!

Нека $\text{opt}(j)$ е минималната цена за разполагане на $w_1 \sqcup \dots \sqcup w_j$. Ако някак знаем, че има оптимално разполагане, в което последният ред започва с w_i , то очевидно

$$\text{opt}(j) = \text{opt}(i-1) + Q(i, j)$$

където $Q(i, j)$ е “наказанието” за последния ред. Но ние не знаем въпросното i поначало. Щом не го знаем, можем да го сметнем, опитвайки всички $i \in \{1, \dots, j\}$:

$$\text{opt}(j) = \min \{ \text{opt}(i-1) + Q(i, j) \mid 1 \leq i \leq j \}$$

Да дефинираме, че $\text{opt}(0) = 0$. Бърза проверка: при това положение, $\text{opt}(1)$ е $Q(1, 1)$, което е коректно. И така, рекурсивната декомпозиция е

$$\text{opt}(j) = \begin{cases} 0, & \text{ако } j = 0 \\ \min \{ \text{opt}(i-1) + Q(i, j) \mid 1 \leq i \leq j \}, & \text{ако } j > 0 \end{cases} \quad (12.51)$$

Остава да видим как се смята $Q(i, j)$. Колко попълващи шпации има на ред, чийто текст е $w_i \sqcup \dots \sqcup w_j$? Да кажем, че $P(i, j)$ е това количество и че го дефинираме за всички i и j , такива че $1 \leq i \leq j \leq n$; последното може да звучи като лоша идея, защото може да има стойности на i и j , за които $w_i \sqcup \dots \sqcup w_j$ не се побира на един ред, но след малко ще се погрижим за това. Засега дефинираме

$$P(i, j) = M - \left(\left(\sum_{k=i}^j \ell_k \right) + j - i \right) = M - j + i - \sum_{k=i}^j \ell_k \quad (12.52)$$

като при тази дефиниция е възможно $P(i, j)$ да е отрицателно. Тогава наказанието $Q(i, j)$ се смята така:

$$Q(i, j) = \begin{cases} \infty, & \text{ако } P(i, j) < 0 \\ 0, & \text{ако } P(i, j) \geq 0 \text{ и } j = n \\ P(i, j)^3, & \text{ако } P(i, j) \geq 0 \text{ и } j < n \end{cases} \quad (12.53)$$

Забележете, че $P(i, j) < 0$ тук $w_i \sqcup \dots \sqcup w_j$ не се побира на един ред. Давайки наказание безкрайност в такъв случай, предотвратяваме разполагане, при което съдържанието на дори един ред има дължина, надхвърляща M , тъй като решение с цена безкрайност не може да е минимално; решение с крайна цена има задължително. Също така забележете, че $j = n$ е същото като $w_i \sqcup \dots \sqcup w_j$ да е на последния ред, а върху него наказание няма, което е същото като наказанието да е нула.

И така, (12.51) ни дава валидна рекурсивна декомпозиция, защото чрез него можем да пресмятаме цените на големите конституенти чрез цените на малките конституенти. Ако освен това пресметнем $Q(i, j)$ предварително и запазим резултатите в таблица[†] $Q[1 \dots n, 1 \dots n]$, можем да пресметнем ефикасно цената на оптимално решение чрез

$$\begin{aligned} \text{opt}[0] &= 0 \\ \text{opt}[j] &= \min \{ \text{opt}[i-1] + Q[i, j] \mid 1 \leq i \leq j \}, \text{ ако } j > 0 \end{aligned} \quad (12.54)$$

На свой ред, $Q[1 \dots n, 1 \dots n]$ може да се пресметне ефикасно, ако първо сме пресметнали $P[1 \dots n, 1 \dots n]$ със смисъла от (12.52).

Дотук имаме решение за олекотения вариант, понеже $\text{opt}[n]$ е желаната оптимална цена. Ще направим директно алгоритъм за тежкия вариант. За тази цел ще конструираме и масив $r[1 \dots n]$ със следния смисъл: за всяко $j \in \{1, \dots, n\}$, ако $\text{opt}[j]$ се пресмята чрез $\text{opt}[i-1]$ в (12.54), то $r[j] = i$. Тогава $r[j]$ има смисъл на номер на дума, с която започва реда, завършващ с w_j , при оптимално разполагане на $w_1 \sqcup \dots \sqcup w_j$. По този начин след изчисляването на $\text{opt}[n]$, чрез масива r можем да направим обратно проследяване, за да видим къде се минава на нови редове и така да получим оптимално разположение на текста в параграфа. Очевидно последният ред завършва с w_n и започва с $w_{r[n]}$, предпоследният ред завършва с $w_{r[n]-1}$ и започва с $w_{r[r[n]-1]}$, и така нататък.

Входният масив $L[1 \dots n]$ съдържа дължините на думите.

DYNAMIC TYPESETTING($L[1 \dots n]$: array of posint, M : posint)

```

1  създай празни масиви  $P[1 \dots n, 1 \dots n]$ ,  $Q[1 \dots n, 1 \dots n]$ ,  $\text{opt}[0 \dots n]$  и  $r[1 \dots n]$ 
2  for  $i \leftarrow 1$  to  $n$ 
3       $P[i, i] \leftarrow M - L[i]$ 
4      for  $j \leftarrow i + 1$  to  $n$ 
5           $P[i, j] \leftarrow P[i, j-1] - L[j] - 1$ 
6  for  $i \leftarrow 1$  to  $n$ 
7      for  $j \leftarrow i$  to  $n$ 
8          if  $P[i, j] < 0$ 
9               $Q[i, j] \leftarrow \infty$ 
10         else if  $j = n$ 
11              $Q[i, j] \leftarrow 0$ 
12         else
13              $Q[i, j] \leftarrow (P[i, j])^3$ 
14   $\text{opt}[0] \leftarrow 0$ 
15  for  $j \leftarrow 1$  to  $n$ 
16       $\text{opt}[j] \leftarrow \infty$ 
17      for  $i \leftarrow 1$  to  $j$ 
18          if  $\text{opt}[j] > \text{opt}[i-1] + Q[i, j]$ 
```

[†]Ако не се притесняваме за разхода на памет, таблицата е квадратна, в противен случай е горно-триъгълна.

```

19         opt[j] ← opt[i - 1] + Q[i, j]
20         r[j] ← i
21     return opt[n], r

```

Сложността както по време, така и по памет е очевидно $\Theta(n^2)$. Заслужава да се отбележи, че и двете сложности може да бъдат подобрени до $\Theta(nM)$, което на практика е сериозно подобрение – на практика, редовете рядко имат над 70 символа. Ключовото наблюдение е, че $\lceil \frac{M}{2} \rceil$ е точната горна граница за броя на думите, които може да се поберат на един ред[†]. Ред, върху който опитваме да разположим $w_{i_1} \cdots w_{j_1}$, би имал точно $j - i + 1$ думи. Ако $j - i + 1 > \lceil \frac{M}{2} \rceil$, то е невъзможно да сторим това, така че $Q[i, j]$ е ∞ автоматично. Следователно,

- налага се да изчисляваме $P[i, j]$ и $Q[i, j]$ само за стойности на i и j , такива че $j - i + 1 \leq \lceil \frac{M}{2} \rceil$, което касае паметта;
- във **for**-цикъла на редове 17–20, достатъчно е променливата i да взема стойности от $\max\{1, j - \lceil \frac{M}{2} \rceil + 1\}$ до j , което касае бързодействието.

Нещо повече, може да подобрим сложността по памет до $\Theta(n)$. Забелязваме, че $Q[i, j]$ се ползва само във **for**-цикъла на редове 17–20. $Q[i, j]$ може да се пресметне в константно време от $P[i, j]$ съгласно (12.53). Ако можем да изчисляваме $P[i, j]$ в константно време всеки път, когато изпълнението е на ред 17, то от него можем да изчисляваме и $Q[i, j]$, така че масивите $P[1 \dots n, 1 \dots n]$ и $Q[1 \dots n, 1 \dots n]$ стават излишни и алгоритъмът може да не ги създава поначало. А как да пресмятаме $P[i, j]$ в реално време на ред 17? Много просто, индексната променлива i се променя с намаляване, започвайки от j и завършвайки с $\max\{1, j - \lceil \frac{M}{2} \rceil + 1\}$, а $P[i, j]$ се инициализира с $M - L[j]$ и после на всяка итерация, на $P[i, j]$ се присвоява $P[i + 1, j] - (L[j] + 1)$. При това положение ни трябва само масивите $opt[0 \dots n]$ и $r[1 \dots n]$.

След като разполагаме с изчисления масив $r[1 \dots n]$, ето как правим принтирането на оптималното разположение на текста. Алгоритъмът LINE PRINT с вход (r, j) в някакъв смисъл принтира текста $w_{i_1} \cdots w_{j_1}$, като обаче не принтира самите думи, които не са дадени, а принтира наредените тройки (a, b, k) , указващи, че $w_a \cdots w_b$ се разполага на ред k . Това е за всички релевантни $w_a \cdots w_b$, и то в ред, който задава разбиване на $w_{i_1} \cdots w_{j_1}$.

```

LINE PRINT(r: array of posints, j: posint)
1  i ← r[j]
2  if i = 1
3      k ← 1
4  else k ← LINE PRINT(r, i - 1) + 1
5  print "(i, j, k)"
6  return k

```

Първоначалното викане е LINE PRINT(r, n). Алгоритъмът работи във време $O(n)$, понеже във всяко викане, вторият аргумент е поне с единица по-малък.

Ето илюстрация на работата на алгоритъма върху текста на стр. 695. Отново, текстът е (дължините на думите, включително и долепените препинателни знаци, са записани отдолу):

Dixit_{ℓ₁=5} algorizmi_{ℓ₂=10}: laudes_{ℓ₃=6} deo_{ℓ₄=3} rectori_{ℓ₅=7} nostro_{ℓ₆=6} atque_{ℓ₇=5} defensori_{ℓ₈=9} dicamus_{ℓ₉=7} dignas_{ℓ₁₀=7}, que_{ℓ₁₁=3}

[†]Ако всяка дума е от една буква!

et debitum ei reddant, et augendo multiplicent laudem,

при ограничение $M = 50$. Масивите P и Q са прекалено големи, за да бъдат написани тук. Масивът `opt` е този:

[91125, 39304, 19683, 12167, 3375, 512, 8, 20167, 11375, 5103, 3887, 2240, 576, 351, 15471, 13572, 7300, 2583, 351]

Излиза, че оптималната цена е 351. Масивът `r` е този:

$$\mathbf{r} = \begin{bmatrix} 1, & 1, & 1, & 1, & 1, & 1, & 1, & 5, & 6, & 6, & 6, & 7, & 7, & 8, & 10, & 10, & 11, & 13, & 15 \\ \textcolor{red}{1}, & \textcolor{red}{2}, & \textcolor{red}{3}, & \textcolor{red}{4}, & \textcolor{red}{5}, & \textcolor{red}{6}, & \textcolor{red}{7}, & \textcolor{red}{8}, & \textcolor{red}{9}, & \textcolor{red}{10}, & \textcolor{red}{11}, & \textcolor{red}{12}, & \textcolor{red}{13}, & \textcolor{red}{14}, & \textcolor{red}{15}, & \textcolor{red}{16}, & \textcolor{red}{17}, & \textcolor{red}{18}, & \textcolor{red}{19} \end{bmatrix}$$

Дори на око можем да установим какво е оптималното решение: $r[19]$ е 15, което означава, че думи 15 до 19 са на последния ред, а предпоследният ред започва с дума 14. Имаме $r[14] = 8$, което означава, че предпоследният ред започва с дума 8, което означава, че предпредпоследният ред започва с дума 7. Но $r[7] = 1$, което означава, че предпредпоследният ред е първият ред номер. Наистина, върху този масив r , алгоритъмът LINE PRINT($r, 19$) връща

$$\begin{pmatrix} 1 & 7 & 1 \\ 8 & 14 & 2 \\ 15 & 19 & 3 \end{pmatrix}$$

което означава (както вече отбелязахме), че думи 1 до 7 отиват на ред 1, думи 8 до 14 отиват на ред 2 и думи 15 до 19 отиват на ред 3:

Dixit_algorizmi: laudes_deo_rectori_nostro_atque
defensori_dicamus_dignas, que_et_debitum_ei
reddant, et_augendo_multiplicent_laudem,

Наистина, цената е $2^3 + 7^3 = 351$.

В току-що разгледания пример, оптималното решение съвпада с алчното, което се вижда на око веднага. Това е случайно – при $M = 50$ и този текст се получава така. Но при $M = 27$ и същия текст, оптималното решение, което DYNAMIC TYPESETTING и LINE PRINT конструират, е това:

Dixit algorizmi: laudes
deo rectori nostro atque
defensori dicamus dignas,
que et debitum ei reddant,
et augendo multiplicent
laudem,

Цената на това решение е $4^3 + 3^3 + 2^3 + 1^3 + 4^3 = 164$. При $M = 27$, алчният алгоритъм би разположил текста така:

Dixit algoritmi: laudes deo
reitori nostro atque
defensori dicamus dignas,
que et debitum ei reddant,
et augendo multiplicent
laudem,

Цената на алчното решение е $0^3 + 7^3 + 2^3 + 1^3 + 4^3 = 416$.

12.8 NP-трудни задачи върху дървета

Лекция 14, Лекция 15 и Лекция 16 разглеждат феномените на NP-пълнотата и NP-трудността в детайли. Тук само ще кажем, че оптимизационна задача е NP-трудна, ако във версия за разпознаване тя е NP-пълна. На свой ред, задача за разпознаване е NP-пълна, ако

- не е известен ефикасен алгоритъм, който намира решение “от нищото”, тоест, разполага единствено с входа (всички досега разгледани алгоритми са такива),
- но има ефикасен алгоритъм, такъв, че ако му се даде кандидат-решение, той може да провери в полиномиално време, че това наистина е решение. Откъде се взема кандидат-решението е въпрос, който не разглеждаме. **Ако** му се даде такова, **то** той може да провери бързо, че то наистина е решение.

Това **не** е дефиницията на “NP-пълна” (сравнете с Определение 139), но за целите на тази лекция върши работа.

Много от NP-трудните задачи са върху графи, било ориентирани, било неориентирани. По правило, NP-трудните задачи върху неориентирани графи стават ефикасно решими, ако графите са дървета. Авторът се сеща за само една естествена графова задача, а именно BANDWIDTH, която остава NP-трудна върху дървета [116].

В тази секция ще разгледаме ефикасни алгоритми по схемата **Динамично Програмиране** за две важни графови задачи, ако екземплярите бъдат ограничени до дървета.

Това, че изключително трудни задачи стават лесни, ако наложим сериозни ограничения върху екземплярите, не е странно. Нещо такова вече видяхме в Допълнение 63. Но кое ограничение е достатъчно сериозно, така че да превърне NP-трудна задача в ефикасно решима задача? На този важен въпрос няма (засега) смислен отговор. Не всяко ограничение върху екземплярите “облекчава” задачата; повечето NP-трудни задачи върху неориентирани графи остават NP-трудни, ако ограничим екземплярите до планарни графи.

12.8.1 Минимално върхово покриване на дърво

Върхово покриване, на английски *vertex cover*, на (обикновен) граф G е всяко подмножество от върхове, чието изтриване води до граф без ребра. Намиране на какво да е върхово покриване е тривиално: самото $V(G)$ е върхово покриване, а също така $V(G) \setminus \{u\}$, за който да е връх u , е върхово покриване. Трудно е да се намери върхово покриване с малка мощност. Задачата намира приложение, примерно, когато графът моделира някакви несъвместимости и искаме да намерим малко множество обекти (върхове), чието отстраняване елиминира несъвместимостите; с други думи, останалите обекти са съвместими по двойки.

Формалното определение на задачата в оптимизационен вариант е следното.

Изч. Задача 51: VERTEX COVER

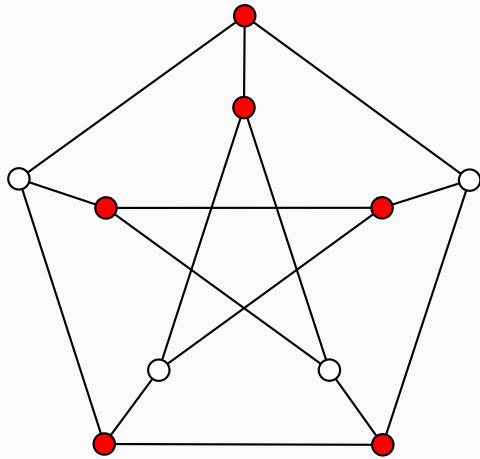
екземпляр: Неориентиран граф $G = (V, E)$.

решение: Подмножество $U \subset V$, такова че $\forall (u, v) \in E : (u \in U \text{ или } v \in U)$ и $|U|$ е минимално.

Мощността на минимално върхово покриване на G се нарича *числото на върховото покриване на G* и се бележи с “ $\tau(G)$ ”.

Пример за минимално върхово покриване е показан на Фигура 12.2. Графът на Petersen, да го наречем P , не може да бъде покрит с по-малко от 6 върха: само външният 5-цикъл иска поне 3 върха, а независимо от това, вътрешният 5-цикъл (вътрешните пет диагонала) иска също поне 3 върха, така че 6 е долна граница за мощността на върховото покриване и върховото покриване на Фигура 12.2 е оптимално. И така, $\tau(P) = 6$.

Фигура 12.2 : Минимално върхово покриване в графа на Petersen.



Задачата МАКСИМАЛНА АНТИКЛИКА, на английски INDEPENDENT SET, е много близка до VERTEX COVER. *Антиклика* в граф G е всяко подмножество на $V(G)$, между нито два върха на което няма ребро.

Изч. Задача 52: INDEPENDENT SET

екземпляр: Неориентиран граф $G = (V, E)$.

решение: Подмножество $U \subset V$, такова че U е антиклика и $|U|$ е максимално.

VERTEX COVER и INDEPENDENT SET са дуални в смисъл, че върху един и същи граф G , U е върхово покриване тстк $V(G) \setminus U$ е антиклика.

Теорема 105: VERTEX COVER и INDEPENDENT SET са дуални

Нека $G = (V, E)$ е граф. За всяко $U \subseteq V$ е вярно, че U е върхово покриване тстк $V \setminus U$ е антиклика.

Доказателство: Нека U е върхово покриване. Ще докажем, че $V \setminus U$ е антиклика. Да разгледаме произволни различни върхове $u, v \in V \setminus U$. Твърдим, че u и v не са съседи. Да допуснем противното. Тогава съществува ребро (u, v) в E . Но това ребро не е покрито в смисъл, че и двата му края са извън U , което влече, че U не е върхово покриване. Полученото противоречие показва, че u и v не са съседи. Щом за всеки два върха извън U е вярно, че не са съседи, то $V \setminus U$ е антиклика.

В обратната посока, нека U е антиклика. Ще докажем, че $V \setminus U$ е върхово покриване. Да разгледаме произволно ребро $(u, v) \in E$. Невъзможно е u и v да са от U , понеже U е антиклика.

Тогава поне единият от u и v е от $V \setminus U$. Покажахме, че за всяко ребро, поне единият му край е от $V \setminus U$. Тогава $V \setminus U$ е върхово покриване. $\square$

Теорема 105 се илюстрира добре от Фигура 12.2: четирите неоцветени върхове са антиклика, и по-голяма антиклика няма. Ако имаше, щеше да има по-малко върхово покриване.

VERTEX COVER е трудна изчислителна задача. Във варианта на задача за разпознаване, тя е една от класическите NP-пълни задачи (Лекция 16).

Допълнение 66: VERTEX COVER и EDGE COVER са различни задачи

Задачата VERTEX COVER (Задача 51) по-скоро говори за покриване на ребрата, а не на върховете! Наистина, въпросното подмножество U трябва да съдържа поне един връх от всяко **ребро**. Човек би казал, че множеството, чието елементи трябва да се окажат покрити от U , е E , а не V . Върховете са атомарни обекти, така че—говорейки от общи съображения—за да реализираме покриване на V от едно множество, това множество трябва да е самото V .

Името на задачата е VERTEX COVER, защото има задача EDGE COVER и тя е различна. Задачата EDGE COVER иска да се намери минимално множество от ребра, такова че всеки връх е инцидентен с поне едно ребро от него [56, стр. 79].

Изч. Задача 53: EDGE COVER

екземпляр: Неориентиран граф $G = (V, E)$.

решение: Подмножество $E' \subseteq E$, такова че $\forall u \in V \exists e \in E' : u \in e$ и $|E'|$ е минимално.

Задачата много прилича на VERTEX COVER на пръв поглед, но всъщност е фундаментално различна. EDGE COVER е решима в полиномиално време от алгоритъм, който намира максимално съчетание и след това го разширява по алчен начин, така че всеки връх да се окаже инцидентен на някое избрано ребро [56, стр. 190]. Терминът *съчетание* е обяснен в Подподсекция 16.2.7.1.

На стр. 79 в [56] двойката задачи (EDGE COVER, VERTEX COVER), и двете във версия за разпознаване, е дадена като пример за задачи с минимална разлика в дефинициите, едната от които е ефикасно решима и е в класа на сложност **P** (Подсекция 14.4.5), а другата е в класа на сложност **NP-с** (Подсекция 14.5.5 и Определение 139) и, по всеобщото мнение, не е ефикасно решима. Други такива двойки са (EULERIAN CYCLE, HAMILTONIAN CYCLE) и (SHORTEST PATH, LONGEST PATH).

Върху дървета обаче, задачата VERTEX COVER е решима ефикасно. Дърветата имат много по-бедна структура от графите по принцип и това позволява да има ефикасно решение. В тази секция ще разгледаме линеен алгоритъм за VERTEX COVER върху дървета, изграден върху схемата **Динамично Програмиране**.

Линеен алчен алгоритъм за VERTEX COVER върху дървета. Нека T е произволно дърво с повече от два върха. Нека U е минималното върхово покриване, която ще изградим. В началото, $U = \emptyset$. Нека L_1 е множеството от висящите върхове на T . Нека L_2 е множеството от техните съседи. Твърдим, че не можем да сбъркаме, добавяйки L_2 към U . Наистина, всяко ребро между висящ връх и негов съсед трябва да бъде покрито в смисъл, че поне единият му край трябва да влезе в U . Нищо няма да загубим, ако вкараме в U връх v от L_2 :

- ако v е съсед на точно един w от L_1 , то поне един от $\{v, w\}$ трябва да влезе в U ,
- ако v е съсед на повече от един върха $w_1, \dots, w_k$, то, ако не сложим v в U , трябва да сложим всички $w_1, \dots, w_k$; а ако сложим v , можем да забравим за $w_1, \dots, w_k$, понеже v покрива ребрата $(v, w_1), \dots, (v, w_k)$.

И така, първо изтриваме върховете от L_1 от T и после правим $U \leftarrow U \cup L_2$ и изтриваме и L_2 от T . Ако не остане нищо от T , или ако останат дървета, всяко с един връх[†], то няма какво повече да правим – всички ребра от оригиналното T са покрити от L_2 . В противен случай, продължаваме така върху дърветата на гората $T - L_1 - L_2$: ако дадено дърво от гората е с едно ребро, добавяме към U кой да е край на реброто, в противен случай идентифицираме висящите върхове, слагаме техните съседи в U , изтриваме и тези висящи върхове, и въпросните съседи, и така нататък. Ефикасната имплементация може да е трикова, но идеята е ясна: “белим” дървото “отвън навътре”, като изхвърляме висящите върхове и вземаме техните съседи, и така докато можем.

Ако искаме да решим МАКСИМАЛНА АНТИКЛИКА, можем да походим дуално: пак “белим” отвън навътре, но сега вземаме висящите върхове, а изхвърляме съседите им.

Това е и краят на описанието на алчния алгоритъм.

Забележете, че следната алчна идея за VERTEX COVER върху дървета **не работи**:

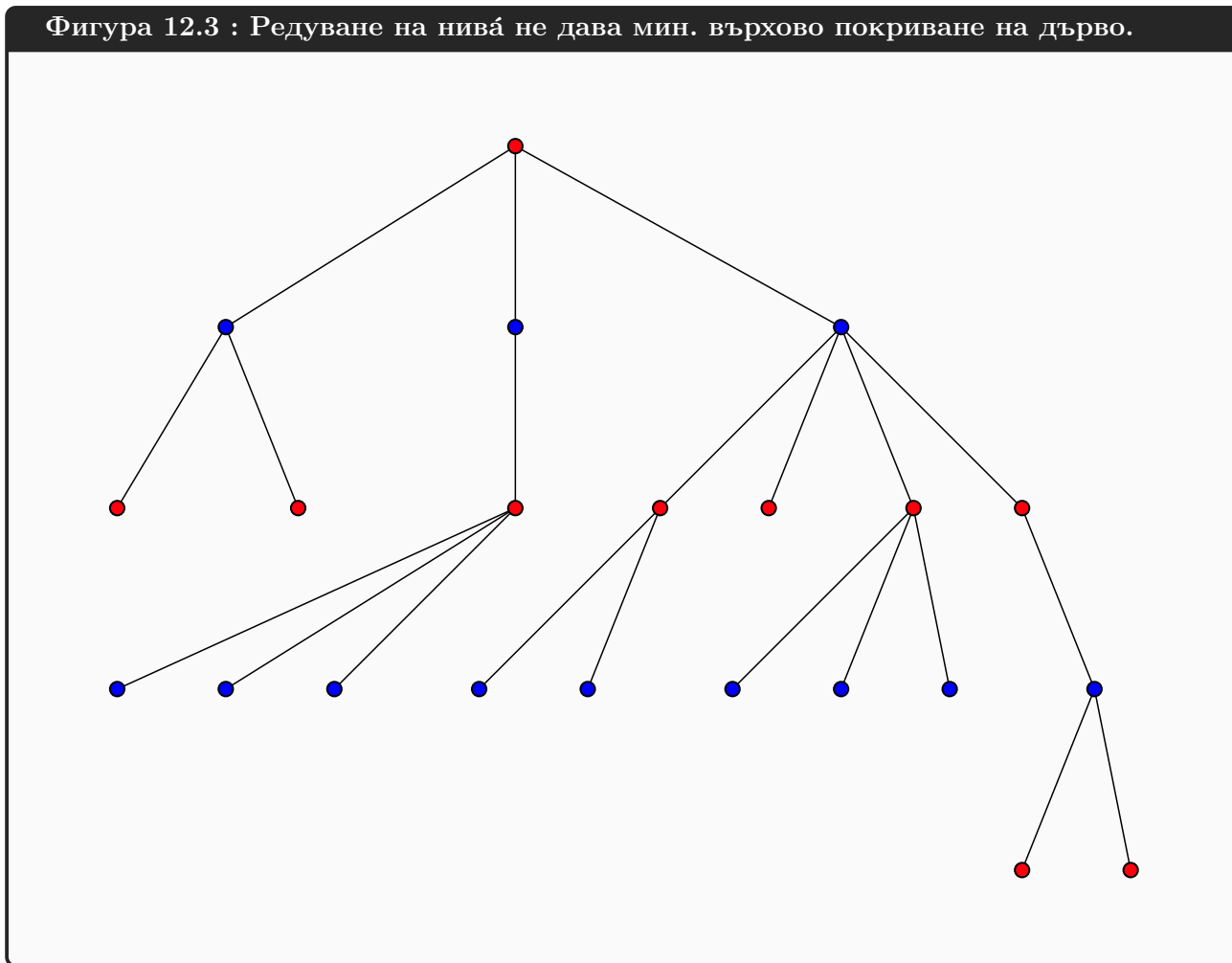
- да пуснем BFS[‡] върху дървото от кой да е връх s ,
- да разбием множеството от върховете на $V(T)$ на две подмножества: V_e , тези на четно разстояние от s , и V_o , тези на нечетно разстояние от s ,
- и да вземем като решение това от V_e, V_o , което е не по-голямо от другото,

не работи. Наистина, всяко от V_e и V_o е върхово покриване, понеже за всяко ребро (v, w) , единият край е във V_e , а другият във V_o , но резултатът може да не е оптимален. Фигура 12.3 показва дърво, на което нито V_e (червените върхове), нито V_o (сините върхове) е минимално върхово покриване. Очевидно $|V_e| = 10$ и $|V_o| = 11$, $\min\{10, 11\} = 10$ и опитът за решение чрез редуване на нива спрямо корена дава решение с тегло 10. Но Фигура 12.4 показва същото дърво след изпълнението на алгоритъма, изграден по схемата динамично програмиране. Онзи пример е за дуалната задача МАКСИМАЛНА АНТИКЛИКА, но щом максималната антиклика е с големина 15, то минималното върхово покриване е с големина 7 (при положение, че броят на върховете е 22).

[†]Забележете, че изтриването на $L_1 \cup L_2$ може да доведе до несвързан граф.

[‡]Или DFS. Върху дървета, DFS също пресмята разстояния, защото всеки два върха са свързани с един единствен път.

Фигура 12.3 : Редуване на нива не дава мин. върхово покриване на дърво.



Ефикасно решение върху дървета, изградено по схемата Динамично Програмиране. Решаваме дуалната задача МАКСИМАЛНА АНТИКЛИКА в олекотения вариант. Нека дървото е T . Първо правим дървото кореново, вкоренявайки в произволен връх r . За всеки връх $v \in V(T)$ поддържахме наредена двойка числа (v^+, v^-) , където

- v^+ е максималната мощност на антиклика в $T[v]$, която **съдържа** връх v ,
- а v^- е максималната мощност на антиклика в $T[v]$, която **не съдържа** връх v .

Ако v е листо, то очевидно $v^+ \leftarrow 1$ и $v^- \leftarrow 0$; това е дъното на рекурсията. Ако v е не-листо, то v има поне едно дете; в такъв случай (v^+, v^-) може да се пресметне ефикасно от наредените двойки на децата на v . В крайна сметка, отговорът е $\max\{r^+, r^-\}$, защото всяко оптимално решение за цялото дърво или съдържа, или не съдържа корена. А как да пресмятаме наредените двойки за не-листата? Ето как. Нека v има деца $w_1, \dots, w_k$. Тогава

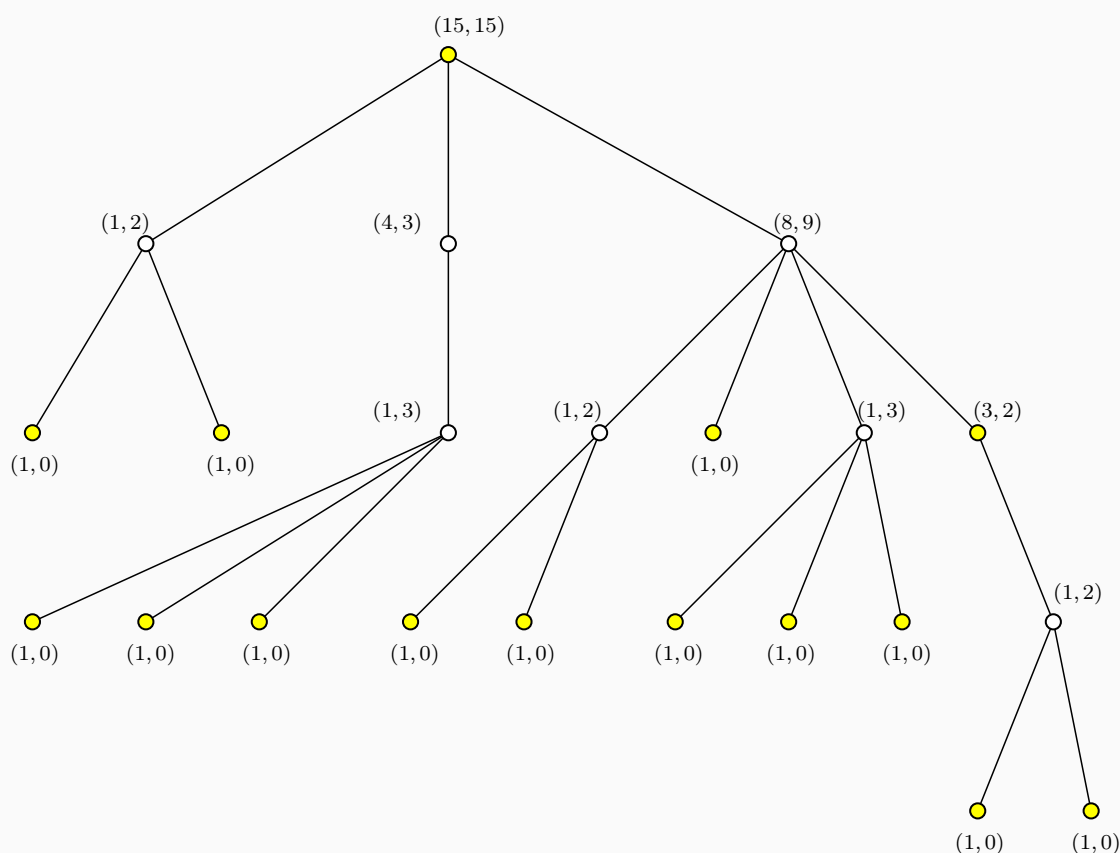
$$v^+ \leftarrow 1 + \sum_{i=1}^k w_i^- \quad (12.55)$$

$$v^- \leftarrow \sum_{i=1}^k \max \{w_i^-, w_i^+\} \quad (12.56)$$

Фигура 12.4 показва примерно кореново дърво с 22 върха и работата на алгоритъм, изграден върху рекурсивната декомпозиция (12.55) и (12.56). Наредената двойка (15, 15) в корена

означава, че има антиклика с размер 15, съдържаща корена, и антиклика с размер 15, не съдържаща корена.

Фигура 12.4 : Намиране на максимална антиклика в дърво с 22 върха.



Петнадесетте върха в жълто са оптимално решение, съдържащо корена. Очевидно размерът на минималното върхово покриване е $22 - 15 = 7$ и белите върхове са едно такова минимално върхово покриване.

Коректността на рекурсивната декомпозиция е очевидна. Сложността по време е линейна, предвид това, че алгоритъмът може да се имплементира като модифициран DFS с добавена само константа работа във всяко рекурсивно викане.

Алгоритъмът, изграден по схемата **Динамично Програмиране**, има значително предимство пред алчния алгоритъм, който скицирахме на стр. 705: алгоритъмът по схемата **Динамично Програмиране** може да се модифицира тривиално така, че да решава **МАКСИМАЛНА АНТИКЛИКА** в тегловен вариант.

Изч. Задача 54: INDEPENDENT SET, WEIGHTED

екземпляр: Неориентиран граф $G = (V, E)$, функция $g : V \rightarrow \mathbb{R}$.

решение: Подмножество $U \subset V$, такова че U е антиклика и $\sum_{u \in U} |g(u)|$ е максимална.

Забележете, че тук говорим за тегла на върховете, а не на ребрата. Досега теглата в тегловните графи бяха все върху ребрата (Определение 86).

Да кажем, че решаваме МАКСИМАЛНА АНТИКЛИКА в тегловен вариант, модифицирайки алгоритъма по схемата **Динамично Програмиране**. Рекурсивната декомпозиция се променя много лесно. Ако v е листо, то $v^+ \leftarrow g(v)$ и $v^- \leftarrow 0$. В противен случай,

$$v^+ \leftarrow g(v) + \sum_{i=1}^k w_i^- \quad (12.57)$$

$$v^- \leftarrow \sum_{i=1}^k \max \{w_i^-, w_i^+\} \quad (12.58)$$

По този начин получаваме алгоритъм за задачата в тегловен вариант със същата линейна сложност. Докато алчният алгоритъм на стр. 705 няма очевидно обобщение, което решава тегловния вариант на задачата.

12.8.2 Минимално доминиращо множество на дърво

Доминиращо множество, на английски *dominating set*, на (обикновен) граф G е всяко подмножество от върхове, такова че за всеки връх е вярно, че се намира в това множество или[†] има съсед в него. Задачата има многобройни приложения, например:

- ако графът моделира пътна мрежа от градове-върхове и пътища-ребра и се иска да се построят минимален брой болници в градовете по такъв начин, че всеки град без болница да има съседен град, в който има болница, то градовете с болниците се явяват минимално доминиращо множество;
- ако графът моделира взаимната видимост на върховете в планарна фигура, то всяко минимално подмножество от върхове U , такова че всеки връх извън U се вижда от някой връх от U , се явява минимално доминиращо множество за графа (вижте Допълнение 87).

Дефинициите на “върхово покриване” и “доминиращо множество” си приличат повърхностно, но всъщност са доста различни. Като екстремален пример да разгледаме пълен граф K_n . Той няма върхово покриване с размер, по-малък от $n - 1$, защото очевидно трябва да изтрием $n - 1$ върха, за да не останат ребра. От друга страна, той има доминиращо множество с размер 1, защото всеки негов връх е съсед на всички останали. Изолираните върхове се “държат” различно спрямо тези две задачи: за минималното върхово покриване те са без значение (все едно ги няма), докато всяко доминиращо множество трябва да съдържа всички изолирани върхове, понеже изолиран връх не може да бъде доминиран от друг връх.

Мнение на автора е, че в практиката задачата за минимално върхово покриване най-често се появява в графи, които моделират несъвместимост, докато задачата за минимално доминиращо множество най-често се появява в графи, които моделират афинитет.

Формалното определение е следното.

Изч. Задача 55: DOMINATING SET

екземпляр: Неориентиран граф $G = (V, E)$.

решение: Подмножество $U \subset V$, такова че $\forall u \in V (u \in U \vee \exists v \in N(u)(v \in U))$ и $|U|$ е минимално.

[†]Това е включващо “или”.

Мощността на минимално по мощност доминиращо множество на G се нарича *числото на доминирането на G* и се бележи с $\gamma(G)$.

Теорема 106: В граф без изол. в-ве, всяко върхово покриване е доминиращо множество

Нека $G = (V, E)$ е граф без изолирани върхове и $U \subseteq V$. Ако U е върхово покриване на G , то U е доминиращо множество в G .

Доказателство: Нека U е произволно върхово покриване на G . Това означава, че за всяко ребро, поне единият край е в U . Ще докажем, че за всеки връх $u \in V$, $u \in U$ или u има съсед в U . Нека u е произволен връх, който не е в U . Ще покажем, че u има съсед в U . Да допуснем противното: нито един съсед на u не е връх от U . Но G няма изолирани върхове, така че u не е изолиран връх, така че u има поне един съсед v . По допускане, $v \notin U$. Тогава нито един от краищата на реброто (u, v) не е в U , следователно U не е върхово покриване. $\downarrow$ $\square$

Забележете следната фундаментална разлика между VERTEX COVER и DOMINATING SET. Ако $G = (V, E)$ е граф, $u, v \in V$ не са съседни и $G' = (V, E \cup \{u, v\})$, то $\tau(G) \leq \tau(G')$, но $\gamma(G) \geq \gamma(G')$. Че $\tau(G) \leq \tau(G')$ е очевидно, понеже всяко новодобавено ребро е нова “грижа” по отношение на върховото покриване – произволно върхово покриване U на G може да не съдържа нито u , нито v , поради което U не се явява покриване на G' .

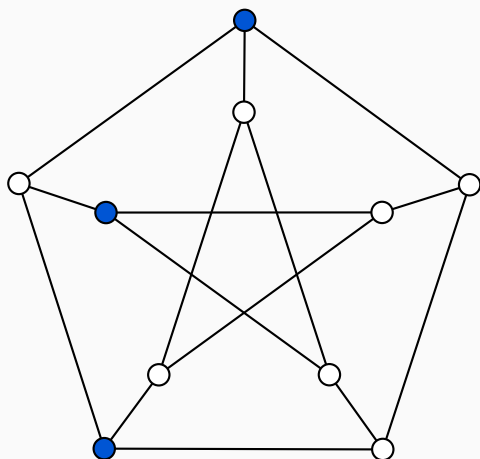
Да се убедим, че $\gamma(G) \geq \gamma(G')$. Нека U е произволно доминиращо множество за G . Със сигурност U доминира и G' :

- ако $\{u, v\} \cap U \neq \emptyset$, то в G' продължава да е вярно, че всеки връх е в U или има съсед в U ;
- ако $\{u, v\} \cap U = \emptyset$, то в G и u , и v имат съсед в U , така че и в G' и u , и v имат съсед в U , понеже добавянето на ребро оставя това истина.

Лесно може да се измисли пример, в който добавянето на ново ребро намалява числото на доминиране, защото предоставя нова възможност връх да доминира свой съсед. И така, по отношението на доминирането, всяко новодобавено ребро е нова възможност, а не грижа.

Пример за доминиращо множество е показан на Фигура 12.5. Графът на Petersen не може да бъде доминиран от по-малко от 3 върха: той е 3-регулярен граф, а във всеки 3-регулярен граф, един връх може да доминира най-много четири върха (сам себе си и трите си съседа), така че два върха не могат да доминират десетте върха на P . Ерго, 3 е долна граница за мощността на доминиращото множество и доминиращото множество на Фигура 12.2 е оптимално. И така, $\gamma(P) = 3$. Виждаме, че конверсното твърдение на Теорема 106 не е вярно: трите върха на Фигура 12.5 не покриват графа на Petersen, който “иска” поне шест върха, за да бъде покрит, както отбелязахме на стр. 704.

Фигура 12.5 : Минимално доминиращо множество в графа на Petersen.



Ефикасно решение на DOMINATING SET върху дървета по схемата Динамично Програмиране. Тъй като обосновката на рекурсията е значително по-неочевидна от тази на МАКСИМАЛНА АНТИКЛИКА върху дървета, ще я направим бавно и внимателно. Решаваме задачата в тежкия вариант.

За удобство да въведем следните нотации. Нека H е произволно кореново дърво. " $D(H)$ " означава някое оптимално доминиращо множество за H . " $D^+(H)$ " означава някое оптимално доминиращо множество за H , което съдържа корена. " $D^-(H)$ " означава някое оптимално доминиращо множество за H , което не съдържа корена.

Нотация 13: $\uplus$

Знакът " $\uplus$ " е за обединение на две множества с празно сечение. Например, може да ползваме " $A \uplus B$ ", ако е известно, че $A \cap B = \emptyset$. Ако това не е известно и напишем " $A \uplus B$ ", то приемаме неявно, че $A \cap B = \emptyset$.

Ако напишем $A_1 \uplus A_2 \uplus \dots \uplus A_k$, неявно приемаме, че $A_1, A_2, \dots, A_k$ са множества, които по двойки имат празно сечение.

На английски се казва *disjoint union*.

Забележете, че ако $A_1, A_2, \dots, A_k$ са непразни множества и $B = A_1 \uplus A_2 \uplus \dots \uplus A_k$, то $\{A_1, \dots, A_k\}$ се явява разбиване на B .

Освен това, знакът " $\uplus$ " не ни е абсолютно необходим в смисъл, че спокойно можем да ползваме " $\cup$ " вместо него. Идеята на " $\uplus$ " е да се напомни или акцентира на читателя, че множествата-операнди нямат общи елементи; ако използваме " $\cup$ " наместо " $\uplus$ ", резултатът би бил същият. В случай, че ни интересува мощността на обединението и знаем, че $B = A_1 \uplus \dots \uplus A_k$, то $|B| = \sum_{i=1}^k |A_i|$ по принципа на разбиването. Ако обаче само знаем, че $B = A_1 \cup \dots \cup A_k$, то трябва да използваме принципа на включването и изключването, за да намерим $|B|$.

В Определение 78 въведохме " argmin ". Тук ползваме " argmin " с незначителна формална злоупотреба.

$$D(H) = \operatorname{argmin} \{|D^+(H)|, |D^-(H)|\} \quad (12.59)$$

Формално, това е невалидно използване на “argmin”. Формално правилно би било $D(H)$ да е множеството от оптималните доминиращи множества, а $D^+(H)$ и $D^-(H)$ да са множествата от оптималните доминиращи множества, съответно съдържащи и несъдържащи корена; при това положение имаме право да напишем

$$D(H) = \operatorname{argmin}_{U \in D^+(H) \cup D^-(H)} |U| \quad (12.60)$$

Това е вярно и формално прецизно, но не можем да го превърнем в ефикасен алгоритъм, защото тези множества може да са грамадни. За ефикасен алгоритъм, ние искаме **едно** оптимално доминиращо множество, а не всички. Поради това ще пишем формално неизрядни изрази като (12.59), а не формално изрядни изрази като (12.60).

(12.59) се интерпретира така: ако разполагаме с **едно** оптимално доминиращо множество, съдържащо корена, и **едно** оптимално доминиращо множество, несъдържащо корена, то не по-голямото от тях е **едно** оптимално решение за H .

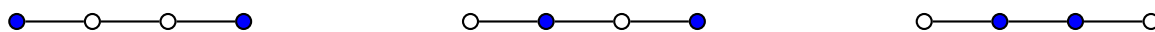
Нека дървото, върху което решаваме задачата, е $T = (V, E)$. БОО, нека T има повече от един връх. Правим T кореново дърво, вкоренявайки го в произволен връх r . Нека децата на r са $w_1, \dots, w_k$. Започваме да разсъждаваме по начина, по който разсъждавахме за МАКСИМАЛНА АНТИКЛИКА върху дървета. Търсим $D(T)$. Коренът r или е в $D(T)$, или не е в $D(T)$.

- Да допуснем, че $r \in D(T)$. Търсим $D^+(T)$. Очевидно $\{r\} \cup \bigcup_{i=1}^k D(T[w_i])$ е доминиращо множество за дървото T , съдържащо корена. Но дали е оптимално решение за T ? Не непременно. Това, че r задължително участва в доминиращо множество ни дава свободата за всяко $T[w_i]$ да вземем множество, което доминира всеки връх, **може би без** w_i . Причината е проста: всички w_i биват доминирани от r така или иначе, ерго, може да си позволим за поддърветата, вкоренени в децата на r , да вземем “почти доминиращи” множества – такива, в които коренът (той е дете на r) може да не е доминиран.

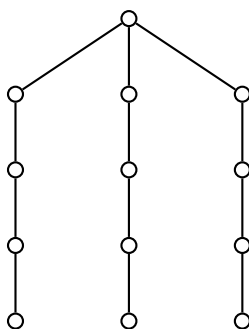
Ето прост пример. Първо забелязваме, че прост път с дължина 3 не може да бъде доминиран от един връх: който и връх да сложим в доминиращото множество, поне един връх остава недоминиран. БОО, достатъчно е да разгледаме следните два случая. Върхът, чрез който опитваме да доминираме пътя, е в синьо. Върховете, които остават недоминирани, са в сиво.



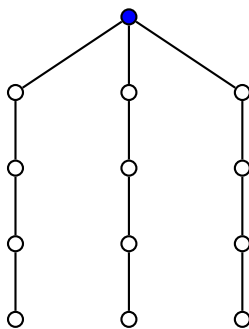
Второ, забелязваме, че два върха могат да доминират пътя по някой от тези три начина:



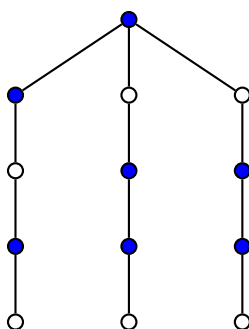
Трето, конструираме самия екземпляр. Както обикновено, коренът е нарисован най-горе.



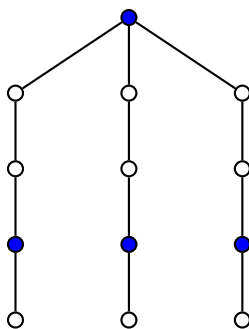
Разглеждаме случая, в който коренът е в доминиращото множество.



Поддърветата, вкоренени в децата на корена, са пътища с дължина 3. Както вече видяхме, всяко от тези поддървета има число на доминиране 2. Ако вземем решение за цялото дърво, което се състои от корена и по едно решение за всяко дете, получаваме решение с мощност 7, като например това:



Всъщност, има решение с мощност 4, съдържащо корена:



Това е краят на примера. Видяхме, че, ако коренът е в решението, за поддърветата, вкоренени в децата, можем да си позволим решения, които не доминират техните корени, понеже същите се оказват доминирани от корена на цялото дърво. Накратко, **не е непременно вярно**, че $|D^+(T)| = 1 + \sum_{i=1}^k |D(T[w_i])|$, поради което $\{r\} \uplus \biguplus_{i=1}^k D(T[w_i])$ не е непременно оптимално решение, съдържащо корена.

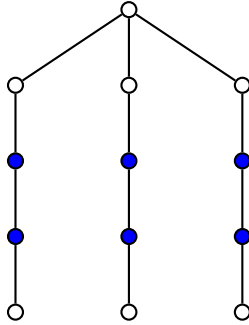
Ще въведем още една нотация. Нека H е произволно кореново дърво. “ $\tilde{D}(H)$ ” означава някое множество с минимална мощност, което доминира всеки връх на H , може би с изключение на корена.

Забележете, че $|\tilde{D}(H)| \leq |D(H)|$, като е възможно строго неравенство, пример за което видяхме преди малко (дървото е път, вкоренен в единия си край).

И така,

$$D^+(T) = \{r\} \uplus \biguplus_{i=1}^k \tilde{D}(T[w_i]) \quad (12.61)$$

- Да допуснем, че $r \notin D(T)$. Човек би се изкушил да каже, че решението е $\biguplus_{i=1}^k D(T[w_i])$, но това множество може да остави корена недоминиран, както се вижда от следния пример.



Коренът може да бъде доминиран само от някое от децата си, при положение, че той самият не е в доминиращото множество. Ерго, налага се да искаме поне едно от децата на корена да е задължително в доминиращото множество. Тогава ще ползваме D^+ за точно едно w_i и D за останалите деца. А за кое i да ползваме $D^+(T[w_i])$? За това, за което получаваме оптимално решение. Накратко,

$$D^-(T) = \operatorname{argmin} \left\{ D^+(T[w_i]) \uplus \biguplus_{j \in \{1, \dots, k\} \setminus \{i\}} D(T[w_j]) : 1 \leq i \leq k \right\} \quad (12.62)$$

Получихме изрази за $D^+(T)$ и $D^-(T)$, съответно (12.61) и (12.62), но с цената на въвеждането на $\tilde{D}(T)$. Налага са да изразим и него. Припомняме си, че $\tilde{D}(T)$ е минимално по мощност множество от върхове, което доминира дървото, но може би с изключение на корена. Фразата “може би” е ключова, както ще видим.

От общи съображение, човек би казал, че $\tilde{D}(T) = \biguplus_{i=1}^k D(T[w_i])$. И би сбъркал. Със сигурност множеството $\biguplus_{i=1}^k D(T[w_i])$ доминира всички върхове, евентуално без корена. Обаче това множество може да не е оптимално! То е оптимално, ако коренът **не е** в решението. Но нищо не забранява коренът **да е** в решението. Разглеждаме множество $\tilde{D}(T)$, което съдържа корена. Това, че коренът се съдържа, ни “развързва ръцете” за децата му $w_1, \dots, w_k$ да вземем не D , а $\tilde{D}$. С други думи, ако коренът е в $\tilde{D}(T)$, можем да си позволим за поддърветата, вкоренени в децата, да вземем решения, които може да не съдържат техните корени. Ето пример за това.

- Вляво е минимално решение, което не съдържа корена, но поддърветата, вкоренени в децата на корена, са доминирани напълно от подрешенията, индуцирани от поддърветата, вкоренени в децата на корена.
- Вдясно е минимално решение, което съдържа корена, и в което поддърветата, вкоренени в децата на корена, не са напълно доминирани от подрешенията, индуцирани от поддърветата, вкоренени в децата на корена.



Очевидно дясното решение, което има мощност 4, е по-добро от лявото решение, което има мощност 6. И за двете решения имаме право да кажем, че са решения, в които коренът (на цялото дърво) не е непременно доминиран. Това, че вдясно той е доминиран, и не просто е доминиран, а се намира в решението, не представлява противоречие.

И така,

$$\tilde{D}(T) = \operatorname{argmin} \left\{ \left| \biguplus_{i=1}^k D(T[w_i]) \right|, \left| \{r\} \uplus \biguplus_{i=1}^k \tilde{D}(T[w_i]) \right| \right\} \quad (12.63)$$

Съставихме четири уравнения с четири неизвестни. Да напишем цялата система (засега изпускаме началните условия). Ще използваме малко променена нотация: ако кореновото дърво е H , с “ $c(H)$ ” означаваме множеството от децата на корена.

$$D(T) = \operatorname{argmin} \{ |D^+(T)|, |D^-(T)| \}$$

$$D^+(T) = \{r\} \uplus \biguplus_{w \in c(T)} \tilde{D}(T[w])$$

$$D^-(T) = \operatorname{argmin} \left\{ \left| D^+(T[w]) \uplus \biguplus_{z \in c(T) \setminus \{w\}} D(T[z]) \right| : w \in c(T) \right\}$$

$$\tilde{D}(T) = \operatorname{argmin} \left\{ \left| \biguplus_{w \in c(T)} D(T[w]) \right|, \left| \{r\} \uplus \biguplus_{w \in c(T)} \tilde{D}(T[w]) \right| \right\}$$

Тази система може да се опрости значително. D се изразява чрез D^+ и D^- , но

- D^+ се изразява чрез $\tilde{D}$,
- D^- се изразява чрез D^+ и D ,
- $\tilde{D}$ се изразява чрез D и $\tilde{D}$.

Очевидно можем да изразим всичко само чрез D и $\tilde{D}$ по този начин:

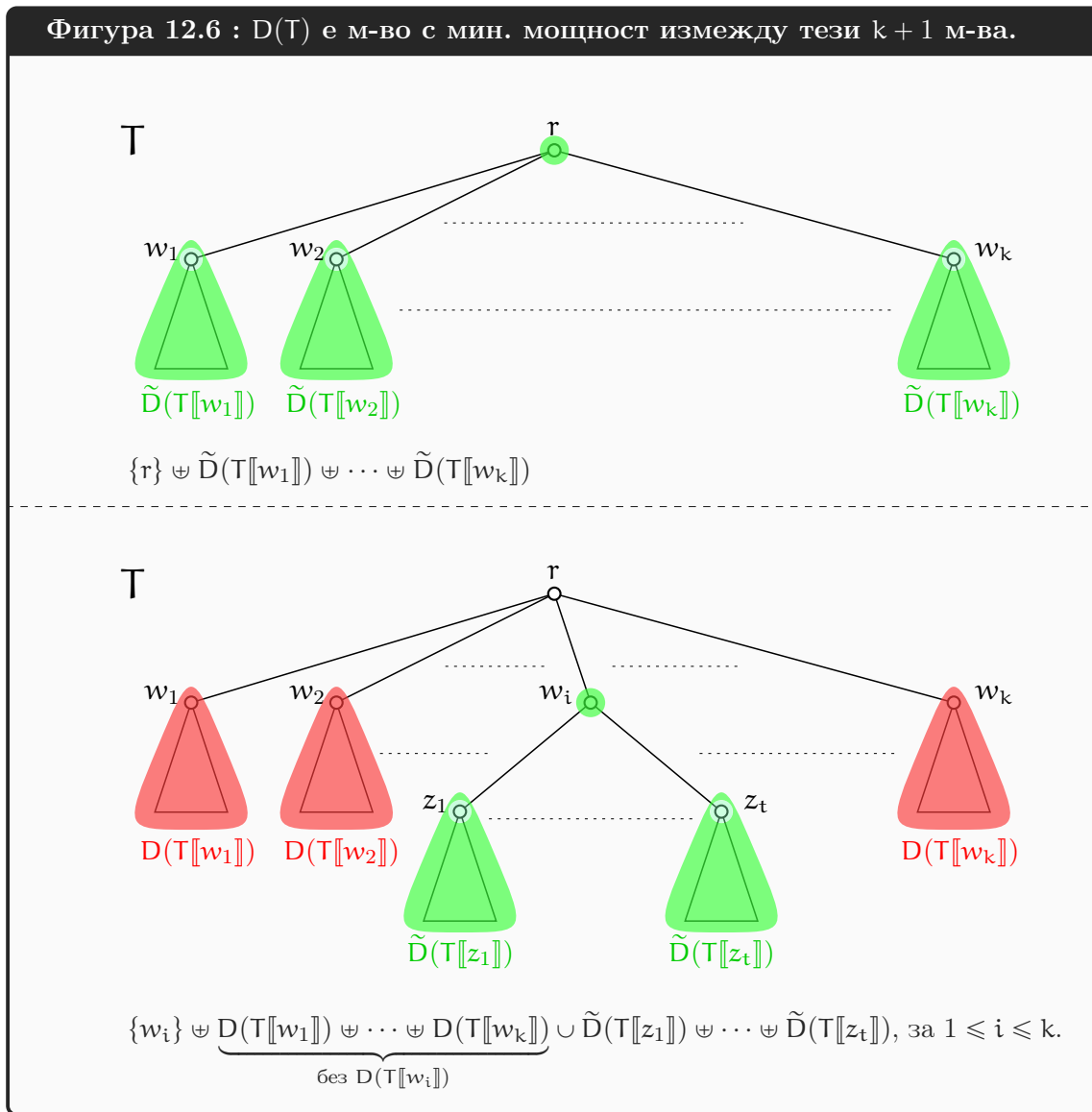
$$D(T) = \operatorname{argmin} \left\{ \left| \{r\} \uplus \biguplus_{w \in c(T)} \tilde{D}(T[w]) \right|, \right. \quad (12.64)$$

$$\left. \operatorname{argmin} \left\{ \left| \{w\} \uplus \biguplus_{z \in c(T[w])} \tilde{D}(T[z]) \uplus \biguplus_{a \in c(T) \setminus \{w\}} D(T[a]) \right| : w \in c(T) \right\} \right\}$$

$$\tilde{D}(T) = \operatorname{argmin} \left\{ \left| \biguplus_{w \in c(T)} D(T[w]) \right|, \left| \{r\} \uplus \biguplus_{w \in c(T)} \tilde{D}(T[w]) \right| \right\} \quad (12.65)$$

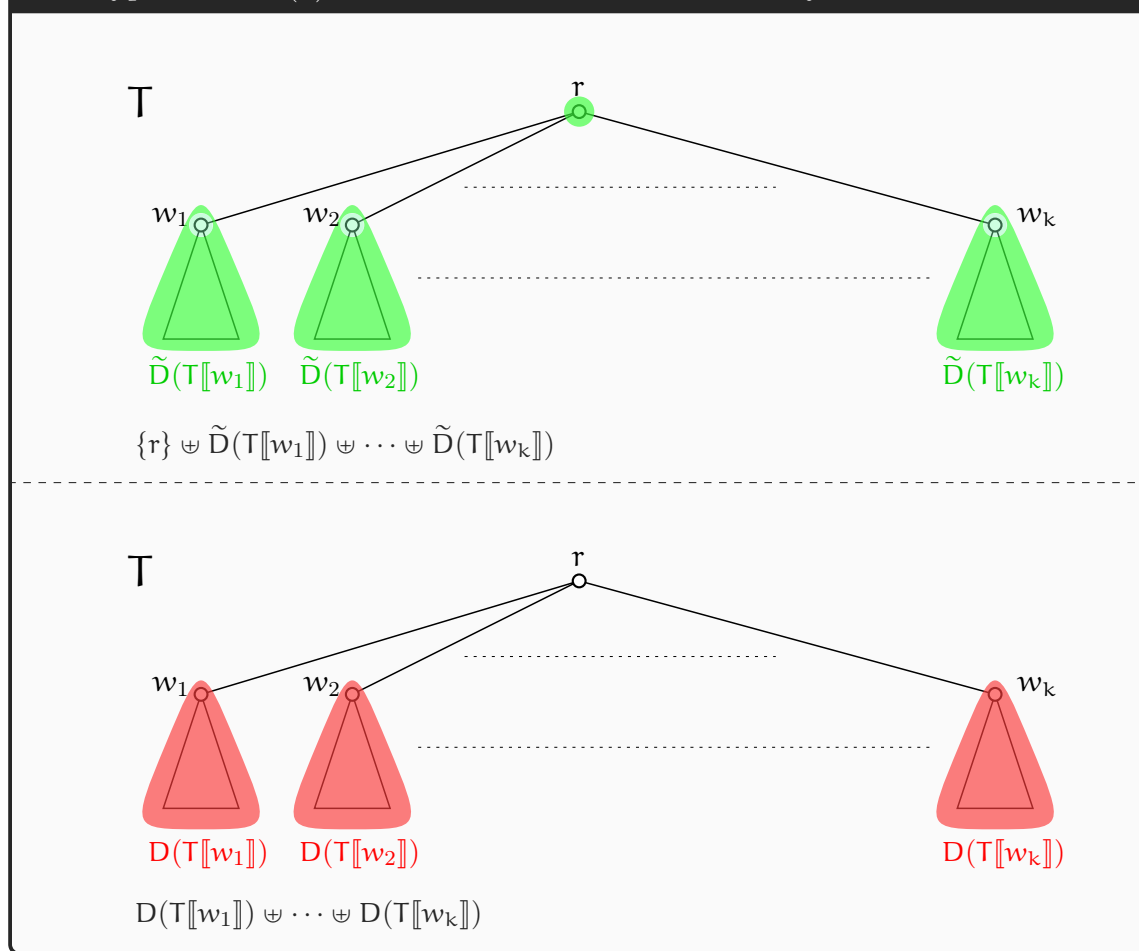
С червено и синьо е показано как D^+ и D^- биват замествани, за да изчезнат и да останат само D и $\tilde{D}$.

Фигура 12.6 илюстрира двата операнда в минимума в 12.64.



Фигура 12.7 илюстрира двата операнда в минимума в 12.65.

Фигура 12.7 : $\tilde{D}(T)$ е м-во с мин. мощност измежду тези две м-ва.



В светлината на опростените изрази, искаме началните условия само за D и $\tilde{D}$. И така, ако $T = (\{r\}, \emptyset)$, то

$$D(T) = r, \tilde{D}(T) = \emptyset \quad (12.66)$$

Тежкия вариант на задачата се решава от рекурсивна декомпозиция, състояща се от (12.64), (12.65) и (12.66). От нея лесно може да се получи линеен алгоритъм по схемата **Динамично Програмиране**.

За олекотения вариант на задачата. Има решение за олекотения вариант на DOMINATING SET върху дървета, което е популярно (може да се открие на много места в Интернет) и с неясен произход (поне за автора на записките), което го прави фолклор. Горедолу, това решение е следното.

DS ON TREES($T = (V, E)$: дърво)

- 1 $A[1, \dots, n], B[1, \dots, n]$: масиви от цели числа
- 2 нека r е произволен връх от V
- 3 направи T кореново дърво с корен r
- 4 работи от листата нагоре по следния начин
- 5 за всяко листо u
- 6 $A[u] \leftarrow 1, B[u] \leftarrow 0$

```

7      за всяко не-листо  $u$ 
8       $A[u] \leftarrow 1 + \min \left\{ \sum_{v \in c(u)} B[v], \min_{v \in c(u)} \left\{ \sum_{w \in c(v)} B[w] + \sum_{z \in c(u) \setminus \{v\}} A[z] \right\} \right\}$ 
9       $B[u] \leftarrow \min \left\{ 1 + \sum_{v \in c(u)} B[v], \sum_{v \in c(u)} A[v] \right\}$ 
10 return  $A[r]$ 

```

$A[u]$ има смисъл на минимално доминиращо множество в поддървото, вкоренено в u . $B[u]$ има смисъл на множество с минимална мощност в поддървото, вкоренено в u , което множество доминира всеки връх, може би с изключение на u .

Да се убедим, че това популярно решение на олекотения вариант следва от (12.66), (12.64) и (12.65):

- 6 отговаря на (12.66),
- 8 отговаря на (12.64),
- 9 отговаря на (12.65).

Това, че 6 отговаря на (12.66) и 9 отговаря на (12.65), е очевидно. Че 8 отговаря на (12.64) може би не е толкова очевидно; за да видите съответствието, забележете, че множествата $\{r\}$ и $\{w\}$ дават по едно $+1$, когато минем от изрази за множества към изрази за мощности на множества, и тези две $+1$ може да бъдат изнесени пред $\min$, както става на ред 8.

Минаването от тежкия вариант, даващ оптимално доминиращо множество, в олекотения вариант, даващ само цената на оптимално доминиращо множество, става толкова лесно, понеже (12.64) и (12.65) ползват само обединения на множества без общи елементи, така че появите на $\cup$ директно се превеждат в суми.

Задачата във вариант с тегла. Задачата е следната.

Изч. Задача 56: DOMINATING SET, WEIGHTED

екземпляр: Неориентиран граф $G = (V, E)$, функция $g : V \rightarrow \mathbb{R}$.

решение: Подмножество $U \subset V$, такова че $\forall u \in V (u \in U \vee \exists v \in N(u) (v \in U))$ и $\sum_{u \in U} |g(u)|$ е минимална.

DS ON TREES, WEIGHTED решава задачата с тегла в олекотения вариант. Има малка разлика, но съществена разлика между него и DS ON TREES: на ред 8 в DS ON TREES добавяме единица вдясно **извън минимума**, докато на ред 9 в DS ON TREES, WEIGHTED се налага да сложим **вътре в минимума** теглото на върха, който задължително участва в съответния операнд (на минимума). Затова на ред 9 в DS ON TREES, WEIGHTED имаме веднъж $g(u)$ в първия операнд и след това $g(v)$ във втория операнд, в който v взема всички стойности на дете на u .

DS ON TREES, WEIGHTED($T = (V, E)$: дърво, g : тегловна функция върху върховете)

- 1 $A[1, \dots, n], B[1, \dots, n]$: масиви от цели числа
- 2 нека r е произволен връх от V
- 3 направи T кореново дърво с корен r

```

4 работи от листата нагоре по следния начин
5   за всяко листо  $u$ 
6    $A[u] \leftarrow g(u), B[u] \leftarrow 0$ 
7   за всяко не-листо  $u$ 
8    $A[u] \leftarrow \min \left\{ g(u) + \sum_{v \in c(u)} B[v], \min_{v \in c(u)} \left\{ g(v) + \sum_{w \in c(v)} B[w] + \sum_{z \in c(u) \setminus \{v\}} A[z] \right\} \right\}$ 
9    $B[u] \leftarrow \min \left\{ g(u) + \sum_{v \in c(u)} B[v], \sum_{v \in c(u)} A[v] \right\}$ 
10 return  $A[r]$ 

```

12.9 Игри

В тази секция разглеждаме игри, в които играят два двама противници, които се редуват. На английски такава игра се нарича *turn based game*. Много настолни игри са такива, например игрите табла, шах и Го. Играта морски шах от Допълнение 71 също е *turn based game*, въпреки че не е настолна игра и при нея всеки играч разполага само с частична информация, за разлика от шаха, таблата и играта Го, в които всеки от играчите вижда цялата позиция[†].

Подчертаваме, че не става дума за компютърни игри, които често не са *turn based*, понеже играчите играят едновременно.

Да кажем, че противниците се наричат Албена и Борис, като Албена играе първа. Кой играе последен или последна обикновено зависи от развитието на играта и не е дефинирано в правилата.

Какво означава да се спечели играта зависи от правилата. В някои игри печалбата е “булева”: единият играч печели, а другият губи, и няма други възможности. Такава е The Daisy Petals Game (Подсекция 12.9.1), в която все някой ще откъсне последен или последна. Тези игри приличат на задачите за разпознаване (Определение 3).

В по-сложни игри като шаха или единият печели и другият губи, или завършват наравно[‡]. В някакъв смисъл може да кажем, че и шахът прилича на задача за разпознаване, защото за конкретна позиция има смисъл да зададем въпроса с булев отговор “Има ли печеливша стратегия за белите в n хода?”.

Има обаче игри, в които печалбата има числено изражение и те приличат на оптимизационните задачи (Определение 4). Пример за такава игра е играта със събирането на монети в Подсекция 12.9.2. В тях въпросът не е **дали** даден играч печели или не, а **колко** най-много може да спечели, ако играта е максимизационна, или колко най-малко може да спечели, ако играта е минимизационна.

От особена важност е да дефинираме какво означава “оптимална стратегия” на даден играч. Оптималната стратегия, както и името подсказва, е набор от правила (може да кажем, че е следване на алгоритъм), при които даден играч постига възможно най-много **дори при**

[†]Авторът не се наема да каже дали настолни игри с много играчи като монопол, или игрите с карти, са *turn based games*, но това няма значение за тази лекция.

[‡]В шаха може противниците да се съгласят на реми, може да се стигне до пат, може да се стигне до трикратно повторение на позиция или поредица от 50 хода, в които не е местена пешка и не е взета фигура, може и двете страни да останат с фигури, които теоретично не могат да дадат мат, например цар и два коня. В някои разновидности на таблата е възможно, макар и много малко вероятно, равенство; авторът има предвид тапата, но може да има и други видове табла, в които е възможно равенство. В играта Го също може да се стигне до равенство.

най-добра игра на противника. Авторът се изкушава да каже “за всяка игра на противника”, но това може е подвеждащо.

Да кажем, че играта е “булева” и равенство е невъзможно, като в играта с венчелистчетата. Тогава “оптимална стратегия” за даден играч се дефинира, ако за нея или него винаги има начин да спечели, както и да играе противникът. Още може да кажем “печели задължително”. Това има точно определен смисъл, който може да се изрази чрез езика на предикатната логика, редувайки екзистенциални и универсални квантори. Албена печели задължително тстк

- съществува ход на Албена, такъв че
- за всеки ход на Борис
- съществува ход на Албена, такъв че
- за всеки ход на Борис
- ...
- за всеки ход на Борис
- съществува ход на Албена, с който тя печели.

Вторият играч Борис печели задължително тстк

- за всеки ход на Албена
- съществува ход на Борис, такъв че
- за всеки ход на Албена
- съществува ход на Борис, такъв че
- ...
- за всеки ход на Албена
- съществува ход на Борис, с който Борис печели.

Очевидно второто се явява негация на първото. Оттам и тези двете са несъвместими: няма как Албена да печели задължително и Борис да печели задължително. В играта с пълна маргаритка, както ще видим, Борис печели задължително. Може също да кажем, че Албена губи задължително (отново: не абсолютно винаги, а при оптимална игра на Борис).

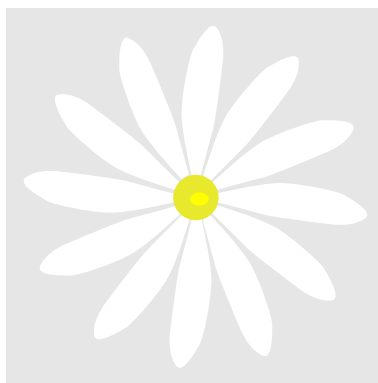
Ако даден играч губи задължително при оптимална игра на противника—какъвто е случаят с Албена в играта с пълна маргаритка—няма смисъл да говорим за оптимална стратегия за него или нея.

При игри с количествен краен резултат, каквато е играта със събирането на монети, “оптимална стратегия” за даден играч означава нещо различно, а именно колко най-много[†] може да реализира даден играч при оптимална игра на противника. При тези игри има смисъл да се говори за оптимална стратегия и за двата противника.

[†]“Колко най-много” казваме, когато целта на играта е максимизация; ако целта беше минимизация, щеше да е “колко най-малко”.

12.9.1 The Daisy Petals Game

Дадена е маргаритка като тази:



За краткост да наречем играчите А и Б. Играчите се редуват да късат едно или две венчелистчета, но ако са две, задължително трябва да са съседни. Печели този, който откъсне последен. Ето едно възможно развитие на играта, при което първият печели (отляво надясно):



Задачата е да се каже дали А печели задължително, или Б печели задължително.

Лесният вариант: “пълна” маргаритка. Доколкото е известно на автора на тези записки, задачата е популярна във варианта, в който се започва от маргаритка, от която не е късано нищо (“пълна” маргаритка).

Ако броят на венчелистчетата е четен и по-голям от 2, както е в примера, който видяхме (бяха 12 поначало), има елементарна стратегия за Б, с която Б печели задължително: след всеки ход на А,

- ако А е откъснала едно венчелистче, то Б къса противоположното венчелистче,
- ако А е откъснала две венчелистчета, то Б къса двете противоположни венчелистчета.

В някакъв смисъл, Б играе “огледално” на А. Гарантирано Б откъсва последно.

И така, ако стартираме от пълна маргаритка с четен брой венчелистчета, поне 4, при оптимална игра вторият играч печели задължително. В примера, който видяхме, А спечели заради грешки на Б.

По-труден вариант: “непълна” маргаритка. Възниква въпрос, ако началната конфигурация не е “пълна” маргаритка, а някакъв общ случай, в който се допуска някои венче-

листчета да липсват, примерно така:



, дали А печели задължително, или Б печели

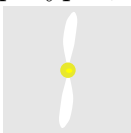
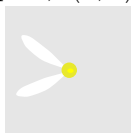
задължително?

Сега началната конфигурация се описва не просто с броя на венчелистчетата. Сега има значение и как са групирани заедно; тоест, къде са празнините. Колко са големи празнините (тоест, във всяка празнина колко венчелистчета е имало поначало) няма значение. Важното е колко групи (максимални по включване области без липсващи венчелистчета) има и колко венчелистчета има във всяка група. В последния пример групите са 3 и могат да бъдат описани с мултимножеството $\{2, 2, 3\}_M$ – очевидно е, че няма значение точно как са разположени една спрямо друга. БОО да допуснем, че елементите на мултимножеството са написани отляво надясно в ненарастващ ред и тогава можем да го опишем като редица $\langle 3, 2, 2 \rangle$. И така, всяка конфигурация е редица от цели положителни числа в ненарастващ ред.

За всяка конфигурация да въведем понятията *печеливша* и *губеща*. Дадена конфигурация, тоест, редица от цели положителни числа, е печеливша тстк А, стартирайки от нея, печели задължително; в противен случай е губеща. Ключовото наблюдение е следното: конфигурация С е печеливша тстк

- или една от $\langle 1 \rangle$ или $\langle 2 \rangle$, при което А печели директно с един ход,
- или има ход на А, от който се получава конфигурация C' , която е губеща за Б; тук А печели индиректно, понеже Б е длъжен да играе.

Губеща конфигурация е, примерно, $\langle 1, 1 \rangle$. Картинно, конфигурацията може да бъде нари-

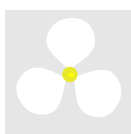
сувана така  или така , това няма значение. Очевидно е, че този, който е на ход, може да направи само едно нещо—да откъсне единственото венчелистче от едната група—след което другият печели директно.

Конфигурациите $\langle 1 \rangle$ и $\langle 2 \rangle$ са базовите конфигурации. Както отбелязахме, те са печеливши. Всяка конфигурация С с общо n венчелистчета се получава

- или от конфигурация C_1 с общо $n + 1$ венчелистчета чрез откъсване на венчелистче,
- или от конфигурация C_2 с общо $n + 2$ венчелистчета чрез откъсване на две съседни венчелистчета.

На практика вече имаме рекурсивна декомпозиция по броя на венчелистчетата. Нека n е броят на венчелистчетата в задачата. Можем да пресметнем рекурсията по схемата **Динамично Програмиране**, като систематично, за $k = 1, 2, \dots, n$ намерим всички конфигурации с k венчелистчета и за всяка от тях, която не е базова, пресметнем дали е печеливша или губеща от конфигурациите с $k - 1$ и $k - 2$ венчелистчета. Всяка конфигурация, за дадена стойност на k , идентифицираме с редица от цели положителни числа, в ненарастващ ред, сумиращи се до k .

Има една особеност. За някои n се налага да различаваме маргаритка с n венчелистчета, от която нищо не е късано, от маргаритка с една единствена група от n венчелистчета, от която обаче е късано. И в двата случая редицата е $\langle n \rangle$, но е възможно едната конфигурация да е печеливша, а другата конфигурация да е губеща. Като пример при $n = 3$:

- тази конфигурация  е губеща, понеже каквото и да откъсне А, след това Б откъсва останалите едно или две венчелистчета и печели,

- а тази конфигурация



е печеливша, защото А трябва само да откъсне средното

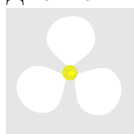
венчелистче, за да “вкара” Б в губещата конфигурация



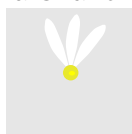
И двете конфигурации се описват от редицата $\langle 3 \rangle$. При редици с една единствена компонента се налага да различаваме някак ситуация с нула “празнини” (маргаритка, от която не е късано) от ситуация с една “празнина” (има късани венчелистчета).

За едно или две венчелистчета няма значение дали има празнина, или няма[†]. За $n = 3$,

както видяхме,

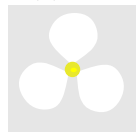


е губеща, а

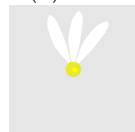


е печеливша. За да различаваме някак $\langle 3 \rangle$

в двата случая, ако конфигурацията е без празнина, ще пишем $\langle \tilde{n} \rangle$, примерно $\langle \tilde{3} \rangle$ описва



, докато при точно една празнина ще пишем $\langle n \rangle$, примерно $\langle 3 \rangle$ описва



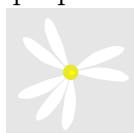
Изключвайки тази особеност, ясно е, че всяка конфигурация на n венчелистчета е едно целочислено разбиване на n (Определение 39), така че можем да мислим в термините на целочислени разбивания. Вместо да говорим за конфигурации от венчелистчета, ще говорим за целочислени разбивания. Можем да изградим триъгълна таблица с информация за разбиванията до каквато стойност на n пожелаем. Таблица 12.6 показва разбиванията до $n = 7$. Тези, които са написани със зелен цвят, са печеливши, а тези с червен цвят са губещи. За по-компактен запис не ползваме никакъв разделител между компонентите на дадено разбиване. Тъй като числата са едноцифрени, двусмислица не може да има.

n	целочислените разбивания на n															
1	$\widetilde{1}$	1														
2	$\widetilde{2}$	2	11													
3	$\widetilde{3}$	3	21	111												
4	$\widetilde{4}$	4	31	22	211	1111										
5	$\widetilde{5}$	5	41	32	311	221	2111	11111								
6	$\widetilde{6}$	6	51	42	33	411	321	222	2211	3111	21111	111111				
7	$\widetilde{7}$	7	61	52	43	511	421	331	322	4111	3211	2221	31111	22111	211111	1111111

Таблица 12.6: Печеливши и губещи целочислени разбивания.

Тъй като при разглеждането на задачата от произволна начална конфигурация започнахме

с примера



, да го разгледаме докрай. Това е 322, което е печеливша конфигурация

[†]Трудно е да си представим цвете, което има точно едно венчелистче по природа, но математически може да разгледаме и такова.

съгласно таблицата. Наистина, от нея с откъсване на едно венчелистче получаваме



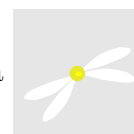
която е **321**. Това е губеща конфигурация, тъй като от нея стигаме до, и само до, следните конфигурации, всяка от които е печеливша. Ето достижимите с едно късане конфигурации:

- , или **32**, е печеливша,
- , или **311**, е печеливша,
- , или **31**, е печеливша,
- , или **221**, е печеливша,
- , или **211**, е печеливша,
- , или **2111**, е печеливша.

Би било прекалено голямо усилие да разгледаме всяка от тях, така че да видим само



или **32**. Тя наистина е печеливша, защото с едно късане става



, или **22**. А **22** на свой ред наистина е губеща, защото от нея може да се отиде единствено в очевидно печелившите **2** и **21**.

Лесно може да се дефинират с математическа прецизност обръщанията към предни редове в таблицата – за всяка редица със сума n , точно тези редици със сума $n - 1$ или $n - 2$, които може да се получат от него. По този начин таблицата може да бъде попълвана систематично.

Забележете връзката между Таблица 12.6 и Таблица 12.4. В някакъв смисъл, Таблица 12.4 е кондензиран вариант на Таблица 12.6: ако игнорираме колоната с тилдите в Таблица 12.6, останалата част на Таблица 12.6 съдържа самите целочислени разбивания, чиито бройки се съдържат в клетките на Таблица 12.4. Примерно, да разгледаме реда с $n = 7$ в Таблица 12.4. Сумата от числата в него е $1 + 3 + 4 + 3 + 2 + 1 + 1 = 15$, и наистина в Таблица 12.6 има петнадесет целочислени разбивания за $n = 7$, експлицитно посочени. От тези петнадесет целочислени разбивания на седмицата, точно три имат по точно две части (това са 61, 52 и

43), а от друга страна в Таблица 12.4, елементът, отговарящ на $n = 7$ и $k = 2$, е 3. И така нататък.

Асимптотично говорейки, този алгоритъм за решаване на задачата с маргаритките не е ефикасен. Броят на целочислените разбивания на n расте *доста бързо*, като асимптотиката е, грубо казано, експонента от $\sqrt{n}$. Но за разумни стойности на n нещата не са толкова черни: за $n = 20, 40, 60$ бройките са съответно 627, 37 338 и 966 467[†]. *Известни са* ефикасни алгоритми за генерирането на самите целочислени разбивания, които вършат работа само $\Theta(1)$ за разбиване, в амортизирания смисъл. За маргаритки, които се срещат в реалния свят, този подход води до успех.

12.9.2 The Coins in a Line Game

Дадени са n монети, разположени една до друга в редица. n е четно число. Монетите са от най-различни деноминации. За целите на задачата, деноминациите са произволни цели положителни числа. Всеки играч, когато е неговият или нейният ред, взема точно една монета или от левия, или от десния край на редицата. Играта продължава до изчерпването на монетите в редицата (което означава, че монетите са разпределени между играчите). След играта всеки задържа монетите, които е събрал(а). Искане се ефикасен алгоритъм, който намира максималната възможна сума за Албена.

Може да мислим за състоянието в даден момент от играта като за редица от цели положителни числа. Тук наредбата е от ключово значение, така че не може да я пренебрегнем и да разглеждаме мултимножеството от числата. Да кажем, че началната наредба е $\langle 1, 2, 100, 2 \rangle$. Съгласно правилата, за Албена са достъпни само единицата вляво и двойката вдясно, така че нейният избор е само между тези две монети. Очевидната алчна стратегия “избери тази от двете достъпни монети, която е не по-малка от другата”, не работи; ако следва тази стратегия, Албена ще вземе двойката вдясно, конфигурацията ще стане $\langle 1, 2, 100 \rangle$ и Борис ще вземе стотицата. Забележете, че при начална конфигурация $\langle 1, 2, 100, 2 \rangle$, Албена може да вземе стотицата, ако първо вземе единицата вдясно, с което Борис получава $\langle 2, 100, 2 \rangle$ и, тъй като е длъжен да играе, взема една от двойките, предоставяйки на Албена или $\langle 100, 2 \rangle$, или $\langle 2, 100 \rangle$, при което тя взема стотицата.

И така, очевидната алчна стратегия не работи. Има и по-изтънчена алчна идея. Да си припомним, че n е четно. Да разбием монетите на тези, които са на четни позиции, и тези които са на нечетни. Условно да ги наречем “червените” и “сините”. Примерно,

1, 3, 6, 3, 1, 3

Играейки първа, Албена лесно може да събере монетите от даден цвят (и само тях). Така че тя може първо да сметне кое множество има не по-малка сума от другото и след това да събере монетите му, принуждавайки Борис да събере останалите. Тази идея гарантира на Албена, че ще вземе не по-малко от Борис, но не ѝ гарантира, че ще вземе максимална сума. В примера: и червеното множество има сума 8, а синьото, 9, така че Албена ще вземе монети на стойност 9, следвайки тази идея. Обаче максималната печалба за Албена всъщност е 10. Да се убедим в това. Албена започва с тройката вдясно, оставяйки на Борис избор от две единици:

1, 3, 6, 3, 1

[†]С други думи, ако продължим Таблица 12.6 надолу, дължината на n -ия ред ще е от порядъка на експонента от $\sqrt{n}$, като, например, за $n = 60$, редът ще има 966 467 колони за самите целочислени разбивания плюс още една за $\widetilde{60}$.

Която и единица да вземе Борис, Албена взема другата единица, след което Борис трябва да избира между две тройки:

3, 6, 3

Която и тройка да вземе Борис, Албена взема шестицата, с което тя взема общо $3 + 1 + 6 = 10$.

Има ефикасен алгоритъм по схемата **Динамично Програмиране**, който гарантира максимална печалба за Албена. Да кажем, че $\text{opt}(i, j)$ е оптималната сума, която Албена може да събере, ако има пред себе си подредицата от i -тата монета включително до j -тата монета включително, за $1 \leq i < j \leq n$. Забележете, че няма смисъл да допускаме $i = j$, защото при четно n , Албена в своя последен ход играе върху редица от две монети. Последното вземане задължително се прави от Борис, докато алгоритъмът, който строим, е за Албена.

Ако Албена може да изчислява тази функция ефикасно, тя разполага и с ефикасно решение, защото ѝ трябва $\text{opt}(1, n)$. Тук разглеждаме само намирането на оптималната печалба, но от това решение лесно може да получим решение, което дава и самите оптимални ходове.

Ще измислим подходяща рекурсия за изчислението на $\text{opt}(i, j)$ чрез някакви $\text{opt}(i', j')$, където $j' - i' < j - i$. С други думи, управляващият параметър е разликата между втория и първия аргумент. Тази функция ще се изчислява чрез запълване на (част от) двумерен масив $\text{opt}[1..n, 1..n]$. Решението е $\text{opt}[1, n]$.

Нека масив $M[1..n]$ съдържа редицата от монети; тоест, $M[i]$ е стойността на i -тата монета отляво, за $1 \leq i \leq n$.

Дъното на рекурсията се реализира чрез

$$\text{opt}[i, i + 1] = \max \{M[i], M[i + 1]\}$$

за $1 \leq i < n$. Идеята е проста: ако Албена играе върху двуелементната редица от монети $M[i], M[i + 1]$, тя взема тази, която е не по-малка от другата, после Борис взема другата и играта приключва.

Същината е изчислението на $\text{opt}[i, j]$ за $j > i + 1$. Албена избира дали да вземе $M[i]$, или да вземе $M[j]$. Други възможности няма.

- Ако Албена вземе $M[i]$, на ход е Борис, който или която избира между $M[i + 1]$ и $M[j]$. Тъй като допускаме, че Борис играе оптимално, той ще направи избора, който му дава максимална печалба, и за Албена ще остане по-лошото (по-малкото) от $\text{opt}[i + 1, j - 1]$ и $\text{opt}[i + 2, j]$. Така че Албена ще вземе най-много, при оптимална игра на Борис,

$$M[i] + \min \{\text{opt}[i + 1, j - 1], \text{opt}[i + 2, j]\}$$

- Ако Албена вземе $M[j]$, на ход е Борис, който избира между $M[i]$ и $M[j - 1]$. Тъй като допускаме, че Борис играе оптимално, той ще направи изборът, който му дава максимална печалба, и за Албена ще остане по-лошото от $\text{opt}[i, j - 2]$ и $\text{opt}[i + 1, j - 1]$. Така че Албена ще вземе най-много, при оптимална игра на Борис,

$$M[j] + \min \{\text{opt}[i, j - 2], \text{opt}[i + 1, j - 1]\}$$

Албена трябва да избере максимума от тези. И така:

$$\text{opt}[i, j] = \max \{M[i] + \min \{\text{opt}[i + 1, j - 1], \text{opt}[i + 2, j]\}, M[j] + \min \{\text{opt}[i, j - 2], \text{opt}[i + 1, j - 1]\}\}$$

Да илюстрираме работата на алгоритъма, изграден върху тази рекурсивна декомпозиция.

Нека входът отново е $\langle 1, 3, 6, 3, 1, 3 \rangle$. Работната матрица е квадратна, като клетките под главния диагонал не се ползват по очевидни причини (те са в черно на илюстрацията), а в останалата част се ползват само клетките $[i, j]$, за които $j - i$ е нечетно; тези, за които $j - i$ е четно (те са в сиво на илюстрацията), отговарят на ходове на Борис, които са нерелевантни за алгоритъма за Албена.

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

Да разгледаме началните условия

$$\text{opt}[i, i + 1] = \max \{M[i], M[i + 1]\}$$

за $1 \leq i \leq 5$. Тогава $\text{opt}[1, 2] = \max \{1, 3\} = 3$, така че имаме:

	1	2	3	4	5	6
1		3				
2						
3						
4						
5						
6						

Аналогично, $\text{opt}[2, 3] = \max \{3, 6\} = 6$, $\text{opt}[3, 4] = \max \{6, 3\} = 6$, $\text{opt}[4, 5] = \max \{3, 1\} = 3$ и $\text{opt}[5, 6] = \max \{1, 3\} = 3$, така че в диагонала над главния имаме:

	1	2	3	4	5	6
1		3				
2			6			
3				6		
4					3	
5						3
6						

С това приключихме с началните условия.

За клетка $[1, 4]$, стойността се получава като

$$\begin{aligned}
 \text{opt}[1, 4] &= \max \{ 1 + \min \{ \text{opt}[2, 3], \text{opt}[3, 4] \}, 3 + \min \{ \text{opt}[1, 2], \text{opt}[2, 3] \} \} \\
 &= \max \{ 1 + \min \{ 6, 6 \}, 3 + \min \{ 3, 6 \} \} \\
 &= \max \{ 1 + 6, 3 + 3 \} \\
 &= 7
 \end{aligned}$$

Наистина, върху вход $\langle 1, 3, 6, 3 \rangle$, Албена може да вземе най-много 7. Аналогично,

$$\begin{aligned}
 \text{opt}[2, 5] &= \max \{ 3 + \min \{ \text{opt}[3, 4], \text{opt}[4, 5] \}, 1 + \min \{ \text{opt}[2, 3], \text{opt}[3, 4] \} \} \\
 &= \max \{ 3 + \min \{ 6, 3 \}, 1 + \min \{ 6, 6 \} \} \\
 &= \max \{ 3 + 3, 1 + 6 \} \\
 &= 7
 \end{aligned}$$

и

$$\begin{aligned}
 \text{opt}[3, 6] &= \max \{ 6 + \min \{ \text{opt}[4, 5], \text{opt}[5, 6] \}, 3 + \min \{ \text{opt}[3, 4], \text{opt}[4, 5] \} \} \\
 &= \max \{ 6 + \min \{ 3, 3 \}, 3 + \min \{ 6, 3 \} \} \\
 &= \max \{ 6 + 3, 3 + 3 \} \\
 &= 9
 \end{aligned}$$

Имаме

	1	2	3	4	5	6
1		3		7		
2			6		7	
3				6		9
4					3	
5						3
6						

Окончателно,

$$\begin{aligned}
 \text{opt}[1, 6] &= \max \{1 + \min \{\text{opt}[2, 5], \text{opt}[3, 6]\}, 3 + \min \{\text{opt}[1, 4], \text{opt}[2, 5]\}\} \\
 &= \max \{1 + \min \{7, 9\}, 3 + \min \{7, 7\}\} \\
 &= \max \{1 + 7, 3 + 7\} \\
 &= 10
 \end{aligned}$$

Таблицата изглежда така:

	1	2	3	4	5	6
1		3		7		10
2			6		7	
3				6		9
4					3	
5						3
6						

Решението е 10.

12.10 Мемоизация

Името на тази секция не е сгрешено. Не става дума за меморизация, което е транслитерация на *memorisation* (по американския правопис е *memorization*) и означава запомняне в разговорния смисъл. *Memoization*, за което авторът на лекционните записки не разполага с по-подходяща българска дума от “мемоизация”, е техника за ефикасна имплементация на

алгоритмични идеи. Понятието е въведено от британския компютърен учен Donald Michie, работил през 60-те години на 20 век в областта на изкуствения интелект. В [статия от 1968 г.](#) той описва метод, с който “компютрите се учат от опита си”. В секцията “Recursively Defined Functions”, Michie описва изчисление на рекурсивни функции, при което резултатите от виканията на дадена функция се съхраняват в таблица. При всяко викане на рекурсивната функция първо се проверява дали желаната стойност вече е била изчислена.

- Ако да, стойността се взема от таблицата.
- Ако не, започва изчисляването на стойността, но при виканията върху по-малки стойности на аргумента отново се проверява дали за тях стойността е била вече изчислена.

Лесно се вижда, че на фундаментално ниво идеята е същата като идеята на схемата **Динамично Програмиране**: подзадачите имат общи подподзадачи, и резултатите от работата върху подподзадачите се съхраняват, за да не бъдат пресмятани отново и отново и отново. Съществена разлика между мемоизацията и схемата **Динамично Програмиране** е посоката на изчислението. Мемоизацията е истинска рекурсия с “ход надолу” и “ход нагоре”, само че резултатите от виканията се пазят. Динамичното програмиране е само втората част от рекурсията, “ходът нагоре”.

Пример с пресмятането на биномни коефициенти. Да разгледаме пресмятане на биномни коефициенти чрез рекурсия съгласно (12.20). Като среда ще ползваме Maple (TM).

Нека изчисляваме $\binom{30}{15}$, който е 155 117 520. Съгласно (12.20), рекурсията първо пресмята $\binom{29}{15}$, после пресмята $\binom{29}{14}$ и после ги сумира. За да се сметне $\binom{29}{15}$ се смятат $\binom{28}{15}$ и $\binom{28}{14}$, а за се сметне $\binom{29}{14}$ се смятат $\binom{28}{14}$ и $\binom{28}{13}$. Проблемът—аналогично на Секция 12.1.2—е в това, че при връщането на стойността на $\binom{29}{15}$ изчислението е “забравило”, че вече е разполагало със стойността на $\binom{28}{14}$, и започва да я смята наново, за да получи $\binom{29}{14}$, и това явление се появява **навсякъде в дървото на рекурсията**.

Може да избегнем този проблем като пълним таблица, подобна на Таблица 12.1, но може да използваме подхода на Michie. При него изчислението е истинска рекурсия, но всеки биомнен коефициент, веднъж получен, се съхранява. Рекурсивната функция е имплементирана така, че когато бъде извикана с n и k , които искат рекурсивни викания, първо се извършва проверка дали вече разполагаме с желаната стойност. Само ако не разполагаме с тази стойност се извършват рекурсивните викания.

За да бъде имплементирана тази идея лесно на практика, трябва програмната среда да осигурява тези проверки **прозрачно** за нас. В противен случай—примерно, ако пишем на C—трябва ние да се погрижим за създаването на таблицата и проверяването в нея. Едно средство за програмиране, което поддържа мемоизация, е Maple(TM). Рекурсия с мемоизация в Maple(TM) се постига изключително лесно: дава се опция `remember` в дефиницията на рекурсивната процедура и след това Maple(TM) се грижи за таблицата и проверяването в нея. Следният пример с $\binom{30}{15}$ показва драстично повишаване на ефикасността на изчислениято с мемоизация в Maple.

Стандартният начин за измерване на чистото време за изчисление в Maple (TM) не е с гледане на часовника, а с използване на функцията `profile` с аргумент-името на функцията или процедурата, която искаме да профилираме. После с `showprofile` се извежда подробна информация за изразходваното изчислително време и памет. Но в примера с биомните коефициенти, това профилиране дава чисто изчислително време, което е в пъти над времето, което Maple използва, за да изчислява без профилиране (става дума за рекурсия без мемоизация; тази с мемоизация ползва, грубо казано, нула време за тези стойности на входа). За да измерим времето без ползване на `profile`, ползваме трика с функцията `time`.

```

Z:\algs> "c:\Program Files\Maple 2018\bin.X86_64_WINDOWS\cmaple.exe"
| \~ / |      Maple 2018 (X86 64 WINDOWS)
._ | \ | _   | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  /   All rights reserved. Maple is a trademark of
 < ____ ____ >   Waterloo Maple Inc.
      |           Type ? for help.
> kernelopts(printbytes=false);
> mybinomial1 := proc(n::nonnegint, k::nonnegint)
> if k > n then 0
> else if k = 0 or k = n then 1
> else mybinomial1(n-1, k) + mybinomial1(n-1, k-1)
> fi; fi; end proc;
> mybinomial2 := proc(n::nonnegint, k::nonnegint) option remember;
> if k > n then 0
> else if k = 0 or k = n then 1
> else mybinomial2(n-1, k) + mybinomial2(n-1, k-1)
> fi; fi; end proc;
> mytime := time();
> mybinomial1(30,15);

155117520

> time() - mytime;

315.453

> mytime := time();
> mybinomial2(30,15);

155117520

> time() - mytime;

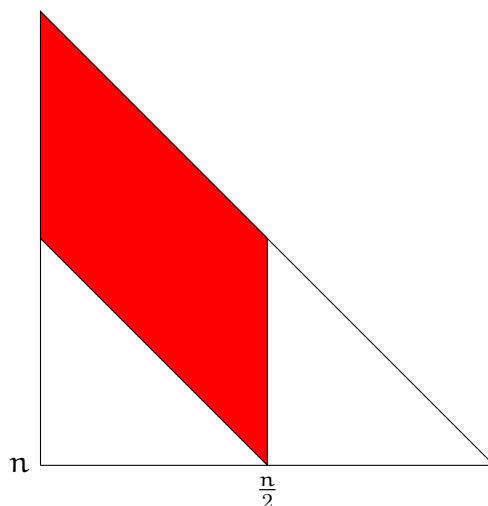
0.

```

И така, `mybinomial1(30,15)` приключва работата си за 315.453 секунди, а `mybinomial2(30,15)` приключва работата си толкова бързо, че този начин на отчитане отчита нула секунди. Единствената разлика между тях е, че `mybinomial2` ползва опцията `remember`.

С мемоизацията трябва да се внимава. Ако изчислението просто трупа данни в таблици, които са огромни, в някакъв момент може да започне да не достига памет. Сякаш този потенциален проблем е по-голям при мемоизацията в сравнение с динамичното програмиране, защото при динамичното разработчикът манипулира таблиците явно и е напълно наясно за тяхното съществуване и възможни последици от не-освобождаването на паметта, използвана за тях. От друга страна, това, че мемоизацията става прозрачно за разработчика, може да го/я изкуши да забрави за нея.

Едно потенциално предимство на мемоизацията пред динамичното програмиране е, че за някои изчисления не ни трябват всички стойности за по-малки аргументи, а само някои. Образно, ако смятаме $\binom{n}{n/2}$, изчислението с мемоизация запълва само тази част от таблицата, която е оцветена в червено:



Докато динамичното програмиране би “напълнило” цялата таблица до ред n включително.

12.11 За границите на възможностите на динамичното програмиране

Алгоритмичната схема **Динамично Програмиране** е чудесно средство за алгоритмичен дизайн. Вече видяхме немалко примери за задачи, които на пръв поглед са невъзможно трудни, но умелото прилагане на схемата **Динамично Програмиране** дава ефикасни алгоритми, които са и лесни за разбиране, ако човек разбере рекурсията. За новаци в областта на алгоритмичния дизайн конструирането на такива алгоритми е нещо като магия.

За съжаление или за щастие, тази магия работи само донякъде. Тя има своите ограничения. Тук ще разгледаме един пример от книгата “The Algorithm Design Manual” на Skiena (вж. [144, стр. 303–304]) за алгоритъм по схемата **Динамично Програмиране**, който има експоненциална сложност. В което няма нищо странно, понеже задачата е **NP**-трудна; странно би било да се намери ефикасен алгоритъм за нея. Изводът е, че алгоритмичната неподатливост отказва да изчезне при магическите думи “динамично програмиране”. “Динамично програмиране” не е непременно синоним на “ефикасно решение”!

Задачата е **НАЙ-ДЪЛЪГ ОРИЕНТИРАН ПЪТ**. Става дума за тегловни графи. Тегло на път е, както винаги, сумата от теглата на ребрата.

Изч. Задача 57: LONGEST DIRECTED PATH

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w , начален връх s , краен връх t .

решение: Дължина на най-дълъг прост път от s до t в G .

Да разгледаме следната рекурсия. Нека $L(i, j)$ означава дължината на най-дълъг прост път от i до j . Тъй като говорим за прости пътища, има смисъл да искаме $i \neq j$. Всеки такъв най-дълъг път има някакъв предпоследен връх x , поради което можем да се изкушим да кажем:

$$L(i, j) = \max_{(x, j) \in E} \{L(i, x) + w(x, j)\}$$

Тази рекурсия обаче има два проблема, всеки от които я прави безполезна.

- Първо, нищо в нея не пречи да бъдат повтаряни върхове, тоест, пътят, чиято дължина се има предвид, да не е прост.
- Второ, не е ясно в какъв ред върви изчислението, поради което практическата ѝ имплементация може да зацikli. Иначе казано, в тази рекурсивна декомпозиция състоянията са наредени двойки от върхове, но не е ясно кои са големите и кои са малките състояния, така че да изчисляваме функцията върху големите състояния от функцията върху малките състояния.

Коректна рекурсия е следната. Нека $LP(i, j, P)$ означава дължината на най-дълъг път от i до j , в който P е редицата от междинните върхове. Щом пътят е прост, P не трябва да съдържа нито i , нито j , нито в P може да има повтаряне на върхове.

$$LP(i, j, P \oplus x) = \max_{(x, j) \in E, j \notin P} \{L(i, x, P) + w(x, j)\}$$

“ $P \oplus x$ ” означава редицата P с “долепен” x в края. $L(i, x, P)$ вдясно изключва възможността $x \in P$, така че е достатъчно да сме сигурни, че $j \notin P$, което присъства като ограничение в максимума.

Тази рекурсия е очевидно коректна, но нейната алгоритмична имплементация би била твърде неефекасна. От общи съображения може да кажем, че има не повече от $(n - 3)!$ възможности за $P^\dagger$, поради което таблицата на динамичното програмиране би била непрактично огромна дори за n от порядъка на няколко десетки.

Малко подобрение може да се получи, ако не помним пермутации, а множества. Всъщност, P се използва само за да се избегне повтаряне на върхове. Ако въведем друга рекурсия $\tilde{L}(i, j, S)$ със смисъл на дължина на най-къс път от i до j , в който S е множеството от междинните върхове, може да кажем:

$$\tilde{L}(i, j, S \cup \{x\}) = \max_{(x, j) \in E, j \notin S} \{\tilde{L}(i, x, S) + w(x, j)\}$$

Пространството от състоянията става от порядъка на 2^n , което е известно подобрение спрямо факториела. Въпреки това, алгоритъмът, съответен на рекурсията, продължава да е неефекасен.

Наблюдение 80

Схемата **Динамично Програмиране** не е всемогъща. Тя има своите естествени ограничения. За задачи от класа **NP-hard**, конструирането на алгоритъм по тази схема не ни дава ефикасно решение, понеже пространството от състоянията е необозримо грамадно.

[†] P не може да има повторения и не съдържа нито i , нито j , нито x .

Част V

Долни граници

Лекция 13

Долни граници върху сложност на задачи.

Резюме: Въвеждаме понятието долна граница върху сложност на задача. Доказваме долни граници върху броя на сравненията в две занимателни задачи. Въвеждаме понятието дърво за вземане на решение. Използвайки този модел, доказваме долна граница $\Omega(n \lg n)$ върху сложностите на СОРТИРАНЕ и УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ, при определени допускания. Въвеждаме редукция на задача P_1 до задача P_2 , по този начин показвайки, че, ако P_1 е трудна, то и P_2 е трудна. С редукция на УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ до НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, МОДА и 2SUM доказваме долна граница $\Omega(n \lg n)$ върху сложностите на тези задачи. Въвеждаме понятието относителна долна граница и доказваме, че за DEGENERACY TESTING е в сила всяка долна граница за 3SUM; дефинираме “hardness” в смисъл, че DEGENERACY TESTING е 3SUM-hard. Въвеждаме аргументиране чрез противник и доказваме няколко долни граници чрез такова аргументиране.

13.1 Въведение

Определение 103: Долна и горна граница на сложността на задача

Нека P е произволна изчислителна задача. *Долна граница върху сложността на P* е всяка функция $\phi(n)$, такава че всеки алгоритъм, който решава P , работи във време $\Omega(\phi(n))$. *Горна граница върху сложността на P* е всяка функция $\psi(n)$, такава че поне един алгоритъм, който решава P , работи във време $O(\psi(n))$.

Забележете разликата между долна и горна граница.

- За да покажем, че дадена функция е горна граница за дадена задача, достатъчно е да покажем един алгоритъм, чиято сложност по време има такава горна граница. Например, от това, че INSERTION SORT работи във време $O(n^2)$ и това, че INSERTION SORT решава задачата СОРТИРАНЕ, следва, че n^2 е асимптотична горна граница за СОРТИРАНЕ.
- От друга страна, за да покажем, че дадена функция е долна граница за дадена задача, не е достатъчно да покажем един алгоритъм, чиято сложност по време има такава долна граница. Например, от това, че MERGESORT работи във време $\Omega(n \lg n)$ и това, че MERGESORT решава задачата СОРТИРАНЕ, по никакъв начин не следва, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ. Доказателство за долна граница е аргумент за всички

алгоритми—а те са безброй много—които решават задачата. Всеки известен алгоритъм за СОРТИРАНЕ, който е базиран на сравнения (Определение 105), работи във време $\Omega(n \lg n)$, но **от това не следва**, че няма по-бърз алгоритъм, работещ във време, примерно, $\Theta(n \lg \lg n)$ или дори $\Theta(n)$. Както ще видим, такъв наистина няма, но това трябва да се докаже.

Доказателство за долна граница на задача обезсмисля опитите да бъдат конструирани по-бързи алгоритми. Долната граница е **фундаментално ограничение**, нещо като природна даденост, с която сме длъжни да се съобразяваме, независимо дали ни харесва или не. Има далечна аналогия с фундаменталните ограничения от физиката, например *невъзможността за пътуване със скорост, по-голяма от скоростта на светлината във вакуум* (“Consequently, the speed of light is a natural absolute speed limit.”), *принципът на неопределеността от квантовата механика* или *първият и втори принцип на термодинамиката*.

Доказателствата на нетривиални долни граници върху сложността на задачи са трудни. Какво е тривиална долна граница, ще обясним с пример. За СОРТИРАНЕ, тривиална долна граница е $\Omega(n)$, защото няма как да сортираме масив, без да сме разгледали всеки елемент[†]. Доказателството, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ е значително по-трудно и затова казваме, че не е тривиално.

Допълнение 67: Долни граници, задачи и изчислителни модели

Ще разгледаме една тривиална долна граница: $\Omega(n^2)$ за сортирането чрез *транспозиции*. Транспозиция е размяна на местата на два съседни елемента от масива. В нашия псевдокод това се записва като “swap(A[i], A[i + 1])”. Сортиране чрез транспозиции е сортиране, при което единствените достъпи до масива за преместване на елементи са транспозиции. Пример за алгоритъм, който сортира чрез транспозиции, е BUBBLESORT [34, стр. 40].

Очевидно е, че в процеса на работата на алгоритъма, всяка извършена транспозиция може да намали броя на инверсиите (Определение 63) с най-много единица. И тъй като инверсиите може да са $\frac{n(n-1)}{2}$ (при обратно сортиран масив от два по два различни елемента), то само “ликвидирането” на инверсиите налага квадратичен брой транспозиции. Масивът не може да е сортиран, ако в него има инверсии (Наблюдение 46), и от това веднага следва, че всеки алгоритъм, сортиращ само чрез транспозиции, има долна граница за сложността $\Omega(n^2)$.

Тази долна граница обаче не е върху самата изчислителна задача СОРТИРАНЕ, а върху СОРТИРАНЕ с някакво допълнително ограничение: да се ползват само транспозиции. Ограничението не е част от дефиницията на задачата. Да си припомним Разяснение 3: задачата е релация, изобразяваща екземпляри в решения, а алгоритъм за нея е някаква приемливо детайлна реализация на тази релация. Ограничението да се ползват само транспозиции е ограничение не върху задачата, а върху алгоритмите за нея.

Ключовото понятие тук е “изчислителен модел”, което (бегло) споменахме в Допълнение 4. Изчислителният модел определя какво можем да правим. Очевидно, за да ограничим сортирането до такова, което ползва само транспозиции, трябва да наложим ограничения върху нашия изчислителен модел. Изследването на долни граници винаги е в контекста на даден изчислителен модел. В тези лекционни записки изчис-

[†]Аналогично, както ще видим в следваща лекция, тривиална долна граница за обхождане на граф, представен със списъци на съседства, е $\Omega(n + m)$, където n е броят на върховете и m е броят на ребрата, поради това, че размерът на списъците на съседства е $\Theta(n + m)$, и няма как да обходим графа, без да сме разгледали всеки елемент от тези списъци.

лителни модели не се разглеждат и само се споменават в допълнения, но внимателния читател ще забележи, че изследването на долни граници иска първо да е дефиниран изчислителен модел.

За ограничените цели на лекционните записки можем да избегнем разглеждането на изчислителни модели при долните граници, като кажем, че долната граница не е само върху дадена изчислителна задача, а върху задачата и някакъв клас алгоритми за нея; например, СОРТИРАНЕ и алгоритмите, ползващи само транспозиции.

И още една забележка. Това, че $\Omega(n)$ е долна граница за СОРТИРАНЕ и това, че $\Omega(n \lg n)$ е долна граница за СОРТИРАНЕ (което все още не сме доказали), не са в противоречие. Просто втората долна граница е по-висока, следователно е и по-добра (и по-интересна). Когато става дума за долни граници, интересуваме се от колкото е възможно по-високи такива. Дуално, при горните граници искаме колкото е възможно по-ниски такива: $O(n^9)$ е валидна горна граница за СОРТИРАНЕ, но тя е безинтересна предвид факта, че $O(n^2)$ също е горна граница, а на свой ред тя е безинтересна предвид това, че $O(n \lg n)$ е горна граница.

Доказателствата за долни граници, които ще разгледаме, се класифицират така.

- Елементарни доказателства. Примерно, СОРТИРАНЕ има тривиална долна граница $\Omega(n)$, защото всеки елемент трябва да бъде разгледан поне веднъж. Също така, ТЪРСЕНЕ В НЕСОРТИРАН МАСИВ има тривиална долна граница $\Omega(n)$ – по същата причина.
- Доказателства, използващи дървета за вземане на решение (на английски, *decision trees*). Примери за такива доказателства има в тази лекция: долните граници 2 за THE BALANCE PUZZLE, 3 за THE TWELVE-COIN PUZZLE, $\Omega(n \lg n)$ за СОРТИРАНЕ и $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ използват дървета за вземане на решение.
- Доказателства, използващи редукции. Примерно, долната граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ. Редукцията там е, УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ се редуцира до НАЙ-БЛИЗКИ ЕЛЕМЕНТИ.
- Аргументиране чрез противник (на английски, *adversary argument*).

13.2 Дървета за вземане на решение

13.2.1 Задачите THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE

Сега ще разгледаме две занимателни задачи, решението на които ще ни даде необходимата интуиция за ключовото понятие *дърво за вземане на решение*[†].

[†]Досадното повтаряне на “решение” в това изречение се дължи на факта, че на български превеждаме и *solution*, и *decision* като *решение*; в “решението на които” имаме предвид “*solution*”, а в “дърво за вземане на решение” имаме предвид “*decision*”.

Задача 64: THE BALANCE PUZZLE

Дадени са 9 номерирани предмета, да речем топки. Осем от тях имат едно и също тегло, а една топка е по-тежка (от коя да е от осемте). Нашата цел е да идентифицираме тежката топка, като използваме везни. Везните нямат стандартни теглилки, така че можем правим единствено измервания от вида: някои топки на едното блюдо срещу други топки на другото блюдо, и да наблюдаваме резултата. Има точно три възможности за резултата: везната да се наклони наляво или везната да се наклони надясно или везната да остане балансирана. Трябва да намерим тежката топка с колкото е възможно по-малко измервания.

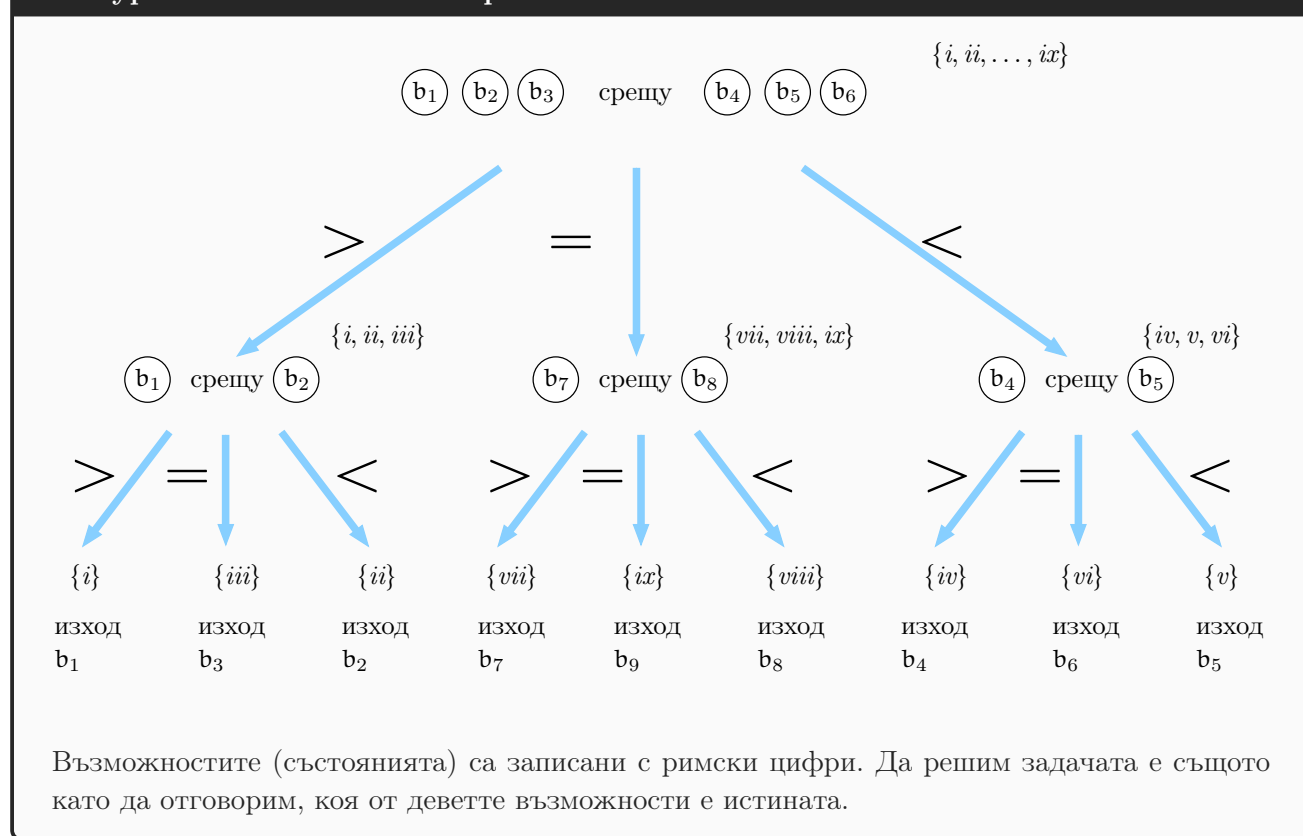
Предложете схема за измервания, която ползва не повече от две измервания. Докажете, че две е долна граница за броя на измерванията.

Решение: Нека деветте топки са $b_1, \dots, b_9$. С едно използване на везната мерим b_1, b_2 и b_3 срещу b_4, b_5 и b_6 . Има точно три възможни изхода.

- Ако b_1, b_2 и b_3 са по-тежки от b_4, b_5 и b_6 , използваме везната втори път с b_1 срещу b_2 . Отново има точно три възможни изхода.
 - ♦ Ако b_1 е по-тежка от b_2 , връщаме “ b_1 е тежката топка”.
 - ♦ Ако b_2 е по-тежка от b_1 , връщаме “ b_2 е тежката топка”.
 - ♦ Ако b_1 и b_2 са еднакво тежки, връщаме “ b_3 е тежката топка”.
- Ако b_4, b_5 и b_6 са по-тежки от b_1, b_2 и b_3 , използваме везната втори път с b_4 срещу b_5 . Отново има точно три възможни изхода.
 - ♦ Ако b_4 е по-тежка от b_5 , връщаме “ b_4 е тежката топка”.
 - ♦ Ако b_5 е по-тежка от b_4 , връщаме “ b_5 е тежката топка”.
 - ♦ Ако b_4 и b_5 са еднакво тежки, връщаме “ b_6 е тежката топка”.
- Ако b_1, b_2 и b_3 , от една страна, и b_4, b_5 и b_6 , от друга страна, са еднакво тежки, използваме везната втори път с b_7 срещу b_8 . Отново има точно три възможни изхода.
 - ♦ Ако b_7 е по-тежка от b_8 , връщаме “ b_7 е тежката топка”.
 - ♦ Ако b_8 е по-тежка от b_7 , връщаме “ b_8 е тежката топка”.
 - ♦ Ако b_7 и b_8 са еднакво тежки, връщаме “ b_9 е тежката топка”.

Фигура 13.1 илюстрира тази схема.

Фигура 13.1 : Схема за измервания за THE BALANCE PUZZLE.



Много полезно е да мислим в термините на възможности, или възможни състояния. По-начало има точно девет възможности: или b_1 е тежката топка, или b_2 е тежката топка, и така нататък, или b_9 е тежката топка. Нека запишем тези девет състояния със, съответно, $i, ii, \dots, ix$. В началото множеството от състоянията е $\{i, ii, \dots, ix\}$. След всяко измерване, множеството от възможностите намалява. Задачата бива решена когато множеството от възможностите стане едноелементно. На Фигура 13.1 до всеки връх е записано множеството от възможностите, които са консистентни с резултатите от измерванията досега, изключвайки текущото измерване. Тоест, консистентни с резултатите от всички върхове от корена до родителя на настоящия връх.

Доказателството за долна граница е следното. Схемата трябва да е такава, че да различава една възможност от общо девет, използвайки тернарно[†] дърво за вземане на решение. Тернарно дърво с височина 1 има най-много три листа, така че не може да различава повече от три възможности. Това не ни върши работа за тази задача. Очевидно дървото трябва да има височина поне 2, за да има поне девет листа, с които да може да различава девет възможности.

Наблюдение 81

Броят на листата на дървото в схемата за измерване е горна граница за броят на състоянията (възможностите), които съответната схема за измерване може да различава.

Забележете, че аргументацията за долната граница няма нищо общо с конкретната схема, която изложихме горе. Това, че изведохме долна граница две за броя на измерванията съвсем не означава автоматично, че тази долна граница е достижима. В случая, тя е достижима,

[†]“Тернарно” кореново дърво е такова, в което всеки връх има най-много три деца.

☐

Задача 65: THE TWELVE-COIN PUZZLE

Дадени са 12 номерирани монети. Измежду тях, 11 имат едно и също тегло, а другата — да я наречем *странната монета* — е или по-лека, или по-тежка. Нашата задача е да идентифицираме странната монета, както и да определим дали е лека или тежка, използвайки везни като тези в Задача 64. Монетите са различни на външен вид, примерно са номерирани $c_1, \dots, c_{12}$.

Предложете схема за измервания, с която идентифицираме странната монета и нейната странност с не повече от три измервания. Докажете, че три е долна граница за броя на измерванията.

Решение: Тази задача е по-сложна от предишната. Броят на възможностите сега е 24: два пъти броят на монетите. Нека i' означава “ c_1 е тежка”, i'' означава “ c_1 е лека”, ii' означава “ c_2 е тежка”, ii'' означава “ c_2 е лека”, и така нататък, xii' означава “ c_{12} е тежка” и xii'' означава “ c_{12} е лека”. Фигура 13.2 показва схема от измервания с най-много три използвания на везните, която определя коя от 24-те възможности е истината.

Фигура 13.2 : Схема от измервания за THE TWELVE-COIN PUZZLE.

Дотук доказахме, че три измервания са достатъчни.

Сега ще докажем, че три измервания са необходими. Тъй като искаме да различим една възможност от общо 24 възможности и всяко измерване има най-много три възможни изхода, долна граница за броя на измерванията е $\lceil \log_3 24 \rceil = 3$. Защо? Защото тернарно дърво с ≥ 24 листа има височина поне три.

Забележете, че долната граница, която следва от такива съображения (височината на дървото като логаритъм от броя на листата) не винаги е достижима. В случая с THE TWELVE-COIN PUZZLE, долната граница наистина е достижима, както виждаме от схемата на Фигура 13.2. Но същата задача с 13 монети **не може** да бъде решена с най-много три измервания в най-лошия случай. Ето защо:

- Няма смисъл да мерим едно срещу друго две множества от монети с различен брой монети, защото, ако блюдото с повечето монети се наклони надолу, ние не придобиваме **никаква нова информация**, в смисъл, че възможностите не намаляват, а остават същите.
- Имайки предвид предното съображение, за всеки избор на две подмножества с една и съща кардиналност (от множеството от 13-те монети), или тези множества имат кардиналност ≥ 5 , или множеството от останалите монети има кардиналност ≥ 5 .
- Множество с кардиналност ≥ 5 има ≥ 10 възможности за странната монета.
- Няма начин да определим една възможност измежду ≥ 10 възможности, използвайки ≤ 2 измервания, защото при не повече от три изхода от измерване, две последователни измервания могат да различат най-много 9 възможности.

Покажахме долна граница четири за броя на измерванията. От друга страна, $\lceil \log_3 26 \rceil = 3$. Няма противоречие между факта, че $\lceil \log_3 26 \rceil = 3$, и факта, че THE THIRTEEN-COIN PUZZLE изисква поне четири измервания в най-лошия случай. Това са различни аргументации за долна граница, като едната аргументация дава долна граница три, а другата, четири. Три си остава валидна долна граница, просто при THE THIRTEEN-COIN PUZZLE долната граница три не е достижима; иначе казано, долната граница три не е точна. $\square$

13.2.2 Долна граница $\Omega(n \lg n)$ за СОРТИРАНЕ

13.2.2.1 Сортиране, базирано на сравнения

Долната граница $\Omega(n \lg n)$, която ще докажем, не е в сила за всеки сортиращ алгоритъм, а **само за сортиращите алгоритми, базирани на сравнения**. В [35] това понятие е обяснено по следния начин.

Определение 104: Comparison sort [35, стр. 191]

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \dots, a_n \rangle$. That is, given two elements a_i and a_j , we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

Както ще се убедим от изложението в Подсекция 13.2.2, Определение 104 е ненужно ограничаващо. Ако променим, примерно, INSERTION SORT, слагайки в самото начало код, който преброява елементите от всеки вид (това може да се направи тривиално в $O(n^2)$ време и $O(n)$

памет), и оставяйки същинското сортиране същото, то това ще остане сортиращ алгоритъм, базиран на сравнения, въпреки че сега “инспектира” стойностите на елементите. Съществено е не дали алгоритъмът инспектира стойности, а дали пресмята изходната сортирана редица само въз основа на сравнения, или не. Поради това в тези записки приемаме друго.

Определение 105: Сортиращ алгоритъм, базиран на сравнения

Сортиращ алгоритъм, базиран на сравнения, е всеки сортиращ алгоритъм, който извършва сравнения от вида $a_i \stackrel{?}{<} a_j$ върху входните елементи и изходът се определя еднозначно от извършените сравнения и резултатите от тях.

Образно казано, всеки алгоритъм за сортиране, базиран на сравнения, използва везна, подобна на везните от Подсекция 13.2.1, само че с два, а не три възможни изхода.

Сортиращите алгоритми от Лекция 4, Лекция 5 и Лекция 6 са базирани на сравнения. COUNTING SORT и RADIX SORT от Лекция 7 не са базирани на сравнения.

13.2.2.2 Дървото на сравненията на INSERTION SORT за $n = 3$

Да си припомним INSERTION SORT.

```

INSERTION SORT(A[1 .. n])
1  for i ← 2 to n
2      key ← A[i]
3      j ← i - 1
4      while j > 0 and key < A[j] do
5          A[j + 1] ← A[j]
6          j ← j - 1
7      A[j + 1] ← key

```

Да разгледаме работата на INSERTION SORT върху вход с големина три. Нека входът е

$$a_1, a_2, a_3$$

С малки букви, примерно “ a_1 ”, означаваме елементите от входа. Те са неизменни, в смисъл че който и да е от тях, да кажем a_1 , е едно и също нещо във всеки момент от работата на алгоритъма. С главни букви, примерно “ $A[1]$ ”, означаваме елементите от текущия масив A . Те може да се менят по време на работата! В началото е вярно, че $A = [a_1, a_2, a_3]$, но след няколко стъпки от изпълнението може да имаме $A = [a_2, a_3, a_1]$.

Да допуснем, че елементите от входа са **уникални**. Точно едно от следните е в сила:

$$a_1 < a_2 < a_3 \tag{13.1}$$

$$a_1 < a_3 < a_2 \tag{13.2}$$

$$a_2 < a_1 < a_3 \tag{13.3}$$

$$a_2 < a_3 < a_1 \tag{13.4}$$

$$a_3 < a_1 < a_2 \tag{13.5}$$

$$a_3 < a_2 < a_1 \tag{13.6}$$

Ще наричаме (13.1), ..., (13.6), *пермутациите*. Говорейки стриктно, това е злоупотреба с терминологията, защото всяко от тези е двойка от неравенства (още по-прецизно, конюнкция

от две съждения), а не пермутация, но е ясно какво се има предвид.

Ключовото наблюдение, без което няма истинско разбиране на понятието “дърво за вземане на решение за базирано на сравнения сортиране”, е следното.

Наблюдение 82

Да сортираме входа със сравнения в някакъв смисъл е същото като да определим, само в резултат на изходите от сравненията, коя от пермутациите е истинната.

Сравненията стават само на ред 4 в $\text{key} \stackrel{?}{<} A[j]$. Първото сравнение е $a_2 \stackrel{?}{<} a_1$. В този момент $i = 2$, $j = 1$, а $\text{key} = a_2$.

Случай I Ако е вярно, че $a_2 < a_1$, то точно тези три пермутации (от началните шест) са възможни:

$$a_2 < a_1 < a_3$$

$$a_2 < a_3 < a_1$$

$$a_3 < a_2 < a_1$$

Тялото на **while**-а се изпълнява и A става $[a_1, a_1, a_3]$. После j става 0, изпълнението излиза от **while**-а, $A[1]$ става a_2 , така че A става $[a_2, a_1, a_3]$. Следващото сравнение е $a_3 \stackrel{?}{<} a_1$. В този момент $i = 3$, $j = 2$, а $\text{key} = a_3$.

Случай I.1 Ако е вярно, че $a_3 < a_1$, то точно тези две пермутации остават възможни:

$$a_2 < a_3 < a_1$$

$$a_3 < a_2 < a_1$$

Тялото на **while**-а се изпълнява и A става $[a_2, a_1, a_1]$. j става 1. Следващото сравнение е $a_3 \stackrel{?}{<} a_2$.

Случай I.1.a Ако е вярно, че $a_3 < a_2$, то остава една единствена възможна пермутация:

$$a_3 < a_2 < a_1$$

Тялото на **while**-а се изпълнява и A става $[a_2, a_2, a_1]$. После j става 0, изпълнението излиза от **while**-а, $A[1]$ става a_3 , така че A става $[a_3, a_2, a_1]$. След това i става 4 и алгоритъмът терминира.

Случай I.1.b Ако не е вярно, че $a_3 < a_2$, то остава една единствена възможна пермутация:

$$a_2 < a_3 < a_1$$

while-цикълът не се изпълнява повече. Тъй като $j = 1$, $A[2]$ става a_3 , така че A става $[a_2, a_3, a_1]$. След това i става 4 и алгоритъмът терминира.

Случай I.2 Ако не е вярно, че $a_3 < a_1$, остава една единствена възможна пермутация:

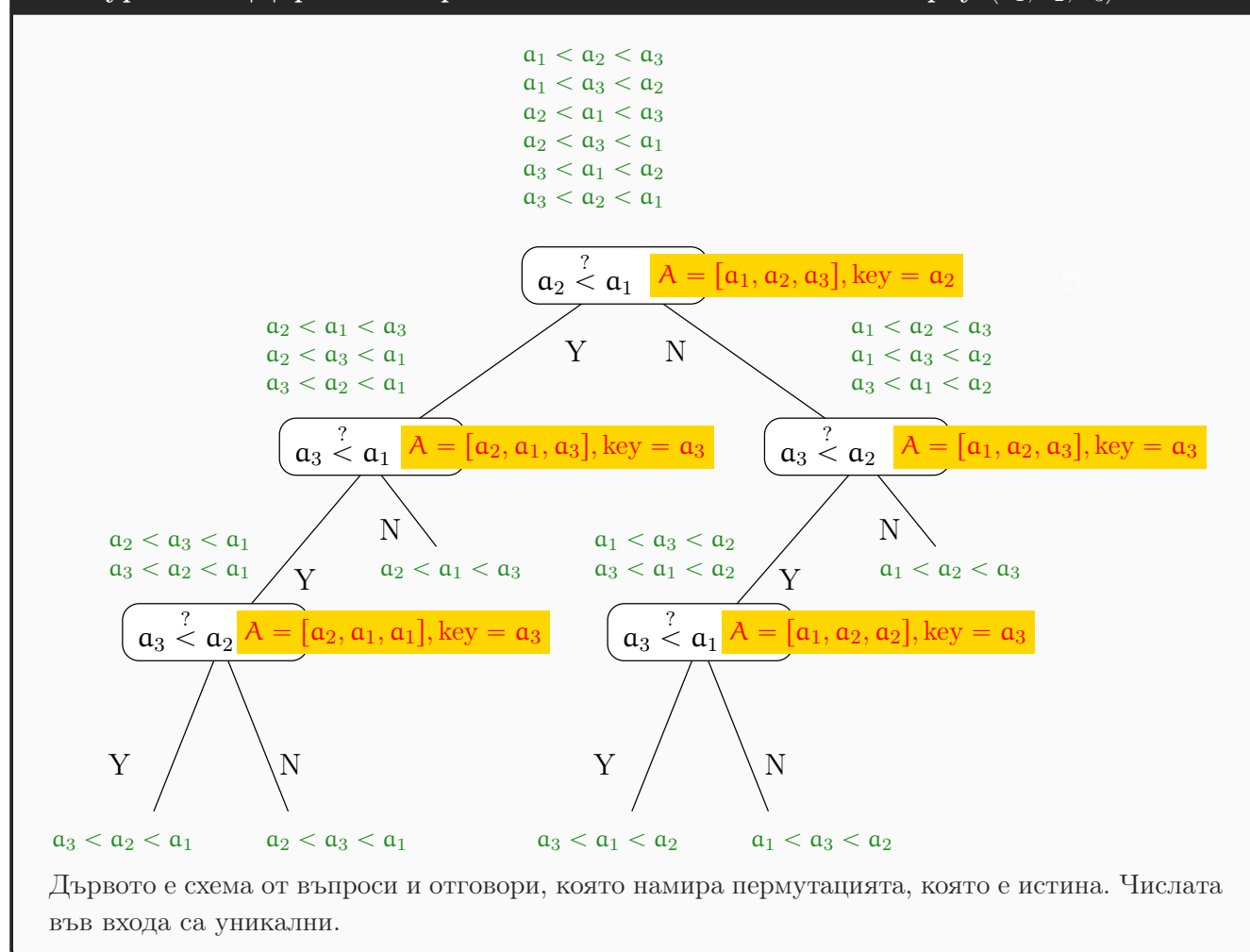
$$a_2 < a_1 < a_3$$

Да си припомним, че при сравнението $a_3 \stackrel{?}{<} a_1$ имаме $i = 3$, $j = 2$, $\text{key} = a_3$ и $A = [a_2, a_1, a_3]$. **while**-ът не се изпълнява повече. $A[3]$ става a_3 , тоест, не се променя, така че A остава

$[a_2, a_1, a_3]$. i става 4 и алгоритъмът терминира. Това е краят на анализа на **Случай I**.

Можем да направим аналогичен анализ и на **Случай II** с неговите подслучаи и подподслучаи, но това удоволствие остава за читателя. Пълният анализ се илюстрира от дървовидната структура, показана на Фигура 13.3. Ключовото нещо са пермутациите, съвместими с досега получените отговори. В началото всички шест пермутации са възможни и затова коренът е асоцииран с множеството от всички тях, но всеки зададен въпрос разбива множеството на асоциираните пермутации на тези, които са съвместими с положителния отговор, и тези, които са съвместими с отрицателния отговор. Всяко листо е асоциирано със само една пермутация. Листата на дървото отговарят точно на различните начини да терминира алгоритъмът; с други думи, на различните състояния на сортирания масив, изразени чрез пермутация на a_1, a_2 и a_3 . За удобство на читателя, във вътрешните върхове на дървото—тези с въпросите—са записани с **червени букви на жълт фон** масивът A и съдържанието на key в момента на задаване на въпроса.

Фигура 13.3 : Дървото на сравненията на INSERTION SORT върху $\langle a_1, a_2, a_3 \rangle$.



Допълнение 68: Защо допуснахме, че a_1, a_2 и a_3 са уникални?

Дървото от Фигура 13.3 е конструирано при допускането, че няма повторения на числа. Какво ще стане, ако допуснем повторения на числа? Ще се промени ли дървото? Със сигурност INSERTION SORT работи коректно и когато входните елементи не са уни-

кални, но как се отразява на дървото възможността да има повторения? Отговорът е, че дървото остава същото, но сега върховете, включително и листата, са асоциирани с повече състояния.

Ако се допуска повторения на стойности във входа, възможностите (състоянията) са значително повече. При $n = 3$ те са 13 на брой:

$$a_1 < a_2 < a_3$$

$$a_1 < a_3 < a_2$$

$$a_2 < a_1 < a_3$$

$$a_2 < a_3 < a_1$$

$$a_3 < a_1 < a_2$$

$$a_3 < a_2 < a_1$$

$$a_1 = a_2 < a_3$$

$$a_3 < a_1 = a_2$$

$$a_1 = a_3 < a_2$$

$$a_2 < a_1 = a_3$$

$$a_2 = a_3 < a_1$$

$$a_1 < a_2 = a_3$$

$$a_1 = a_2 = a_3$$

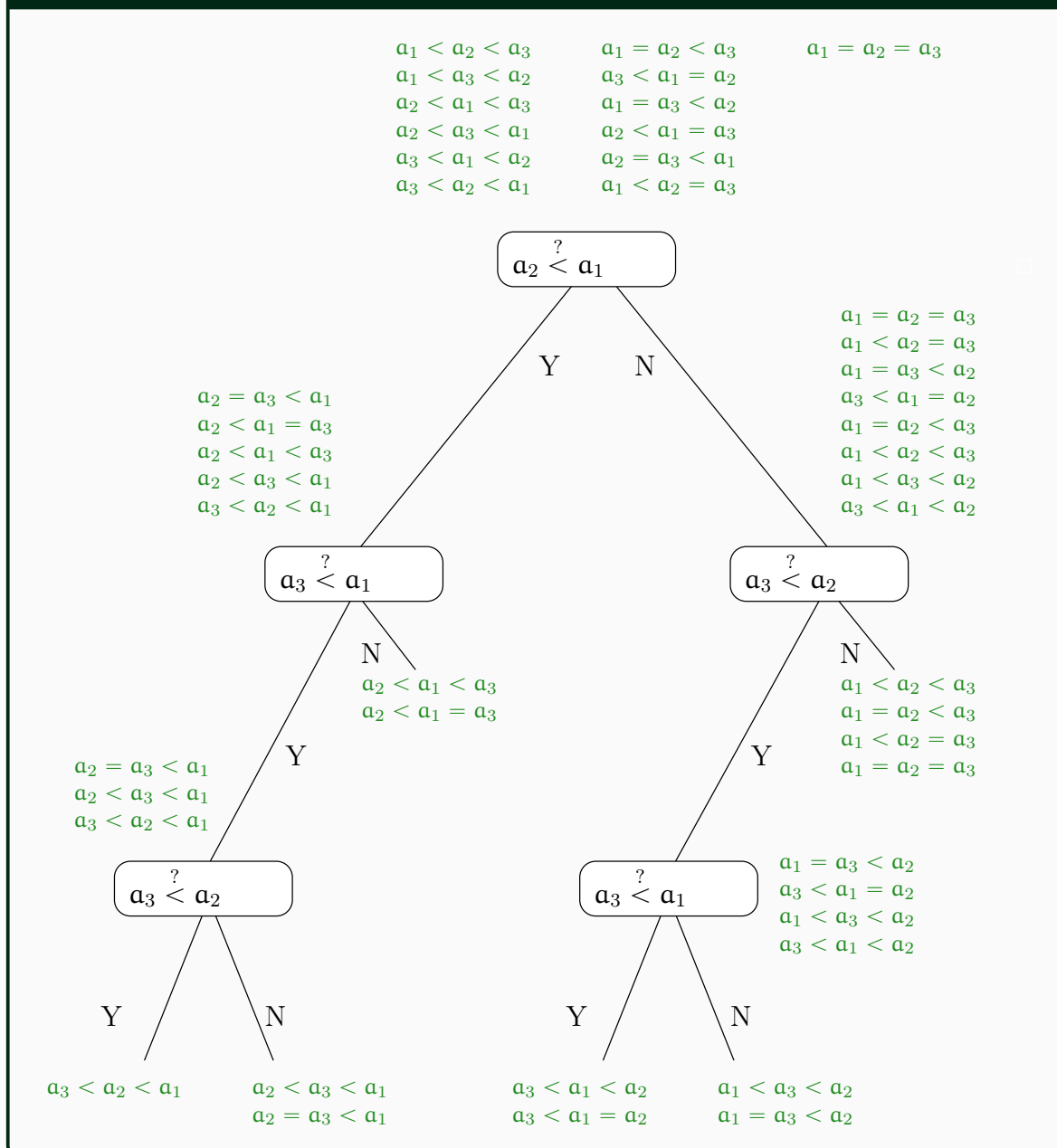
Това са слабите наредби от Допълнение 13 за $n = 3$. Както знаем от Допълнение 13, броят им е $P_n = \sum_{k=1}^n \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} k!$ при n елемента. Наистина, за $n = 3$ имаме

$$P_3 = \sum_{k=1}^3 \left\{ \begin{smallmatrix} 3 \\ k \end{smallmatrix} \right\} k! = \left\{ \begin{smallmatrix} 3 \\ 1 \end{smallmatrix} \right\} \cdot 1! + \left\{ \begin{smallmatrix} 3 \\ 2 \end{smallmatrix} \right\} \cdot 2! + \left\{ \begin{smallmatrix} 3 \\ 3 \end{smallmatrix} \right\} \cdot 3! = 1 \cdot 1 + 3 \cdot 2 + 1 \cdot 6 = 1 + 6 + 6 = 13$$

Ще наричаме тези състояния *псевдо-пермутациите*, а не “слабите наредби”, за да има аналогия с термина “пермутациите” на стр. 742. Терминът “псевдо-пермутации” е взет от статията [97].

Дървото на сравненията на INSERTION SORT за $n = 3$ с псевдо-пермутациите е показано на Фигура 13.4.

Фигура 13.4 : Дървото на сравнения на INSERTION SORT не различава псевдо-пермутациите.



Някои листа са асоциирани с повече от едно състояние (псевдо-пермутация). Какво връща дървото? Има смисъл да дефинираме, че всяко листо на дървото връща множеството от псевдо-пермутациите, асоциирани с него. Примерно, листото най-вляво връща $\{a_3 < a_2 < a_1\}$, това вдясно от него връща $\{a_2 < a_3 < a_1, a_2 = a_3 < a_1\}$, и така нататък. Може ли да кажем, че това дърво сортира? В някакъв смисъл, да: ако целта е само да се изчисли наредба на входните числа, съвместима с получените отговори, то очевидно наредбата (тя е точно една), отговаряща на строгите неравенства, върши работа. Но ако целта е да се намери единствената псевдо-пермутация на входните числа, която е истина, това дърво не върши работа.

И така, причината да се ограничим до входове с уникални елементи в търсенето на нетривиална долна граница за СОРТИРАНЕ е, че при допускането за уникалност из-

ложението е по-просто и състоянията са по-малко, откъдето работата на дърветата е по-лесно разбираема. А получената долна граница $\Omega(n \lg n)$ при допускането за уникалност е валидна и без него.

13.2.2.3 Дървото на сравненията на SELECTION SORT за $n = 3$

Да си припомним SELECTION SORT.

SELECTION SORT($A[1 \dots n]$: array of integers)

```

1  for  $i \leftarrow 1$  to  $n - 1$ 
2      for  $j \leftarrow i + 1$  to  $n$ 
3          if  $A[j] < A[i]$ 
4              swap( $A[i], A[j]$ )
```

Фигура 13.5 показва дървото на сравненията на SELECTION SORT върху вход $\langle a_1, a_2, a_3 \rangle$. То е различно от дървото на INSERTION SORT (Фигура 13.3). Забележете, че дървото на SELECTION SORT е съвършено двоично дърво (Определение 53), за разлика от дървото на INSERTION SORT. Ясно е защо е така: SELECTION SORT върши един и същи брой сравнения при дадено n без оглед на конкретиката на входа, а именно

- $n - 1$ сравнения при $i = 1$,
- $n - 2$ сравнения при $i = 2$,
- и така нататък,
- 1 сравнение при $i = n - 1$,

което прави общо $\binom{n}{2}$ сравнения. Следва, че в дървото на SELECTION SORT, всеки път от корена до листо има една и съща дължина $\binom{n}{2}^\dagger$. От друга страна, INSERTION SORT е адаптивен и, при дадено n , в общия случай прави различен брой сравнения за едно и също i в зависимост от конкретиката на входа.

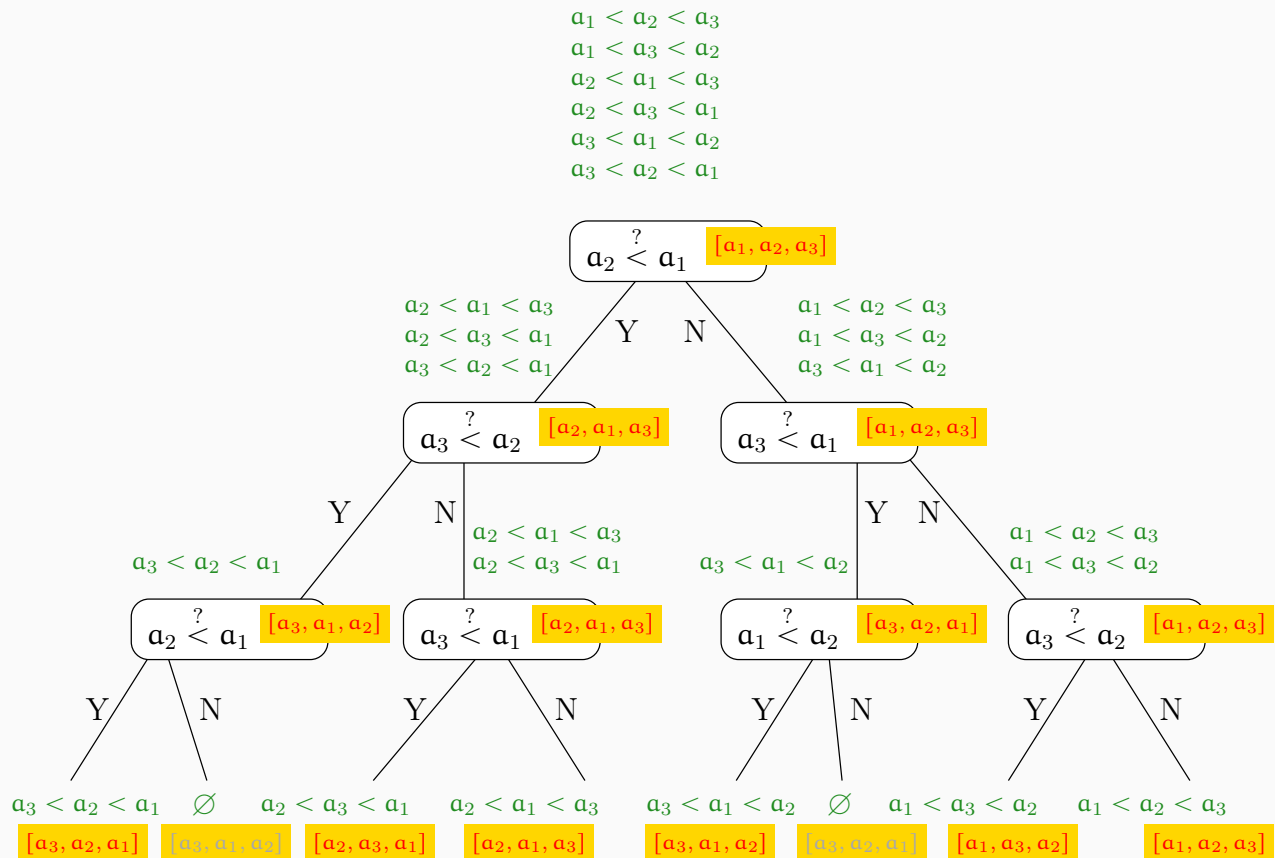
Също така забележете двете листа, маркирани с “ $\emptyset$ ”, в дървото на SELECTION SORT на Фигура 13.5. Всяко от тях отговаря на редица от въпроси и отговори, в която два пъти се сравняват a_1 и a_2 ; $\emptyset$ се асоциира с отговора, който е несъвместим с прежде получения отговор.

Във всеки връх на дървото е записано с червени букви на жълт фон съдържанието на масива A , като

- в корена е началното състояние $[a_1, a_2, a_3]$,
- във всеки друг връх е състоянието на A след извършеното сравнение в родителя и съответния отговор, водещ към въпросния връх.

Двете недостижими листа просто повтарят състоянието на родителя си, като всеки не-корен, реброто към който е маркирано с N ; текстът за тях е в **синьо**.

[†]Има $\binom{n}{2}$ вътрешни върхове, които отговарят на сравненията, и още един връх, който е листото, така че общо върховете са $\binom{n}{2} + 1$, откъдето дължината на пътя е $\binom{n}{2}$.

Фигура 13.5 : Дървото на сравненията на SELECTION SORT върху $\langle a_1, a_2, a_3 \rangle$.

Дървото на сравненията на SELECTION SORT е различно от дървото на сравненията на INSERTION SORT дори при $n = 3$.

13.2.2.4 Дървото на вземане на решение на произволен сортиращ алгоритъм

В Подподсекция 13.2.2.2 и Подподсекция 13.2.2.3 изведохме дървовидна структура от въпроси и отговори чрез подробно симулиране на работата на конкретни алгоритми при всички възможни резултати от сравненията. Такава дървовидна схема от въпроси и отговори се нарича *дърво за вземане на решение*, на английски *decision tree*. Определението в [34, стр. 192] е следното:

A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored.

Забележете, че има смисъл да разглеждаме такива дървета и извън контекста на сортирането. Както видяхме в миналата секция, известните задачи THE BALANCE PUZZLE (Задача 64) и THE TWELVE-COIN PUZZLE (Задача 65) са податливи на анализ чрез дървета за вземане на решение, само че тернарни. И така, дърветата за вземане на решение няма нужда да са двоични и не са ограничени до задачата СОРТИРАНЕ.

При всяко от тези:

- THE BALANCE PUZZLE,
- THE TWELVE-COIN PUZZLE,

- INSERTION SORT върху вход с размер три,
- SELECTION SORT върху вход с размер три,

дървото за вземане на решение е едно единствено. Очевидно е, че INSERTION SORT по принцип има не едно такова дърво, а безкрайно множество от дървета, по едно за всеки размер на входа. Същото е вярно и за SELECTION SORT[†].

Сега да се ограничим до дървета за вземане на решение, съответни на сортиращи алгоритми.

Наблюдение 83

Нека SOME SORT е произволен сортиращ алгоритъм, базиран на сравнения. SOME SORT има не едно единствено дърво за вземане на решение, а безкрайно множество от такива дървета, по едно за всеки размер на входа. Ако кажем “дървото за вземане на решение на SOME SORT”, имаме предвид общия елемент на това множество.

13.2.2.5 Дървото за вземане на решение е изчислителен модел

В Подподсекция 13.2.2.2 и Подподсекция 13.2.2.3 ние изведохме дървото от алгоритъма. Но може да мислим, че дървото е първичен обект, а не е дериват на сортиращ алгоритъм.

Определение 106: Пермутациите спрямо дърветата за вземане на решение

Нека са дадени числа $a_1, \dots, a_n$, две по две различни. *Пермутациите на $a_1, \dots, a_n$ са $n!$ на брой конюнкции от неравенства, където всяка конюнкция е над $n-1$ неравенства и е от вида*

$$a_{i_1} < a_{i_2} < \dots < a_{i_n}$$

за някоя същинска пермутация $\langle i_1, i_2, \dots, i_n \rangle$ на $\{1, 2, \dots, n\}$.

Щом числата са две по две различни, точно една от пермутациите е истина.

Определение 106 е в сила само в рамките на Лекция 13. По принцип “пермутация” е биекция от множество в себе си, което сега наричаме “същинска пермутация”.

Определение 107: Сортиращо дърво за вземане на решение

Нека са дадени числа $a_1, \dots, a_n$, две по две различни. *Сортиращо дърво за вземане на решение* е всяка дървовидна схема от сравнения от вида $a_i \stackrel{?}{<} a_j$ и техните отговори, където $1 \leq i < j \leq n$, всяко листо на която е асоциирано с точно една пермутация. Изходът от изчислението на дървото върху конкретен вход е пермутацията, която е асоциирана с листото, в което изчислението приключва.

Когато разглеждахме тези дървета като деривати (на сортиращи алгоритми), видяхме, че може листо да е асоциирано с нула пермутации; примерно, дървото на Фигура 13.5. Това не е в противоречие с Определение 107. Можем да мислим, че листата с по нула пермутации са изтрети, понеже са недостижими.

[†]Апропо, THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE също имат обобщения за n топки или монети, където n е произволно цяло положително число. Всяко от тези обобщения има решение, което се състои от безкрайно множество от дървета за вземане на решение: по едно за всяко n .

Дървото, в някакъв смисъл, сортира, само че **не чрез местене** на елементи, а чрез намиране на единствената пермутация, която е истина. Може да мислим, че дървото е първичният обект, а алгоритъмът е дериват, като работата на алгоритъма е да мести елементи по такъв начин, че в правилният момент “на везната” да бъдат поставени правилните елементи, за да бъдат сравнени.

Всяко дърво за вземане на решение, независимо дали е сортиращо или е за друга задача, е *неуниформен изчислителен модел*, на английски *nonuniform computational model*, защото за всяка големина на входа дървото е различно. Друг пример за неуниформен изчислителен модел са логическите схеми (на английски, *logic circuits*; вижте [136, стр. 16]). В контраст на това, машината с произволен достъп [136, стр. 19] е униформен изчислителен модел, защото за всяка големина на входа тази машината е една и съща. Машината на Turing (Определение 120) също е униформен изчислителен модел.

13.2.2.6 За структурата на сортиращите дървета за вземане на решение

Определение 108: Дърво за вземане на решение различава пермутациите

Нека $a_1, \dots, a_n$ са числа, две по две различни, и T е дърво за вземане на решение върху тях. T *различава всяка двойка пермутации*, накратко *различава пермутациите*, тстк всяко негово листо е асоциирано с ≤ 1 пермутация.

Наблюдение 84: Сортиращите дървета за вземане на решение различават пермутациите

По определение (Определение 107), сортиращите дървета за вземане на решение различават пермутациите.

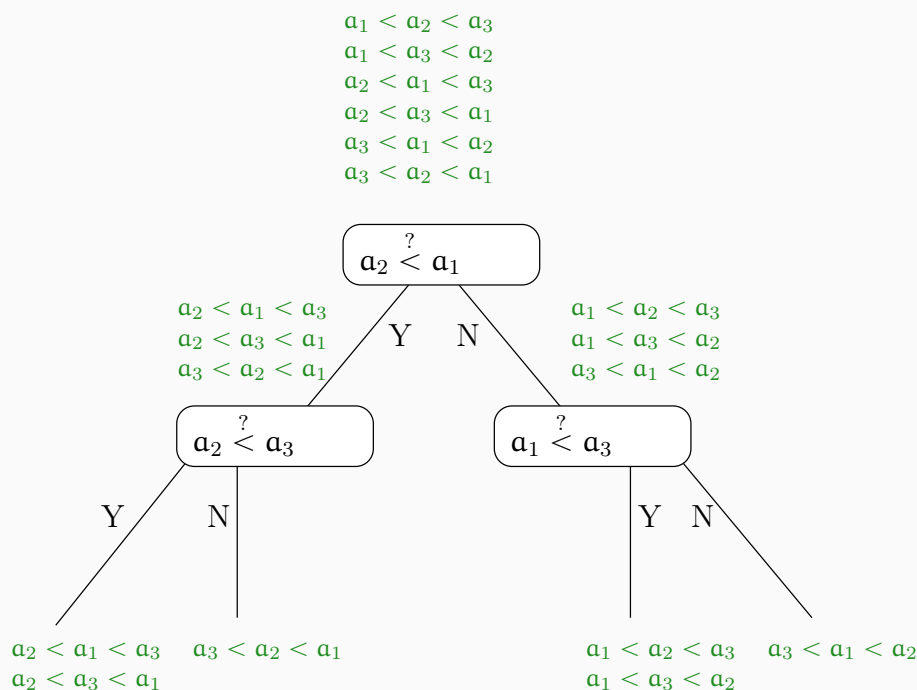
Каква е структурата на дърво T , което хем различава пермутациите, хем е възможно най-малко? Следните две наблюдения показват, че отговорът не е тривиален.

Наблюдение 85

За да различава T пермутациите, не е достатъчно T да прави сравнение $a_i \stackrel{?}{<} a_j$, за всеки i и j , такива че $1 \leq i < j \leq n$, **в някой връх**.

С други думи, не е достатъчно за всяко $\{i, j\} \subseteq \{1, \dots, n\}$ да има поне един вътрешен връх с въпроса $a_i \stackrel{?}{<} a_j$. Пример за това е дървото на Фигура 13.6.

Фигура 13.6 : Не е достатъчно всяка двойка да е сравнена някъде в дървото, за да бъдат различавани пермутациите.



Това дърво не различава $a_2 < a_1 < a_3$ от $a_2 < a_3 < a_1$, а също и $a_1 < a_2 < a_3$ от $a_1 < a_3 < a_2$.

Дървото на Фигура 13.6 сравнява a_1 с a_2 , и a_2 с a_3 , и a_2 с a_3 , и въпреки това не различава някои двойки пермутации. Лесно се вижда защо е така: пермутациите $a_2 < a_1 < a_3$ и $a_2 < a_3 < a_1$ се асоциират с лявото дете на корена, а въпросът $a_1 < a_3$, който би могъл да ги различи, се намира в дясното дете на корена. Ерго, различаващите въпроси трябва да са разположени адекватно по всички пътища от корена до листата. Но не трябва да залитаме в другата крайност!

Наблюдение 86

За да различава T пермутациите, не е необходимо по всеки път от корена до листо да се прави сравнение $a_i < a_j$ за всеки i и j , такива че $1 \leq i < j \leq n$.

Със сигурност, ако по всеки път от корена “надолу” към някое листо сравняваме всяка двойка числа, дървото различава пермутациите. Но това не е необходимо. Защо? Защото T би имало височина $\binom{n}{2}$, което означава да има $2^{\binom{n}{2}}$ листа. Това би било грамадно дърво, което наистина няма листо с повече от една пермутация, но почти всички листа са асоциирани с $\emptyset$.

За да се убедим в последното, да съобразим, че $\lim_{n \rightarrow \infty} \frac{2^{\binom{n}{2}}}{n!} = \infty$. Това на свой ред се извежда тривиално с логаритмуване на числителя и знаменателя, което води до $\lim_{n \rightarrow \infty} \frac{\Theta(n^2)}{\Theta(n \lg n)}$, което очевидно е ∞ .

Ерго, да сравняваме винаги (по всеки път от корена към листо) всяка двойка числа означава да сортираме във време $\Omega(n^2)$. Но ние знаем, че може да се сортира във време $\Theta(n \lg n)$ от алгоритми като HEAPSORT и MERGESORT. Техните дървета със сигурност не сравняват всяка двойка числа по всеки път от корена към листо.

13.2.2.7 Долна граница $\Omega(n \lg n)$ чрез дървета за вземане на решение

Сега ще видим как дърветата за вземане на решение дават долна граница за изчислителната задача, с която са свързани.

Теорема 107: Височината на сортиращо дърво за вземане на решение е $\Omega(n \lg n)$

Нека T е сортиращо дърво за вземане на решение с вход n уникални елементи. Нека h е височината на T . В сила е $h = \Omega(n \lg n)$.

Доказателство: По определение (Определение 107), всяко дърво T за вземане на решение, което сортира n уникални елементи, има $n!$ листа. От курса Дискретни Структури знаем, че щом T е двоично дърво, в сила е $h \geq \log_2 \ell$, където ℓ е броят на листата. Щом $\ell \geq n!$, в сила е $h \geq \log_2 n!$. Но съгласно Теорема 33, $\log_2 n! \asymp n \lg n$. Заклучаваме, че $h = \Omega(n \lg n)$. $\square$

Следствие 32: $\Omega(n \lg n)$ е долна граница за сортиране на уникални числа, базирано на сравнения

Нека $h_{\min}(n)$ е минималната височина на дърво за вземане на решение, което сортира n уникални елементи. $h_{\min}(n)$ е асимптотична долна граница за сортиране на уникални елементи, базирано на сравнения, понеже всеки сортиращ алгоритъм, базиран на сравнения, има съответно дърво за вземане на решение. Максималният брой въпроси, който задава дървото, за да сортира n уникални елементи, е точно височината на дървото.

Както видяхме в Теорема 107, $h_{\min}(n) = \Omega(n \lg n)$. Тогава $\Omega(n \lg n)$ е долна граница за сортиране, базирано на сравнения, ако елементите са уникални.

Следствие 33: $\Omega(n \lg n)$ е долна граница за сортиране, базирано на сравнения

Елементите във входа да са уникални е строго частен случай на общия случай, в който се допуска повтаряне на елементи във входа. Щом $\Omega(n \lg n)$ е долна граница за частния случай, тя е валидна долна граница и за общия случай.

13.2.2.8 Минималният брой на сравненията в най-добрия случай е $n - 1$

Този резултат е без значение за долната граница, но е любопитен сам по себе си. Вече видяхме, че височината на сортиращо дърво за вземане на решение е $\Omega(n \lg n)$. Това е долна граница за броя на сравненията и тя е точна, което означава, че в най-лошия случай се налага дървото да зададе толкова въпроси, за да изчисли пермутацията на входните елементи. “Най-лошият случай” може да се интерпретира като “когато задаващият въпросите е абсолютен кутсуз”. Най-лошият случай е дължината на най-дълъг път в дървото от корена до листо; с други думи, височината на дървото за вземане на решение.

Сега разсъждаваме над това, с колко въпроса може да се изчисли пермутацията, ако задаващият въпросите е абсолютен късметлия? Тук става дума за дължината на най-къс път в дървото за вземане на решение от корена до листо. На автора на записките не е известен термин, аналогичен на “височина”, за това понятие, така че ще ползва “минималното разстояние от корена до листо”.

Теорема 108: Сортиращите дървета сравняват всяка двойка съседни по големина елементи

Нека са дадени числата $a_1, \dots, a_n$, две по две различни. Нека $\langle i_1, i_2, \dots, i_n \rangle$ е уникалната пермутация на $1, 2, \dots, n$, такава че

$$a_{i_1} < a_{i_2} < \dots < a_{i_n}$$

Нека $A = \langle a_1, \dots, a_n \rangle$. Тогава всяко сортиращо дърво T за вземане на решение с вход A :

- сравнява a_{i_1} с a_{i_2} ,
- сравнява a_{i_2} с a_{i_3} ,
- и така нататък,
- сравнява $a_{i_{n-1}}$ с a_{i_n} .

Не се твърди, че тези сравнения стават в този ред. Това, което се твърди е, че по време на работата на $T(A)$, всяко от тези сравнения се случва, рано или късно. Може да има и други сравнения, но тези се случват със сигурност.

“Сравнява a_p с a_q ” означава, че извършва или сравнението $a_p \stackrel{?}{<} a_q$, или сравнението $a_q \stackrel{?}{<} a_p$. Да се извършват и двете е безсмислено, понеже числата са уникални по условие.

Доказателство: Да допуснем противното. Нека при работата си върху входа A , за някое $k \in \{1, \dots, n-1\}$, дървото T не сравнява a_{i_k} с $a_{i_{k+1}}$ и въпреки това връща коректно пермутацията

$$\pi = a_{i_1} < a_{i_2} < \dots < a_{i_{k-1}} < a_{i_k} < a_{i_{k+1}} < a_{i_{k+2}} < \dots < a_{i_n}$$

Нека A' е друг вход, който се получава от A с размяна на стойностите на a_{i_k} и $a_{i_{k+1}}$.

От ключово значение е на този етап от доказателството да сте разбрали, че пермутацията π се определя само от стойностите. Примерно, ако $n = 5$ и $a_1 = 8, a_2 = 4, a_3 = 1, a_4 = 6$ и $a_5 = 5$, тоест $A = \langle 8, 4, 1, 6, 5 \rangle$, то $\pi = a_3 < a_2 < a_5 < a_4 < a_1$, което отговаря на пермутацията-биекция на индекси $\langle 3, 2, 5, 4, 1 \rangle$.

В същия пример, ако $k = 2$, получаваме A' от A чрез размяна на втория **по големина** с третия **по големина** елемент. Вторият по големина елемент на A е $a_2 = 4$, а третият по големина елемент на A е $a_5 = 5$. Тогава $A' = \langle 8, 5, 1, 6, 4 \rangle$.

По отношение на A' е в сила

$$a_{i_1} < a_{i_2} < \dots < a_{i_{k-1}} < \textcolor{red}{a_{i_{k+1}}} < \textcolor{red}{a_{i_k}} < a_{i_{k+2}} < \dots < a_{i_n}$$

Тази пермутация е различна от π ; вижте елементите в червено. $T(A')$ трябва да върне нея. Но $T(A')$ връща π —също като $T(A)$ —което е некоректно.

Сега ще докажем, че $T(A')$ връща π . Ключовото наблюдение е, че ако $\{p, q\} \neq \{k, k+1\}$, то резултатът от сравнението $a_p \stackrel{?}{<} a_q$ е един и същи за $T(A)$ и за $T(A')$. Това е напълно очевидно и не си заслужава да се доказва формално, но е добре да се спомене експлицитно.

В нашия пример с

$$A = \langle 8, 4, 1, 6, 5 \rangle$$

$$A' = \langle 8, 5, 1, 6, 4 \rangle$$

резултатът от всяко сравнение на елементи е един и същи, освен ако не сравняваме втория и петия елемент. Наистина,

- $a_1 \stackrel{?}{<} a_2$ има отговор N както при вход A, така и при вход A',
- $a_1 \stackrel{?}{<} a_3$ има отговор N както при вход A, така и при вход A',
- $a_2 \stackrel{?}{<} a_3$ има отговор N както при вход A, така и при вход A',
- $a_2 \stackrel{?}{<} a_4$ има отговор Y както при вход A, така и при вход A',
- и така нататък.

От това наблюдение веднага следва, че в $T(A)$ се изминава един и същи път от корена до листо, какъвто се изминава в $T(A')$. Тогава изходът на $T(A)$ и $T(A')$ е един и същи, а именно π . Тогава T не е коректно дърво за вземане на решение. $\square$

Заслужава си да се подчертае следното за Теорема 108. Сортиращото дърво “не знае” поначало кои са съседните по големина елементи; то не знае, че a_{i_1} е най-малкото число, следвано от a_{i_2} и така нататък[†]. Това, че всяко сортиращо дърво рано или късно слага на везната всяка двойка съседни по големина елементи е неизбежно вярно и е съществено важно за дървото, въпреки че дървото “не знае” предварително кои са тези двойки.

Следствие 34: За всеки път p от корена до листо в сортиращо дърво, $|p| \geq n - 1$

Нека T е сортиращо дърво за вземане на решение. Тогава дължината на всеки път от корена до листо е поне $n - 1$.

Доказателство: Твърдението следва веднага от Теорема 108. $\square$

Ще въведем едно много полезно понятие, което се ползва в няколко доказателства за долни граници.

Определение 109: Граф на сравненията за уникални елементи

Нека са дадени числа $a_1, \dots, a_n$, две по две различни. Нека са извършени сравнения от вида $a_i \stackrel{?}{<} a_j$, като $i \neq j$. *Графът на сравненията* е ориентиран граф с върхове $a_1, \dots, a_n$, чиито ребра са генерирани по следния начин. За всяко сравнение $a_i \stackrel{?}{<} a_j$:

- ако изходът от сравнението е Y, то има ребро (a_i, a_j) и казваме, че a_i е изгубило от a_j , а също и че a_j е спечелило от a_i ,
- ако изходът от сравнението е N, то има ребро (a_j, a_i) и казваме, че a_j е изгубило от a_i , а също и че a_i е спечелило от a_j .

Други ребра няма.

[†]Ако дървото знаеше това поначало, задачата щеше да се обезсмисли. Аналогично, съответният сортиращ алгоритъм не знае предварително кой е най-малкият елемент във входа, кой е следващият и така нататък – ако знаеше това предварително, сортирането щеше да става в линейно време.

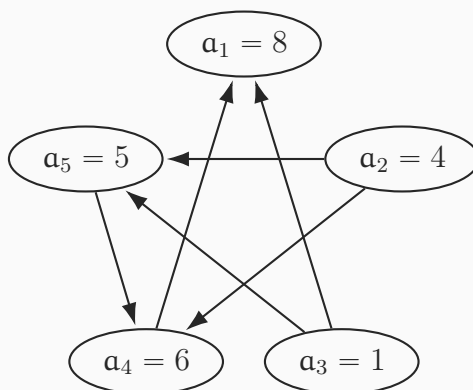
Наблюдение 87: Графът на сравненията е даг

Графът на сравненията е ацикличен, което следва веднага от факта, че релацията $<$ е антисиметрична.

Предвид това, отгук наметне ще говорим за *дага на сравненията*.

Фигура 13.7 показва примерен даг на сравненията при уникални елементи.

Фигура 13.7 : Пример за даг на сравненията при гарантирано уникални елементи.



Направени са сравненията $a_3 \stackrel{?}{<} a_5$, $a_3 \stackrel{?}{<} a_1$, $a_1 \stackrel{?}{<} a_4$, $a_2 \stackrel{?}{<} a_5$, $a_2 \stackrel{?}{<} a_4$ и $a_4 \stackrel{?}{<} a_5$ с изходи съответно Y, Y, N, Y, Y и N.

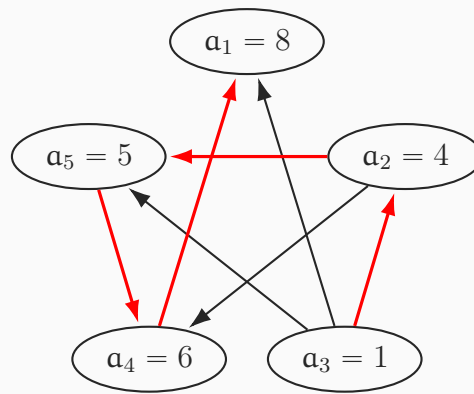
Теорема 108 влече, че след извършването на всички сравнения от сортиращо дърво за вземане на решение върху вход от уникални числа, полученият даг на сравненията съдържа Хамилтонов път, като Хамилтоновият път точно отговаря на уникалната пермутация-изход. В терминологията на Теорема 108, Хамилтоновият път е

$$p = a_{i_1}, a_{i_2}, \dots, a_{i_n}$$

От курса Дискретни Структури знаем, че ако даг има Хамилтонов път, той е единствен. Ерго, въпросният p е единственият Хамилтонов път в дага на сравненията, което отговаря на факта, че сортиращото дърво за вземане на решение връща точно една пермутация.

В дага на сравненията на Фигура 13.7 няма Хамилтонов път, понеже a_2 и a_3 не са сравнени. Ако бъдат сравнени и те, в дага вече има Хамилтонов път, както е показано на Фигура 13.8.

Фигура 13.8 : Даг на сравненията с Хамилтонов път.



Направени са сравненията $a_3 \stackrel{?}{<} a_5$, $a_3 \stackrel{?}{<} a_1$, $a_1 \stackrel{?}{<} a_4$, $a_2 \stackrel{?}{<} a_5$, $a_2 \stackrel{?}{<} a_4$, $a_4 \stackrel{?}{<} a_5$ и $a_2 \stackrel{?}{<} a_3$ с изходи съответно Y, Y, N, Y, Y, N и N. Хамилтоновият път е в червено.

13.2.3 Долна граница $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ

Става дума за Задача 19: дали дадени числа $a_1, \dots, a_n$ са уникални, или има повторения. Задачата е задача за разпознаване. За краткост пишем “EU”, от **E**LEMENT **U**NIQUENESS, наместо “УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ”.

Разглеждаме задачата само в контекста на изчислителен модел, базиран на сравнения. Алгоритъм за задачата EU е базиран на сравнения, ако изходът се намира само чрез извършване на сравнения от вида $a_i \stackrel{?}{<} a_j$. В Подсекция 4.2.3 видяхме два алгоритъма за EU: тривиалния квадратичен алгоритъм, опитващ всички ненаредени двойки, и по-изтънчения алгоритъм, който първо сортира числата и след това с едно сканиране на сортираната редица изчислява дали числата са уникални. И двата алгоритъма са базирани на сравнения, ако сортирането е базирано на сравнения. Наистина, в псевдокода в Подсекция 4.2.3 сравненията са от вида $A[i] \stackrel{?}{=} A[j]$, но това може да се реализира чрез $A[i] \stackrel{?}{<} A[j] \wedge A[j] \stackrel{?}{<} A[i]$.

Сега няма предварително допускане за уникалност на елементите. Когато правихме разсъжденията за сортиращите дървета за вземане на решение, допуснахме предварително, че елементите са уникални. В контекста на сортирането е допустимо да се постулира **преди** конструкцията на алгоритъма/дървото, че елементите са уникални. Това допускане не променя нищо, нито в алгоритъма, нито в дървото за вземане на решение, и долната граница $\Omega(n \lg n)$ е в сила.

В контекста на сортирането, допускането за уникалност на елементите води до това, че състоянията са точно пермутациите, и ни дава право да твърдим, че дърветата различават състоянията с листата си. Допълнение 68 показва, че в отсъствието на допускане за уникалност на елементите, състоянията не са пермутациите, а псевдо-пермутациите, които са значително повече от $n!$ [†], така че листата са асоциирани с по няколко състояния. Това не е фатално за извеждането на долната граница $\Omega(n \lg n)$ за сортирането; тя може да се изведе и без допускане за уникалност, но може би тогава нещата са по-неясни, понеже дърветата не различават състоянията с листата си. Фигура 13.4 показва дървото за вземане на решение

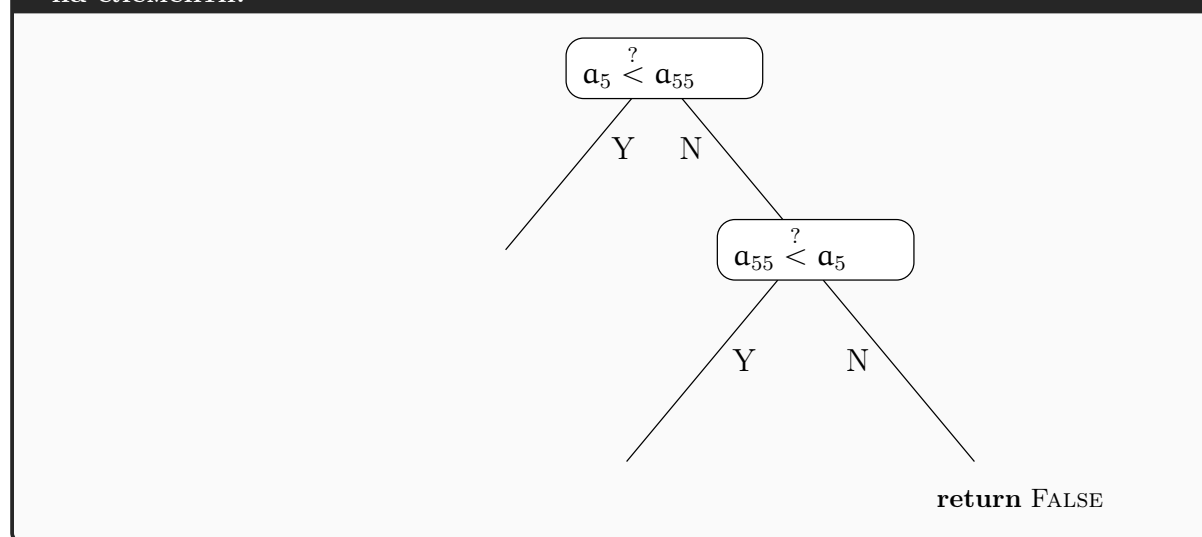
[†] А именно $P_n = \sum_{k=1}^n \binom{n}{k} k!$. Вижте Допълнение 13 за числата на Bell с наредба.

на INSERTION SORT при допускане за повтаряне на елементи. Дървото е същото като това на Фигура 13.3, но не различава състоянията (псевдо-пермутациите).

В контекста на EU обаче би било груба грешка да постулираме предварително, че елементите са уникални. Това би обезсмислило задачата, защото всеки екземпляр би бил ДА-екземпляр и алгоритъмът, който изпълнява само връщане на TRUE, би я решавал. Наистина, в разсъжденията долу ние ще разглеждаме само входове с уникални елементи, но това е **съвсем друго нещо**. Алгоритъмът, съответно и дървото, за EU е конструиран без допускане за уникалност, а когато ние подаваме входове след това (при вече конструиран алгоритъм за общия случай, позволяващ повтаряне), имаме право да подаваме само входове с уникални елементи. Ако получим долна граница $\Omega(n \lg n)$ при това, тя е в сила и в общия случай.

Следствие 34 не е в сила за EU. Повтаряне на елементи може да се установи много бързо, със само едно сравнение $a_i \stackrel{?}{=} a_j$, ако задаващият въпроса е късметлия и улучи двойка повтарящи се елементи. Ако са разрешени само въпроси от вида $a_i \stackrel{?}{<} a_j$, то двете сравнения $a_i \stackrel{?}{<} a_j$ и $a_j \stackrel{?}{<} a_i$ ще свършат работа: отрицателен отговор и на двете означава, че $a_i = a_j$. Фигура 13.9 илюстрира това.

Фигура 13.9 : В най-добрия случай, с две сравнения се намира повтаряне на елементи.



Доказателството за долната граница за EU. Ще докажем, че всеки алгоритъм за EU, базиран на сравнения, има сложност $\Omega(n \lg n)$. От това следва, че няма съществено по-бърз начин да се изследва уникалността от това да се сортират числата; вече знаем, че сортирането със сравнения става във време $\Omega(n \lg n)$.

Ще направим доказателство с дърво за вземане на решение. Всеки алгоритъм ALGEU за EU, базиран на сравнения, има безкрайно множество $\mathcal{T}$ от дървета за вземане на решение, по едно за всяка големина на входа, а общият елемент на $\mathcal{T}$ наричаме “дървото за вземане на решение”. За разлика от дървото на сортиращ алгоритъм, което изчислява пермутацията и това е изходът му, изходът на дървото на ALGEU е бинарен: то или връща TRUE, ако елементите са уникални, или връща FALSE, ако не са. Това е така, защото EU е задача за разпознаване.

Ще докажем, че всяко листо, връщащо TRUE, е асоциирано с не повече от една пермутация; става дума за пермутация в смисъла на Определение 106, а не за псевдо-пермутация.

Тогава тези листа са поне $n!$. Но тогава всички листа са поне $n!$ и доказателството на долната граница се довършва като при сортирането: Теорема 107, тривиално модифицирана за EU, ни казва, че височината е $\Omega(n \lg n)$.

Щом разсъждаваме само за листата, връщащи TRUE, разглеждаме само ДА-екземпляри на EU.

И тук помага осмислянето на нещата с граф на сравнението, но се налага да променим дефиницията. В контекста на сортирането допуснахме *a priori*, че числата от входа са уникални, поради което ребро се слага в графа на сравненията и при положителен, и при отрицателен отговор на сравнение (припомнете си Определение 109). Сега ребро се слага само при положителен отговор, защото искаме ребрата да са само от по-малки към по-големи елементи.

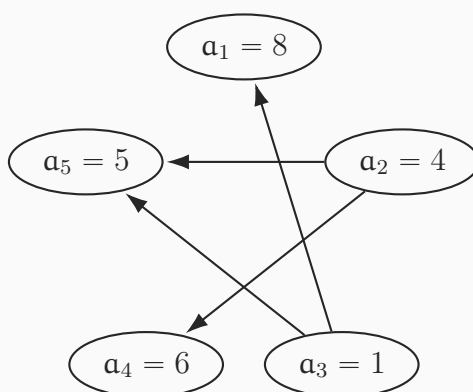
Определение 110: Граф на сравненията при възможни повтаряния

Нека са дадени числа $a_1, \dots, a_n$, не непременно уникални. Нека са извършени сравнения от вида $a_i \stackrel{?}{<} a_j$, като $i \neq j$. *Графът на сравненията* е ориентиран граф с върхове $a_1, \dots, a_n$, чиито ребра са генерирани по следния начин. За всяко сравнение $a_i \stackrel{?}{<} a_j$, ако изходът от сравнението е Y, то има ребро (a_i, a_j) и казваме, че a_i е изгубило от a_j , а също и че a_j е спечелило от a_i . Други ребра няма.

Графът на сравненията и при тази дефиниция е даг.

Фигура 13.10 показва примерен даг на сравненията съгласно Определение 110. Числата и сравненията са същите като в примера на Фигура 13.7, но ребрата в дага на Фигура 13.10 са по-малко, понеже двата отрицателни отговора не пораждаят ребра.

Фигура 13.10 : Пример за даг на сравненията при възможни повтаряния.



Направени са сравненията $a_3 \stackrel{?}{<} a_5$, $a_3 \stackrel{?}{<} a_1$, $a_1 \stackrel{?}{<} a_4$, $a_2 \stackrel{?}{<} a_5$, $a_2 \stackrel{?}{<} a_4$ и $a_4 \stackrel{?}{<} a_5$ с изходи съответно Y, Y, N, Y, Y и N.

Въпреки че заглавието на Фигура 13.10 казва “при възможни повтаряния”, конкретните числа са уникални. Това е напълно допустимо – “възможно е да има повтаряния” е различно от “трябва да има повтаряния”. Но, тъй като дагът на сравненията е конструиран с оглед на възможни повтаряния, той не съдържа ребрата, съответни на двете сравнения с отрицателен отговор.

Теорема 109: За дърветата на EU върху ДА-екземплярите

Нека са дадени числата $a_1, \dots, a_n$, две по две различни. Нека $A = \langle a_1, \dots, a_n \rangle$. Нека $\langle i_1, i_2, \dots, i_n \rangle$ е уникалната пермутация на $1, 2, \dots, n$, такава че

$$a_{i_1} < a_{i_2} < \dots < a_{i_n}$$

Нека T е произволно дърво за вземане на решение на EU. Вярно е, че $T(A)$:

- извършва сравнението $a_{i_1} \stackrel{?}{<} a_{i_2}$,
- извършва сравнението $a_{i_2} \stackrel{?}{<} a_{i_3}$,
- и така нататък,
- извършва сравнението $a_{i_{n-1}} \stackrel{?}{<} a_{i_n}$.

Не се твърди, че тези сравнения стават в този ред. Това, което се твърди е, че по време на работата на T с вход A , всяко от тези сравнения се случва, рано или късно. Може да има и други сравнения, но тези се случват със сигурност, щом входът е от уникални числа.

Забележете разликата с Теорема 108. Там се твърдеше, че за всяка двойка индекси на съседни по големина елементи се извършва поне едно от двете възможни сравнения на съответните елементи от входа. Тук се твърди, че за всяка двойка индекси на съседни по големина елементи се извършва не кое да е, а **точно определено** сравнение на съответните елементи от входа. Това е така, защото сортиращото дърво в Теорема 108 е направено при допускането за уникалност на елементите и е все едно кое от двете сравнения се прави. Докато дървото на EU не е направено при допускане на уникалност и отрицателен отговор на сравнение $a_p \stackrel{?}{<} a_q$ не води до еднозначен извод.

Доказателство: Да допуснем противното. Нека при работата си върху входа A , за някое $k \in \{1, \dots, n-1\}$, дървото T не извършва $a_{i_k} \stackrel{?}{<} a_{i_{k+1}}$ и въпреки това връща коректно TRUE. Нека A' е друг вход, който се получава от A с присвояването

$$a_{i_k} \leftarrow a_{i_{k+1}}$$

Забелязваме, че A' е НЕ-екземпляр и $T(A')$ би трябвало да върне FALSE.

Като пример, нека $n = 5$ и $a_1 = 8, a_2 = 4, a_3 = 1, a_4 = 6$ и $a_5 = 5$, тоест $A = \langle 8, 4, 1, 6, 5 \rangle$. Нека $k = 2$. A' се получава от A , като на втория по големина елемент, а именно a_2 , се присвои стойността на третия по големина елемент a_3 . Тази стойност е 5, така че $A' = \langle 8, 5, 1, 6, 5 \rangle$. Наистина, A' е НЕ-екземпляр, понеже има два елемента-петици.

Само че $T(A')$ връща TRUE. Ключовото наблюдение е, че ако $(p, q) \neq (k, k+1)$, то резултатът от сравнението $a_p \stackrel{?}{<} a_q$ е един и същи за $T(A)$ и за $T(A')$. Дори сравнението да е $a_{k+1} \stackrel{?}{<} a_k$, резултатът е N както върху вход A , така и върху вход A' .

В нашия пример с

$$A = \langle 8, 4, 1, 6, 5 \rangle$$

$$A' = \langle 8, 5, 1, 6, 5 \rangle$$

резултатът от всяко сравнение на елементи е един и същи, освен ако сравнението не е $a_2 \stackrel{?}{=} a_5$. Наистина,

- $a_1 \stackrel{?}{<} a_2$ има отговор N както при вход A, така и при вход A',
- $a_1 \stackrel{?}{<} a_3$ има отговор N както при вход A, така и при вход A',
- $a_3 \stackrel{?}{<} a_4$ има отговор Y както при вход A, така и при вход A',
- $a_5 \stackrel{?}{<} a_2$ има отговор N както при вход A, така и при вход A',
- и така нататък.

Полученото противоречие показва, че такова T не съществува. □

Следствие 35: Установяването на уникалност генерира Хамилтонов път

В контекста на Теорема 109, нека дагът на сравненията (Определение 110), който $T(A)$ генерира, е G. Тогава G съдържа Хамилтонов път p. Нещо повече:

$$p = a_{i_1}, a_{i_2}, \dots, a_{i_n}$$

Това, че сортиращите дървета различават пермутациите, е заложено в дефиницията им (Определение 107), така че просто го отбелязахме в Наблюдение 84. В контекста на EU обаче, изобщо не е очевидно, че дърветата (на EU) различават пермутациите. Но ние вече почти сме доказали това.

Теорема 110: Дърветата за вземане на решение на EU различават пермутациите

Всяко дърво за вземане на решение на EU различава пермутациите.

Доказателство: Нека имената от Теорема 109 и Следствие 35 са в сила. Нека ℓ е листото на T, чрез което $T(A)$ връща TRUE. Твърдим, че ℓ е асоциирано с една единствена пермутация, а именно

$$\pi = a_{i_1} < a_{i_2} < \dots < a_{i_n}$$

Съгласно изучаваното по Дискретни Структури, даг не може да има повече от един Хамилтонов път. От това следва, че p е единственият Хамилтонов път в G. А от това следва, че π е единствената пермутация, асоциирана с ℓ . □

След като сме доказали Теорема 110, долната граница $\Omega(n \lg n)$ за задачата EU, базирана на сравнения, следва от същите съображения като тази на СОРТИРАНЕ.

13.2.4 Долна граница $\Omega(\lg n)$ за ТЪРСЕНЕ

Вече разгледахме алгоритми за двоично търсене в итеративен вариант (Алгоритъм 4) и в рекурсивен вариант (Допълнение 28). И в двата варианта, сложността по време в най-лошия случай е $\Theta(\lg n)$.

Сега твърдим, че $\Omega(\lg n)$ е асимптотична долна граница за ТЪРСЕНЕ. Забележете, че тук не говорим за двоично търсене, а за **всякакво** търсене! Ако търсенето е базирано на сравнения, всеки алгоритъм работи във време $\Omega(\lg n)$. Причината е много проста: има $\Theta(n)$ различни състояния по отношение на даден вход. Нека входът е масив $A[1..n]$ (сортиран или не, няма значение) и търсената стойност е key . Да допуснем, че елементите от A са два по два различни. Това не е ограничение на общността, понеже, ако получим долна граница при това допускане, в общия случай долната граница не може да е по-ниска.

Всеки алгоритъм за ТЪРСЕНЕ, базиран на сравнения, трябва да може да различава (поне) следните $n + 1$ състояния:

състояние 1: $key = A[1]$,

състояние 2: $key = A[2]$,

...

състояние n : $key = A[n]$,

състояние $n + 1$: $key \notin A$.

в смисъл, че това са съществено различни ситуации и дървото за вземането на решение трябва да е такова, че всяко листо да е асоциирано с не повече от едно от тези състояния. В началото на алгоритъма всички тези състояния (ситуации, ако предпочитате) са възможни, а след всяко извършване на сравнение между key и елемент на масива, някои от тях стават невъзможни[†]. Ако в края на алгоритъма останат две или повече възможни състояния, то алгоритъмът “не си е свършил работата” и отговорът му, какъвто и да е, е невалиден. Ерго, всеки коректен алгоритъм има такова дърво, че всяко листо е асоциирано с най-много едно от тези състояния.

Но това са $\Theta(n)$ състояния, което дава долна граница $\Omega(n)$ за броя на листата, което дава долна граница $\Omega(\lg n)$ за височината, което дава долна граница $\Omega(\lg n)$ за сложността по време на алгоритъма.

13.2.5 Лампи и ключове

Преди повече от петнадесет години авторът на тези лекционни записки чу следната задача.

В една стая има три ключа за лампи. Лампите се намират в друга стая, като между двете стаи няма видимост. Всеки ключ пали/гаси точно една от лампите. Целта е да разберем кой ключ за коя лампа е чрез само един експеримент.

“Да направим експеримент” в този контекст означава да сложим всеки от ключовете в някакво положение и след това да отидем в стаята с лампите и да извършим наблюдение – да видим коя лампа е запалена и коя е загасена.

Първоначално авторът изпадна в когнитивен дисонанс, защото за него беше очевидно, че за повече от две лампи не е достатъчен един експеримент. С един експеримент може да определим кой ключ на коя лампа отговаря само при два ключа и две лампи. При повече ключове и лампи са необходими поне два експеримента. А задачата иска решение със само един експеримент!

[†]Може дори да няма такива—които стават невъзможни след някое сравнение—ако сравнението е безсмислено. Това може да се случи, ако алгоритъмът е глупав, но с нищо не променя валидността на аргументацията.

Естествено, задачата мълчаливо допуска, че ключовете са *позиционни*. Това означава, че всеки ключ има две външно различни положения и ние знаем кое положение затваря електрическа верига и кое положение отваря електрическа верига. Повечето ключове за осветление са точно такива, като при правилно монтиране положението *надолу* прекъсва верига (тоест, гаси лампата), а положението *нагоре* затваря верига (тоест, пали лампата)[†]. Но има ключове, които нямат две видимо различни положения – при такъв ключ няма как да знаем дали палим или гасим, натискайки еднократно, ако не виждаме резултата от действието[‡].

Ако ключовете не са позиционни, очевидно няма решение дори при само два ключа и две лампи – каквото и да направим с ключовете, ако при наблюдението се окаже, че нито една от лампите не свети, или и двете светят, няма как да разберем кой ключ на коя лампа съответства без допълнителни експерименти. И така, допускаме, че ключовете са позиционни.

Но дори при позиционни ключове не изглежда да има решение с един експеримент за три лампи и три ключа. Има четири възможности за лампите: и трите светят, точно две светят, точно една свети и нито една не свети. Няма смисъл да поставяме ключовете в положения, при които нито една лампа не свети, защото тогава не ни трябва да правим наблюдение – ние знаем какво ще видим, а именно, че нито една от лампите не свети. Аналогично, няма смисъл да палим и трите лампи. Остава да запалим една лампа или две лампи. Но и в двата случая при последващото наблюдение няма да решим задачата, въпреки че ще получим някаква информация. Примерно, ако пуснем точно една лампа с ключ едно, при наблюдението ще разберем ключ едно на коя лампа съответства, но ни трябва още един експеримент, за да разберем кой от ключове две и три на коя от останалите две лампи съответства.

“Решението” на задачата се оказа дразнещо глупаво: с точно един от ключовете палим лампа (не знаем коя, понеже не сме извършили наблюдение, но знаем, че я палим, понеже ключовете са позиционни), държим я запалена известно време, след което я гасим, палим друга лампа с някой от другите два ключа и бързо отиваме в стаята с лампите. Точно една от лампите свети—това е тази, която пуснахме с второто пускане на ключа—и точно една от несветещите лампи е гореща заради това, че с първото пускане на ключ сме я били включили; по този начин разбираме и първият ключ, с който запалихме лампа (която после угасихме), на коя лампа съответства – на горещата измежду двете несветещи. Решението е глупаво, понеже “чупи” модела, който мислено сме изградили, а именно, че разрешените действия са манипулиране на ключовете и наблюдения на лампите. Има много сценарии, в които това “решение” не работи: може лампите са на десетметров таван и не можем да ги достигнем с ръка, може лампите да светят със студена светлина и да не се нагряват осезаемо при работа, може лампите да са в гореща среда, примерно гореща фурна, в която всички лампи са горещи, може коридорът между двете стаи да е толкова дълъг, че лампата да изстине, докато стигнем до нея, и така нататък.

Допълнение 69: Thinking outside the box – добро или лошо?

Посоченото решение на задачата с трите ключа и лампи, което си позволих да нарека “глупаво”, е типично нестандартно решение, за чието измисляне е необходимо да “излезем от рамката”, което на английски се нарича *thinking outside the box* или *thinking out*

[†]Тази конвенция идва от шалтерите в електротехниката. Тъй като гравитацията действа винаги, шалтерът може да падне сам, но не може да се вдигне сам. Много по-добре е непредвидено падане на шалтера да прекъсне електрическата верига, което може да остави някакви хора временно без ток, отколкото да затвори веригата и да убие човек, който в момента работи по нея и разчита, че шалтерът я е прекъснал.

[‡]Някои фарове на автомобили са с такива ключове.

of the box. Това понятие, поначало смислено, се е превърнало в досадно клише, като битува разбиране, че излизането от рамката е непременно нещо добро и положително, така че трябва да излизаме от рамката колкото е възможно по-често, едва ли не постоянно, за да сме оригинални, креативни и неповторими.

Всъщност, излизането от рамката може да е положително, а може и да не е. **Всяка** словесна задача без изключение предполага някакъв модел, като решението на задачата е съгласно конвенциите и ограниченията на модела; решението е **в рамката на модела**, а не извън нея. По правило този модел е имплицитен и се подразбира както от автора на задачата, така и от решаващия задачата. Примерно, в основното училище решавахме задачи от типа на

“Влак тръгва в 10:00 ч. от град А за град Б с 80 км/ч. В 11:00 ч. от Б за А тръгва друг влак с 60 км/ч. Намерете разстоянието между А и Б, ако влаковете се срещат в 12:00 ч.

Решението 220 км. се намира наум. Обаче, за да намерим това решение, ние се “облегнахме” на допускания, които не са споменати изрично:

- всеки от влаковете развива съответната скорост мигновено, за нула време, което в реалния свят е невъзможно;
- влаковете не спират на междинни гари;
- веднъж достигнал съответната скорост, всеки влак се движи абсолютно равномерно, което, ако говорим педантично, е невъзможно за реални влакове.

Решението 220 км. е някаква идеализация, апроксимация на някаква реална ситуация. А какво бихте казали за такова “решение”:

Влаковете всъщност не са срещнали, защото мотрисата на първия влак се е запалила и пътниците са били прехвърлени в автобуси, а вторият влак е заседнал на междинна гара поради кражби на семафори.

Това “решение” вече излиза напълно извън рамката, която се подразбира за задачата, въпреки че съвсем не е изключено в живота да стане именно това.

Такива драстични излизания от рамката на модела обезсмислят всяка словесна задача и изобщо всяка задача, чието решение изисква спазване/съблюдаване на правила. Като друг пример, шахматните задачи от типа “Намерете мат в четири хода за белите” се обезсмислят от решения, при които играчът с черните фигури прибегва към насилие, виждайки, че губи. Не че е невъзможно на практика да стане точно това и със сигурност историята познава такива случаи, но задачата е дадена при допускането, че се играе по правилата на шаха и се признават само решения по правилата на шаха. Аналогично, недопустимо е да се предлага решение на задачата за влаковете, което излиза от рамките на разумния имплицитен модел.

Трябва ясно да различаваме решаването на теоретична задача съгласно някакви правила, в рамките на даден модел, от решаването на житейска задача, в която наистина трябва да мислим “извън кутията”, защото правила няма. Последното важи особено силно за компютърната сигурност. Често теорията разглежда модел, в който комуникират две страни, да кажем А и Б, по информационен канал, който е общодостъпен и

който се следи от подслушвач. Подслушвачът следи пасивно какво минава по канала или активно прехваща връзката, подменяйки информация. Много от задачите в областта на компютърната сигурност са в рамките на този модел и решенията се основават на неговите ограничения. В реалния свят обаче нещата са много, много по-сложни и действията на противника може да не се ограничат до подслушване или прехващане на комуникацията! Ако информацията е от огромно значение и противникът е мощен и мотивиран да я получи на всяка цена, доста сигурно е, че той няма да спазва ограниченията на модела, а ще прибегне до подкупване, изнудване, проникване с взлом, откровено насилие и така нататък. Потърсете с гугъл за *rubber hose crypto* за повече информация. В по-кратки и по-малко драматични сценарии, излизането извън модела по отношение на сигурността става не чрез прибегване към насилие, а чрез използване на, например, *hardware backdoors*.

Потенциалната възможност да се излезе от рамката на модела, който допускаме, и да започне да се действа по неочакван начин, колкото и да е житейски важна, не обезсмисля задачите, които са поставени в рамката на даден модел и чието решение трябва да е съобразено с модела. Умението да решаваме задачи в рамките на моделите е много важно и ценно:

- първо, защото е възможно житейската ситуация да се развие по, или близо до, модела; в примера с комуникацията, възможността противникът да прибегне към насилие е по-скоро изключение, а backdoors по правило се използват от малко на брой, супермощни агенти, срещу които нямаме шанс така или иначе,
- второ, защото решенията в рамката на модела показват аналитични способности, за разлика от много от “решенията”, чупещи модела.

Да се върнем към задачата за ключовете и лампите. Моделът, който е нормално да допуснем, е следният: имаме право само да нагласим ключовете в някакви положения, после да извършим наблюдение, да правим изводи от наблюдението, въз основа на тях да променим някои от ключовете, пак да правим наблюдение, и така нататък, докато не разберем кой ключ коя лампа пуска/спира. Решението с пипане на несветещите лампи, за да се разбере коя от тях е била пусната допреди малко, излиза напълно извън този модел; иначе казано, чупи модела. В някои житейски ситуации то може да е полезно, в други да е неприложимо (лампите са на висок таван, примерно), но по-важното е, че чупи модела. Ако започнем да чупим модела, може да стигнем до екстремни “решения”, в които разбираме кой ключ към коя лампа е изобщо без наблюдения. Възможностите за това са неограничени:

- изкъртваме с кирка мазилката, за да проследим кабелите в стената,
- питаме електротехника (или който и да е човек, знаещ свързането),
- подкупваме електротехника,
- изнудваме електротехника,
- намираме планове на сградата,
- и така нататък.

Започнем ли веднъж да чупим модела, спиране няма.

За да избегнем чупене на модела, може да направим условието огромно, с много ограничения, изредени експлицитно, надявайки се така да предотвратим екстаравагантни решения и оставяйки като възможно само решението, което имаме предвид. Но ако решаващият задачата е решил сериозно да чупи модела (или просто да се лигави), многото ограничения в условието няма да го спрат. Колкото и прецизно и подробно да е условието на словесна задача, винаги може да се намери интерпретация, която е малко или много различна от тази, която е имал предвид този, който е измислил задачата и която интерпретация води до решение, различно от оригиналното.

В резюме, всяка словесна задача изисква изготвяне на модел и решение в рамката на този модел. При достатъчно “добро желание” винаги може да се намери интерпретация, при която се изготвя различен модел и решението в неговата рамка има малко общо с решението на този, който дава задачата. За задачите, които не изискват правене на модел, това не е в сила. Например, задачата да се реши дадено рекурентно уравнение не изисква правене на модел и при нея този, който решава, няма възможност да шикалкави: или решава уравнението, или не го решава. За разлика от това, ако е дадена словесна задача, която трябва да се моделира с рекурентно уравнение, има неограничени възможности за шикалкаване при достатъчно “добро желание”.

И най-важното. За **житейски** задачи мисленето “извън кутията”, тоест, чупенето на общоприетия модел, може да е изключително полезно. Но от това не следва, че решаването на **теоретични** задачи при стриктно съблюдаване на рамката на модела е безполезно или второкачествено, нито че чупенето на модела при теоретичните задачи е смислено.

И така, задачата няма решение с един експеримент при три ключа и лампи, ако съблюдаваме ограниченията на разумния модел.

Нека n е броят на ключовете/лампите. При $n = 3$ и $n = 4$ има решение с два експеримента. При $n = 5$ не изглежда да има решение с два експеримента, но има решение с три експеримента. И така нататък. Изглежда, че в общия случай има решение с $\lceil \log_2 n \rceil$ експеримента, но не с по-малко от това. Решението е много просто и ще го опишем с думи. Нека $n = 2^m$, за някое $m \in \mathbb{N}$. Ако $m = 0$, има един ключ и една лампа и този ключ пали тази лампа, което заключаваме без да правим експерименти; тоест, нула експерименти. Ако $m > 1$, нека множеството от ключовете е $K = \{k_1, \dots, k_n\}$, а множеството от лампите е $L = \{\ell_1, \dots, \ell_n\}$. Разбиваме K на две равномошни множества, да кажем $K_1 = \{k_1, \dots, k_{n/2}\}$ и $K_2 = \{k_{(n/2)+1}, \dots, k_n\}$. Поставяме ключовете от K_1 в положение “запалено”, поставяме ключовете от K_2 в положение “загасено”, отиваме в стаята с лампите и наблюдаваме. Точно $n/2$ лампи светят. Нека множеството от светещите лампи е L_1 , а от останалите лампи е L_2 . Сега вече знаем, че ключовете от K_1 съответстват на лампите от L_1 , а тези от K_2 , на лампите от L_2 . Решаваме задачата рекурсивно върху (K_1, L_1) и (K_2, L_2) . Това решение по същество е по схемата разделяй-и-владей, само че, ако си го представяме като алгоритъм, той е паралелен алгоритъм, защото с едно действие научаваме множеството лампи, съответни на K_1 , и после решенията за (K_1, L_1) и (K_2, L_2) се получават паралелно.

В тази лекция обаче ние се интересуваме от долни граници, така че за нас е по-важен въпросът, защо задачата не може да се реши с по-малко от логаритмичен брой експерименти. Ще докажем асимптотична логаритмична долна граница за броя на експериментите. Ще използваме дърво за вземане на решение.

Лесно се вижда, че това, което се иска в задачата, е да се намери точно една пермутация измежду $n!$ пермутации: при фиксирано изброяване на ключовете $k_1, \dots, k_n$, това, което се иска, е пермутация $\pi: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, такава че k_i пали лампа $\ell_{\pi(i)}$, за $1 \leq i \leq n$. На всяка схема, която е серия от “разцъкване” на лампи и последващо наблюдение, съответства дърво за вземане на решение. Листата на дървото са поне $n!$, защото то изчислява една пермутация измежду $n!$. Да видим каква е разклонеността му. За разлика от дърветата за вземане на решение, които разгледахме досега, това дърво има разклоненост, която не е константа, а е функция на n . Разклонеността съответства на броя на различните резултати, които можем да получим от едно наблюдение. Колкото по-бързо растяща е тази функция, толкова по-малка е височината на дървото и съответно долната граница за задачата е по-ниска.

При n различни лампи, възможните резултати от наблюденията са не повече от 2^n , понеже всяка лампа може да свети или да не свети. Следователно, разклонеността на дървото е най-много 2^n . Ако $h(n)$ е височината на дървото, то

$$h(n) \geq \log_{2^n} n!$$

Това не е грешка – основата на логаритъма е 2^n . В сила е

$$\log_{2^n} n! = \frac{\log_2 n!}{\log_2 2^n} = \frac{\Theta(n \lg n)}{n} = \Theta(\lg n)$$

Показахме, че $h(n) = \Omega(\lg n)$. Следователно, броят на експериментите е поне логаритмичен, в асимптотичния смисъл.

13.3 Редукции

13.3.1 Долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ

Тази изчислителна задача въведохме в Подсекция 4.2.2. Да си я припомним.

Изч. Задача: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ

екземпляр: Числа $a_1, a_2, \dots, a_n$.

решение: Минимално $|a_i - a_j|$ за $1 \leq i < j \leq n$.

Това е едномерният вариант на задачата: входът се състои от числа (а не от наредени двойки числа, примерно). Без съмнение съществува директно доказателство на долната граница $\Omega(n \lg n)$ за тази задача с дърво за вземане на решение. Но, както видяхме при задачата EU, директното доказателство може да е дълго и тежко. Тук ще извършим *редукция*: ще редуцираме задачата EU до задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ. По този начин ще използваме наготово резултата за долна граница на EU.

Редукцията се прави така. Нека ALGCP е произволен алгоритъм за задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ (базиран на сравнения, естествено). Да разгледаме следния алгоритъм AlgX.

```

ALGX(Числа  $a_1, a_2, \dots, a_n$ )
1   $d \leftarrow \text{ALGCP}(a_1, a_2, \dots, a_n)$ 
2  if  $d = 0$ 
3      return FALSE

```



```

4  else
5      return TRUE

```

Очевидно ALGX решава задачата EU. Нещо повече. Нека сложността по време на ALGX е $T(n)$. Извън извикването на ALGCP на ред 1, ALGX извършва само константна допълнителна работа (присвояването на ред 1 и редове 2–5). Тогава

$$T(n) = \Theta(1) + S(n)$$

където $S(n)$ е сложността по време на ALGCP. Но ние знаем, че $T(n) = \Omega(n \lg n)$, защото има долна граница $\Omega(n \lg n)$ за EU, ако се ползва алгоритъм, базиран на сравнения. Тогава трябва да е вярно, че $S(n) = \Omega(n \lg n)$. Но $S(n)$ е произволен алгоритъм за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, базиран на сравнения. Заключаваме, че има долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ако се ползва алгоритъм, базиран на сравнения.

В частност, ако допуснем, че ALGCP има сложност $o(n \lg n)$, да кажем $\Theta(n \lg \lg n)$, то и ALGX щеше има сложност $\Theta(n \lg \lg n)$, от което директно следва, че EU има горна граница $\Theta(n \lg \lg n)$, което е несъвместимо с вече доказаната долна граница $\Omega(n \lg n)$ за EU.

Една забележка. В предния параграф се казва “в частност”, защото негацията на съжденияето

всеки алгоритъм, базиран на сравнения, за EU, има долна граница $\Omega(n \lg n)$

не е

съществува алгоритъм, базиран на сравнения, за EU, който има горна граница $o(n \lg n)$

Причината е обяснена подробно в Допълнение 70.

Допълнение 70: $f(n) \neq \Omega(g(n))$ не влече $f(n) = o(g(n))$

Какво следва от факта, че $f(n) \neq \Omega(g(n))$? С други думи, какво влече $f(n) \not\geq g(n)$? Читателят може да се изкуши да помисли, че тогава непременно $f(n) < g(n)$, по аналогия с импликацията $x \not\geq y \rightarrow x < y$ върху реалните числа. Само че $\geq$ и $<$ са релации върху функции и при тях нещата са по-сложни. Импликация $f(n) \neq \Omega(g(n)) \rightarrow f(n) \prec g(n)$ **няма**.

Съгласно Теорема 24, има точно два случая, в $f(n) \not\geq g(n)$ и $f(n) < g(n)$:

1. (2.52), тоест функциите са асимптотично несравними;
2. (2.53), тоест $f(n) \leq g(n)$ и само това.

Тези случаи обаче са екзотика, що се отнася до функции, които описват алгоритмична сложност. За да бъдат функциите асимптотично несравними или едната да изостава от, и после да догонва, другата, и така до безкрай, едната или двете функции трябва да имат “подскачащ характер”. Обикновено срещашите се в практиката функции, описващи алгоритмична сложност, като n , $n \lg n$, n^2 , 2^n и така нататък не са такива. И така, за нормално държащите се, срещашите се на практика функции е вярно, че $f(n) \not\geq g(n)$ влече $f(n) < g(n)$.

Казваме, че задачата ЕУ *се редуцира* до задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ в смисъл, че задачата ЕУ има решение, което се състои в решение на задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, плюс пренебрежимо малко допълнителна работа. Накратко и не особено формално казано:

задача Х се редуцира до задача Y, ако алгоритъм за задача Y може да се използва за решаване на задача Х.

Засега ще се задоволим с това разбиране за “редукция”. В Секция 14.6 ще разгледаме редукциите много по-подробно и прецизно.

Грубо казано, от факта, че P_1 се свежда до P_2 правим два извода:

- Ако P_1 е трудна задача (каквото и да значи това), то P_2 е трудна. Няма как някаква вътрешна, иманентна сложност да изчезне при редукцията.
- Ако P_2 е лесна (каквото и точно да значи това), то P_1 е лесна. По същата причина: не може иманентната сложност да изчезне при редукцията.

Важно е да знаем, че следните изводи не са верни.

- Ако P_1 е лесна, то P_2 е лесна. Такъв извод няма. Винаги можем да направим нещата произволно сложни.
- Ако P_2 е трудна, то P_1 е трудна. Такъв извод няма. Причината отново е, че винаги можем да направим нещата сложни.

Наблюдение 88

Трансформирайки задача в задача, можем да добавим колкото си искаме допълнителна сложност отгоре. Това, което е невъзможно, е при такива трансформации да намалим иманентната сложност.

13.3.2 Долна граница $\Omega(n \lg n)$ за МОДА

Да си припомним определението на “мода” от Подсекция 4.2.5. Формално, изчислителната задача МОДА е следната.

Изч. Задача 58: МОДА

екземпляр: Числа $a_1, a_2, \dots, a_n$.

решение: Най-често срещано число измежду дадените.

Ще докажем долна граница $\Omega(n \lg n)$ за МОДА. Естествено, имаме предвид намиране на мода чрез сравнения. Нека ALGMODE е произволен алгоритъм за задачата МОДА, базиран на сравнения. Той връща стойността на някоя мода. Да разгледаме следния алгоритъм AlgY.

```

ALGY(Числа  $a_1, a_2, \dots, a_n$ )
1   $m \leftarrow \text{ALGMODE}(a_1, a_2, \dots, a_n)$ 
2   $c \leftarrow 0$ 
3  for  $i \leftarrow 1$  to  $n$ 
4      if  $a_i = m$ 
5           $c \leftarrow c + 1$ 

```



```

6  if c = 1
7      return TRUE
8  else
9      return FALSE

```

Очевидно ALGY решава задачата EU. Нещо повече. Нека сложността по време на ALGY е $T(n)$. Извън извикването на ALGMode на ред 1, ALGY извършва само линейна допълнителна работа. Тогава

$$T(n) = \Theta(n) + S(n)$$

където $S(n)$ е сложността по време на ALGMode. Но ние знаем, че $T(n) = \Omega(n \lg n)$, защото има долна граница $\Omega(n \lg n)$ за EU, ако се ползва алгоритъм, базиран на сравнения. Тогава трябва да е вярно, че $S(n) = \Omega(n \lg n)$. Но $S(n)$ е произволен алгоритъм за МОДА, базиран на сравнения. Заклучаваме, че има долна граница $\Omega(n \lg n)$ за МОДА, ако се ползва алгоритъм, базиран на сравнения.

13.3.3 Долна граница $\Omega(n \lg n)$ за 2SUM във версия за броене

В Подсекция 4.2.7 въведохме задачата kSUM в два варианта: като задача за разпознаване и като задача за търсене. Тук имаме предвид версията за търсене при $k = 2$, но се търси броят на двойките, сумиращи се до нула, а не се търси двойка, сумираща се до нула. Поради това ще кажем, че версията е за броене.

Изм. Задача: 2SUM, версия за броене

екземпляр: Масив от числа $A[1..n]$.

решение: Броят на наредените двойки (i, j) , такива че $1 \leq i < j \leq n$ и $A[i] + A[j] = 0$.

Нека ALG2SUMCOUNT е произволен алгоритъм за задачата 2SUM във версия за броене, базиран на сравнения. Да разгледаме следния алгоритъм AlgZ.

ALGZ($A[1..n]$: масив от положителни числа)

```

1  B ← [-A[1], -A[2], ..., -A[n], A[1], A[2], ..., A[n]]
2  sum ← ALG2SUMCOUNT(B)
3  if sum = n
4      return ДА
5  else
6      return НЕ

```

Ще покажем, че ALGZ решава задачата EU. Първо да допуснем, че в $A[1..n]$ няма повторения. Тогава единствените двойки различни елементи от B, които се сумират до 0, са двойките $(-A[i], A[i])$, за $1 \leq i \leq n$. Тогава sum на ред 2 получава стойност n и ALGZ връща ДА на ред 4.

Сега да допуснем, че съществуват $i_1, \dots, i_k$, такива че $k > 1$ и $1 \leq i_1 < \dots < i_k \leq n$ и $A[i_1] = \dots = A[i_k]$. На тези k на брой еднакви елементи от A съответстват $2k$ на брой елементи в B с еднаква абсолютна стойност, половината отрицателни и другата половина положителни. Всеки от отрицателните се сумира до нула с всеки от положителните, така че на въпросните k еднакви елементи от A съответстват k^2 различни двойки елементи с нулева сума в B. Тогава в B има поне $n - k + k^2$ двойки елементи, сумиращи се до нула, като

$n - k + k^2 > n$ при $k > 1$. С други думи, броят на двойките различни елементи от B , които се сумират до 0, е по-голям от n . Тогава sum на ред 2 получава стойност, по-голяма от n , и ALGZ връща НЕ на ред 6.

И така, ALGZ решава задачата EU. Нека сложността по време на ALGZ е $T(n)$. Извън извикването на ALG2SUMCOUNT на ред 2, ALGZ извършва само линейна допълнителна работа. Тогава

$$T(n) = \Theta(n) + S(n)$$

където $S(n)$ е сложността по време на ALG2SUMCOUNT . Но ние знаем, че $T(n) = \Omega(n \lg n)$, защото има долна граница $\Omega(n \lg n)$ за EU, ако се ползва алгоритъм, базиран на сравнения. Тогава трябва да е вярно, че $S(n) = \Omega(n \lg n)$. Но $S(n)$ е произволен алгоритъм за 2SUM, ВЕРСИЯ ЗА БРОЕНЕ, базиран на сравнения. Заклучаваме, че има долна граница $\Omega(n \lg n)$ за 2SUM, ВЕРСИЯ ЗА БРОЕНЕ, ако се ползва алгоритъм, базиран на сравнения.

13.3.4 Долна граница $\Omega(n \lg n)$ за INTERVAL SCHEDULING.

Става дума за Задача 43. При определени ограничения, задачата СОРТИРАНЕ се свежда до нея. Нека разглеждаме само алгоритми за INTERVAL SCHEDULING, които връщат не само стойността на оптималното решение, а и самото решение. И го връщат не като множество, а като **редица** от интервали, подредени по начина, по който са подредени във времето.

Ако имаме такъв алгоритъм, можем да го ползваме като процедура, за да сортираме n на брой, две по две различни, **цели** числа $a_1, \dots, a_n$.

- Първо построяваме във време $\Theta(n)$ наредените двойки $(a_1, a_1 + \epsilon), \dots, (a_n, a_n + \epsilon)$ за някакво достатъчно малко ϵ . Тези наредени двойки представляват интервалите.
- Върху тези интервали пускаме хипотетичния алгоритъм за INTERVAL SCHEDULING.

Тъй като цялото множество от тези интервали е безконфликтно, този алгоритъм връща редицата от всички интервали, сортирани по първо число, и оттук намираме сортираната редица от входните числа. Ерго, този алгоритъм сортира числата $a_1, \dots, a_n$. Но, както знаем, това става във време $\Omega(n \lg n)$. Тъй като първата фаза е линейна, трябва втората фаза да се изпълнява за време $\Omega(n \lg n)$.

13.3.5 Относителна долна граница: DEGENERACY TESTING е 3SUM-hard

В тази подсекция ще видим пример за *условна*, или *относителна*, долна граница на задача. Задачата е DEGENERACY TESTING (вижте надолу). Долната граница за нея е същата като на 3SUM. Тук говорим за 3SUM във версия за разпознаване.

Дълго време се е смятало, че 3SUM има долна граница $\Omega(n^2)$. Това е доказано през 1999 г. от Jeff Erickson [46], но за доста слаб изчислителен модел. През 21 век са открити няколко алгоритъма за 3SUM със сложност $o(n^2)$ (в нормалния изчислителен модел), най-бързият от които е със сложност $O\left(n^2 \cdot \frac{\lg \lg n}{\lg n}\right)$ [59]. Текущата хипотеза (недоказана обаче!) е, че няма алгоритъм за 3SUM със сложност $O(n^{2-\epsilon})$ за никоя положителна константа ϵ в нормалния изчислителен модел.

Каква е истината за долната граница на 3SUM, ние не знаем. Но ще видим една лесна редукция на 3SUM до DEGENERACY TESTING. От съществуването на тази редукция следва,

че всяка долна граница за 3SUM е и долна граница за DEGENERACY TESTING, стига да е по-висока, в асимптотичния смисъл, от $n^{2-\epsilon}$ за всяка положителна константа ϵ .

Една полезна дефиниция, която конкретизира какво разбираме под “редукция”.

Определение 111: 3SUM-hard

Нека P е изчислителна задача за разпознаване. Казваме, че P е 3SUM-hard, ако произволен екземпляр с размер n на 3SUM може да бъде решен чрез

1. решаване на P върху константен брой екземпляри (на P), всеки с размер $O(n)$,
2. и още $O(n^c)$ допълнителна работа, където c е константа, такава че $0 < c < 2$.

Сравнете тази дефиниция с дефиницията на “NP-hard” от Лекция 14. Интуитивно, “ P е 3SUM-hard” казва “ P е поне толкова трудна, колкото 3SUM”.

Забележете, че ако P е 3SUM-hard, то 3SUM се свежда до P , но “ P е 3SUM-hard” казва нещо доста конкретно и добре дефинирано, предвид Определение 111, за разлика от неформалното и не особено конкретно “3SUM се свежда до P ”.

Изч. Задача 59: DEGENERACY TESTING

екземпляр: $p_1, p_2, \dots, p_n$: точки в равнината

въпрос: Дали поне три от точките са колинеарни?

Ще докажем, че DEGENERACY TESTING е 3SUM-hard. Първо един помощен факт.

Лема 47

За всеки три реални числа a, b и c , две по две различни, е вярно, че точките (a, a^3) , (b, b^3) и (c, c^3) са колинеарни тстк $a + b + c = 0$.

Доказателство: В едната посока, нека точките (a, a^3) , (b, b^3) и (c, c^3) са колинеарни. Тогава те лежат върху права с уравнение $y = \alpha x + \beta$. Тогава a, b и c са корени на $x^3 = \alpha x + \beta$. Но това е кубично уравнение, така че трябва да е вярно

$$x^3 - \alpha x - \beta = (x - a)(x - b)(x - c)$$

Но коефициентът пред x^2 вляво е нула. Следователно, вдясно коефициентът пред x^2 също е нула. Но вдясно коефициентът пред x^2 е $-(a + b + c)$. Тогава $a + b + c = 0$.

В обратната посока, нека $a + b + c = 0$. Да разгледаме $(x - a)(x - b)(x - c)$. Отваряйки скобите, ще получим $x^3 - \alpha x - \beta$ за някакви реални α и β , тъй като коефициентът пред x^2 е нула, което на свой ред се дължи на факта, че $a + b + c = 0$.

Тогава a, b и c са корените на уравнението $x^3 - \alpha x - \beta = 0$. Но тогава точките (a, a^3) , (b, b^3) и (c, c^3) са колинеарни, понеже лежат върху правата $y = \alpha x + \beta$. $\square$

Теорема 111: Задачата DEGENERACY TESTING е 3SUM-hard

DEGENERACY TESTING е 3SUM-hard.

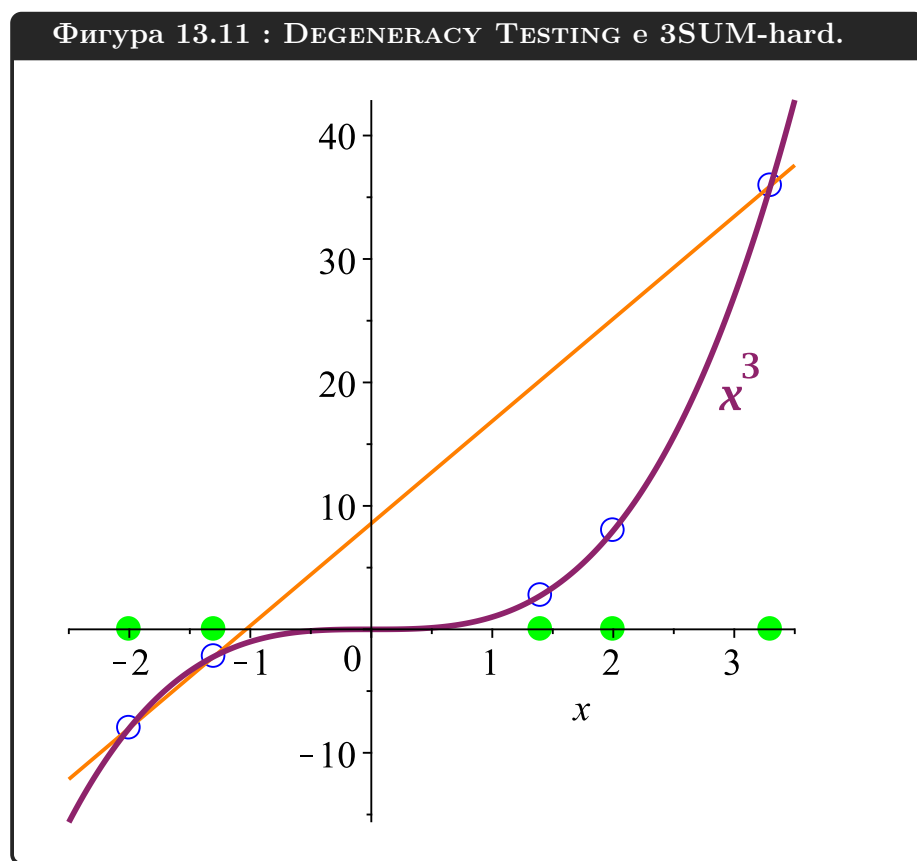
Доказателство: Нека е даден екземпляр на 3SUM: множество от реални числа $\{s_1, \dots, s_n\}$. От него конструираме екземпляр $\{(s_1, s_1^3), \dots, (s_n, s_n^3)\}$ на DEGENERACY TESTING. Съгласно

Лема 47, отговорът на въпроса “Дали има поне три точки от $\{(s_1, s_1^3), \dots, (s_n, s_n^3)\}$, които са колинеарни?” е същият като отговорът на въпроса “Дали има три елемента на $\{s_1, \dots, s_n\}$, които се сумират до нула?”. Конструкцията работи достатъчно бързо според изискването на Определение 111, а именно във време $O(n^c)$ за $c = 1$. Броят на екземплярите на DEGENERACY TESTING, които конструираме, е само един, така че изпълняваме и изискването решаването да става чрез константен брой екземпляри на задачата, към която свеждаме. $\square$

Фигура 13.11[†] илюстрира факта, че DEGENERACY TESTING е 3SUM-hard. Екземплярът на 3SUM е $\{-2, -1.3, 1.4, 2, 3.3\}$. Нему съответстват точките в зелено върху абсцисата. Той е ДА-екземпляр, понеже $-2 + (-1.3) + 3.3 = 0$. От този екземпляр на 3SUM строим следния екземпляр на DEGENERACY TESTING:

$$\{(-2, -8), (-1.3, -2.197), (1.4, 2.744), (2, 8), (3.3, 35.937)\}$$

Нему съответстват точките, нарисувани със сини кръгчета и лежащи върху тъмно лилавата крива $y = x^3$. Той е ДА-екземпляр на DEGENERACY TESTING, понеже трите точки $(-2, -8)$, $(-1.3, -2.197)$ и $(3.3, 35.937)$ са колинеарни. Правата в оранжево показва, че тези точки са колинеарни.



[†]Фигура 13.11 е направена с Maple(TM).

13.4 Аргументиране чрез противник (Adversary argument)

13.4.1 Неформално обяснение

Вие искате да решите някаква задача върху някакви входни данни. Вие обаче нямате директен достъп до входа. Входните данни се държат от друг субект, когото наричаме условно “дявола”. Дяволът позволява да му задавате въпроси за данните, и то адаптивно: задавате въпрос, той отговаря, а какъв ще бъде следващият Ваш въпрос (може да) зависи от получения отговор. Дяволът може да знае, но може и да не знае алгоритъма, който следвате. Какви са въпросите, които можете да задавате, зависи от правилата на играта. Може да са въпроси с булев отговор, примерно дали измежду две числа от данните, едното е по-малко от другото. Може да не са с булев, а с числен отговор, примерно, каква е стойността на число от входа, или каква е сумата на някакви числа от входа; такива въпроси е по-удачно да бъдат наречени “достъпи до входните данни”.

Лошотията на дявола се изразява единствено в това, че той Ви кара да зададете максимален брой въпроси/да направите максимален брой достъпи. Всеки отговор на дявола е такъв, че оттук насетне Ви се налага да зададете максимален брой въпроси, за да получите решение. Дяволът е **неограничено умен**. По-академично казано, той **има неограничени изчислителни способности** и във всеки момент знае точно каква информация Ви е разкрил досега и какъв отговор да даде на следващия Ви въпрос, за да Ви затрудни максимално. Както вече казахме, дяволът може да знае алгоритъма, който следвате, при което той знае всеки следващ Ваш въпрос, но може и да не го знае, при което той трябва да има предвид всяка възможна редица от следващи Ваши въпроси. Подчертаваме, че дяволът **няма магически способности** и, ако не знае Вашата стратегия за решаване на задачата, няма как да знае следващия Ви въпрос.

Дяволът трябва да е *консистентен*: той е длъжен да се ангажира с отговорите, които е дал до момента. Ако веднъж каже, примерно, че $a_5 < a_{10}$, всички негови следващи отговори трябва да са консистентни с това; той вече нито може да каже, че $a_5 \geq a_{10}$, нито може да даде отговори, от които следва, че $a_5 \geq a_{10}$. Това е много важно. Ако дяволът имаше свободата да дава неконсистентни отговори, то цялото усилие щеше да е безсмислено.

Забележете, че не се твърди, че дяволът не лъже! Това, което се твърди, че лъжите на дявола не трябва да си противоречат; иначе казано, в края на играта той не може да бъде изобличен като лъжец. Както ще стане ясно от изложението долу, **поначало вход няма**. Дяволът създава входните данни в процеса на отговаряне на Вашите въпроси така, че да са консистентни с отговорите, които Ви е дал. Ако след края на играта поискате от дявола да Ви покаже целия вход, той е длъжен да го направи. Но това, което той всъщност ще направи е, че в момента, в който поискате да видите входа, той ще генерира вход, който е консистентен с отговорите, които Ви е предоставил. Той се е постарал такъв вход да съществува. Ако тогава го попитате:

– Наистина ли това беше входът поначало?

Той ще излъже:

– Да, разбира се.

И може да добави:

– Нима не ми вярвате?!

Ето пример. Нека задачата е СОРТИРАНЕ, и то базирано на сравнения. Да кажем, че дяволът не знае алгоритъма, който изпълнявате. Вие задавате въпроси от вида $a_i \stackrel{?}{<} a_j$, а работата на дявола е да Ви накара да зададете максимален брой въпроси. Във всеки момент дяволът знае какви отговори е дал до момента, колко пермутации на n елемента са консистентни с досега дадените отговори и за всяко следващо запитване от вида $a_i \stackrel{?}{<} a_j$, дяволът отговаря или ДА, или НЕ въз основа на един единствен критерий: максимизиране на броя на пермутациите, които остават възможни[†]. В началото дяволът изобщо не си направил труда да генерира вход – вход не е имало. В края на играта, ако поискате от дявола да ви покаже входа, той ще изчисли вход, който е консистентен с всички отговори, които е дал, и ще излъже, че точно това е бил входът през цялото време.

Сега да конкретизираме примера. Нека задачата е СОРТИРАНЕ НА ТРИ ЕЛЕМЕНТА, базирано на сравнения. Да кажем, че трите елемента са уникални. Нека дяволът не знае Вашия алгоритъм. В този пример лошотията на дявола е в това, че ще Ви кара да зададете (поне) три въпроса. Очевидно, в някои случаи задачата може да се реши само с два въпроса, ако входът е подходящ за тях. Примерно, ако $a_1 < a_2 < a_3$ и въпросите са $a_1 \stackrel{?}{<} a_2$ и $a_2 \stackrel{?}{<} a_3$. Но при игра срещу дявола никога няма да “минете” само с два въпроса, защото дяволът ще генерира, по отношение на Вашите въпроси, вход, който е “неподходящ”. Вашият първи въпрос е или $a_1 \stackrel{?}{<} a_2$, или $a_1 \stackrel{?}{<} a_3$, или $a_2 \stackrel{?}{<} a_3$. Кой от тези въпроси ще зададете, това дяволът не знае. Дяволът обаче знае, че има точно 6 възможни пермутации на входа и за всеки от тези въпроси, точно 3 пермутации са съвместими с положителния отговор и точно 3, с отрицателния. Така че няма значение какво ще отговори на първото запитване; да кажем, че дяволът винаги отговаря с ДА на първия въпрос. На втория въпрос обаче дяволът не отговаря произволно! Тъй като след отговора на първия въпрос остават 3 възможни пермутации, то, какъвто и втори въпрос да зададете, съществува отговор—или ДА, или НЕ, в зависимост от конкретиката—такъв че поне две пермутации са съвместими и с първия, и с втория отговор. Е, дяволът ще даде точно този отговор на втория въпрос и по този начин ще Ви принуди да зададе и трети въпрос. Забележете, че той ще ви принуди да зададете и трети въпрос независимо от това, какви са първия и втория Ви въпрос. След като получите решението и играта приключи, дяволът лесно може да предостави такъв вход, който е консистентен с всички отговори, които е дал, така че не може да бъде уличен в лъжа.

Забележете, че ако дяволът не знае алгоритъма Ви, той е длъжен да допусне, че Вие изпълнявате максимално ефикасен алгоритъм. Няма смисъл от дявол, който допуска, че изпълнявате някакъв неефикасен алгоритъм, за да се опита да Ви прецака въз основа на тази неефикасност. Това може да се онагледява добре с шах. Да кажем, че играете шах с дявола и той няма представа колко сте добър/добра, каква стратегия предпочитате и така нататък. Тук дяволът няма за цел да проточи играта максимално, а просто да победи. Единственото смислено допускане е, че дяволът ще играе “на макс”, правейки оптимални ходове като за най-силен опонент. Много трудно се анализира игра с дявол, който нарочно играе субоптимално, разчитайки на Вашата слабост; примерно, за да може играта да приключи най-скоро.

Допълнение 71: Морски шах с дявола

Няколко игри се наричат *Морски шах* на български. Тук ще разгледаме една от тях (на английски е *Battleships*), която е удачна илюстрация на основната идея на игра срещу злонамерен противник, разполагащ с неограничена изчислителна мощ.

[†]Тоест, които са консистентни с дадените отговори.

Правилата са следните. Играят двама играчи. Всеки разполага с квадратна мрежа, да кажем 11×11 , с номерирани редове и колони, наречена *море*:

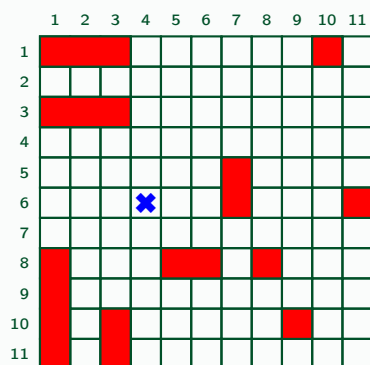
	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											

и няколко фигури, наречени *кораби*. Всеки кораб е правоъгълник, който може да бъде поставен, хоризонтално или вертикално, върху няколко съседни квадратчета от мрежата, запълвайки ги точно. Корабите са следните: един кораб 1×4 , два кораба 1×3 , три кораба 1×2 и четири кораба 1×1 . Всеки играч крие своето море от другия играч. Преди началото на играта, всеки играч разполага своите кораби в своето море както намери за добре, но скришно от другия играч. Има ограничение два кораба да не се припокриват и да не се докосват, нито по хоризонтал, нито по вертикал, нито по диагонал. Ето примерно разполагане на корабите на единия играч:

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											

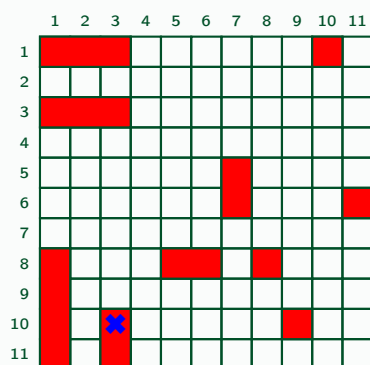
Самата игра се състои в редуващи се “изстрели” на всеки играч в морето на другия играч, като “изстрел” е просто казване на координатите на някое квадратче от морето на противника. Всеки играч записва изстрелите на противника си с хиксове върху своето море. Единият играч е първи. Той или тя казва на втория някаква наредена двойка (i, j) , където $1 \leq i, j \leq 11$. Вторият играч прави следното.

- Ако на (i, j) няма кораб (в нашия пример, нека $(i, j) = (6, 4)$), вторият казва “вода”, отбелязва хикс на (i, j) в своето море:

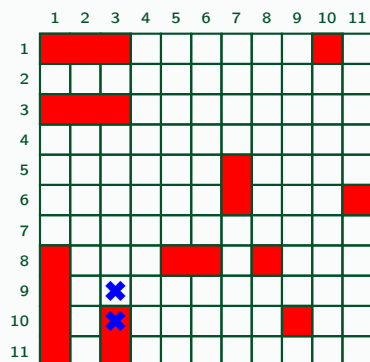


и на свой ред играе, казвайки координатите на някакво квадратче от морето на първия играч.

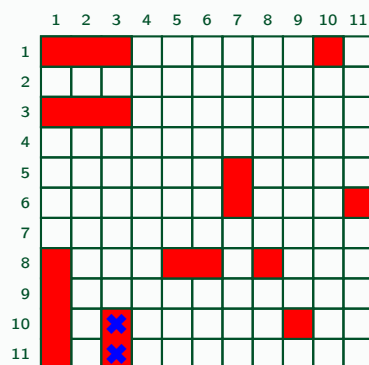
- Ако на това квадратче има кораб, който е по-голям от 1×1 и има поне едно неударено квадратче освен (i, j) (в нашия пример, нека $(i, j) = (10, 3)$), то вторият играч казва “удар” и отбелязва удара с хикс:



След това отново играе първият играч – този, който току-що нанесе удара. Първият продължава да играе, докато или не уцели вода:

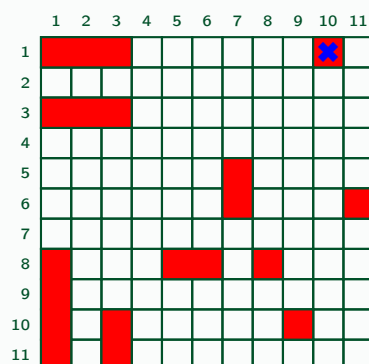


при което вече играе вторият играч, или не удари и последното досега неударено квадратче:

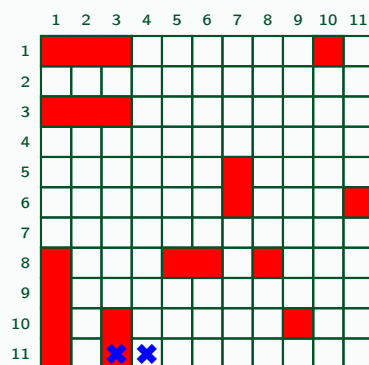


при което вторият играч казва “потопен” и първият играч продължава да играе.

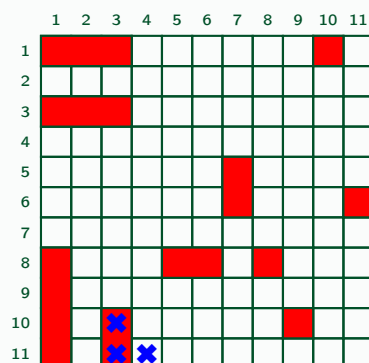
- Ако или на това квадратче има кораб 1×1 —в нашия пример, нека $(i, j) = (1, 10)$:



или (i, j) е единственото неударено квадратче на кораб, по-голям от 1×1 —в нашия пример, нека $(i, j) = (10, 3)$ и досега първият играч е направил тези два изстрела:

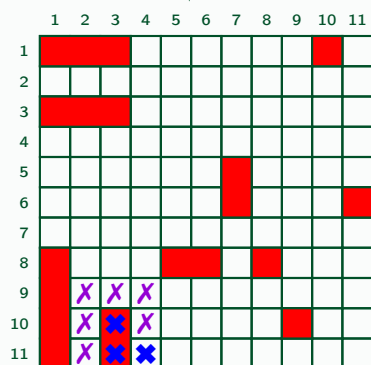


то вторият играч казва “потопен” и отбелязва мястото на удара с хикс.



После първият играч играе отново.

Печели играчът, който пръв или първа потопи всички кораби на противника си. Естествено, по време на играта няма смисъл играч да стреля върху квадратче, върху което вече е стрелял. Също така няма смисъл да стреля в района, ограждащ вече потопен кораб; ако разгледаме отново последния пример, след потапянето на кораба на $\langle(10, 3), (11, 3)\rangle$, всички квадратчета около него, в които досега не е стреляно, стават безсмислени като цели:



Това са правилата на играта. Сега да допуснем, че играете с дявола на морски шах. Почти сигурно ще загубите! Като добросъвестен и почтен човек, Вие поставяте корабите си в самото начало и после отговаряте вярно на изстрелите на противника, които улчват Вашите вече поставени кораби, с “удар” или “потопен”. Дяволът обаче не е почтен и със сигурност на Вашите начални изстрели, където и да са те, той ще отговори с “вода”. Същността на неговата стратегия е, че той изобщо не е поставил кораби. Дяволът не може да отговаря с “вода” прекалено дълго, защото съществува множество от изстрели от Ваша страна, които са така разположени, че неговите кораби нямат възможни позиции спрямо тях – но Вие с почти пълна сигурност няма да имате възможност да дадете толкова изстрели, защото ще сте загубили играта отдавна, играейки почтено.

Ако след загубата си поискате от дявола да ви покаже как са били разположени корабите му през цялото време, той ще изчисли такова разположение на корабите си, че нито един от Вашите изстрели не е попадение, и ще ви покаже него.

Имайте предвид, че дяволът е неограничено умен и ще отлага до последно първото попадение върху свой кораб. Както вече казахме, най-вероятно Вие ще сте загубили играта много преди да имате възможност да дадете толкова изстрели, че дяволът да не може повече да отлага първото попадение.

По друг начин казано: корабите общо заемат $4 + 6 + 6 + 4 = 20$ квадратчета, а морето е $11 \times 11 = 121$ квадратчета. Вероятността при първи изстрел да бъде ударен кораб, ако се играе почтено и кораби са разположени поначало, а даващия изстрел стреля по случайно квадратче, е $\frac{20}{121} \approx \frac{1}{6}$. Ако играете достатъчно дълго с дявола и се редувате за пръв изстрел, в горе-долу $\frac{1}{6}$ от игрите неговият пръв изстрел ще е попадение (удар или потапяне). Дяволът няма магически способности и не вижда корабите Ви^a, така че в около $\frac{5}{6}$ от игрите първият му изстрел ще е “вода”. Ще се изненадате обаче (ако не сте наясно, че играете срещу мошеник) от това, че **всеки** Ваш пръв изстрел е “вода”. Защото дяволът не слага кораби поначало. Нещо повече, Вие ще загубите преди да можете да реализирате попадение, така че **всеки** Ваш изстрел по време на **цялата** игра ще е “вода”.

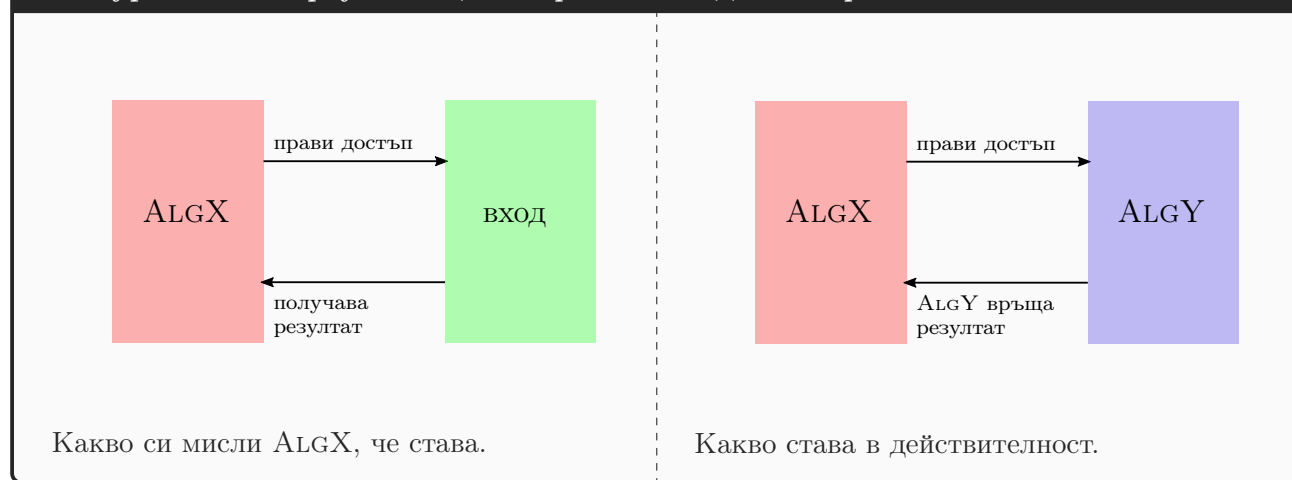
Примерът с морския шах не е перфектна илюстрация за даване на долна граница на изчислителна задача, понеже при решаване на изчислителна задача играете само Вие (само Вие задавате въпроси), а дяволът само отговаря; в морския шах Вие се редувате с дявола за задаване на въпроси. Въпреки тази разлика, би трябвало този пример да даде добра интуиция за стратегията на опонент, който е максимално злонамерен и има неограничена изчислителна мощ.

^aТова е същото като дяволът да не знае алгоритъма Ви. Да знае алгоритъма Ви включва да знае как сте си разположили корабите. Ако дяволът знаеше това, всеки негов изстрел би бил попадение, така че след първия му изстрел Вие бихте гледали как той играе до победа.

13.4.2 Формално определение

Долна граница за някакъв предварително известен алгоритъм $ALGX$. Разглеждаме $ALGX$ и друг алгоритъм $ALGY$, който играе ролята на противника (дявола). Всеки достъп на $ALGX$ до входните данни всъщност е едно запитване (query) от страна на $ALGX$ към $ALGY$. Може да си представяме, че $ALGY$ прехваща всеки опит на $ALGX$ за достъп до входа и той ($ALGY$) връща стойност, която, от гледна точка на $ALGX$, е легитимен резултат от достъпа до входа. Може още да си представяме, че $ALGX$ не знае за съществуването на $ALGY$ и няма как да разбере, че $ALGY$ прехваща всеки опит за достъп до входа и резултатът от достъпа е това, което $ALGY$ решава да предостави. И накрая, можем да мислим, че вход изобщо няма и $ALGY$ връща, каквото намери за добре. Фигура 13.12 илюстрира всичко това.

Фигура 13.12 : Аргументация с противник: два алгоритъма.



Основните допускания са следните.

- $ALGY$ разполага с $ALGX$ и оттам знае каква ще е редицата от достъпи до входа, които ще извърши $ALGX$.
- $ALGY$ е неограничено мощен. По отношение на $ALGY$ не се интересуваме от това, какви ресурси ползва като време и памет. С други думи, не ни интересува колко ефикасен е $ALGY$, стига да е ефективен.
- $ALGY$ връща такава редица от отговори на $ALGX$, че $ALGX$ да направи поне някакъв брой достъпи (queries), който брой е долната граница, която доказваме.

- ALG_Y трябва да е консистентен. Редицата резултати $\mathcal{R}$, които той дава на ALG_X , трябва да е такава, че да има поне един вход със съответната големина, за който редицата от резултатите е точно $\mathcal{R}$.

Дизайнерите на ALG_Y обаче сме ние. Нашата цел е да докажем долна граница за работата на ALG_X в най-лошия случай. За тази цел ние конструираме ALG_Y , който наричаме “противник”. Противникът е в състояние да накара ALG_X да извърши поне толкова достъпи, колкото е долната граница, която доказваме. Противникът не генерира вход поначало, с който би се ангажирал, а прави нещо много по-изтънчено и перфидно: той **създава входа в процеса на достъпи и връщане на резултати**, отчитайки конкретните достъпи, извършени от ALG_X , и данните, които ALG_X е получил до момента.

Да кажем, че ALG_Y кара ALG_X да направи максимален брой достъпи е прекалено амбициозно. В предната подсекция се твърдеше, че дяволът, бивайки неограничено умен и абсолютно злонамерен, връща отговори, които да ни накарат да работим максимално много. Когато конструираме противник-алгоритъм, целта ни е по-скромна, но затова пък постижима. Ние конструираме противник за доказване на някаква долна граница $f(n)$, където n е големината на входа. Ако конструираме противник ALG_Y , който кара ALG_X да извърши поне $f(n)$ достъпа за всяко n , и докажем това, ние сме си свършили работата. При това остава възможно да има друг противник ALG_Y' , който кара ALG_X да извърши дори повече достъпи; да кажем, $f'(n)$, където $f'(n) > f(n)$ за всички достатъчно големи n . Това, че може да има дори “по-лош” противник от този, който сме намерили и анализирали, не е в противоречие с нашето доказателство, че $f(n)$ е долна граница.

Наблюдение 89

Да допуснем, че сме доказали, че конструираният от нас противник кара ALG_X да прави поне $f(n)$ достъпа за всяко n . Ако освен това докажем, че $f(n)$ е **точна долна граница** за броя на достъпите, то ние сме конструирали оптимален противник, който точно отговаря на дявола от Подсекция 13.4.1. В противен случай просто сме конструирали противник, който върши работа за дадено доказателство, оставяйки възможността да има дори по-лош противник.

Долна граница за изчислителна задача П. Това е сравнително по-труден за конструиране и анализиране противник. Сега вече противникът ALG_Y не знае нищо за конкретния алгоритъм ALG_X , който решава П. От гледна точка на ALG_Y , ALG_X е просто черна кутия, от която излизат достъпи до входа (queries). Фигура 13.12 илюстрира адекватно и тази ситуация, но сега имаме предвид, че

- ALG_Y не знае нищо за ALG_X .
- “ ALG_X ” не означава конкретен алгоритъм, а означава кой да е алгоритъм за изчислителната задача П.

Допусканията за неограничена мощност, “злонамереност” и консистентност на ALG_Y са в сила и сега.

13.4.3 Долна граница $\lceil \log_2 n \rceil$ за намиране на число от $\{1, \dots, n\}$

Тук става дума за игра между играчи А и Б. Дадено е някакво цяло положително число n , което е известно и на А, и на Б. Б намисля някакво число $x \in \{1, \dots, n\}$, а А трябва да

познае числото. Единствените въпроси, които А може да задава на Б, са от вида “Дали x е по-малко от k ?”, за някакво естествено k .

Тъй като съдържанието на въпроса зависи изцяло от k , може да мислите, че А подава стойност k на Б, после Б отговаря Y или N , после А подава друга стойност k' на Б, после Б отговаря Y или N , и така нататък. По този начин идентифицираме редицата от въпроси с редицата $\langle k, k', \dots \rangle$.

Ще докажем, че в най-лошия случай, броят на въпросите за намиране на x е поне $\lceil \log_2 n \rceil$. Ще направим доказателството чрез аргументация с противник. Б е противникът и той ще накара А да зададе поне $\lceil \log_2 n \rceil$ въпроси, за да намери x . Както подобава на противник, Б е много умен и, естествено, злонамерен. Разбира се, поначало Б не намисля никакво x , а редицата от отговорите е такава, че да “прецака” А максимално. В края на краищата, А ще получи някакво число x —единственото, което е консистентно с въпросите и отговорите—но това ще коства на А поне $\lceil \log_2 n \rceil$ въпроса.

Нека $S \subseteq \{1, \dots, n\}$ е множеството от точно тези числа, всяко от които може да е x при получените досега отговори на въпроси.

В самото начало $S = \{1, \dots, n\}$, защото не са зададени никакви въпроси и x може да е кое да е число от $\{1, \dots, n\}$. Ерго, $|S| = n$ в самото начало.

След всеки въпрос на А “Дали x е по-малко от k ?” и получения отговор, $|S|$ може да намалее, но не може да нарастне. Естествено, ако А играе умно, $|S|$ със сигурност ще намалее, но дори А да играе глупаво[†], $|S|$ няма как да нарастне. Противникът Б преценява кой отговор— Y или N —води до по-голямо S . По този начин S може да намалее наполовина, но не повече.

За да може А да отговори, трябва $|S| = 1$. Ако $|S| > 1$ и А се опита да отговори, Б ще го опровергае моментално! Нека $|S| > 1$. За което и да е $c \in S$ играчът А да каже “ x е c ”, противникът Б ще отвърне на секундата “Всъщност x е d ”, за кое да е $d \in S \setminus \{c\}$. При това А няма как да опровергае Б.

Нека S_0 е началното S , а S_i е полученото S след i -ия въпрос. В сила е следното:

$$\begin{aligned} |S_0| &= n \\ |S_{i+1}| &\geq \left\lceil \frac{|S_i|}{2} \right\rceil, \text{ ако } i \geq 1 \end{aligned}$$

При това положение, минималната стойност за i , такава че $|S_i| = 1$, е $i = \lceil \log_2 n \rceil$.

13.4.4 Долна граница $n - 1$ за намиране на максималния елемент

Задачата е да се намери чрез, и само чрез, сравнения максимално число измежду произволни числа $a_1, a_2, \dots, a_n$. Итеративният алгоритъм MAX UNSORTED, ITER от Подподсекция 2.1.3.1 намира максимума на n числа, но не е оптимален като брой сравнения, защото ползва sentinel $-\infty$ за инициализация на временния максимум и оттам прави n сравнения. Ето как може да намерим максимума с $n - 1$ сравнения.

MAX UNSORTED($a_1, \dots, a_n$: реални числа)

```

1  max ← a1
2  for i ← 2 to n
3      if ai > max
```

[†]Примерно: първият въпрос е “Дали x е по-малко от 200?”, отговорът е Y , вторият въпрос е “Дали x е по-малко от 300?”. Това би било глупаво от страна на А.

```

4      max ←  $a_i$ 
5  return max

```

Сега ще покажем, че с по-малко от $n - 1$ сравнения е невъзможно да се намери максимумът.

Първо забелязваме, че всяко a_i трябва да участва в поне едно сравнение. Да допуснем противното: нека има коректен алгоритъм ALGMAX за тази задача, който върху поне един вход работи така, че някой a_i не участва в нито едно сравнение. Тогава противникът манипулира входа по следния начин:

- Ако ALGMAX върне $a_i^\dagger$, то противникът прави a_i по-малко от всяко друго число от входа. При това положение няма как a_i да е максимумът и ALGMAX се оказва опроверган. След края на работата на ALGMAX, ако поискаме от противника да видим входа, той ще ни покаже вход, в който a_i е минималния елемент, а останалите числа са такива, че за всички извършени сравнения резултатите са точно тези, които противникът е дал като отговори. Очевидно противникът може да генерира такъв вход и поради това не може да бъде уличен в лъжа.
- Ако ALGMAX върне a_j , където $j \neq i$, противникът дава на a_i стойност, по-голяма от тази на кое е да друго число. При това положение a_i е максимумът и ALGMAX се оказва опроверган. И в този случай не можем да уличим противника в лъжа, защото a_i не е бил сравняван изобщо.

И в двата случая ALGMAX не работи коректно, така че той не е коректен алгоритъм и допускането е опровергано.

Долната граница, която следва от съображението, че всеки елемент трябва да е бил сравнен поне веднъж с друг, е $\lceil \frac{n}{2} \rceil$. За да се убедим в това, да си представим графа на сравненията, но в неориентиран вариант. Ние вече въведохме “ориентиран граф на сравненията” (Определение 110). Неориентиран граф на сравненията е нещо подобно: върховете са числата $a_1, a_2, \dots, a_n$, а ребрата са ненаредените двойки числа, които са били сравнени от алгоритъма. Твърдението, че всяко число трябва да е било сравнено поне веднъж е същото като твърдението, че този граф няма изолирани върхове. Очевидно е, че за всеки граф без изолирани върхове е в сила $m \geq \lceil \frac{n}{2} \rceil$.

Но долната граница $\lceil \frac{n}{2} \rceil$ е твърде слаба. Не че всичко това не е истина, но има по-висока долна граница за броя на сравненията, която сега ще докажем.

Теорема 112: Намирането на максималния елемент иска поне $n - 1$ сравнения

В текущия контекст, ALGMAX извършва поне $n - 1$ сравнения.

По правило даваме само едно доказателство на теорема. Тук ще направим изключение.

Първо доказателство: Да си представим отново неориентирания граф на сравненията G . Твърдим, че G е свързан.

Да допуснем, че G не е свързан. По-подробно казано, допускаме, че има вход $a_1, a_2, \dots, a_n$, върху който ALGMAX прави такива сравнения, че G има поне две свързани компоненти. Освен това допускаме, че ALGMAX работи коректно върху $a_1, a_2, \dots, a_n$ и числото a_i , което той връща, е максимално. Но a_i е връх в някоя свързана компонента G' на G . Тъй като G не е свързан, той има поне една друга свързана компонента G'' . Противникът манипулира

[†]Звучи безумно алгоритъмът да върне елемент, който изобщо не е бил сравнен, но тук правим теоретично доказателство и разглеждаме всички случаи.

входа така: прави всеки връх в G'' по-голям от всеки връх от G' , като относителната наредба между елементите-върхове от G'' се запазва. При това положение максимумът няма как да е от G' , така че ALGMAX не работи коректно върху $a_1, a_2, \dots, a_n$. Противникът не може да бъде уличен в лъжа *post factum*, защото той ще генерира (постфактум) вход, за който изходите от сравненията са точно такива, каквито е получил ALGMAX по време на работата си.

Ключовият факт, върху който почиват тези разсъждения, е, че ако неориентираният граф на сравненията не е свързан, за всеки две негови различни свързани компоненти G' и G'' е вярно, че не може да заключим нищо за относителните големина на елемент-връх от G' и елемент-връх от G'' . Може всеки елемент от G' да е по-голям от всеки елемент от G'' , може всеки елемент от G'' да е по-голям от всеки елемент от G' , може всички елементи от G' да са с големина между два съседни по големина елементи от G'' , и така нататък – всичко е възможно за относителните им големина.

Заклучаваме, че неориентираният граф на сравненията е свързан. Добре известен факт от теорията на графите е, че всеки свързан граф е изпълнено $m \geq n - 1$. Тогава броят на сравненията е поне $n - 1$.

Второ доказателство: Сега ще използваме “даг на сравненията”. $\text{BOO}^\dagger$, нека числата са две по две различни, така че имаме предвид Определение 109. Нека ALGMAX е произволен алгоритъм за намиране на максимума. Нека $\text{ALGMAX}(a_1, \dots, a_n)$ връща a_i , като $\tilde{G}$ е дагът на сравненията, генериран за намирането на a_i .

Наблюдение 90

В текущия контекст, за всяко $j \in \{1, \dots, n\}$ е вярно, че

- a_j е сифон в $\tilde{G}$ тстк a_j не е загубил нито едно сравнение; иначе казано, a_j е спечелил всички сравнения, в които е участвал;
- a_j е източник в $\tilde{G}$ тстк a_j не е спечелил нито едно сравнение; иначе казано, a_j е загубил всички сравнения, в които е участвал.

Лема 48

В текущия контекст, $\tilde{G}$ има точно един сифон и той е a_i .

Доказателство: По конструкция, $a_i = \max \{a_1, \dots, a_n\}$. Очевидно a_i не е губил сравнения. От Наблюдение 90 следва, че той е сифон в $\tilde{G}$. Ще покажем, че в $\tilde{G}$ други сифони няма.

Да допуснем противното. Тогава някой a_k , такъв че $k \neq i$, е сифон в $\tilde{G}$, което е същото (Наблюдение 90) като a_k да не е губил сравнения. Тогава противникът манипулира входа, присвоявайки на a_k стойност, която е по-голяма от стойността на всеки друг елемент. Сега вече a_k е максималният елемент във входа, но резултатът от всяко сравнение, в което участва a_k , си остава същият, понеже a_k нараства при манипулацията. Резултатите от сравненията, в които a_k не участва, очевидно не се променят от увеличаването на стойността на a_k . Но тогава ALGMAX продължава да връща a_i , защото резултатите от всички сравнения са

[†]Наистина няма загуба на общност, понеже, ако докажем долна граница $n - 1$ за частния случай на две по две различни числа, тя е в сила и за общия случай на възможни повторения.

същите като тези преди увеличаването на a_k . Полученото противоречие показва, че такъв a_k не съществува. $\square$

Желаният резултат следва веднага от Лема 48, понеже всички елементи освен a_i са $n - 1$ на брой и всеки от тях трябва да е загубил сравнение, за да не е сифон. Но, за да е загубил сравнение, трябва да е бил сравнен поне веднъж. $\square$

Трето доказателство: Твърдим, че дагът на сравненията $\tilde{G}$ е слабо свързан. Наистина, ако $\tilde{G}$ има две или повече слабо свързани компоненти, от която и от тях ALGMAX да вземе максимума, противникът може *post factum* да увеличи произволно стойностите на елементите от друга слабо свързана компонента, така че максимумът да се окаже в нея (другата). При това противникът няма да бъде уличен в лъжа, защото стойностите, които ще генерира като вход, ще са консистентни с резултатите от сравненията.

Щом $\tilde{G}$ е слабо свързан, той има поне $n - 1$ ребра, което значи, че са направени поне $n - 1$ сравнения. $\square$

По същество третото доказателство е същото като първото доказателство.

Тъй като Подсекция 13.4.4 е за лесна, да не кажем “тривиална”, долна граница, не описавме стратегията, която следва противникът, а само показахме, че ако са извършени по-малко от $n - 1$ сравнения, противникът би опровергал алгоритъма, който намира максимума. В следващите подсекции долните граници са доста по-трудни за доказване и в тях ще описваме точно стратегията, която противникът следва. Не е грешка да се каже още “алгоритъмът на противника”, само трябва да се прави ясна разлика между алгоритъма, който решава задачата, и алгоритъма на противника – това са принципино различни неща.

13.4.5 Долна граница $\lceil \frac{3n}{2} \rceil - 2$ за намиране на максимален и минимален елемент

Задачата е да се намери чрез, и само чрез, сравнения максимално число и минимално число измежду произволни числа $a_1, a_2, \dots, a_n$, където $n \geq 2$. Ще допуснем, че числата са две по две различни, така че ще казваме, че търсим максимума и минимума. Очевидно може да намерим максимума и минимума поотделно с общо $(n - 1) + (n - 1) = 2n - 2$ сравнения, но това не е оптимално. Има по-добър начин.

Намиране на максимума и минимума с $\lceil \frac{3n}{2} \rceil - 2$ сравнения. Това не е доказателството на долната граница. Тук описваме алгоритъм, който намира желаните елементи с $\lceil \frac{3n}{2} \rceil - 2$ сравнения, но от това не следва, че няма начин те да бъдат намерени с по-малко сравнения. Чак на следващата страница ще докажем долната граница. Въпросният алгоритъм разглеждаме само за “загриване”.

Първо да допуснем, че n е четно. Групираме числата в $\frac{n}{2}$ двойки, във всяка двойка намираме максимума и минимума с едно сравнение и получаваме $\frac{n}{2}$ максимума, по един от всяка двойка, и още $\frac{n}{2}$ минимума, по един от всяка двойка. Глобалния максимум намираме от тези $\frac{n}{2}$ максимума чрез $\frac{n}{2} - 1$ сравнения. Аналогично, глобалния минимум намираме от тези $\frac{n}{2}$ минимума чрез $\frac{n}{2} - 1$ сравнения. Общият брой на използваните сравнения е

$$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \left(\frac{n}{2} - 1\right) = \frac{3n}{2} - 2$$

което при четно n можем да напишем като

$$\left\lceil \frac{3n}{2} \right\rceil - 2$$

Сега да допуснем, че n е нечетно. Броят на двойките е $\lfloor \frac{n}{2} \rfloor$ и едно число a_j остава извън двойките. Намираме $\lfloor \frac{n}{2} \rfloor$ максимума, по един от всяка двойка, и още $\lfloor \frac{n}{2} \rfloor$ минимума, по един от всяка двойка. После глобалният максимум се намира измежду тези максимуми и a_j с $(\lfloor \frac{n}{2} \rfloor + 1) - 1 = \lfloor \frac{n}{2} \rfloor$ сравнения. После глобалният минимум се намира от тези минимума и a_j (разглеждаме най-лошият случай, в който a_j не се е оказал глобален максимум и сме длъжни да проверим дали не е глобален минимум) пак с $(\lfloor \frac{n}{2} \rfloor + 1) - 1 = \lfloor \frac{n}{2} \rfloor$ сравнения. Общият брой на сравненията е $3 \lfloor \frac{n}{2} \rfloor$. При $n = 2k + 1$, това е

$$3 \left\lfloor \frac{2k+1}{2} \right\rfloor = 3k = (3k+2) - 2 = \frac{6k+4}{2} - 2 = \left\lceil \frac{6k+3}{2} \right\rceil - 2 = \left\lceil \frac{3(2k+1)}{2} \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Виждаме, че и при четно, и при нечетно n тази идея води до решение с $\lceil \frac{3n}{2} \rceil - 2$ сравнения. Следният алгоритъм реализира тази идея. Алгоритъмът очевидно може да се реализира in-place, но сега сложността по памет не ни интересува, така че можем да си позволим реализация с линейна сложност по памет.

MAXNMIN($A[1 \dots n]$): цели числа без повторения, $n \geq 2$)

```

1   $k \leftarrow \lfloor \frac{n}{2} \rfloor$ 
2  създай масиви MAX[1, ..., k], MIN[1, ..., k]
3  for  $i \leftarrow 1$  to  $k$ 
4      if  $A[2i-1] > A[2i]$ 
5          MAX[i]  $\leftarrow A[2i-1]$ , MIN[i]  $\leftarrow A[2i]$ 
6      else
7          MAX[i]  $\leftarrow A[2i]$ , MIN[i]  $\leftarrow A[2i-1]$ 
8  tmpmax  $\leftarrow$  MAX[1], tmpmin  $\leftarrow$  MIN[1]
9  for  $i \leftarrow 2$  to  $k$ 
10     if MAX[i] > tmpmax
11         tmpmax  $\leftarrow$  MAX[i]
12     if MIN[i] < tmpmin
13         tmpmin  $\leftarrow$  MIN[i]
14  if isodd(n)
15     if  $A[n] > \text{tmpmax}$ 
16         tmpmax  $\leftarrow A[n]$ 
17     if  $A[n] < \text{tmpmin}$ 
18         tmpmin  $\leftarrow A[n]$ 
19  return (tmpmax, tmpmin)
```

Долна граница $\lceil \frac{3n}{2} \rceil - 2$ за броя на сравненията за намиране на максимума и минимума. Ще докажем, че има противник, който може да манипулира входа така, че да накара всеки алгоритъм за задачата да направи поне $\lceil \frac{3n}{2} \rceil - 2$ сравнения. От това ще следва, че алгоритъмът MAXNMIN е оптимален, поне що се отнася до броя на извършените сравнения.

Да си представим произволен алгоритъм ALGMXMIN за тази задача. Нека G е дагът на сравненията, генериран от работата на ALGMXMIN върху вход с n елемента. От Лема 48

знаем, че G има точно един сифон и това е максимумът. Напълно аналогично, G има точно един източник и това е минимумът. Максимумът е спечелил всички сравнения, а минимумът е загубил всички сравнения.

Наблюдение 91

В текущия контекст, всяко число, различно от максимума и от минимума, е спечелило поне едно сравнение и е загубило поне едно сравнение.

В началото нито едно число не е било сравнено. След това алгоритъмът започва да сравнява двойки числа и в края на работата му

- едно число (максимумът) само е печелило сравнения,
- едно число (минимумът) само е губило сравнения,
- всяко от останалите $n - 2$ числа е спечелило поне по едно сравнение и загубило поне по едно сравнение.

Има смисъл да въведем *състояние* на всяко от числата по време на работата на алгоритъма. Във всеки момент, всяко число x е в точно едно от следните четири състояния:

N , ако x още не е било сравнявано.

W , ако x е спечелило поне едно сравнение и не е загубило нито едно сравнение.

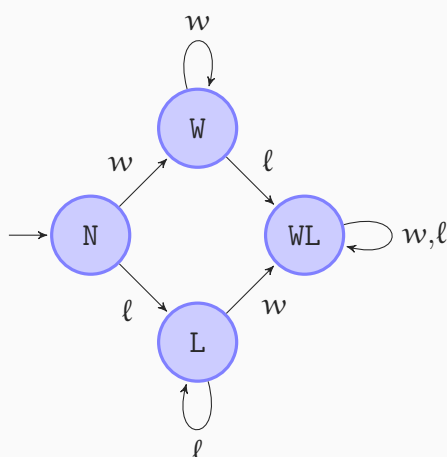
L , ако x е загубило поне едно сравнение и не е спечелило нито едно сравнение.

WL , ако x е спечелило поне едно сравнение и е загубило поне едно сравнение.

В началото всяко число е в състояние N . В края всички числа без две са в състояние WL , точно едно е в състояние W и точно едно е в състояние L . В някакъв смисъл, работата на алгоритъма е да “вкара” числата в тези състояния – когато, и само когато, всички числа са в така описаните състояния, задачата е решена. Работата на противника е да забави колкото е възможно повече постигането на съвкупността от тези състояния.

Да помислим как число минава от състояние в състояние след извършване на сравнения. Началното състояние е N . WL е мъртво състояние – от него не може да се излезе. Всеки преход става след сравнение на това число с друго, което означава, че нашето число или е спечелило, или е загубило. Сравнение, в което то е спечелило, означаваме с “ w ”, а сравнение, в което то е загубило, означаваме с “ l ”. Фигура 13.13 показва диаграмата на състоянията и преходите за произволно число.

Фигура 13.13 : Диаграма на състоянията и преходите.



За достигане на WL от N са необходими поне два прехода.

Сега ще опишем стратегия на противника при сравняване на две числа, която максимизира минималния брой на сравненията. Да кажем, че се извършва сравнение $a_i <^? a_j$. Всяко от a_i и a_j има състояние, което е елемент на $\{N, W, L, WL\}$. Противникът държи сметка за текущото състояние на всеки елемент. Да разгледаме всички възможни комбинации от състояния. Те са само $(2^{4-1}) = 10$, а не $4^2 = 16$, защото наредбата е без значение, така че комбинациите са двueleментните мултимножества над опорното множество $\{N, W, L, WL\}$. Таблица 13.1 описва как противникът манипулира данните във всеки възможен случай.

Съст-ние на a_i	Съст-ние на a_j	Избор на противника	Можеше ли да избере обратното?	Ново съст-ние на a_i	Ново съст-ние на a_j	Промяна в броя на съст-нията
N	N	$a_i < a_j$	да	L	W	+2
N	W	$a_i < a_j$	не	L	W	+1
N	L	$a_i > a_j$	не	W	L	+1
N	WL	$a_i < a_j$	да	L	WL	+1
W	W	$a_i < a_j$	да	WL	W	+1
W	L	$a_i > a_j$	не	W	L	0
W	WL	$a_i > a_j$	не	W	WL	0
L	L	$a_i < a_j$	да	L	WL	+1
L	WL	$a_i < a_j$	не	L	WL	0
WL	WL	$a_i < a_j$	да	WL	WL	0

Таблица 13.1: Стратегията на противника в задачата за намиране на максимален и минимален елемент.

От особен интерес са петте комбинации, в които противникът няма избор за своя отговор (оцветените редове). Примерно, да разгледаме втория ред. Състоянието на a_i е N, а това на a_j е W; тоест, a_i не е участвал в сравнения изобщо, а a_j е участвал в поне едно сравнение и спечелил всички сравнения, в които е участвал.

- Ако противникът отговори $a_i < a_j$ (както прави според таблицата), той променя състоянието само на a_i . Очевидно a_i не може да остане N, щом вече е участвал в едно

сравнение. Но a_j си остава в W , защото продължава да е вярно, че само е печелил сравнения.

- Ако противникът отговори $a_i > a_j$ (което той няма да направи, защото е неограничено умен и знае, че би било в негов ущърб да отговори така), то a_i би се оказал в W , спечелвайки срещу a_j , но a_j би се оказал в WL , след като е отбелязал първата си загуба. В този случай и a_i , и a_j биха променили състоянията си.

От този пример трябва да е ясно каква е стратегията на противника: изходът от всяко сравнение е такъв, че промяната в състоянията **и на двата участника** да е минимална. Ако се сравняват N с N (първият ред), промяна настъпва в състоянията и на двете числа, независимо от отговора на противника (съответно, при това сравнение противникът отговаря произволно). Колкото и да не иска, противникът е принуден да промени състоянието на всяко от двете числа. Естествено, сравненията N срещу N са най-много $\lfloor \frac{n}{2} \rfloor$ на брой. Във всеки друг случай (освен N срещу N), противникът отговаря по такъв начин, че или да не промени нито едно от състоянията (например W срещу L на ред шести), или да промени само едно от състоянията.

Да въведем понятието *макросъстояние* като редицата от състоянията на всички числа. Началното макросъстояние е $\langle N, N, \dots, N \rangle$. Да кажем, че *крайно макросъстояние* е всяка от редиците, имаща един елемент W , един елемент L , а останалите $n - 2$ елемента WL . Както вече отбелязахме, имаме категоричен отговор на задачата тогава и само тогава, когато е достигнато крайно макросъстояние. За достигането на крайно макросъстояние от началното макросъстояние, трябва да бъдат извършени поне $2n - 2$ прехода, защото:

- за всяко число, което завършва в WL са необходими поне 2 прехода (вижте Фигура 13.13), което означава поне $2(n - 2) = 2n - 4$ прехода общо.
- за числото, което завършва в W е необходим поне 1 преход,
- за числото, което завършва в L е необходим поне 1 преход.

И така, трябва да бъдат извършени поне $2n - 4 + 2 = 2n - 2$ прехода общо от всички числа. Това обаче **не означава**, че трябва да се направят общо $2n - 2$ сравнения! Както видяхме, има сравнения, при които **и двете** участващи числа си променят състоянието.

Наблюдение 92

В текущия контекст, сравненията, при които и двете числа си променят състоянието, са точно сравненията от вида N срещу N , а те са най-много $\lfloor \frac{n}{2} \rfloor$ на брой.

При всички останали сравнения, противникът може да манипулира входа така, че да има не повече от 1 преход на сравнение **и за двете** участващи числа. Това означава, че противникът винаги може да наложи

$$2n - 2 - \left\lfloor \frac{n}{2} \right\rfloor = 2n - 2 + \left\lceil -\frac{n}{2} \right\rceil = \left\lceil 2n + \left(-\frac{n}{2}\right) \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2 \quad (13.7)$$

сравнения.

Ето по-подробно обяснение на това заключение. Нека X е множество от сравнения с минимална мощност за намиране на максимума. Знаем от Подсекция 13.4.4, че $|X| = n - 1$. Нека Y е множество от сравнения с минимална мощност за намиране на минимума. От симетрични съображения имаме $|Y| = n - 1$. Да разгледаме $Z = X \cup Y$. Ясно е, че Z е множество от

сравнения, с които намираме както максимума, така и минимума. В тази подсекция търсим долна граница за $|Z|$.

Ключовото наблюдение е, че $|Z|$ **не е** просто $|X| + |Y|$, защото някои сравнения може да се ползват както за намиране на максимума, така и за намиране на минимума. С други думи, $X \cap Y$ може да е непразно. От комбинаторния принцип на включването и изключването имаме

$$|Z| = |X| + |Y| - |X \cap Y|$$

Щом $|X| = n - 1$ и $|Y| = n - 1$, в сила е

$$|Z| = 2n - 2 - |X \cap Y|$$

Наблюдение 93

В текущия контекст, ако противникът играе оптимално, $X \cap Y$ се състои от точно тези сравнения, при които нито един от двата участника не е бил сравняван досега.

Ако това не е очевидно, вижте пак Таблица 13.1. Само първият ред е сравнение, което “върши работа” както за намиране на максимума, така и за намиране на минимума. Това точно отговаря на факта, че първият ред е единственият, при който има точно две промени на състояние. Вторият ред върши работа само за намиране на минимума. Третият ред върши работа само за намиране на максимума. Четвъртият ред върши работа само за минимума, ако противникът отговори $\langle L, WL \rangle$, както е написано в таблицата; ако отговори $\langle W, WL \rangle$, което е възможно, то четвъртият ред би вършил работа само за максимума. И така нататък. Редове 6, 7, 9 и 10 не вършат работа нито за минимума, нито за максимума.

Допълнение 72: Безсмислено за макс&мин сравнение е смислено за сорт.

В текущия контекст, това, че сравненията от редове 6, 7, 9 и 10 не вършат работа, е вярно, но само що се отнася до намиране на максимума и минимума. Ако задачата е, да кажем, СОРТИРАНЕ, може да има има смисъл от сравнение $\langle WL, WL \rangle$. Ето малък пример. Да кажем, че $n = 6$ и са направени сравнения на a_1 с a_2 , a_2 с a_3 , a_4 с a_5 и a_5 с a_6 . Да кажем, че резултатите са тези:

$$\overset{L}{a_1} < \overset{WL}{a_2} < \overset{W}{a_3}$$

$$\overset{L}{a_4} < \overset{WL}{a_5} < \overset{W}{a_6}$$

Наистина, ако търсим максимума и минимума, напълно безсмислено е да сравняваме a_2 с a_5 . Ние вече знаем, че минимумът е от $\{a_1, a_4\}$, а максимумът е от $\{a_3, a_6\}$, ерго, информацията за това, кой е по-голям от a_2 и a_5 , е нерелевантна. За да намерим максимума и минимума, необходимо и достатъчно е да направим още две сравнения: a_1 с a_4 и a_3 с a_6 .

Но, ако искаме да намерим сортираната пермутация, сравнението на a_2 с a_5 е смислено. Забележете, че има точно $\binom{6}{3} = 20$ пермутации, съвместими с $a_1 < a_2 < a_3$ и $a_4 < a_5 < a_6$. Тогава ни трябва поне $\lceil \log_2 20 \rceil = 5$ сравнения, за да открием истината.

Ако направим $a_2 \overset{?}{<} a_5$ и отговорът е Y , то има 10 пермутации, съвместими с този

отговор. А именно

$$\begin{aligned}
 a_4 &< a_1 < a_2 < a_3 < a_5 < a_6 \\
 a_4 &< a_1 < a_2 < a_5 < a_3 < a_6 \\
 a_4 &< a_1 < a_2 < a_5 < a_6 < a_3 \\
 a_1 &< a_4 < a_2 < a_3 < a_5 < a_6 \\
 a_1 &< a_4 < a_2 < a_5 < a_3 < a_6 \\
 a_1 &< a_4 < a_2 < a_5 < a_6 < a_3 \\
 a_1 &< a_2 < a_3 < a_4 < a_5 < a_6 \\
 a_1 &< a_2 < a_4 < a_3 < a_5 < a_6 \\
 a_1 &< a_2 < a_4 < a_5 < a_3 < a_6 \\
 a_1 &< a_2 < a_4 < a_5 < a_6 < a_3
 \end{aligned}$$

Тогава има точно 10 пермутации, съвместими с отговора N. Излиза, че сравнението $a_2 \stackrel{?}{<} a_5$ води до перфектна дихотомия (разбиване на множеството на пермутациите на два равномошни дяла) и е съвсем смислено, ако искаме да сортираме.

Съгласно Наблюдение 92 и Наблюдение 93, $|X \cap Y| \leq \lfloor \frac{n}{2} \rfloor$. Тогава

$$|Z| \geq 2n - 2 - \lfloor \frac{n}{2} \rfloor$$

Ползваме (13.7) и получаваме

$$|Z| \geq \left\lceil \frac{3n}{2} \right\rceil - 2$$

Явно описание на стратегията на противника. Най-подробните доказателства за долни граници с противник се правят с описание на стратегията, която следва противникът. Може още да се каже “на алгоритъма на противника”, тъй като противникът е алгоритъм (а не физически човек или демон), но този израз крие опасността читателят да се обърка, понеже решението на задачата също е алгоритъм. Затова е по-добре да се говори за “стратегията на противника”.

Ние вече доказахме (Подсекция 13.4.3) долна граница $\lceil \log_2 n \rceil$ за намиране на число от $\{1, \dots, n\}$ чрез противник, като описахме в явен вид стратегията на противника Б. Стратегията на Б се оказва изключително проста: на всяка итерация той отговаря Y или N по такъв начин, че мощността на подмножеството, в което може да се намира k след сравнението, да е поне половината на това преди сравнението. Но доказахме (Подсекция 13.4.4) и долна граница $n - 1$ за намиране на максимума, без да описваме стратегията на противника в явен вид. Това доказателство беше по-абстрактно и по същество казваше “ако направим по-малко от $n - 1$ сравнения, противникът ще ни опровергае”, но не казваше как точно отговаря противникът на запитване $a_i \stackrel{?}{<} a_j$.

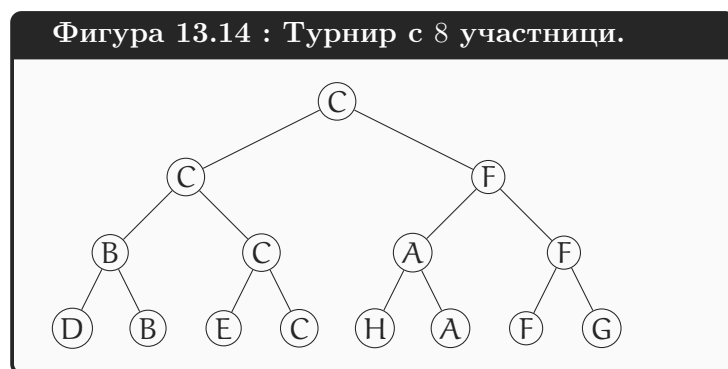
Доказателството за долната граница $\left\lceil \frac{3n}{2} \right\rceil - 2$ за намиране на минимума и максимума, което направихме току-що, не съдържа пълно и явно описание на стратегията на противника. Таблица 13.1 дава общи указания за това как да отговаря противникът, но не е съвсем конкретна. Примерно, ако и двете числа са в състояние WL, Таблица 13.1 казва (най-долният ред),

че на противникът му е все едно как ще отговори. Това е вярно само доколкото сравнението на две числа в състояние WL е груба грешка, що се отнася до ефикасно намирането на минимума и максимума, и противникът не разкрива никаква нова информация независимо дали отговори Y или N, така че може да мислим, че отговаря произволно (или хвърля монета, ако смятате, че противникът не е съзнателно същество и няма свободна воля).

13.4.6 Долна гр. $n - 2 + \lceil \log_2 n \rceil$ за намиране на втория по големина елемент

Въведение: пример с турнир. Представете си турнир с директно елиминирание с n участници. Това е състезание, което се провежда на кръгове. В първия кръг влизат всички участници. Във всеки кръг, в който има повече от един участник, състезателите играят по двойки един срещу друг. Всяка игра завършва с победа – равен резултат е невъзможен. Победителят от всяка двойка продължава в следващия кръг, а победеният приключва участието си. В последния кръг се играе една единствената игра и победителят в нея е и победителят в турнира.

Всичко това се моделира с кореново двоично дърво. Листата на дървото са n на брой и са именувани с участниците в състезанието. Върховете са разположени в нива. Най-ниското ниво се състои от листата (състезателите в началото), а всяко следващо ниво се състои от победителите от предния кръг. Игра между състезатели x и y с победител x в даден кръг на турнира се моделира с това, че в съответното ниво има точно един връх с етикет x и точно един връх с етикет y , а техният родител (в следващия кръг) има етикет x . Фигура 13.14 илюстрира турнир с 8 участници A, ..., H с победител C.



Изложението дотук допускаше негласно, че n е точна степен на двойката; тоест, че съществува естествено m , такова че $2^m = n \leftrightarrow m = \log_2 n$. Тогава броят на върховете във всяко ниво над листата е точно половината от броя на върховете в предното ниво и дървото е свършено двоично дърво.

А ако n не е точна степен на двойката? Тогава организираме турнира по следния начин.

Определение 112: Турнир на произволен брой участници

Нека n е произволно цяло положително число. Организираме турнир в кръгове така.

- Ако броят на участниците в даден кръг е четен, групираме участниците по двойки. Двойките играят и в следващия кръг отиват победителите.
- Ако обаче броят на участниците в даден кръг е нечетен и е по-голям от 1, групираме всички участници без един по двойки. Да наречем x този, който не е групиран. Двойките играят и в следващия кръг отиват победителите и x . В някакъв смисъл x е късметлия, понеже отива в следващия кръг на турнира, без да е играл(а).

Ако в даден кръг на турнира има k участници и $k > 1$, в следващия кръг има $\left\lceil \frac{k}{2} \right\rceil$ участници. Броят на участниците в кръговете на турнира се пресмята чрез итератора (Определение 6)

$$k \mapsto \left\lceil \frac{k}{2} \right\rceil$$

с начална стойност n . Очевидно този итератор има фиксирана точка $k = 1$, но правилата на турнира казват, че турнирът спира, когато остане само един участник.

Ако и само ако n е точна степен на двойката 2^m , итераторът минава само през четни стойности, преди да достигне единицата

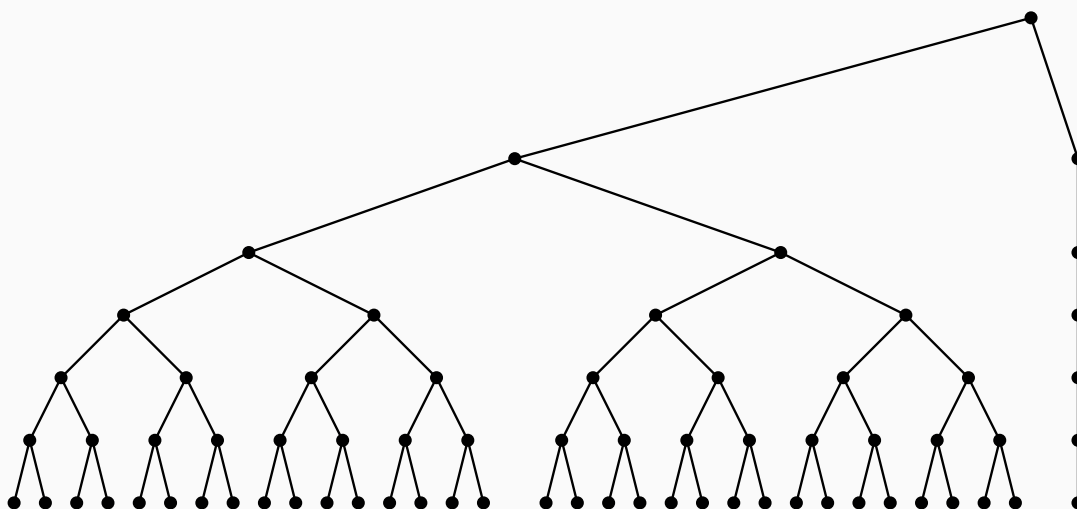
$$n = 2^m \mapsto \frac{n}{2} = 2^{m-1} \mapsto \frac{n}{4} = 2^{m-2} \mapsto \dots \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1$$

Лесно се вижда, че ако n не е точна степен на двойката, поне една от не-крайните стойности е нечетна. Примерно

$$33 \mapsto 17 \mapsto 9 \mapsto 5 \mapsto 3 \mapsto 2 \mapsto 1$$

На тази редица съответства кореновото дърво, показано на Фигура 13.15. Дървото се определя напълно от броя на листата, в случая 33.

Фигура 13.15 : Дървото на турнир с 33 участници.



Ако в даден кръг на турнира има k участници, броят на игрите в този кръг е $\lfloor \frac{k}{2} \rfloor$. Тук вземаме долната граница, защото, ако k е нечетно, един играч продължава нагоре без да играе. Поначало участниците са n на брой, така че броят на игрите в първия кръг е $\lfloor \frac{n}{2} \rfloor$. Броят на участниците, достигнали до втория кръг, е $\lfloor \frac{n}{2} \rfloor$, защото при нечетно n , един участник отива на втори кръг автоматично. Броят на игрите във втория кръг тогава е $\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor$. И така нататък. Броят на всички изиграни игри в турнира е

$$\lfloor \frac{n}{2} \rfloor + \left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor}{2} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor}{2} \right\rfloor}{2} \right\rfloor + \dots + 1$$

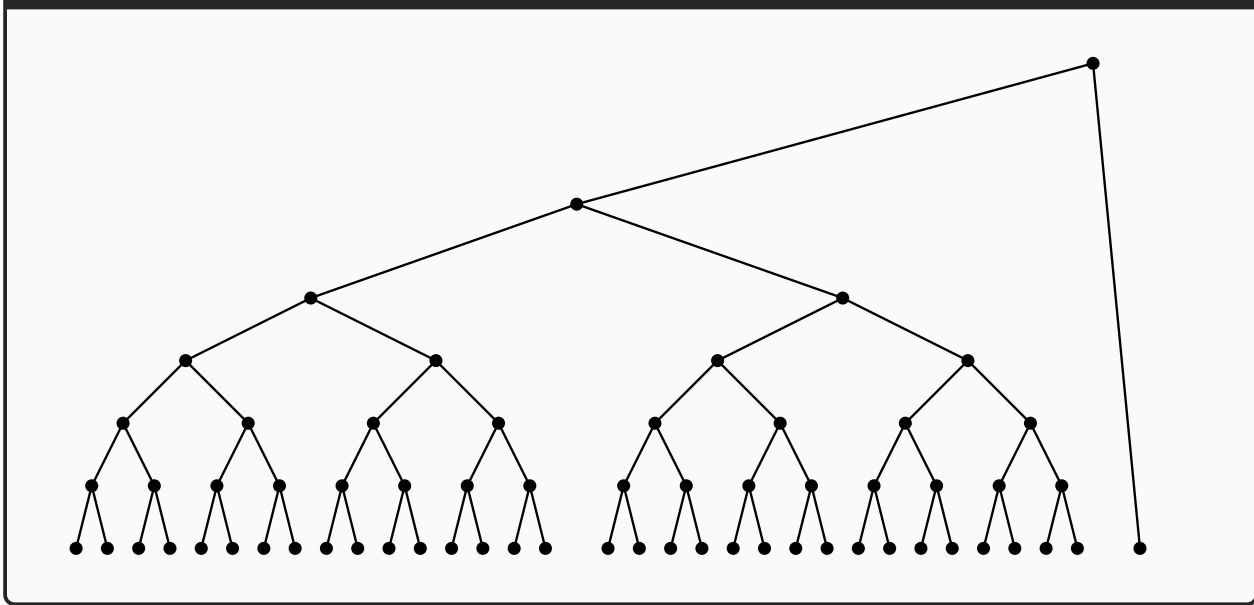
Следствие 23 ни позволява да се освободим от итерираното прилагане на ceiling в числителя, опростявайки до

$$\lfloor \frac{n}{2} \rfloor + \left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\lfloor \frac{n}{4} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\lfloor \frac{n}{8} \rfloor}{2} \right\rfloor + \dots + 1$$

Сумата може да се намери със съображения от теорията на графите. Търси се броят на вътрешните върхове на дървото на турнира, които имат две деца. Това е броят на изиграните игри. Както видяхме, ако n не е точна степен на двойката, неизбежно има вътрешни върхове с точно едно дете и те не отговарят на изиграни игри. Да елиминираме върховете с точно едно дете по най-очевидния начин: ако връх v има точно едно дете w (забележете, че v не може да е коренът) и родителят на v е връх u , то v се изтрива, а w става дете на u [†]. По този начин получаваме редуцирано дърво, което има същата структура като оригиналното, що се отнася до върховете с две деца. Фигура 13.16 показва редуцираното дърво, получено от дървото от Фигура 13.15.

[†]По Дискретни Структури, в дефиницията на хомеоморфизъм, подобно действие нарекохме “серийна редукция”. Но серийна редукция не е точно същото като това, което правим в момента. В турнира дървото е кореново и двете ребра, инцидентни с корена, не подлежат на серийно редуциране.

Фигура 13.16 : Дървото от Фигура 13.15 след редуцирането.



Търсим броя на вътрешните върхове на редуцираното дърво. То е пълно двоично дърво. От курса Дискретни Структури знаем, че всяко пълно k -ично кореново дърво с ℓ листа има точно $\frac{\ell-1}{k-1}$ вътрешни върхове. В случая $\ell = n$, а $k = 2$, така че вътрешните върхове са точно $n - 1$.

И така, броят на изиграните игри в турнир с n участници е точно $n - 1$, за всяко цяло положително n . Има смисъл от едно предупреждение: в този контекст n не е броят на върховете, а броят на листата. Общият брой на върховете на редуцираното дърво е $n - 1 + n = 2n - 1$. Типично, буквата “ n ” означава броя на върховете и това може да доведе до привидни противоречия между резултати, които получаваме тук, и общоизвестни формули.

Намиране на втория по големина елемент с $n - 2 + \lceil \log_2 n \rceil$ сравнения. Ако допуснем, че всеки участник в турнира има сила, която е някакво положително число, и силата на всеки участник е константа във времето, и няма два участника с еднаква сила, победителят е участникът с най-голямата сила. В този смисъл, намирането на участника с най-голяма сила е същото като намирането на максимума на n числа, но само чрез сравнения. Както видяхме в Подсекция 13.4.4, с $n - 1$ сравнения може да намерим победителя, но с по-малко не можем.

В тази задача обаче се иска да се намери **вторият по сила** участник. Бихме могли да подходим така: да намерим победителя с $n - 1$ сравнения и после да намерим с $n - 2$ сравнения победителя сред останалите; очевидно той е вторият по сила сред началните участници. Но този подход ползва $n - 1 + n - 2 = 2n - 3$ сравнения, което не е оптимално.

Можем да намерим втория по сила участник, ползвайки много по-малко сравнения, въз основа на следното наблюдение.

Наблюдение 94

По време на игрите в турнира, с които бива определен победителят, същият играе срещу някакво множество S от участници. Вторият по сила е един от S .

С други думи, по време на турнира, първият и вторият по сила неизбежно са се срещали. Ако това не е очевидно, разгледайте следната аргументация: ако допуснем, че първият и вторият по сила не са играли един срещу друг, противникът може да увеличи силата на втория,

правейки го по-силен от този, който се е оказал победител, без да променя друго. Но след тази манипулация на входа, изходите от игрите-сравнения остават същите, така че победителят не се променя. Само че той/тя вече не е най-силен/най-силна измежду участниците.

Наблюдение 94 е в сила за всеки турнир. Дори турнирът да не е организиран съгласно Определение 112, а да е съгласно алгоритъма MAX UNSORTED на стр. 781, наблюдението е в пълна сила. Всеки проведен турнир, независимо от правилата за провеждане, е двоично кореново дърво, чиито листа са участниците, а коренът е победителят. Ако вътрешните върхове са само изиграните двубои, дървото е и пълно двоично дърво. В някакъв смисъл, победителят “изминава” пътя от листо до корена и е единственият, достигнал корена. Всички останали участници са “изпокапали”, рано или късно. Изминавайки пътя до корена, победителят играе срещу множество от участници, което множество е наречено S в Наблюдение 94. В турнира, показан на Фигура 13.14, $S = \{B, E, F\}$. Кой точно от тях е вторият по сила, не можем да кажем засега. От общи съображения е ясно, че проведеният турнир не ни дава никаква информация за разпределението на силите на елементите на S , понеже нито двама от тях не са играли помежду си. Трябва да направим втори турнир, но само върху елементите на S , за да намерим втория по сила. Това иска $|S| - 1$ сравнения.

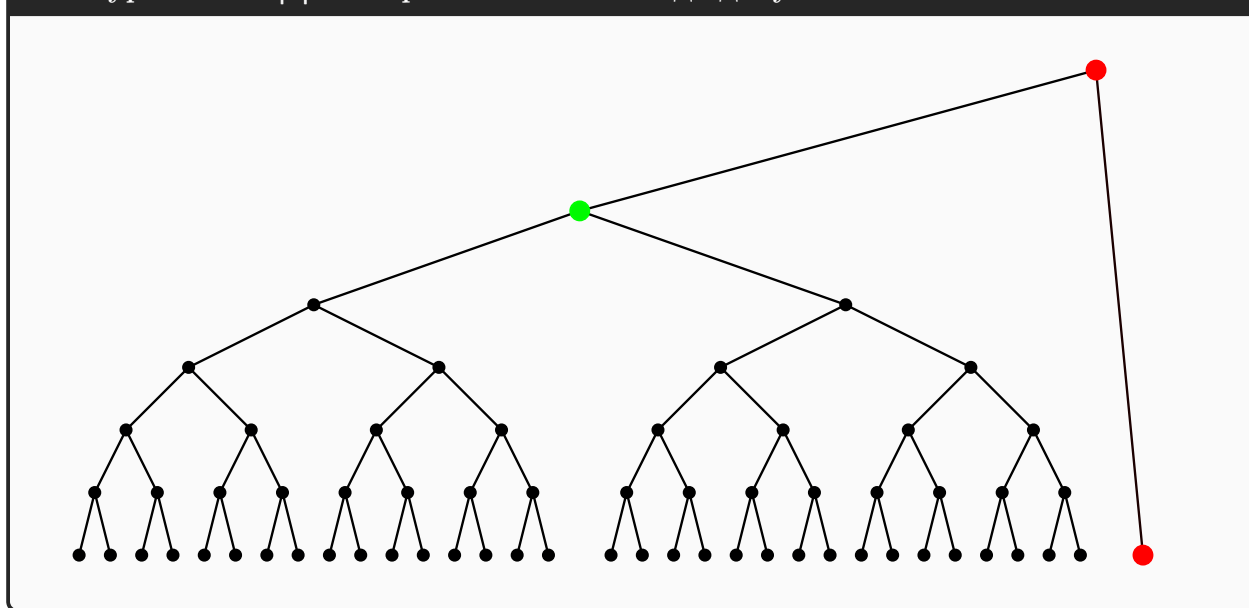
Наблюдение 95

В текущия контекст, намирането на втория по сила участник става с $n - 1 + |S| - 1 = n + |S| - 2$ сравнения.

Ако следваме този подход, $|S|$ трябва да е минимално, от гледна точка на алгоритъма, който търси втория по големина елемент. От гледна точка на противника обаче, $|S|$ трябва да е максимално, за да бъде принуден алгоритъмът да работи максимално дълго.

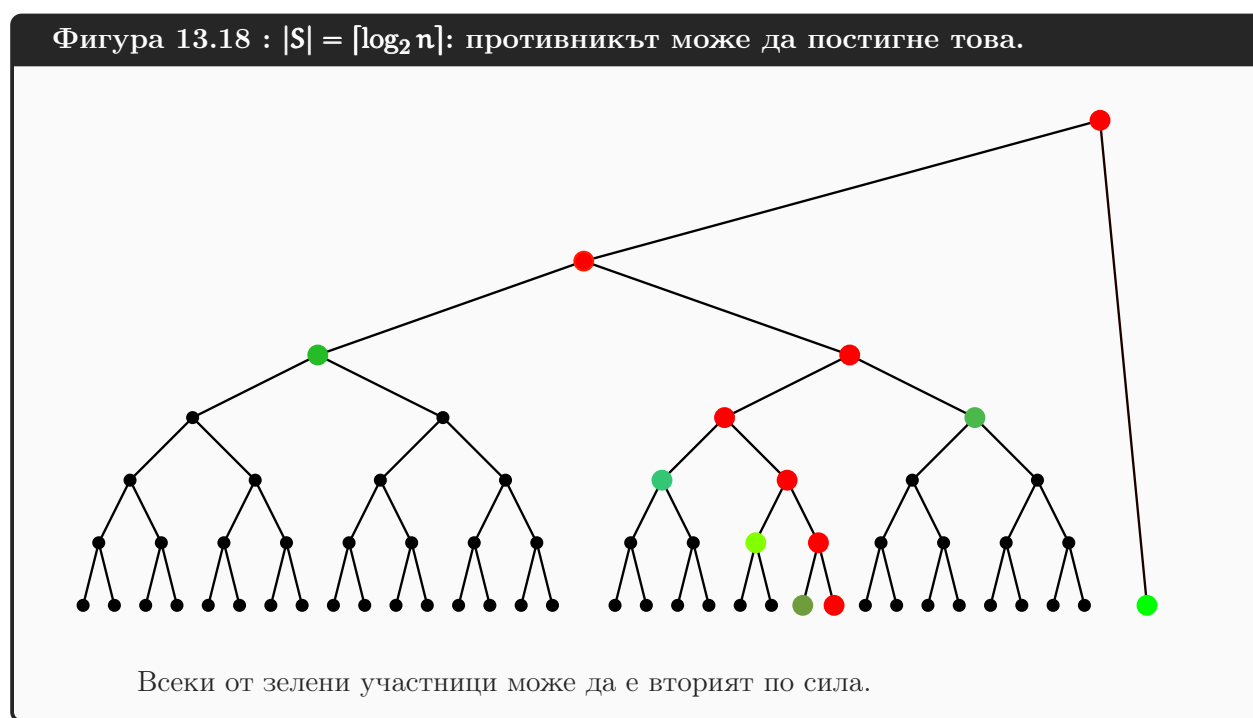
Колко най-малко може да е $|S|$? От най-общи съображения, отговорът е $|S| = 1$. Ако това е така, броят на сравненията за намиране на втория участник е не повече от $n - 1 + 1 = n$. За да се случи този най-добър вариант, достатъчно е да има листо, което е дете на корена, и то да е най-силният участник. Такъв изход от турнир е показан на Фигура 13.17, като в него $n - 1$ сравнения ни дават и първия, и втория по сила.

Фигура 13.17 : $|S| = 1$: противникът няма да допусне това.



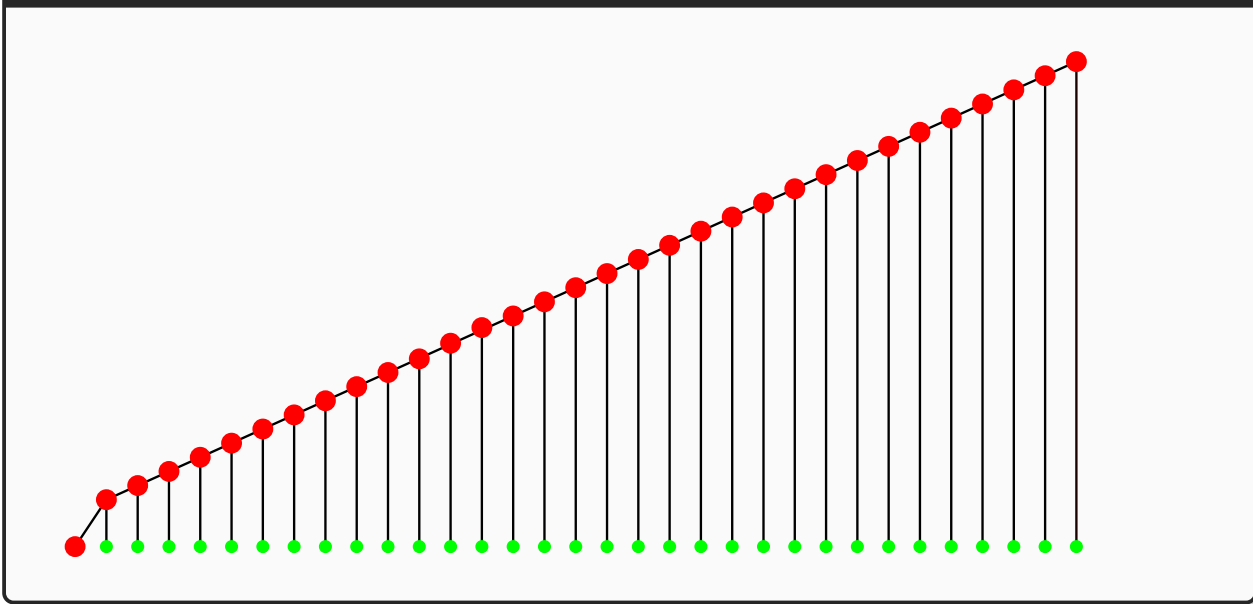
Ние обаче се интересуваме от най-лошия случай, а $|S| = 1$ е по-скоро най-добрият случай. Противникът никога няма да допусне $|S|$ да е само единица. Противникът ще отговаря по такъв начин, че $|S|$ да се окаже равна на височината на дървото. Това е най-много (най-лошото за бързодействието на алгоритъма), което противникът може да постигне: $|S|$ да е височината. Че противникът винаги може да “докара” $|S|$ да е височината, ще докажем нататък в изложението.

Фигура 13.18 илюстрира най-лошия случай, ако ние сме организирали турнира оптимално; което ще рече, съгласно Определение 112. Наистина, на фигурата $|S| = \lceil \log_2 33 \rceil = 6$ и това е точно височината.



Ако обаче сме организирали турнира неоптимално, височината може да достигне $n - 1$ [†]. Примерно, ако турнирът е организиран съгласно алгоритъма MAX UNSORTED на стр. 781, височината е $n - 1$, както е показано на Фигура 13.19.

[†]Не забравяйте, че n е броят на листата, а не броят на всички върхове в дървото!

Фигура 13.19 : $|S| = n - 1$, ако турнирът е организиран неоптimalно.

В обобщение: съгласно Наблюдение 95, можем да намерим втория по сила участник с $n - 2 + \lceil \log_2 n \rceil$ сравнения. **Но това не е доказателство за долна граница!** Ние получихме този резултат при допускането, че намирането на втория по сила става чрез намирането на първия по сила измежду всички, последвано от намирането на първия по сила измежду елементите на S . А ако алгоритъмът за намирането на втория не следва тази обща идея? Доказателството за долна граница не може да почива върху допускания за работата на алгоритъма.

Доказателство за долната граница. Изложението се базира на онлайн записки на Bang-Jensen [9]. Дадено е множество от числа $A = \{a_1, \dots, a_n\}$. Да си представим редица от извършени сравнения, в резултат на които бива намерен втория по големина елемент на A . Нека G е съответният даг на сравненията (Определение 109). Разглеждаме процеса на създаването на G , така че мислим за G като за динамичен граф, който е $G = (A, \emptyset)$ преди първото сравнение, а всяко сравнение добавя по едно ребро[†], докато G не стане такъв, че да е ясно кой е вторият по големина елемент. Ребрата на G обаче (може да) са два вида, които различаваме чрез цветове; това е съществена разлика с дага на сравненията от Определение 109, чиито ребра са само един вид.

Представяме си G като гора $\mathcal{F}$ от антиарборесценции $D_1, D_2, \dots, D_k$, плюс може би още ребра. Очевидно $\biguplus_{i=1}^k V(D_i) = A$. За всяко ребро e на G ,

- или e принадлежи на една от антиарборесценциите, в който случай казваме, че e е *черно ребро*,
- или началото и краят на e са от различни антиарборесценции, в който случай казваме, че e е *червено ребро*.

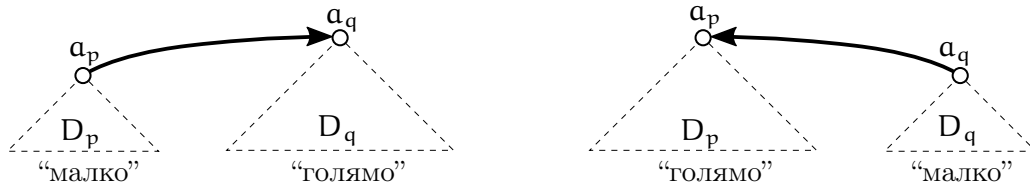
Ребрата, както знаем, са изходите от направените сравнения. Ето стратегията на противника, която генерира ребрата. Да кажем, че алгоритъмът извършва сравнение $a_p \stackrel{?}{<} a_q$.

- ① Ако a_p и a_q са корени на антиарборесценции, да наречем тези антиарборесценции D_p и D_q , то,

[†]Няма да правим безсмислени сравнения на елементи, които вече са били сравнени.

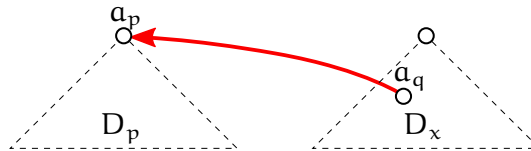
- ♦ ако $|V(D_p)| \leq |V(D_q)|$, противникът отговаря Y и слага черно ребро (a_p, a_q) ,
- ♦ ако $|V(D_p)| > |V(D_q)|$, противникът отговаря N и слага черно ребро (a_q, a_p) .

Посоката на реброто се определя по принципа “от корена на малкото към корена на голямото”.



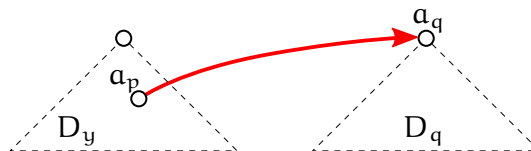
Във всеки случай, D_p и D_q се сливат в една антиарборесценция, така че $\mathcal{F}$ се променя ($|\mathcal{F}|$ намалява с единица).

- ② Ако a_p е корен на антиарборесценция, да я наречем D_p , а a_q принадлежи на антиарборесценцията D_x , но не е неин корен, то противникът отговаря N и слага червено ребро (a_q, a_p) .



D_p и D_x не се сливат, така че $\mathcal{F}$ остава непроменено.

- ③ Ако a_q е корен на антиарборесценция, да я наречем D_q , а a_p принадлежи на антиарборесценцията D_y , но не е неин корен, то противникът отговаря Y и слага червено ребро (a_p, a_q) .



D_q и D_y не се сливат, така че $\mathcal{F}$ остава непроменено.

- ④ Ако нито a_p , нито a_q е корен на антиарборесценция, то противникът отговаря по начин, който е консистентен с досега дадените отговори на сравнения, слагайки или червено ребро (a_p, a_q) , или червено ребро (a_q, a_p) .

a_p и a_q може да са от една антиарборесценция или от различни антиарборесценции. Дори да са от различни антиарборесценции, тези антиарборесценции не се сливат, така че $\mathcal{F}$ остава непроменено.

Очевидно е, че в началото—преди да са направени каквито и да е сравнения—всяка антиарборесценция D_i е тривиален граф $(a_i, \emptyset)$, така че $|\mathcal{F}| = n$. Всяко a_i може да е максимумът в този момент.

Наблюдение 96

Както в началото, така и след всяко сравнение, корените на антиарборесценциите в $\mathcal{F}$ са точно тези елементи, всеки от които може да е максимумът съгласно отговорите на извършените досега сравнения.

Наблюдение 96 се доказва елементарно по индукция по броя на извършените сравнения. Неговото значение става ясно от Лема 49.

Лема 49: Не може намерим втория по големина, преди да намерим максимума

Всяка редица от сравнения, която води до намирането на втория по големина елемент, неизбежно намира и максимума.

Доказателство Да допуснем противното. Нека максимумът е a_i , а вторият по големина елемент е a_j . Допускаме, че коректен алгоритъм $ALGX$ за намиране на втория по големина елемент връща a_j , като от извършените сравнения не става ясно кой е максимумът. Съгласно Лема 48, да бъде намерен максимумът чрез сравнения е същото като дагът на сравненията да има точно един сифон. Ерго, допускането ни е, че $ALGX(a_1, \dots, a_n)$ връща a_j като втори по големина елемент въз основа на сравнения, чиито даг има поне два сифона.

Ако a_j е един от сифоните, противникът може *post factum* да манипулира големините на елементите по такъв начин, че a_j да се окаже максималният елемент. По този начин противникът би опровергал коректността на $ALGX$, защото $ALGX$ би трябвало да връща **втория по големина** елемент, а не максимума. Ако a_j не е един от сифоните, противникът може *post factum* да манипулира големините на елементите по такъв начин, че и двата сифона да са по-големи от a_j , като по този начин a_j не е вторият по големина елемент. И в този случай противникът би опровергал коректността на $ALGX$.

Заклучаваме, че сравненията, извършени за намирането на втория по големина елемент, неизбежно намират и максимума, дори това да не се иска експлицитно в задачата. $\square$

Лема 49 влече, че след приключването на сравненията за намирането на втория максимум, G има само една антиарборесценция, чийто корен е единственият сифон в G и този връх е максимумът според извършените сравнения. Разгледайте пак стратегията на противника ①–④: в някакъв смисъл, противникът отлага колкото може момента, в който остава само една антиарборесценция. С други думи, противникът иска колкото може по-дълго да няма яснота кой е максимумът. Само в случай ①, когато се сравняват корени на антиарборесценции—съгласно Наблюдение 96, тези корени са възможните максимуми до момента—се извършва сливане на антиарборесценции. В този случай сливането е неизбежно, защото все единият трябва да бъде обявен за по-голям от другия, с което другият преставя да е кандидат за максимум. Но ако бъде сравнен корен на антиарборесценция с не-корен на друга антиарборесценция (② или ③), противникът дава отговор, който не слива антиарборесценции; с други думи, кандидатите за максимум остават същите.

Но стратегията на противника е повече от това да отлага момента, в който става ясно кой е максимумът. Както предстои да видим, противникът отговаря така, че налага появата на известен брой червени ребра (които не сливат антиарборесценции и не допринасят за определянето на максимума). В подкрепа на това: крайната антиарборесценция има точно $n - 1$ ребра, които отговарят на $n - 1$ извършени сравнения. Ако това са всички сравнения

при намирането на втория по големина елемент, то долната граница за намирането на втория по големина би била $n - 1$ (колкото е за намиране на максимума).

Лема 50: Стратегията на противника е непротиворечива

Стратегията на противника, описана в ①–④ на стр. 797, генерира граф на сравненията, който е ацикличен.

Доказателство: Ще докажем твърдението по индукция по редицата от извършените сравнения. По същество, това е доказателство с инвариант, който гласи следното: във всеки момент, графът на сравненията е ацикличен и освен това корените на антиарборесценциите имат изходни степени нула.

Последното казва, че от корените не “излизат” червени ребра. Има смисъл това да се акцентира, защото антиарборесценциите са дефинирани като такива по отношение на черните ребра!

Базовият случай е моментът преди началото на сравненията, когато ребра няма изобщо, нито червени, нито черни. Очевидно тогава цикли няма. Да разгледаме произволно сравнение $a_p \stackrel{?}{<} a_q$, допускайки, че твърдението е изпълнено преди него. В случаи ①, ② и ③, генерираното ребро сочи към корена на антиарборесценция, който по допускане има изходна степен нула. Очевидно е, че добавянето на ребро, чийто край е връх от изходна степен нула, не може да доведе до поява на цикъл. Да разгледаме ④, в който случай нито a_p , нито a_q са корени на антиарборесценции и всеки от тях може да е край на черни или червени ребра преди сравнението. Казано е “противникът отговаря по начин, който е консистентен с досега дадените отговори на сравнения”. Това означава следното. Ако от единия връх измежду a_p , a_q има път s до другия преди сравнението $a_p \stackrel{?}{<} a_q$ (тогава от другия няма път до първия, инак би имало цикъл преди сравнението $a_p \stackrel{?}{<} a_q$), то противникът слага реброто така, че да не се получи цикъл (тоест, новото ребро е “по посока на” s). Ако обаче такъв път няма, тоест, a_p и a_q са несравними преди $a_p \stackrel{?}{<} a_q$, то противникът слага произволно или ребро (a_p, a_q) , или ребро (a_q, a_p) ; с други думи, отговаря произволно Y или N . Лесно се вижда, че няма да се получи цикъл от това. Това е и краят на доказателството. $\square$

Допълнение 73: Една задача върху графи и Лема 50

Следната задача е алгоритмичен фолклор.

Задача 66

Даден е граф $G = (V, E)$ с частична ориентация на ребрата. Това означава, че някои от ребрата му са ориентирани, а други са неориентирани. Известно е, че ориентираните му ребра не образуват (ориентиран) цикъл. Да се докаже, че е възможно за да се даде ориентация на всяко от неориентираните ребра по такъв начин, че да не се получи цикъл.

Това е граф-теоретичната формулировка на задачата. Алгоритмичната формулировка иска да се предложи и анализира ефикасен алгоритъм, който дава ориентация на неориентираните ребра, без да образува цикли.

Решението е много просто. Нека E' е множеството от ориентираните ребра, а E'' е множеството от неориентираните ребра. Очевидно $E = E' \cup E''$. Разглеждаме дага $G' = (V, E')$. Топо-сортираме го с линейния алгоритъм и получаваме топо-сортиране L . След

това, за всяко ребро $e = (u, v)$ от E'' , даваме на e такава ориентация, че $L[u] < L[v]$. Неформално казано, ориентиране неориентираните ребра по начин, консистентен с топо-сортирането. Алгоритъмът е с линейна сложност, а коректността му е очевидна: няма как това даване на ориентации на ребра да доведе до цикъл.

Интересно е да се отбележи, че решението на Задача 66 и доказателството на Лема 50 имат доста общо. И в двете, в крайна сметка се получава даг. Произволното отговаряне на сравнението $a_p \stackrel{?}{<} a_q$ от страна на противника, в случай че a_p и a_q не са корени и са несравними според сравненията до момента, в някакъв смисъл е аналог на даването на ориентация, консистентна с топо-сортирането, на неориентираните ребра в решението на Задача 66.

- В Задача 66 имаме подмножество ориентирани от ребра, които индуцират даг, което означава, че те (заедно с върховете) подлежат на топо-сортиране. Даването на посоки на неориентираните ребра, консистентни с топосортирането, е същото като добавяне на ориентирани ребра към даг по начин, запазващ ацикличността.
- В Лема 50 имаме a_p и a_q , които са несравними преди $a_p \stackrel{?}{<} a_q$. Тогава графът на сравненията остава даг независимо от това дали добавим (a_p, a_q) или (a_q, a_p) (не и двете, естествено). Ацикличността се запазва, което е същностното за консистентността на противника.

Теорема 113: Намирането на втория по големина елемент иска $\geq n - 2 + \lceil \log_2 n \rceil$ сравнения

Намирането на второто по големина число измежду n на брой, две по две различни, числа става с поне $n - 2 + \lceil \log_2 n \rceil$ сравнения.

Доказателство: Нека ALGSECMAX е алгоритъм за намиране на втория по големина елемент. Да си представим работата на $\text{ALGSECMAX}(A)$, където A е множество от n числа. Нека алгоритъмът “играе” срещу противник, който следва стратегията, описана в ①–④. Това означава, че поначало A не е фиксирано, а противникът може *post factum* да предостави множество A , което е консистентно с отговорите, които е давал.

Нека $a = \max A$. Нека G е дагът на сравненията в края на работата на $\text{ALGSECMAX}(A)$. Нека D е единствената антиарборесценция в този момент. Очевидно D е подграф на G , понеже има същите върхове, но ребрата на D са само черните ребра. Както знаем от Лема 49, намирайки втория по големина елемент със сравнения, неизбежно намираме и a , поне в теоретико-информационния смисъл: разполагаме с достатъчно информация, за да го идентифицираме, ако искаме. Знаем, че a е единственият сифон в G , но a е единственият сифон и в D , понеже сифоните на антиарборесценциите по време на работата на алгоритъма са точно кандидатите за максимум.

Да разгледаме множеството от върхове

$$S = \{b \in A \mid (b, a) \in E(D)\}$$

Очевидно S се състои точно от децата на a в антиарборесценцията D . Тогава $\text{secondmax } A \in S$, тъй като всеки елемент на $A \setminus (S \cup \{a\})$ е на поне две (черни) ребра разстояние до a (в D); тоест, е загубил от поне две числа.

Лема 51

В текущия контекст, $|S| \geq \lceil \log_2 n \rceil$.

Доказателство: Ще докажем твърдението с индукция по сливането на антиарборесценции. А именно, ще докажем, че за всяка антиарборесценция D_i , получена по време на сравненията, ако S_i е множеството от децата на нейният корен и n_i е броят на нейните върхове, то $|S_i| \geq \lceil \log_2 n_i \rceil$.

Ключово за доказателството е, че броят на върховете на антиарборесценция най-много бива удвоен при добавяне на черно ребро.

Базата е началният момент преди сравненията. Всяка антиарборесценция в този момент има точно един връх, което означава, че множеството от децата на корена е празното множество. Очевидно е изпълнено $|\emptyset| \geq \lceil \log_2 1 \rceil$. ✓

Да допуснем, че по време на работата на алгоритъма има две антиарборесценции D_1 и D_2 с корени съответно u_1 и u_2 . Да кажем, че $|V(D_i)| = n_i$ и S_i е множеството от децата u_i в D_i , за $i = 1, 2$. БОО, нека $n_1 \leq n_2$. Индуктивното допускане е, че $|S_i| \geq \lceil \log_2 n_i \rceil$, за $i = 1, 2$. Нека u_1 и u_2 бъдат сравнени. Съгласно ①, D_1 и D_2 биват слети в една антиарборесценция D' чрез добавяне на черно ребро (u_1, u_2) . Нека S' е множеството от децата на корена на D' . Очевидно $S' = \{u_1\} \cup S_2$, ерго $|S'| = |S_2| + 1$. Но

$$\begin{aligned} |S'| &= |S_2| + 1 \quad // \text{от индуктивното предположение} \\ &\geq \lceil \log_2 n_2 \rceil + 1 = \lceil (\log_2 n_2) + 1 \rceil = \lceil \log_2 2n_2 \rceil \quad // \text{допуснахме, че } n_1 \leq n_2 \\ &\geq \lceil \log_2 (n_2 + n_1) \rceil = \lceil \log_2 |V(D')| \rceil \quad \square \end{aligned}$$

И така, разглеждаме пак края на работата на ALGSECMAX. S се състои точно от елементите, всеки от които може да е вторият по големина, поне що се отнася до черните ребра. Ерго, алгоритъмът може да определи точно кой е вторият по големина само с помощта на червени ребра, чиито краища са елементи на S . Тогава G съдържа поне $|S| - 1$ червени ребра с краища от S ; аргументацията по същество е тази от Подсекция 13.4.4. Но Лема 51 казва, че $|S| \geq \lceil \log_2 n \rceil$. Тогава G съдържа поне $\lceil \log_2 n \rceil - 1$ червени ребра.

Но G също така съдържа и черните ребра, които са точно $n - 1$, понеже те индуцират антиарборесценция с n върха. Тогава ребрата на G са поне

$$n - 1 + \lceil \log_2 n \rceil - 1 = n - 2 + \lceil \log_2 n \rceil$$

Заклучаваме, че ALGSECMAX прави поне толкова сравнения. □

Допълнение 74: Грешно д-во на долната гр. $n - 2 + \lceil \log_2 n \rceil$ за втори макс.

Следното доказателство на долната граница $n - 2 + \lceil \log_2 n \rceil$ за броя на сравненията за намиране на втория най-голям елемент на n -елементно множество е широко разпространено онлайн. Доколкото авторът на записките успя да проучи, оригиналът е помощен учебен материал (handout) на Michael Goldwasser от 2003 г., озаглавен "Sample proofs involving selection problems"^a. Това доказателство е сравнително по-просто от доказателството, което вече видяхме горе. За съжаление, то е грешно. Първо да видим

доказателството и после ще видим контрапример.

Търси се вторият максимален елемент на $A = \{a_1, \dots, a_n\}$ чрез сравнения. Да кажем, че $a_i = \max A$ и $a_j = \text{secondmax } A$, като очевидно $j \neq i$. Да допуснем, че алгоритъм за намиране на втория максимален елемент е идентифицирал a_j като такъв в резултат на някакви сравнения и G е дагът на сравненията.

Лема 52

В текущия контекст, има поне $n - 2$ сравнения, в които a_i не участва.

Доказателство: Нека $G' = G - a_i$. Твърдим, че G' има един единствен сифон и това е a_j . Първо, забелязваме, че a_j е сифон в G' . Иначе противникът *post factum* може да манипулира големините на елементите по такъв начин, че a_j да не е вторият по големина елемент, като резултатите от сравненията се запазят и той (противникът) не може да бъде уличен в лъжа.

Второ, забелязваме, че a_j е единственият сифон в G' . Иначе противникът *post factum* може да манипулира големините на сифоните по такъв начин, че a_j да не е вторият по големина елемент, като резултатите от сравненията се запазят и той (противникът) не може да бъде уличен в лъжа.

И така, a_j е единственият сифон в G' . Тогава G' е слабо свързан, така че има поне $n - 2$ ребра (защото има $n - 1$ върхове). Тези ребра са сравнения, в които a_i не участва. $\square$

Същността на доказателството е да се покаже, че освен тези (поне) $n - 2$ сравнения, в които a_i не участва, има още поне $\lceil \log_2 n \rceil$, в които a_i участва. Ако това бъде доказано, цялото доказателство е готово, тъй като тези множества от сравнения имат празно сечение и мощността на обединението им е $n - 2 + \lceil \log_2 n \rceil$.

Лема 53

В текущия контекст, има поне $\lceil \log_2 n \rceil$ сравнения, в които a_i участва.

Доказателство: Ще опишем стратегия на противника, която принуждава алгоритъма, намиращ втори по големина елемент, да извърши такива сравнения. По време на извършване на сравненията противникът поддържа (изчислява за себе си) множества $S_1, S_2, \dots, S_n$. За всяко $k \in \{1, \dots, n\}$, S_k е състои от точно тези елементи x , за които от направените досега сравнения следва, че $x \leq a_k$. Очевидно S_k се инициализира с $\{a_k\}$, за $k \in \{1, \dots, n\}$, понеже поначало противникът не се е ангажирал с никакви отговори.

Когато му бъде зададен въпрос $a_p \stackrel{?}{<} a_q$, за някакви различни $p, q \in \{1, \dots, n\}$, противникът отговаря по следния начин съгласно мощностите на S_p и S_q .

- Ако $a_p \in S_q$, то досега извършените сравнения влекат, че $a_p < a_q$ и противникът отговаря Y . Инак би бил уличен в лъжа. Ако $a_q \in S_p$, то досега извършените сравнения влекат, че $a_p > a_q$ и противникът отговаря N . Инак би бил уличен в лъжа. Забележете, че е невъзможно $a_p \in S_q$ и $a_q \in S_p$, инак противникът би бил уличен в лъжа.

Забележете също така, че е глупаво от страна на алгоритъма да прави сравнение, резултатът от който е предизвестен от вече извършени сравнения, но това е за сметка на този, който прави сравненията. В случая противникът се радва, че

е направено излишно сравнение, но внимава отговорът му да е консистентен с предишния отговор.

- Нека $a_p \notin S_q$ и $a_q \notin S_p$. Тогава a_p и a_q са несравними според досега наличната информация и противникът има свободата да отговори както Y , така и N , без това да даде основание да бъде уличен в лъжа.

- ♦ ако $|S_p| < |S_q|$, противникът отговаря Y и прави $S_q \leftarrow S_p \cup S_q$.
- ♦ ако $|S_p| > |S_q|$, противникът отговаря N и прави $S_p \leftarrow S_p \cup S_q$.
- ♦ ако $|S_p| = |S_q|$, противникът отговаря Y и прави $S_q \leftarrow S_p \cup S_q$.

Ключовото наблюдение е, че след всяко сравнение, за всяко S_k е вярно, че или $|S_k|$ остава същата, или нараства, но ако нараства, най-много се удвоява. Но след края на сравненията очевидно е изпълнено $|S_i| = n$; няма как да е намерен максимумът a_i , ако от извършените сравнения не следва, че той и всеки друг елемент е по-малък или равен на a_i .

И така, в началото $|S_i| = 1$, а в края $|S_i| = n$, и освен това всяко сравнение най-много удвоява $|S_i|$. Заклучаваме, че са извършени поне $\log_2 n$ сравнения, единият участник в които е a_i . Но броят на сравненията е цяло число. Ерго, извършени са поне $\lceil \log_2 n \rceil$, в които a_i участва. $\square$

Теорема: Намирането на втория по големина елемент става с $\geq n - 2 + \lceil \log_2 n \rceil$ сравнения

Намирането на втория по големина елемент става с поне $n - 2 + \lceil \log_2 n \rceil$ сравнения.

Доказателство: Следва веднага от Лема 52 и Лема 53. $\square$

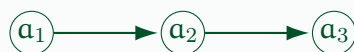
Твърдението е вярно, но доказателство обаче е **грешно**. Лема 52 е вярна. Проблемът е в Лема 53. Ето контрапример. Нека $n = 3$. Тогава $A = \{a_1, a_2, a_3\}$. Очевидно $S_1 = \{a_1\}$, $S_2 = \{a_2\}$ и $S_3 = \{a_3\}$ преди правенето на сравненията. В този момент, дагът на сравненията е:



Да кажем, че първото сравнение е $a_2 \stackrel{?}{<} a_3$. Тъй като $|S_2| = |S_3|$, съгласно стратегията на противника, посочена в Лема 53, неговият отговор е Y . Вече $S_1 = \{a_1\}$, $S_2 = \{a_2\}$ и $S_3 = \{a_2, a_3\}$. Дагът на сравненията става:



Да кажем, че второто сравнение е $a_1 \stackrel{?}{<} a_2$. Очевидно отговор Y подрежда елементите в линейна наредба и става ясно кой е вторият елемент. Ерго, ако противникът е умен, **трябва** да отговори N . Обаче противника ще отговори Y , ако следва стратегията от Лема 53, понеже в сила е $|S_1| \geq |S_2|$. При това множествата стават $S_1 = \{a_1\}$, $S_2 = \{a_1, a_2\}$ и $S_3 = \{a_1, a_2, a_3\}$, а дагът на сравненията става:



Противникът, следвайки стратегията от Лема 53, допусна нашият алгоритъм да намери вторият по големина елемент със само 2 сравнения. Но $n - 2 + \lceil \log_2 n \rceil$ при $n = 3$ е

$$3 - 2 + \lceil \log_2 3 \rceil = 3 - 2 + 2 = 3$$

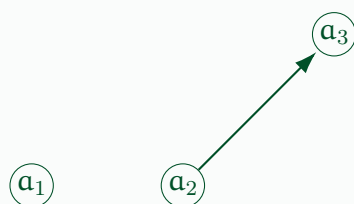
Ерго, противникът, следващ стратегията от Лема 53, не е (достатъчно) умен. Формално, не е вярно, че за всяко n , стратегията на противника, описана в Лема 53, води до поне $n - 2 + \lceil \log_2 n \rceil$ сравнения.

Проблемът със стратегията от Лема 53 е, че релацията $<$ е транзитивна, така че отговор на въпроса $a_p \stackrel{?}{<} a_q$ може да увеличи мощностите на множества S_k за $k \neq p, q$. В контрапримера, който видяхме току-що, отговаряйки Y на сравнението $a_1 \stackrel{?}{<} a_2$, противникът променя S_3 от $\{a_2, a_3\}$ на $\{a_1, a_2, a_3\}$. “Доказателството” на Лема 53 е основано на невярното допускане, че отговорът на въпроса $a_p \stackrel{?}{<} a_q$ не засяга S_k за $k \neq p, q$.

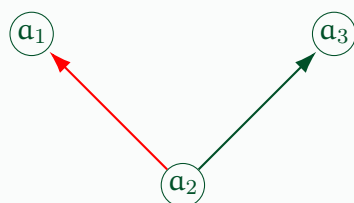
Сега да видим как действа противникът, дефиниран на стр. 797, върху примера $\{a_1, a_2, a_3\}$. Началото е същото:



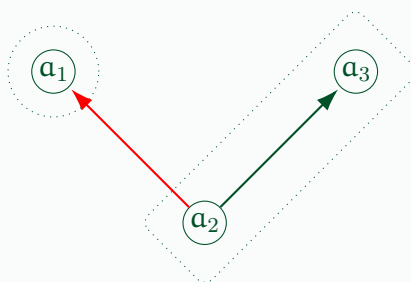
Нека първото сравнение е $a_2 \stackrel{?}{<} a_3$. И a_2 , и a_3 е корен на антиарборесценция. Множествата на върховете на тези антиарборесценции са равномощни. Съгласно ①, противникът казва, че $a_2 < a_3$ и слага черно ребро (a_2, a_3) :



Дотук нещата са същите. Нека второто сравнение е $a_1 \stackrel{?}{<} a_2$, както и преди. Сега a_1 е корен на антиарборесценция, но a_2 не е. Съгласно ②, противникът отговаря N и слага червено ребро (a_2, a_1) :



Забележете, че червеното ребро не слива анти-арборесценции, така че арборесценциите остават две:



Противникът принуждава алгоритъма да направи и трето сравнение, с което “потвърждава” долната граница $n - 2 + \lceil \log_2 n \rceil$. Третото сравнение е $a_1 \stackrel{?}{<} a_3$ (или $a_3 \stackrel{?}{<} a_1$, няма значение). Отговорът на противника е, че $a_1 < a_3$ по принципа, че посоката е от по-малко към по-голямо, но това няма значение за това, което правим в момента.

^aИзглежда, че този handout вече не е намерим с търсачка онлайн. Може би авторът му го е премахнал?

13.4.7 Долна граница $\lceil \frac{3n-2}{2} \rceil$ за намиране на медиана

Фундамент. Става дума за Задача 20. Естествено, разглеждаме намиране на медиана само чрез сравнения. Алгоритъмът COMPUTE MEDIAN на стр. 289 намира медианата точно така, ако сортирането е базирано на сравнения. Алгоритъмът SELECT от Подсекция 6.2.2 може да се използва за намиране на медиана, като и той е базиран на сравнения.

Историческата справка накратко е следната. Blum, Floyd, Pratt, Rivest и Tarjan [23], [24] първи описват алгоритъм, който намира k -тото по големина число измежду n числа. Ние видяхме този алгоритъм в Подсекция 6.2.2 под името SELECT. В частност, този алгоритъм може да се ползва за намиране на медианата в линейно време. Авторите доказват, че броят на сравненията не надхвърля $5.4305n$. В края на статията [24] е доказана и долна граница за броя на сравненията, която за медианата е $1.5n$. В тази подсекция ще видим точно това доказателство за долна граница, само че обяснено бавно и подробно.

Маржът между горната граница $5.4305n$ и долната граница $1.5n$ е значителен. Подобреения има и от двете страни. Доколкото е известно на автора на записките, най-ефикасното намиране на медиана става с $\leq 2.95n + o(n)$ сравнения [39], а най-високата долна граница за броя на сравненията е $2n + o(n)$ [14]. Всичко това е добре обяснено в [40].

Тук, разбира се, не говорим за асимптотика! Асимптотиката е ясна: от една страна, имаме тривиална долна граница $\Omega(n)$ за намирането на медианата, а от друга страна можем да намерим медианата с $O(n)$ алгоритъм SELECT. Ерго, на ниво асимптотика имаме съвпадение на долната и горната граница. Разминаването се получава само ако отчитаме мултипликативната константа.

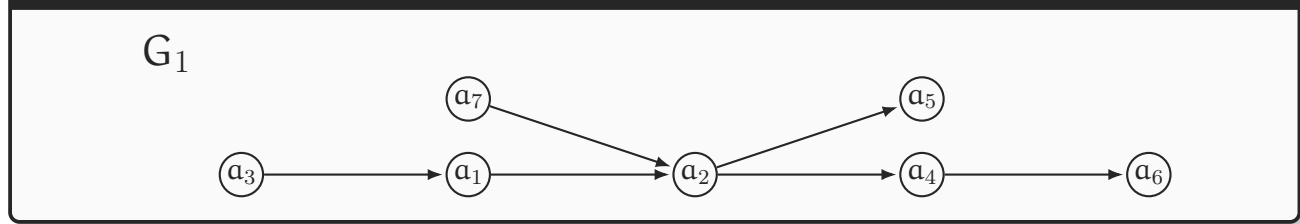
При задачата за намиране на максимума и минимума имаме перфектно, с точност до единица, съвпадение на долната и горната граница, а именно $\lceil \frac{3n}{2} \rceil - 2$ сравнения (Подсекция 13.4.5). Въпросното разминаване при медианата без съмнение се дължи на това, че намирането на медиана е доста по-трудна задача.

Дагът на сравненията и медианата. Ключово понятие и тук е даг на сравненията (Определение 109). Нека входът е $a_1, a_2, \dots, a_n$. Нека $A = \{a_1, \dots, a_n\}$. За простота допусваме, че числата са две по две различни и че са нечетен брой. За всеки даг на сравненията G , за всяко $a_i \in A$, ще казваме, че a_i е *възможна медиана по отношение на G* , ако съществува топологическо сортиране h на G , за което $h(a_i) = \lceil \frac{n}{2} \rceil$. Като екстремален пример, ако G е празен (не са извършени никакви сравнения), всеки елемент е възможна медиана. В другата крайност, ако G съдържа Хамилтонов път p , има една единствена възможност за медиана и това е връх номер $\lceil \frac{n}{2} \rceil$ в p . Известно е, че даг има Хамилтонов път тстк този даг има единствено топологическо сортиране. Ерго, наличието на Хамилтонов път в G влече, че има само една възможна медиана по отношение на извършените сравнения и резултатите от тях.

Но не е необходимо да има Хамилтонов път в дага на сравненията, за да има единствена възможна медиана. Фигура 13.20 показва такъв пример със седем елемента. Дагът на

сравненията е наречен G_1 . Той няма Хамилтонов път, така че има няколко топологически сортирания—девет на брой, ако трябва да сме прецизни—но във всяко от тях, елементът на четвърта позиция е a_2 , така че a_2 е единствената възможна медиана.

Фигура 13.20 : Няма Хамилтонов път, но a_2 е в средата на всяко топо-сортиране.



Това да има една единствена възможна медиана е ключово. Ако спрямо дага на сравненията има поне две възможни медиани a_i и a_j , не можем да определим коя е медианата само от извършените сравнения. Тук аргументацията с противника е в пълна сила.

- Ако, по каквато и да е причина, алгоритъмът ни върне a_i , противникът ще даде такива стойности на елементите от A , че медианата ще се окаже a_j .
- Ако, по каквато и да е причина, алгоритъмът ни върне a_j , противникът ще даде такива стойности на елементите от A , че медианата ще се окаже a_i .

Забележете колко лесна е задачата на противника. Щом има две различни топологически сортирания на дага на сравненията, в едното от което a_i е средният елемент, а в другото a_j е средният елемент, той просто трябва да наложи това топологическо сортиране, което е несъвместимо с изхода на алгоритъма. Бивайки неограничено умен, за него е детска игра да подбере такива стойности на елементите от входа, че тяхната сортирана редица да е тази от “лошото” топо-сортиране.

Забележете също, че противникът не се интересува изобщо от конкретиката на нашия алгоритъм (който той опровергава). По какви точно причини нашият алгоритъм връща a_i или a_j , него не го интересува. За него е важно, че може да опровергае изхода на алгоритъма, манипулирайки входа по начин, който е съвместим резултатите от сравненията.

Да разсъждаваме върху това, какво е необходимото и достатъчно условие да има само една възможна медиана спрямо дага на сравненията G . Както вече казахме, наличието на Хамилтонов път е достатъчно, но не е необходимо. Ако дагът на сравненията трябваше да съдържа Хамилтонов път, задачата за намиране на медиана щеше да има сложността на сортирането $\Omega(n \lg n)$. А ние знаем, че медиана може да се намира в линейно време. Затова няма да настояваме G да има Хамилтонов път.

Да въведем следните подмножества на входа по отношение на кой да елемент от входа a_m .

$$S(a_m) \stackrel{\text{def}}{=} \{a_i \in A \setminus \{a_m\} \mid \text{в } G \text{ има път от } a_i \text{ до } a_m\}$$

$$L(a_m) \stackrel{\text{def}}{=} \{a_i \in A \setminus \{a_m\} \mid \text{в } G \text{ има път от } a_m \text{ до } a_i\}$$

В примера от Фигура 13.20, $S(a_3) = \emptyset$, $L(a_3) = \{a_1, a_2, a_4, a_5, a_6\}$, $S(a_1) = \{a_3\}$, $L(a_1) = \{a_2, a_4, a_5, a_6\}$, $S(a_2) = \{a_1, a_3, a_7\}$, $S(a_2) = \{a_4, a_5, a_6\}$ и така нататък. Това, че задължително $S(a_m) \cap L(a_m) = \emptyset$, е напълно очевидно: общ елемент в тези множества би бил индикация за цикъл в дага G .

От една страна, елементът a_2 е единственият, който може да е медиана спрямо тези резултати от сравненията – това е доста очевидно от Фигура 13.20. От друга страна, той е единственият a_m , за който $S(a_m)$ и $L(a_m)$ представляват разбиване на $A \setminus \{a_m\}$ и освен това са равномошни. Това не е случайно, както показва Лема 54. Първо една дефиниция.

Определение 113: Елементи, сравними с друг елемент

Елементите на A , които са *сравними* с a_m , са елементите на $S(a_m) \cup L(a_m)$.

На прост български, сравнимите с a_m елементи са точно тези, за които можем да кажем въз основа на извършените сравнения, че са по-малки или по-големи от a_m . За тези, които не са сравними с a_m , не можем да кажем дали са по-големи или по-малки от a_m въз основа на извършените сравнения. По отношение на G , сравнимите елементи са точно тези върхове, различни от a_m , които или са достижими от a_m , или a_m е достижим от всеки от тях.

Лема 54

В текущия контекст, a_m е единствената възможна медиана т.е. $|S(a_m)| = |L(a_m)| = \lfloor \frac{n}{2} \rfloor$.

Доказателство: В едната посока, нека a_m е единствената възможна медиана. Веднага забелязваме, че всеки елемент от $A \setminus \{a_m\}$ е сравним с a_m – ако има поне един $a_i \neq a_m$, който не е сравним с a_m , то противникът може *post factum* да даде на a_i или стойност, по-голяма от a_m , или стойност, по-малка от a_m , без да бъде уличен в лъжа, понеже a_i не е сравним с a_m . По този начин винаги има възможност противникът да генерира *post factum* такъв вход, който е съвместим с резултатите от сравненията, но в който a_m не е средният по големина елемент.

От това и очевидния факт, че $S(a_m)$, $L(a_m)$ и $\{a_m\}$ имат две по две празно сечение, следва, че $\{S(a_m), L(a_m), \{a_m\}\}$ е разбиване на A .

Сега съобразяваме, че за всяко топологическо сортиране h на G е вярно, че елементите на $S(a_m)$ са вляво от a_m , а елементите на $L(a_m)$ са вдясно от a_m . Това е тривиално наблюдение, което е истина независимо от това дали a_m е възможна медиана или не. Ако допуснем обратното, веднага заключаваме, че за поне едно ребро $(u, v) \in E(G)$ е изпълнено $h(v) < h(u)$, което е невъзможно съгласно Определение 73.

Щом $\{S(a_m), L(a_m), \{a_m\}\}$ е разбиване на A и елементите на $S(a_m)$ са вляво от a_m , а елементите на $L(a_m)$ са вдясно от a_m и $|A| = n$ и n е нечетно, веднага заключаваме, че $|S(a_m)| = |L(a_m)| = \lfloor \frac{n}{2} \rfloor$.

В обратната посока, нека $|S(a_m)| = |L(a_m)| = \lfloor \frac{n}{2} \rfloor$. Но трите множества $S(a_m)$, $L(a_m)$ и $\{a_m\}$ имат две по две празно сечение, така че $|S(a_m) \cup L(a_m) \cup \{a_m\}| = |S(a_m)| + |L(a_m)| + |\{a_m\}|$. Щом $|S(a_m)| = |L(a_m)| = \lfloor \frac{n}{2} \rfloor$, в сила е $|S(a_m) \cup L(a_m) \cup \{a_m\}| = \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor + 1$. Но n е нечетно, така че $\lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2} \rfloor + 1 = n$. Заключаваме, че $|S(a_m) \cup L(a_m) \cup \{a_m\}| = n$. Но $n = |A|$. Щом $S(a_m)$, $L(a_m)$ и $\{a_m\}$ имат две по две празно сечение и мощността на обединението им е $|A|$, $\{S(a_m), L(a_m), \{a_m\}\}$ е разбиване на A .

От това и от факта, че за всяко топологическо сортиране h на G е вярно, че елементите на $S(a_m)$ са вляво от a_m , а елементите на $L(a_m)$ са вдясно от a_m веднага следва, че a_m е единствената възможна медиана. $\square$

Фигура 13.21 илюстрира Лема 54, показвайки два дага на сравненията.

- G_1 е дагът от Фигура 13.20, в който всички елементи на сравними с a_2 и a_2 е единствената възможна медиана.

- G_2 е различен даг. В него a_6 не е сравним с a_2 и наистина a_2 не е единствената възможна медиана. Примерно, ако противникът даде стойности $a_3 = 1$, $a_1 = 2$, $a_7 = 3$, $a_2 = 10$, $a_4 = 11$, $a_5 = 12$ и $a_6 = 9$, сортираната редица ще стане

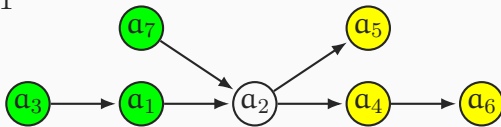
$$\langle a_3, a_1, a_7, a_6, a_2, a_4, a_5 \rangle$$

и медианата ще е a_6 , а не a_2 .

И в G_1 , и в G_2 , върховете от $S(a_2)$ са в зелено, а върховете от $L(a_2)$ са в жълто. В сиво е a_6 от G_2 , който е извън $S(a_2) \cup L(a_2)$, що се отнася до G_2 .

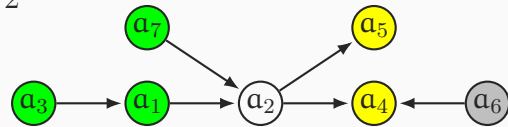
Фигура 13.21 : Единствена медиана е сравнима с всички елементи.

G_1



a_2 задължително е медианата.

G_2



a_2 може да е, но може и да не е, медианата.

Следствие 36: Ако медианата е единствено възможна, дагът на ср. е слабо свързан

Ако дагът на сравненията определя медианата еднозначно, то той е слабо свързан.

Доказателство: Съгласно Лема 54, за всеки даг на сравненията, който определя медианата еднозначно, е вярно, че всеки друг елемент е сравним с медианата. Ако допуснем, че дагът на сравненията има поне две слабо свързани компоненти, веднага стигаме до противоречие: от която и слабо свързана компонента да е медианата, върховете от другите слабо свързани компоненти не са сравними с нея. $\square$

Следствие 37

Ако дагът на сравненията определя медианата еднозначно, той има поне $n - 1$ ребра.

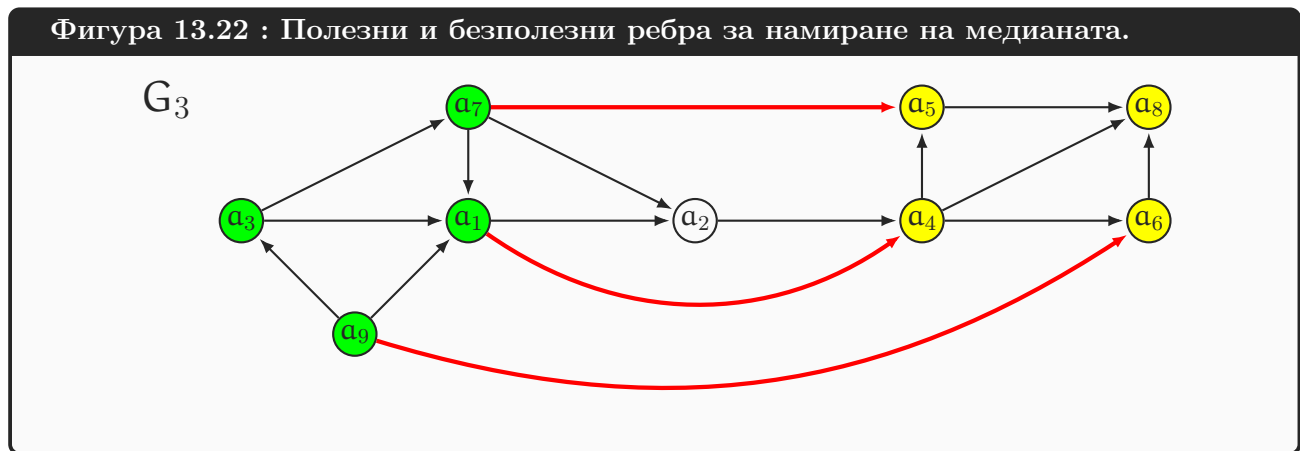
Доказателство: Следва веднага от Следствие 36: всеки слабо свързан ориентиран граф има поне $n - 1$ ребра по същите причини, по които всеки свързан неориентиран граф има поне $n - 1$ ребра. $\square$

Долната граница за броя на сравненията $n - 1$ обаче е твърде слаба. Сега ще я подобрим. Ще опишем стратегия на противника, която води до това, че всеки алгоритъм за намиране на медиана прави поне $\lfloor \frac{3n-2}{2} \rfloor$ сравнения.

Определение 114: Полезните и безполезните ребра на дага на сравненията

В текущия контекст, да кажем, че има една единствена възможна медиана a_m . *Полезните пътища* в G са точно тези пътища, които съдържат a_m като връх. Всяко ребро на G , което се съдържа в поне един полезен път, е *полезно ребро*. Всяко друго ребро на G е *безполезно ребро*.

Фигура 13.20 не е добра илюстрация на тези понятия, защото дагът е максимално “икономичен”, имайки точно $n - 1$ ребра, всички от които са полезни. Вижте дага G_3 на Фигура 13.22. Той има и полезни, и безполезни ребра. Полезните са ребрата в черно, а безполезните, в червено. Единствено възможната медиана е a_2 . Както и преди, върховете от $S(a_2)$ са в зелено, а върховете от $L(a_2)$ са в жълто.



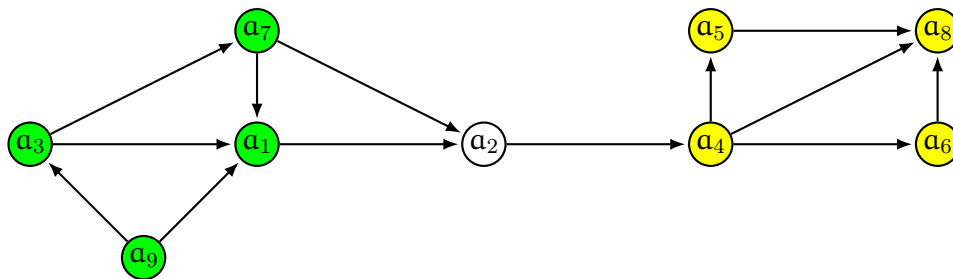
Лесно се вижда, че всяко ребро (u, v) е полезно тстк или $u, v \in S(a_m) \cup \{a_m\}$, или $u, v \in L(a_m) \cup \{a_m\}$. Иначе казано, то е безполезно тстк $u \in S(a_m)$ и $v \in L(a_m)$. Образно казано, безполезните ребра са точно тези, които “прескачат над” a_m .

Изборът на термина “безполезно ребро” е уместен. По отношение на намирането на медианата, сравненията, отговарящи на безполезните ребра, не ни дават **нищо**. Тях можеше и да ги няма, и пак щяхме да намерим a_m като единствено възможната медиана. За да се убедите, че е така, погледнете пак Лема 54 и си припомнете, че

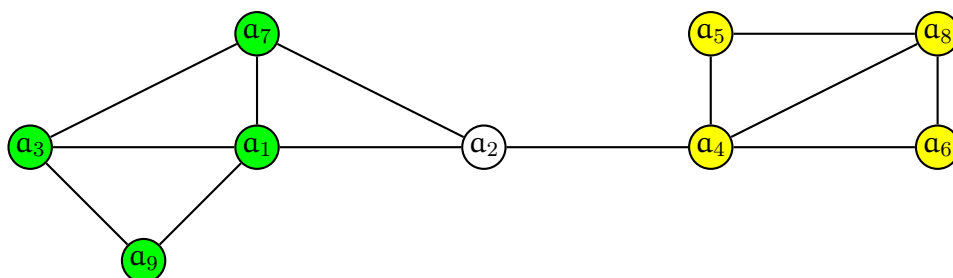
- $S(a_m)$ са върховете, от които a_m е достижим, а това става само чрез пътища, завършващи с a_m ; тези нито един от тези пътища не може да съдържа безполезно ребро.
- $L(a_m)$ са върховете, достижими от a_m , а това става само чрез пътища, започващи с a_m ; тези нито един от тези пътища не може да съдържа безполезно ребро.

Терминът “полезно ребро” може да е спорен. В общия случай не е вярно, че всяко полезно ребро е от ключово значение за намиране на единствената медиана a_m . В дага на Фигура 13.20 наистина всяко полезно ребро е от ключово значение за установяване, че a_2 е единствената медиана, но това е така, защото този даг е минимален като брой ребра – изтрием ли ребро от него, a_2 няма да е единствената възможна медиана. В дага на Фигура 13.22, от друга страна, всяко полезно ребро без (a_2, a_4) може да бъде изтрито и при това останалите полезни ребра продължават да “налагат” a_2 като единствено възможната медиана. Реброто (a_2, a_4) може да бъде наречено *критично*, понеже без сравнението $a_2 \stackrel{?}{<} a_4$ не може да сме сигурни, че a_2 е единствено възможната медиана.

Работата е там, че **всяко** полезно ребро **може** да служи за установяване на единствеността на медианата. Да изтрием безполезните ребра, за да не ни се пречат в разсъжденията. Графът от Фигура 13.22 ще стане този:



За да бъде a_m единствено възможната медиана, достатъчно е да имаме подмножество на полезните ребра с мощност $n - 1$, което индуцира покриващ ориентиран граф. Да се убедим в това. Да премахнем ориентацията на ребрата, получавайки неориентиран граф. Той е свързан, защото дагът беше слабо свързан. В примера, който разглеждаме, получаваме този граф.



В този неориентиран граф всяко покриващо дърво (което очевидно има $n - 1$ неориентирани ребра) налага a_m като единствено възможната медиана. В смисъл, че съответните ориентирани полезни ребра в оригиналния даг представляват множество от сравнения, при чиито резултати единствено възможната медиана е a_m . Сега е напълно ясно защо ориентираниято ребро (a_2, a_4) бе наречено критично преди малко в контекста на дага: в неориентирания граф на него съответства мост, а от курса Дискретни Структури знаем, че всеки мост участва във всяко покриващо дърво.

Лесно се показва, че във всеки свързан граф, всяко ребро участва в някое покриващо дърво. Пренесено в дага на сравненията, това казва, че всяко полезно ребро участва в някое подмножество полезни ребра с минимална мощност (а именно, $n - 1$), което налага a_m да е единствено възможната медиана. А критичните полезни ребра участват във всяко подмножество полезни ребра с минимална мощност (а именно, $n - 1$), което служи за установяване, че a_m е единствено възможната медиана.

След като се уверихме, че термините “безполезни ребра” и “полезни ребра” са смислени, да се върнем на стратегията на противника. Както знаем, противникът не си прави труда да генерира вход поначало. Поначало вход няма. Алгоритъмът задава въпроси от вида $a_i \stackrel{?}{<} a_j$, а противникът отговаря, като при това генерира масив, от който после може да генерира вход, консистентен с отговорите, които е дал.

Противникът категоризира елементите на A като *малки* и *големи*, като за целта поддържа масив $S[1..n]$ от числа. Знаем, че n е нечетно; нека $n = 2k + 1$ за някое естествено k . Подмасивът $S[1..k]$ е за малките числа, а подмасивът $S[k + 2..n]$ е за големите числа. $S[k + 1]$ не е нито малко, нито голямо – то е за медианата. В самото начало S е празен, но след всяко сравнение $a_i \stackrel{?}{<} a_j$ противникът слага или и двете, или точно едното, или нито едното, от a_i, a_j в S съгласно правило, което ще видим след малко. След приключването на сравненията, S представлява линейна наредба на числата от A , като малките са в клетки 1 до k , големите са в клетки $k + 2$ до n , а клетка $k + 1$ съдържа медианата. Ако след

приключването на сравненията поискаме от противника да ни покаже входа, той ще покаже числата в строго нарастващ ред на стойностите според позициите им в S . Най-лесно ще му бъде, на всяко число от S да даде стойност, равна на позицията му в S . По този начин най-малкото число във входа ще има стойност 1, най-голямото ще има стойност n , а медианата ще има стойност $k + 1$.

Масивът S е за “вътрешна употреба” от противника. Алгоритъмът, тоест, ние, не виждаме съдържанието му по време на сравненията. Чак в края, ако поискаме да видим какви точно са числата от входа, противникът ще ползва S , за да ни даде стойности на входа, които са консистентни с дадените отговори на сравнения.

В самото начало противникът инициализира две индексни променливи s и ℓ така:

$$s \leftarrow 1$$

$$\ell \leftarrow n$$

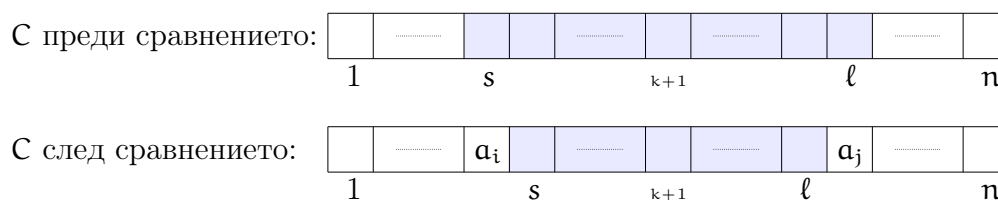
Когато направим сравнение $a_i \stackrel{?}{<} a_j$, противникът слага a_i и a_j в S по следното правило.

- Ако нито a_i , нито a_j са участвали в сравнение досега, то противникът отговаря Y и прави

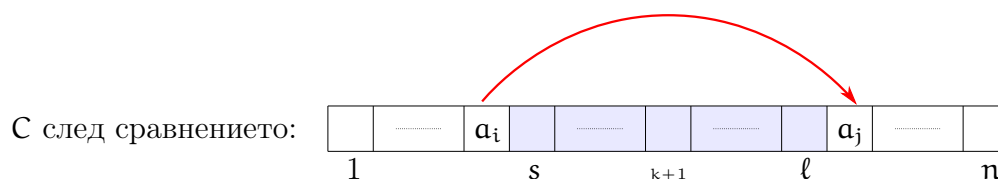
$$S[s] \leftarrow a_i, s++$$

$$S[\ell] \leftarrow a_j, \ell--$$

Нагледно, ето какво става. Със светлосин фон са маркирани свободните клетки в S .



По този начин сравнението става безполезно – очевидно в дага на сравненията това сравнение дава безполезно ребро, което “прескача над” медианата.

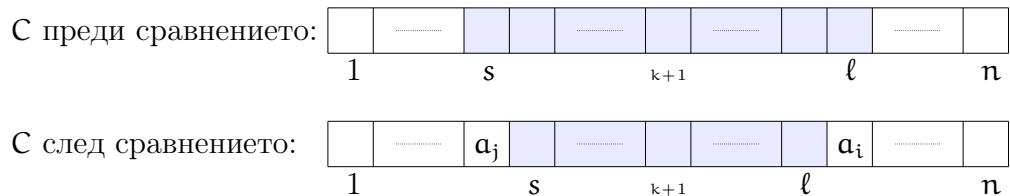


Естествено, решението a_i да стане малко число, а a_j да стане голямо число е произволно. Със същия успех противникът можеше да отговори N и да направи

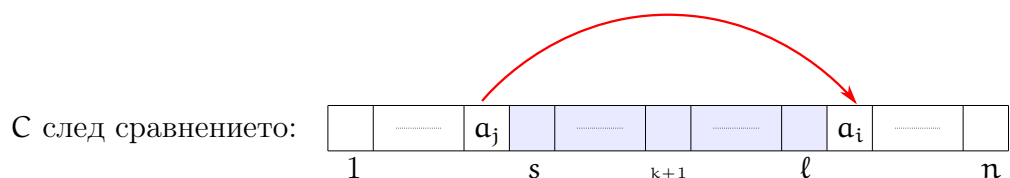
$$S[s] \leftarrow a_j, s++$$

$$S[\ell] \leftarrow a_i, \ell--$$

Нагледно,



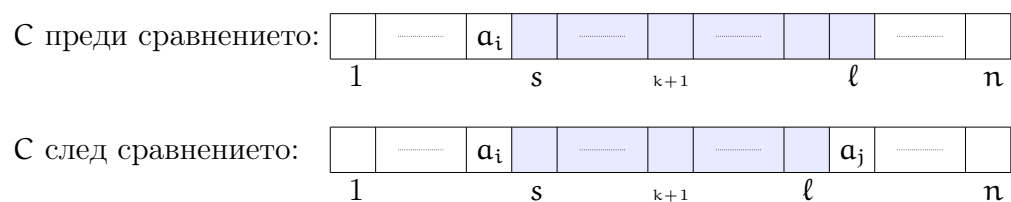
Сравнението пак е безполезно, защото реброто пак е безполезно.



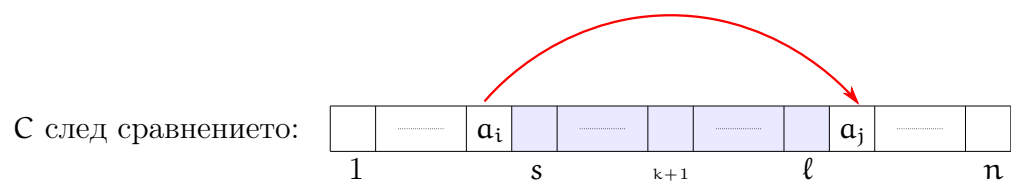
- Нека точно едното от a_i , a_j е участвало в сравнение досега. БОО, нека това е a_i .
 - ♦ Ако a_i е малко число, то противникът отговаря Y и прави a_j голямо число така

$$C[\ell] \leftarrow a_j, \ell--$$

Нагледно:

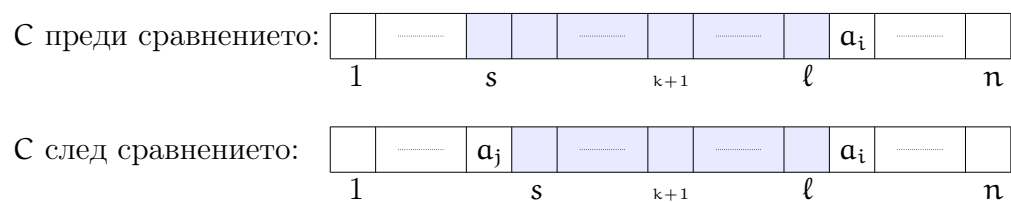


По този начин сравнението става безполезно.

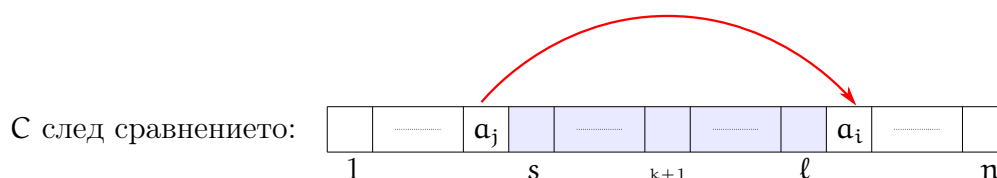


- ♦ Ако a_i е голямо число, то противникът отговаря N и прави a_j малко число така
- $$C[s] \leftarrow a_j, s++$$

Нагледно:



По този начин сравнението става безполезно.

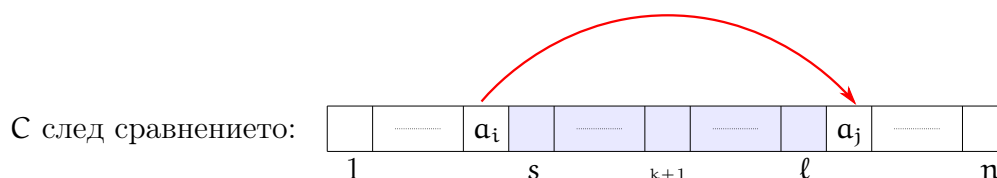


- Остава да видим възможността и a_i , и a_j да са участвали в сравнения досега[†]. Тогава противникът казва истината според съдържанието на S , без да променя нищо в S .

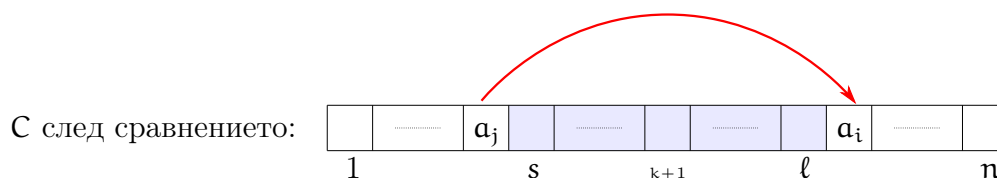
Щом и a_i , и a_j са били сравнявани вече, всяко от тях е или малко, или голямо. Това обаче го знае противникът! Той поддържа S . Алгоритъмът няма представа какви са статусите на a_i и a_j според противника.

- ♦ Да допуснем, че a_i и a_j имат различен статус.

Ако a_i е малко, а a_j е голямо, противникът връща Y . Сравнението $a_i \stackrel{?}{<} a_j$ е очевидно безполезно.

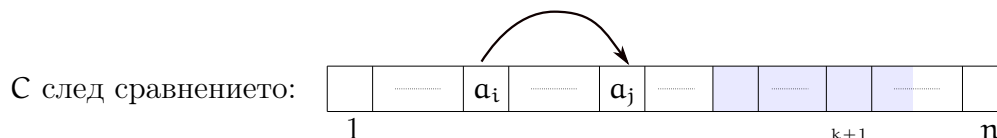


Ако a_i е голямо, а a_j е малко, противникът връща N . Сравнението $a_i \stackrel{?}{<} a_j$ е очевидно безполезно.



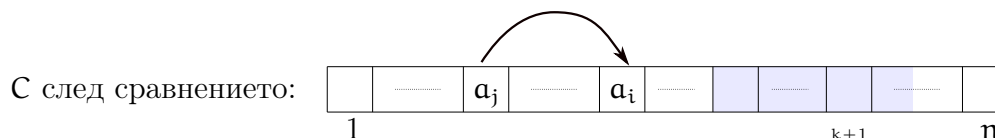
- ♦ Да допуснем, че a_i и a_j имат еднакъв статус. БОО, нека и двата са малки. Сега противникът казва истината съгласно взаимното им разположение в S .

Ако a_i е вляво от a_j , отговорът на противника е Y . Сравнението е полезно, понеже му отговаря полезно ребро в дага на сравненията.



Ако a_j е вляво от a_i , отговорът на противника е N . Сравнението е полезно, понеже му отговаря полезно ребро в дага на сравненията.

[†]Естествено, те не са били сравнявани помежду си; алгоритъмът не е глупав и не би сравнил същите елементи от входа повторно.



Медианата ще се окаже елементът, който най-дълго не е участвал в сравнения. Тъй като в края не може да има елементи, които не са участвали в сравнения, все в даден момент и този елемент ще участва—за първи път—в сравнение. В този момент всички клетки за малки и големи числа в C вече са заети и това число отива в $C[k + 1]$ и става медиана.

Забележете, че противникът не знае *a priori* коя е медианата. Той знае каква е стойността k , а именно $k+1$, но не знае кой елемент от входа ще се окаже това. Редът, в който алгоритъма задава въпросите, решава кой входен елемент ще се окаже медианата, а противникът не знае алгоритъма.

Ключовото наблюдение е, че противникът може да направи така, че поне $\lfloor \frac{n}{2} \rfloor$ от сравненията да се окажат безполезни, и алгоритъмът е безсилен да предотврати това. Алгоритъмът може да минимизира броя на безполезните сравнения, ако $\lfloor \frac{n}{2} \rfloor$ пъти сравнява числа, които досега не са били сравнявани. По този начин $\lfloor \frac{n}{2} \rfloor$ двойки клетки в C , един в малките и един в големите числа, дават по едно безполезно ребро.

Ако обаче алгоритъмът сравнява число, несравнявано досега, с число, което е било участник в сравнение, една позиция (голяма или малка, няма значение) в S дава едно безполезно ребро и общия брой на безполезните ребра става по-голям от $\left\lfloor \frac{n}{2} \right\rfloor$.

И така, противникът винаги може да “накара” алгоритъма да направи поне $\lfloor \frac{n}{2} \rfloor$ безполезни сравнения; тоест, в дага на сравненията да има поне $\lfloor \frac{n}{2} \rfloor$ безполезни ребра. принуждава алгоритъма да извърши още поне $n - 1$ сравнения,

От друга страна, за намирането на медианата са необходими поне $n - 1$ полезни сравнения, които дават поне $n - 1$ полезни ребра, които ребра покриват всички върхове и задават еднозначно медианата (Следствие 37).

Тези две независими съображения дават следната долна граница за броя на сравненията за намиране на медиана:

$$\left\lfloor \frac{n}{2} \right\rfloor + n - 1 = \left\lfloor \frac{n}{2} + n - 1 \right\rfloor = \left\lfloor \frac{3n - 2}{2} \right\rfloor$$

Което и трябваше да покажем.

13.4.8 Долна граница $2n - 1$ за сливане на сортирани масиви

Нека $A[1 \dots n]$ и $B[1 \dots n]$ са сортирани масиви. За удобство допускате, че никой от тях няма повтарящи се елементи и те нямат общи елементи. Ще докажем, че $2n - 1$ е долна граница за броя на сравненията при сливането (*merge*) на тези два масива в един сортиран масив $C[1 \dots 2n]$.

Да допуснем, че съществува алгоритъм ALGX , такъв че $\text{ALGX}(A[1..n], B[1..n])$ връща $C[1..2n]$, който се състои от елементите на A и B , но в сортиран вид, като ALGX ползва най-много $2n - 2$ сравнения за целта. Тъй като C има точно $2n$ елемента, съседните двойки елементи в C са точно $2n - 1$. Щом са ползвани най-много $2n - 2$ сравнения, съществува двойка съседни в C елементи, които не са били сравнени от алгоритъма.

Нека A и B са такива, че

$$A[1] < B[1] < A[2] < B[2] < \dots < A[n] < B[n]$$

Тогава в C се редуват елемент от A с елемент от B :

$$C = [A[1], B[1], A[2], B[2], \dots, A[n], B[n]]$$

Тук няма загуба на общност, понеже, ако долната граница е в сила за такива A и B , тя е в сила и изобщо.

Щом $C = [A[1], B[1], A[2], B[2], \dots, A[n], B[n]]$ и двойка съседни в C не са били сравнени, поне едното от следните е вярно.

- $A[i]$ не е бил сравнен с $B[i]$, за някое $i \in \{1, \dots, n\}$. Тогава противникът прави $A[i]$ по-голям от $B[i]$, но така, че $A[i] < A[i+1]$ да остане вярно. Спрямо тази промяна на входа, изходът би трябвало да бъде

$$[A[1], B[1], A[2], B[2], \dots, A[i-1], B[i-1], B[i], A[i], A[i+1], B[i+1], \dots, A[n], B[n]]$$

Но $ALGX$ продължава да връща $[A[1], B[1], A[2], B[2], \dots, A[n], B[n]]$, понеже резултатите от останалите сравнения са същите като тези преди промяната на входа. Този масив вече не е коректен изход. По този начин противникът опровергава $ALGX$, без да може да бъде уличен в лъжа.

- $A[i]$ не е бил сравнен с $B[i-1]$, за някое $i \in \{2, \dots, n\}$. Тогава противникът прави $B[i-1]$ по-голям от $A[i]$, но така, че $B[i-1] < B[i]$ да остане вярно. Спрямо тази промяна на входа, изходът би трябвало да бъде

$$[A[1], B[1], A[2], B[2], \dots, A[i-1], A[i], B[i-1], B[i], A[i+1], B[i+1], \dots, A[n], B[n]]$$

Но $ALGX$ продължава да връща $[A[1], B[1], A[2], B[2], \dots, A[n], B[n]]$, понеже резултатите от останалите сравнения са същите като тези преди промяната на входа. Този масив вече не е коректен изход. По този начин противникът опровергава $ALGX$, без да може да бъде уличен в лъжа.

Заклучаваме, че такъв $ALGX$ не съществува.

13.4.9 Долна граница $2n - 1$ за търсене в $n \times n$ таблица на Young

Въведохме таблици на Young в Допълнение 36 като правоъгълни таблици от числа и елементи ∞ , които таблици са сортирани по редове и колони. За целите на тази подсекция, таблиците на Young са квадратни и само от числа, като пак са сортирани по редове и колони. Двумерна $n \times n$ таблица M от числа е сортиран по редове и колони тстк:

$$\forall i \in \{1, \dots, n\} \forall j \in \{1, \dots, n-1\} : M[i, j] \leq M[i, j+1]$$

$$\forall j \in \{1, \dots, n\} \forall i \in \{1, \dots, n-1\} : M[i, j] \leq M[i+1, j]$$

Ето пример за таблица на Young при $n = 6$.

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	7	8	9	10	48	56
3	19	21	25	45	75	78
4	28	29	30	65	81	88
5	29	40	58	70	91	95
6	50	55	60	74	99	100

Дадена е $n \times n$ таблица на Young M и число key . Пита се дали key се среща в M . Също като при задачата ТЪРСЕНЕ на стр. 278, и тук задачата

- може да е задача за разпознаване,
- а може да се искат индексите на key , ако key се среща в M , или индикация, че key не се среща в M , в противен случай.

В какъв вариант е задачата, не е съществено.

Първо ще разгледаме една недобра идея за бърз алгоритъм, после една добра идея, и накрая ще намерим точна долна граница, в асимптотичния смисъл, с което ще покажем, че добрата идея е асимптотично оптимална.

Недобра идея за бърз алгоритъм. Очевидно можем да открием key , ако го има, или да сме сигурни, че го няма, ако проверим всички n^2 клетки на M . Това би било прекалено бавно. Трябва някак да се възползваме от факта, че M е сортирана по редове и колони.

Да опитаме така. Да разгледаме главния диагонал. По отношение на примерата таблица горе, това е главният диагонал:

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	7	8	9	10	48	56
3	19	21	25	45	75	78
4	28	29	30	65	81	88
5	29	40	58	70	91	95
6	50	55	60	74	99	100

Да търсим key с двоично търсене по главния диагонал, което означава да го сравняваме с елементи от вида $M[i, i]$, за $i \in \{1, \dots, n\}$. Ако при това го открием, връщаме координатите му. В противен случай, има някакъв индекс j , такъв че $M[j, j] < \text{key}$ и $M[j + 1, j + 1] > \text{key}$. По отношение на нашия пример, нека $\text{key} = 52$. Не откриваме key в главния диагонал, като въпросното j е 3:

	1	2	3	4	5	6
1	1 < 52	2	3	4	5	40
2	7	8 < 52	9	10	48	56
3	19	21	25 < 52	45	75	78
4	28	29	30	65 > 52	81	88
5	29	40	58	70	91 > 52	95
6	50	55	60	74	99	100 > 52

Точката, маркирана с червен кръст, задава четири подтаблицы. В две от тях, маркирани с червен фон, числото 52 не може да се срещне. В другите две, маркирани със зелен фон, числото 52 може да се срещне.

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	7	8	9	10	48	56
3	19	21	25	45	75	78
4	28	29	30	65	81	88
5	29	40	58	70	91	95
6	50	55	60	74	99	100

Продължаваме да търсим рекурсивно в двата зелени подтаблицы. Спирачката на рекурсията е главен диагонал с дължина единица.

Тази идея не е съвсем тривиална за реализиране. В примера, който току-що видяхме, подтаблиците, върху които се вика рекурсивно алгоритъма, са с размери 3×3 ; тоест, това са пак квадратни таблици, но всеки със страна $n/2$. Но може n да е нечетно, а може и точката, която дели главния диагонал, да не го дели наполовина, при което подтаблиците, в които търсенето продължава, са далече от квадратни. Примерно, нека числото, което търсим, е пак 52, но в тази таблица:

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	7	8	9	10	48	56
3	19	21	125	145	175	178
4	28	29	130	165	181	188
5	29	40	158	170	191	195
6	50	55	160	174	199	999

Сега зелените подтаблицы, където трябва да продължим с търсенето, не са квадратни. И за тях можем да въведем някакъв “обобщен главен диагонал”, върху който да търсим с двоично търсене, но това е безсмислено. Както ще видим, самата идея на този алгоритъм не е добра. Дори при перфектна дихотомия върху главния диагонал, както в примера, в който точката-разделител се оказва точно в средата му, рекурентното уравнение, което описва сложността по време в най-лошия случай, е

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n \quad (13.8)$$

Това е само ако на **всяко** ниво на рекурсията точката-делител на главния диагонал се пада по средата му. Събираемостта $\lg n$ отразява работата на двоичното търсене върху главния диагонал, а $2T\left(\frac{n}{2}\right)$ отразява работата на двете рекурсивни викания, всяко върху таблица със страна $n/2$. Съгласно първия случай на Мастър теоремата, решението е $T(n) \asymp n$. Както предстои да видим, има много по-прост и елегантен линеен алгоритъм за тази задача, който не по схемата **Разделяй-и-Владей**.

Добра идея за бърз алгоритъм. Да допуснем, че key не се среща в M , но ние не знаем това и тепърва трябва да го установим. Да сравним $M[n, 1]$ с key . Има две възможности.

1. Ако $M[n, 1] < key$, то със сигурност $M[n - 1, 1] < key, \dots, M[1, 1] < key$.
2. Ако $M[n, 1] > key$, то със сигурност $M[n, 2] > key, \dots, M[n, n] > key$.

В първата възможност има смисъл да сравним $M[n, 2]$ с key . Във втората възможност има смисъл да сравним $M[n - 1, 1]$ с key . Това дава следната идея. Да въведем текущ елемент $M[p, q]$ от таблицата, като поначало $p = n$ и $q = 1$.

- Ако $M[p, q] = key$, или връщаме индикация, че има елемент, равен на key , или връщаме $\langle p, q \rangle$.
- Ако $M[p, q] < key$, правим “ход надясно” $q++$, стига това да не излиза извън таблицата.
- Ако $M[p, q] > key$, правим “ход нагоре” $p--$, стига това да не излиза извън таблицата.

Ако има опит за излезане извън таблицата, тоест, $p < 1$ или $q > n$, извеждаме съобщение, че key не се среща в M . Псевдокодът е следният.

YOUNG TABLEAU SEARCH($M[1..n, 1..n]$: таблица на Young, key : число)

```

1   $p \leftarrow n, q \leftarrow 1$ 
2  while  $p \geq 1$  and  $q \leq n$  do
3      if  $M[p, q] = key$ 
4          return  $\langle p, q \rangle$ 
5      else if  $M[p, q] < key$ 
6           $q++$ 
7      else
8           $p--$ 
9  return "key не се среща в  $M$ "

```

Коректността се доказва тривиално със следния инвариант: при всяко достигане на ред 2, всички появи на key в $M[1..n, 1..n]$ са в $M[1..p, q..n]$.

Да разгледаме сложността по време. Тъй като "придвижването" става, по нещо като начупена линия, която започва от долу-ляво и върви само нагоре и надясно, дължината на тази начупена линия е горна граница за сложността по време, в асимптотичния смисъл. А дължината на начупената линия е най-много $2n - 1$. Следователно, сложността по време в най-лошия случай на този алгоритъм е $\Theta(n)$. Читателят лесно може да се убеди, че засуканият, разделяй-и-владей алгоритъм с двоичното търсене по главния диагонал, който разгледахме преди малко, не прави нищо друго освен да схожда към въпросната начупена линия. Очевидно е много по-смислено да се движим по нея последователно.

Ето илюстрация на работата на алгоритъма. Използваме масива, с който започнахме примера горе, а key отново е 52. Илюстрацията показва въпросната начупена линия с цветни стрелки върху масива и по този начин виждаме редицата от стойностите, които наредената двойка (p, q) е вземала по време на работата.

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	7	8	9	10	48	56
3	19	21	25	45	75	78
4	28	29	30	65	81	88
5	29	40	58	70	91	95
6	50	55	60	74	99	100

"Начупената линия" може и да не толкова начупена, и въпреки това да има дължина $2n - 1$, както се вижда от следната илюстрация:

	1	2	3	4	5	6
1	1	2	3	4	5	40
2	107	108	109	110	148	156
3	119	121	125	145	175	178
4	128	129	130	165	181	188
5	129	140	158	170	191	195
6	150	155	160	174	199	200

Възможно е key да не се среща и “начупената линия” да е с дължина само n :

	2	3	4	5	6	
1	101	102	103	104	105	140
2	107	108	109	110	148	156
3	119	121	125	145	175	178
4	128	129	130	165	181	188
5	129	140	158	170	191	195
6	150	155	160	174	199	200

Възможни са и подобрения, при които “начупената линия” не започва непременно с $M[n, 1]$, а нейното начало се намира с двоично търсене върху долната страна $M[n, 1], \dots, M[n, n]$ или върху дясната страна $M[1, 1], \dots, M[1, n]$. По отношение на най-лошия случай тези подобрения нямат значение.

Долна граница $\Omega(n)$ за задачата. Ще видим прост аргумент чрез противник, който дава желаната долна граница. Да видим следната илюстрация за $n = 6$. За краткост, пишем “ k ” вместо “key”. Допуснете, че ϵ е положително число, много по-малко от k , а ∞ е положително число, много по-голямо от k .

	1	2	3	4	5	6
1	ϵ	ϵ	ϵ	ϵ	ϵ	$k + \epsilon$
2	ϵ	ϵ	ϵ	ϵ	$k - \epsilon$	$k + \epsilon$
3	ϵ	ϵ	ϵ	$k - \epsilon$	$k + \epsilon$	∞
4	ϵ	ϵ	$k - \epsilon$	$k + \epsilon$	∞	∞
5	ϵ	$k - \epsilon$	$k + \epsilon$	∞	∞	∞
6	$k - \epsilon$	$k + \epsilon$	∞	∞	∞	∞

Очевидно k не се съдържа в M . Същественото обаче е, че всеки алгоритъм ALGA, който коректно заключава това, трябва да провери всяка от оцветените клетки (които съдържат $k - \epsilon$ или $k + \epsilon$). Ако ALGA пропусне дори една от тези клетки, противникът ще промени нейното съдържание на k , но ALGA ще продължава да “твърди”, че k не се среща, защото резултатите от всички достъпи до клетките на M ще е същият.

Като пример, основан на предния пример, да кажем, че алгоритъмът никога не проверява $M[4, 3]$. При това положение алгоритъмът няма как да различи предния масив от този:

	1	2	3	4	5	6
1	ϵ	ϵ	ϵ	ϵ	ϵ	$k + \epsilon$
2	ϵ	ϵ	ϵ	ϵ	$k - \epsilon$	$k + \epsilon$
3	ϵ	ϵ	ϵ	$k - \epsilon$	$k + \epsilon$	∞
4	ϵ	ϵ	k	$k + \epsilon$	∞	∞
5	ϵ	$k - \epsilon$	$k + \epsilon$	∞	∞	∞
6	$k - \epsilon$	$k + \epsilon$	∞	∞	∞	∞

Сега k се съдържа, но алгоритъмът няма как да извлече този факт при положение, че не прави достъп до $M[4, 3]$. Забележете, че M и сега е сортиран по редове и по колони.

И така, има вход за $n = 6$, за който се налага да бъдат проверени $2n - 1 = 11$ клетки, преди да се направи заключението, че k не се среща в масива. Очевидно този аргумент се мащабира за всяко n , с което наблюдение приключваме с доказателството.

Една заключителна забележка в духа на Конвенция 3. В тази задача символът “ n ” означава не големината на входа, а корен-квадратен от големината на входа. Входът всъщност е с големина n^2 (плюс още 1 заради key). Поради това, YOUNG TABLEAU SEARCH на стр. 820 не е линеен алгоритъм, а “корен-квадратен алгоритъм”; тоест, има сложност по време $\Theta(\sqrt{N})$,

където N е големината на входа. И долната граница, която доказахме, е $\Omega(\sqrt{N})$, изразена чрез големината на входа N .

13.4.10 Долна граница n^2 за установяване на свързаност на граф с $2n$ върха

Да си припомним СВЪРЗАНОСТ НА ГРАФИ (Задача 1). Ще докажем, че всеки алгоритъм за тази задача, който прави достъпи до графа само чрез задаване на въпроси от вида “Има ли ребро между връх i и връх j ?”, задава поне n^2 въпроса, ако броят на върховете е $2n$.

Има аргументация чрез противник. Противникът разбива произволно множеството от върховете на две подмножества U и W , такива че $|U| = |W| = n$, и след това действа така. Получавайки запитване за върхове i и j ,

- ако $i \in U$ и $j \in U$ или $i \in W$ и $j \in W$, противникът отговаря ДА,
- ако $i \in U$ и $j \in W$ или $i \in W$ и $j \in U$, противникът отговаря НЕ.

Съществуват точно n^2 двueleментни множества $\{x, y\}$, такива че $x \in U$ и $y \in W$ или $x \in W$ и $y \in U$. Ще покажем, че всеки алгоритъм ALGCONN за тази задача трябва да направи запитване за всяко от тези множества. Да допуснем обратното: ALGCONN е коректен алгоритъм за СВЪРЗАНОСТ и съществува $x \in U$ и съществува $y \in W$, такива че ALGCONN не е направил запитване за x и y .

- Ако ALGCONN отговори ДА, то противникът конструира несвързан граф, в който U и W са клики и няма нито едно ребро между връх от U и връх от W . По този начин противникът опровергава ALGCONN, тъй като всички отговори, които е дал, са консистентни с този граф, а графът не е свързан.
- Ако ALGCONN отговори НЕ, то противникът конструира свързан граф, в който U и W са клики и има ребро между x и y . По този начин противникът опровергава ALGCONN, тъй като всички отговори, които е дал, са консистентни с този граф, а графът е свързан.

13.4.11 Задачата за яйцето

13.4.12 Трудноуловими подстрингове и свойства

Терминът на английски е *evasive*, което на български се превежда като *уклончив* или *неуловим*. “Уклончив” не е адекватен термин на български за това, което имаме предвид в тази подсекция, а “неуловим” звучи прекалено силно. Заради това ще използваме *трудноуловим*. Изложението в Подподсекция 13.4.12.1 и Подподсекция 13.4.12.2 се базира на главата за аргументация с противник в книгата Algorithms на Jeff Erickson, само че тази глава не е част от свободно достъпното копие на книгата, посочено в библиографията, а може да бъде свалена свободно [оттук](#).

13.4.12.1 Трудноуловими подстрингове

Определение 115: Трудноуловим подстринг

Нека е дадена крайна непразна азбука Σ . Нека $\mathcal{A}$ е множеството от алгоритмите $\text{ALGX}(\mathbf{a}, \mathbf{b})$, които изчисляват дали $\mathbf{a}$ съдържа $\mathbf{b}$ като подстринг, където $\mathbf{a}, \mathbf{b} \in \Sigma^+$. Нека $X \subseteq \Sigma^+$ и $\sigma \in \Sigma^+$. Нека $L = \{\ell \in \mathbb{N}^+ \mid (\exists \mathbf{y} \in X : |\mathbf{y}| = \ell)\}$. Ще казваме, че σ е *трудноуловим по отношение на X* , ако е изпълнено следното:

- $\forall \text{ALGX} \in \mathcal{A} \quad \forall \ell \in L$
- $\exists \mathbf{x} \in X$, такова че $|\mathbf{x}| = \ell$ и $\text{ALGX}(\mathbf{x}, \sigma)$ прави достъп до всяка буква на $\mathbf{x}$.

На прост български, стринг σ е трудноуловим (evasive) по отношение на X , ако на всеки алгоритъм за намиране на подстринг, който търси σ в стринг от $\mathbf{x} \in X$, в най-лошия случай “се налага” да прочете целия $\mathbf{x}$. “Се налага” означава, че, ако целият $\mathbf{x}$ не бъде прочетен, не може да има сигурност дали σ е подстринг на $\mathbf{x}$. Ключовото тук е **в най-лошия случай**. Не се иска за всеки $\mathbf{x} \in X$ да бъде прочетен целия $\mathbf{x}$ (за да стане ясно дали σ е подстринг на $\mathbf{x}$), а да има поне един такъв $\mathbf{x}$ за всяка възможна дължина. Още по-просто казано, ако търсим σ в $\mathbf{x} \in X$ и играем срещу противник, противникът може да ни накара да прочетем целия $\mathbf{x}$.

Да фиксираме азбуката: нека оттук насетне $\Sigma = \{0, 1\}$. Напълно очевидно е, че 0 и 1 са трудноуловими по отношение на Σ^+ , но нека докажем това формално. БОО, ще го докажем само за 0. Доказателството е с противник. Да кажем, че е търсим 0 в някакъв $\mathbf{x} = x_1x_2 \cdots x_n$. Ето стратегията на противника: той отговаря 1 на всяко запитване на алгоритъма, като запитванията на алгоритъма са за стойности x_i , където $i \in \{1, \dots, n\}$. Ако алгоритъмът пропусне да провери стойността на поне едно x_j , противникът ще го опровергае винаги:

- ако алгоритъмът върне ДА, то противникът ще генерира $\mathbf{x} = 11 \cdots 1$, като това е консистентно с отговорите, които е давал на алгоритъма,
- ако алгоритъмът върне НЕ, то противникът ще генерира такъв $\mathbf{x}$, че $x_j = 0$, като това е консистентно с отговорите, които е давал на алгоритъма.

Jeff Erickson *пише*, че 0 и 1 са единствените трудноуловими стрингове по отношение на $\{0, 1\}^+$. Сега ще разгледаме трудноуловимост по отношение на някакви строги подмножества на $\{0, 1\}^+$. Първо да се убедим, че 01 не е трудноуловим по отношение на множеството от стрингове с нечетна дължина, която очевидно трябва да е поне три. Ще докажем, че, ако искаме да установим дали 01 е подстринг на стринг $\mathbf{x}$ с нечетна дължина, не налага четенето на всяка буква на $\mathbf{x}$. Нека $|\mathbf{x}| = n = 2k + 1$ за някое $k > 0$. Нашият алгоритъм първо чете буквите на четните позиции: $x_2, x_4, \dots, x_{2k}$. Следните случаи са изчерпателни и взаимно изключващи се.

- Ако всички тези букви са нули, то 01 се среща като подстринг на $\mathbf{x}$ тстк поне една буква от $x_3, x_5, \dots, x_{2k+1}$ е единица. Ерго, няма нужда да четем x_1 .
- Ако всички тези букви са единици, то 01 се среща като подстринг на $\mathbf{x}$ тстк поне една буква от $x_1, x_3, \dots, x_{2k-1}$ е нула. Ерго, няма нужда да четем x_{2k+1} .
- Ако редицата от тези букви започва с нули, последвани от поне една единица, то отговорът е ДА. Ерго, няма нужда да се четат други букви от $\mathbf{x}$.

- Ако редицата от тези букви се състои от единици, последвани от нули, нека най-дясната единица е x_t , което означава, че най-лявата нула е x_{t+2} . Тогава 01 се среща като подстринг на x тук 01 е подстринг на $x_1 \cdots x_t$ или $x_{t+2} \cdots x_n$, понеже стойността на x_{t+1} е без значение. Ерго, няма нужда да четем x_{t+1} .

Но 01 е трудноуловим в стринговете с четна дължина.

Теорема 114: 01 е трудноуловим по отношение на четните булеви стрингове

01 е трудноуловим по отношение на $\{x \in \{0, 1\}^+ : |x| \equiv 0 \pmod{2}\}$.

Доказателство: Нека търсим 01 в $x = x_1 x_2 \cdots x_n$, такъв че $n = 2k$. По време на работата на алгоритъма, противникът отговаря така, сякаш 01 не се среща. Забележете, че ако x не съдържа 01, то $x = 1^t 0^{n-t}$ за някое $t \in \{0, \dots, n\}$. Ерго, противникът отговаря по такъв начин, сякаш x е $11 \cdots 100 \cdots 0$, но, ако пропуснем да прочетем поне един бит на x , той (противникът) винаги може да ни опровергае. За тази цел той може да промени (не напълно прочетения) x , ако пожелае, по такъв начин, че 01 да се появи като подстринг.

Противникът поддържа два индекса ℓ и r , като подстрингът $X_L = x_1 \cdots x_\ell$ се състои само от единици, подстрингът $X_R = x_r \cdots x_n$ се състои само от нули, а подстрингът $X_M = x_{\ell+1} \cdots x_{r-1}$ е неясен до момента. В самото начало $\ell = 0$, $r = n + 1$, така че x съвпада с X_M . След всяко запитване за стойността на x_i от страна на алгоритъма, за какво да е $i \in \{1, \dots, n\}$, противникът отговаря ето така.

- Ако $i \in \{1, \dots, \ell\}$, то противникът дава стойност 1 на x_i , понеже x_i попада в X_L .
- Ако $i \in \{r, \dots, n\}$, то противникът дава стойност 0 на x_i , понеже x_i попада в X_R .
- В противен случай е в сила $i \in \{\ell + 1, \dots, r - 1\}$, тоест x_i попада в X_M . Точно един от $x_{\ell+1} \cdots x_i$ и $x_i \cdots x_{r-1}$ има четна дължина.
 - ♦ Ако $x_{\ell+1} \cdots x_i$ е с четна дължина, което е същото като $i - \ell$ да е четно, противникът прави $\ell \leftarrow i$, което означава, че X_L нараства с четен брой букви. Съответно X_M намалява със същия брой букви, а X_R не се променя.
 - ♦ Ако $x_i \cdots x_{r-1}$ е с четна дължина, което е същото като $r - i$ да е четно, противникът прави $r \leftarrow i$, което означава, че X_R нараства с четен брой букви. Съответно X_M намалява със същия брой букви, а X_L не се променя.

Твърдението “точно един от $x_{\ell+1} \cdots x_i$ и $x_i \cdots x_{r-1}$ има четна дължина” се доказва тривиално с инвариант, който гласи, че при всяко четене на бит на x от страна на алгоритъма, подстрингът X_M е с четна дължина. Това е вярно в самото начало, понеже тогава X_M съвпада с x , а след това X_M намалява с четен брой букви след всяко четене на бит от страна на алгоритъма. Щом $X_M = x_{\ell+1} \cdots x_{r-1}$ има четна дължина и $i \in \{\ell + 1, \dots, r - 1\}$, очевидно точно единият от $x_{\ell+1} \cdots x_i$ и $x_i \cdots x_{r-1}$ има четна дължина.

Да допуснем, че нашият алгоритъм, който проверява дали 01 се среща в x , пропусне да прочете поне един бит от x . Ако подстрингът X_M е празен, противникът ни опровергава веднага: ако сме отговорили ДА, той прави $x = 11 \cdots 100 \cdots 0$, примерно записвайки само нули в X_M , а ако сме отговорили НЕ, той слага 01 някъде в X_M ; забележете, че щом X_M е с четна дължина, по-голяма от нула, място за 01 има. А ако подстрингът X_M е празен, което е същото като $x = X_L X_R$, то,

- ако сме отговорили ДА, то противникът прави $x = 11 \cdots 100 \cdots 0$, попълвайки непрочетените битове по подходящ начин,

- а ако сме отговорили НЕ и непрочетен бит е от X_L , то този непрочетен бит не е най-десният бит на X_L и противникът го прави 0 и така се появява подстринг 01,
- а ако сме отговорили ДА и непрочетен бит е от X_R , то този непрочетен бит не е най-левият бит на X_L и противникът го прави 1 и така се появява подстринг 01. $\square$

13.4.12.2 Трудноуловими графови свойства

В Подсекция 13.4.10 доказахме долна граница n^2 за броя на запитванията от вида “Има ли ребро между връх i и връх j ?” върху граф с $2n$ върха, имащи за цел да установят дали графът е свързан. Ако кажем, че броят на върховете е n , където n е четно, тази долна граница става $\frac{n^2}{4}$. Сега ще докажем за същата задача по-висока долна граница, а именно $\binom{n}{2}$. Съществуването на долната граница $\binom{n}{2}$ ни дава основание да кажем, че свойството “свързаност” е трудноуловимо, защото в най-лошия случай се налага да проверим наличието на всяко възможно ребро. Ясна е аналогията с Определение 115, въпреки че тази аналогия е само донякъде.

Често срещано допускане, формулирано още в началото на доказателство за трудноуловимо графово свойство е, че графът е представен с матрица на съседство. В горепосоченият материал на Jeff Erickson е точно така:

Another class of problems for which adversary arguments give good lower bounds is graph problems where the graph is represented by an adjacency matrix, rather than an adjacency list.

Това допускане не е необходимо, ако запитванията от вида “Има ли ребро между връх i и връх j ?” са абстрактни, тоест, не ни интересува как са имплементирани и се интересуваме само от броя им. Ако ни интересува само броя на запитванията, няма значение как точно е представен графът. Допускането, че графът е представен с матрица на съседство има смисъл по следните две причини.

- Първо, матричното представяне ни дава възможност да получим автоматично оптимална долна граница за сложността на алгоритъма, реализиращ запитванията. Матричното представяне гарантира, че отговорът на всяко запитване за ребро се получава във време $\Theta(1)$, поради което, ако броят на запитванията е $f(n)$, то сложността на алгоритъма автоматично е $\Omega(f(n))$, и тази долна граница е оптимална, защото всяко запитване трябва да се случи. Ако графът беше представен със списъци, $\Omega(f(n))$ пак щеше да е долна граница, но не непременно оптимална, защото не е ясно как отговорите на запитванията биха се получавали в $\Theta(1)$, дори и само в амортизирания смисъл.
- Второ, матричното представяне ни дава право да мислим за графа като за булев стринг с дължина $\binom{n}{2}$. Защо $\binom{n}{2}$, а не n^2 ? – защото графът се определя напълно от горната триъгълна форма на матрицата. Главният диагонал се състои само от нули и не е информативен и може да бъде игнориран, а долната триъгълна форма е огледално копие на горната триъгълна форма и също може да бъде игнорирана. Ако си представим горната триъгълна форма като линеаризирана по редове, да кажем отгоре надолу, тя става едномерен булев обект, тоест булев стринг.

И така, можем да мислим за графа като за булев стринг, при което търсенето на дадено свойство става търсене на свойство на булев стринг и Определение 115 започва да става релевантно. Тук обаче нещата са доста по-сложни, защото в общия случай търсим не просто някакъв подстринг—което е лесно—а някакво свойство. В контекста на тази подподсекция, свойството е това: графът, който се престава от стринга, да е свързан.

Поради тези съображения ще разглеждаме матрично представяне на графите.

Определение 116: Трудноуловимо графово свойство

Нека $\mathcal{G}$ е безкрайното множество от графите. Нека $P(x)$ е едноместен предикат с домейн $\mathcal{G}$. С други думи, $P(x)$ е свойство на графите. Казваме, че P е *трудноуловимо графово свойство*, ако за всеки алгоритъм $ALGP$, пресмятащ P , за всяко $n \in \mathbb{N}^+$, съществува $G \in \mathcal{G}$, такъв че $|V(G)| = n$, за който е вярно, че $ALGP(G)$ прави достъп до всеки бит на $\sigma(G)$, където $\sigma(G)$ е линеаризация на горната триъгълна форма на $M(G)$, където $M(G)$ е матрицата на съседство на G .

Графово свойство, за което е очевидно, че е трудноуловимо, е *празнота*, свойството на графа да няма ребра. Аргументацията с противник е елементарна: празнотата на графа G е същото като $\sigma(G)$ да е $00 \cdots 0$. Противникът отговаря “0” на всяко запитване за стойността на бит от $\sigma(G)$, а ако запитващият алгоритъм пропусне дори един бит, противникът ще опровергае *post factum* този алгоритъм при всеки от двата възможни отговора ДА и НЕ. Далече не толкова очевидна е трудноуловимостта на свързаността.

В тази подподсекция ще докажем, че свързаността е трудноуловима. Формалното доказателство е Теорема 115. Ще започнем с интуитивно и неформално изложение. Разглеждаме произволен алгоритъм, проверяващ дали граф G е свързан, правейки запитвания към противник за битовите на $\sigma(G)$, където $\sigma(G)$ има смисълът от Определение 116. Стратегията на противника е следната. Поначало $\sigma(G) = 11 \cdots 1$. Естествено, противникът пази този стринг скрит по време на запитванията и го показва чак след като алгоритъмът, правещ запитванията, е приключил работата си. Ще казваме, че по време на запитванията, $\sigma(G)$ е записан в *тефтера* на противника и само той (противникът) го вижда.

След всяко запитване за стойността на бит b от $\sigma(G)$, нека $\tilde{\sigma}(G)$ е стрингът, който се получава от $\sigma(G)$ след записване на 0 в b .

- Ако графът, който се представя от $\tilde{\sigma}(G)$, е свързан, противникът отговаря “0” и заменя $\sigma(G)$ със $\tilde{\sigma}(G)$.
- В противен случай, противникът отговаря “1” и не променя $\sigma(G)$.

Както предстои да се убедим, тази проста стратегия принуждава питащия алгоритъм да прочете всеки бит на $\sigma(G)$, за да може да даде отговор, който не може да бъде опроверган от противника.

Най-общо казано, графът, към който се стреми противникът, е дърво, чието множество от върхове е $V(G)$. Или, ако предпочитате, G е пълният граф през цялото време (понеже в началото $\sigma(G) = 11 \cdots 1$), а противникът се стреми да изгради негово покриващо дърво, представено чрез текущия $\sigma(G)$. Ако алгоритъмът направи запитване за всеки бит, графът ще се окаже свързан, но ацикличен, тоест, дърво, и крайният $\sigma(G)$ ще представлява това дърво – той ще има точно $n - 1$ единици, които са разположени по такъв начин, че $\sigma(G)$ представлява дърво.

Важно е да се разбере, че противникът поначало няма представа как ще изглежда това дърво, тоест, кои ребра ще са в него. Това зависи от реда, в който ще се правят запитванията, а това е не е известно на противника в началото.

Неформална обосновка с пример. Да разгледаме какво прави противника в един частен случай при $n = 6$. Приемаме, че върховете са $1, \dots, 6$. За по-голяма яснота ще мислим за стринга $\sigma(G)$ не като едномерен стринг от 15 бита, а като за горна триъгълна форма. Ето

как противникът си представя графа преди запитванията. Битовите в клетките на $\sigma(G)$, за които все още не е правено запитване, са в сиво. Преди запитванията всички са такива.

Нека—тук има загуба на общност!—по някаква причина в началото алгоритъмът се фиксира върху връх 3 и прави запитвания само за неговите съседства. За удобство на читателя съответните клетки на $\sigma(G)$ са нарисувани с **жълт фон**.

	1	2	3	4	5	6
1		1	1	1	1	1
2			1	1	1	1
3				1	1	1
4					1	1
5						1
6						

За който и съсед на връх 3 да е първото запитване, противникът ще отговори 0. Да кажем, че първото запитване е за стойността на клетка (2,3). Противникът отговаря “0” и съответно записва 0 в тефтера си. До края на примера, в **червен цвят** са битовите, които за които противникът вече е дал отговор.

	1	2	3	4	5	6
1		1	1	1	1	1
2			0	1	1	1
3				1	1	1
4					1	1
5						1
6						

Да кажем, че следващите запитвания са за (1,3), (3,5) и (3,6), без значение в какъв ред. Противникът отговаря “0” и на трите запитвания. Представянето на графа в тефтера му вече изглежда така.

	1	2	3	4	5	6
1		1	0	1	1	1
2			0	1	1	1
3				1	0	0
4					1	1
5						1
6						

Но на запитването за $(3, 4)$, противникът отговаря “1”:

	1	2	3	4	5	6
1		1	0	1	1	1
2			0	1	1	1
3				1	0	0
4					1	1
5						1
6						

Би било **сериозна грешка** да отговори “0” и на последното запитване за съсед на връх 3.

	1	2	3	4	5	6
1		1	0	1	1	1
2			0	1	1	1
3				0	0	0
4					1	1
5						1
6						

// противникът би се прецакал,
// ако отговори така.

Ако противникът отговори “0” за всички потенциални съседни връхове 3, алгоритъмът може да заключи, че връх 3 е изолиран връх и да не задава повече въпроси, защото граф с поне

два върха, който има изолиран връх, не е свързан. За да поддържа “интригата”, противникът отговаря по такъв начин за $(3, 4)$, че да накара алгоритъма да продължи с въпросите. И така, върхове 3 и 4 се оказват съседни според противника и алгоритъмът продължава да пита.

Запитванията за съседствата на връх 3 може да са в друг ред, при което друг връх да се окаже съседът на връх 3. Примерно, ако последното запитване беше за връх 6, представянето на графа в тефтера на противника щеше да е следното.

	1	2	3	4	5	6
1		1	0	1	1	1
2			0	1	1	1
3				0	0	1
4					1	1
5						1
6						

Накратко, ако алгоритъмът се фиксира в началото само върху съседствата на един и същи връх, противникът ще отговори “0” на първите $n - 2$ запитвания и “1” на $(n - 1)$ -вото запитване. Това е следствие на стратегията, която той (противникът) следва.

Верификация на стратегията на противника. Става дума за стратегията, описана на стр. 827. За всеки случай, ето стратегията отново. Контекстът е този: по отношение на фиксирано множество върхове V , да кажем $V = \{1, \dots, n\}$, противникът конструира в тефтера си описание на граф G , като самото начало G е пълният граф, и отговаря на запитвания за него. Това ще рече, запитвания за ребрата, понеже върховете са едни и същи през цялото време. Питането на алгоритъма е дали има ребро между x и y , които са различни върхове на G . Близко е до ума, че досега не е правено питане за x и y – би било безумие алгоритъмът да пита няколко пъти за една и съща двойка върхове. Засега приемаме на доверие, че при всяко запитване, G е свързан. Надолу ще докажем това формално.

СТРАТЕГИЯ НА ПРОТИВНИКА(x, y : върхове в G , за които не е правено питане)

- 1 ако $G - (x, y)$ остава свързан, изтрий (x, y) от G и отговори НЕ
- 2 в противен случай, не променяй G и отговори ДА

Следното твърдение е инвариант за СТРАТЕГИЯ НА ПРОТИВНИКА. Тъй като СТРАТЕГИЯ НА ПРОТИВНИКА няма цикъл (loop) в себе си, може да звучи странно, че наричаме това съждение “инвариант”. Цикълът, на който това е инвариант, е в питащия алгоритъм. По отношение на неговия цикъл на задаване на въпроси на противника има смисъл да кажем, че това е инвариант, защото е винаги вярно съждение.

Нека във всеки момент, Y е множеството от ребрата, за които противникът вече е отговорил ДА, а M е множеството от ребрата, за които все още не са задавани въпроси. Забележете, че графът G в тефтера на противника е $(V, Y \cup M)$.

Инвариант 29: Инвариант за СТРАТЕГИЯ НА ПРОТИВНИКА

При всяко достигане на ред 1,

- ❶ графът $G = (V, Y \cup M)$ е свързан и
- ❷ за всеки цикъл c в G , нито едно ребро на c не е в Y .

Доказателство: Доказателството е по индукция по реда на достиганията на ред 1, но не в едно изпълнение на СТРАТЕГИЯ НА ПРОТИВНИКА, а във всички изпълнения на СТРАТЕГИЯ НА ПРОТИВНИКА. Базата е първото достигане. В този момент $Y = \emptyset$, а $M = \{(i, j) \mid 1 \leq i < j \leq n\}$. Тогава G е пълният граф на n върха, който е свързан, така че ❶ е вярно. Тъй като Y е празното множество в този момент, ❷ е вярно в празния смисъл. ✓

Да допуснем, че Инвариант 29 е в сила при някое достигане, което не е последното. Аргументацията надолу неявно ползва факта, че Y се мени монотонно: ако ребро “влезе” в Y , то вече не може да “излезе” от Y и остава в Y докрая, понеже противникът се ангажира с отговорите си. Алгоритъмът пита за някакви x и y и противникът се готви да отговори съгласно своята стратегия. Следните два случая са изчерпателни и взаимно изключващи се.

- $G - (x, y)$ е свързан. Тогава противникът отговаря НЕ и изтрива (x, y) от G . Забележете, че изтриването на (x, y) всъщност е премахването му от M , защото в този случай (x, y) не е бил в Y . Новият G очевидно е свързан, така че ❶ остава в сила.

Да разгледаме произволен цикъл c в G след отговора на противника. В този случай Y не се променя, така че нито едно ребро на c не е в Y заради индуктивното допускане.

- $G - (x, y)$ е несвързан. Тогава противникът отговаря ДА и не “пипа” G . Забележете, че отговорът ДА за (x, y) има смисъл на прехвърлянето на (x, y) от множеството M в множеството Y . Щом G се запазва, то той си остава свързан, така че ❶ остава в сила.

Да разгледаме произволен цикъл c в G след отговора на противника. Да допуснем, че поне едно ребро $e \in E(c)$ е в Y . Но това e може да е само (x, y) , понеже всяко друго ребро на c е извън Y заради индуктивното допускане. Но този извод противоречи на факта, че $G - (x, y)$ е несвързан: ако изтриването на (x, y) от G прави G несвързан, то (x, y) не е ребро от цикъл в G . Полученото противоречие показва, че допускането е невярно, така че и ❷ остава в сила. □

Да си припомним, че “гора” означава ацикличен граф, откъдето свързаните ѝ компоненти са дървета, откъдето гора е несвързана тстк тя има повече от едно дърво.

Лема 55: Още един инвариант за СТРАТЕГИЯ НА ПРОТИВНИКА

При всяко достигане на ред 1, графът $F = (V, Y)$ е гора. Нещо повече, $M \neq \emptyset$ тстк тази гора е несвързана.

Доказателство: Очевидно $F \subseteq G$. Това, че F е гора, е тривиално следствие от част ❷ на Инвариант 29.

Ще докажем, че ако $M \neq \emptyset$, то F има поне две дървета. Очевидно F има само едно дърво тстк самата F е дърво. Да допуснем, че F има само едно дърво, но M е непразно. Тогава съществува $e \in M$. Тогава e , добавено към дървото F , образува цикъл c в текущия $G = (V, M \cup Y)$. Но този цикъл c има поне още две ребра освен e , които са елементи на Y . Но това противоречи на част ❷ на Инвариант 29.

Ще докажем, че ако $M = \emptyset$, то F има точно едно дърво. Но това следва веднага от част ① на Инвариант 29: щом $(V, Y \cup M)$ е свързан през цялото време и $M = \emptyset$, очевидно (V, Y) е свързан. С други думи, F е свързана гора, което по дефиниция е дърво. $\square$

Теорема 115: Коректността на противника при трудноуловимостта на свързаността

СТРАТЕГИЯ НА ПРОТИВНИКА “кара” питащият алгоритъм да направи запитване за всяко ребро в смисъл, че ако не направи запитване за поне едно ребро, противникът може да го опровергае винаги.

Доказателство: Доказателството, което предстои да направим, отговаря на фазата **Терминация** в доказателствата с инвариант, които сме разглеждали в предни лекции. Тук има една особеност обаче – не знаем колко запитвания ще направи алгоритъма. Тоест, не знаем кога потокът от запитвания ще терминира. Тъй като сме заели позицията на противника и гледаме от неговата гледна точка, за нас питащият алгоритъм е енигма. В един екстремен случай, питащият алгоритъм може да не направи нито едно запитване и въпреки това да отговори, че графът е свързан (или обратното). Звучи налудничаво, но не е невъзможно. Другата екстремна възможност е питащият алгоритъм да попита за всяко ребро. И нито един междинен вариант не е невъзможен. Противникът трябва да е готов за всичко.

Ако алгоритъмът запита за всяко ребро, то противникът ще даде отговори, от които следва, че графът е дърво. Това следва веднага от Лема 55, но не касае Теорема 115. В Теорема 115 се интересуваме какво става, ако питащият алгоритъм пропусне да запита за поне едно ребро.

Ако питащият алгоритъм пропусне да запита за поне едно ребро, съгласно Лема 55, графът $F = (V, Y)$ е несвързан. Но очевидно F е подграф на $G = (V, M \cup Y)$, а G е свързан съгласно Инвариант 29. При това положение противникът може да опровергае алгоритъма винаги. Ако алгоритъмът отговори, че графът е свързан, то противникът ще покаже несвързания F , който е консистентен с всички отговори на запитвания. Ако алгоритъмът отговори, че графът е несвързан, то противникът ще покаже свързания G , който е консистентен с всички отговори на запитвания. $\square$

Любопитна странична забележка в [учебното помагало на Jeff Erickson](#) е, че СТРАТЕГИЯ НА ПРОТИВНИКА на стр. 830 всъщност е алгоритъма ANTI-KRUSKAL от Допълнение 50, но за построяване на **максимално** покриващо дърво. Алгоритъмът разглежда ребрата в нарастващ ред[†] по тегла, като теглото на ребро е времето, в което се прави питане за него, и, ако изтриването на реброто води до несвързаност, то става част от максималното покриващо дърво (“влиза” в Y), в противен случай то се “изхвърля” (“маха се” от M).

Приключихме с доказателството на факта, че свойството свързаност е трудноуловимо. Много по-задълбочено, но все пак на ниво лекция, изложение на трудноуловимостта има в статията [110]. Най-интересното, изложено в нея, е една хипотеза, която, доколкото е известно на автора, до ден днешен нито е доказана, нито е опровергана. Това е хипотезата на Aanderaa–Kar–Rosenberg (AKR), гласяща, че всяко нетривиално монотонно свойство върху графи е трудноуловимо. “Нетривиално свойство” е такова свойство, че поне един граф го притежава и поне един граф не го притежава. Свързаността и Хамилтоновостта са нетривиални, понеже съществуват свързани графи и Хамилтонови графи, но съществуват и несвързани и не-Хамилтонови графи. Пример за тривиално графово свойство е това: сумата от степените на върховете е два пъти броя на ребрата – няма граф, за който това не е изпълнено. Монотонно свойство, за целите на това изложение, е свойство, което се запазва при добавяне на

[†]“Истинският” ANTI-KRUSKAL разглежда ребрата в намаляващ ред по тегла.

ребра. Примерно, не-планарността е монотонно свойство, защото граф, който не е планарен, няма как да стане планарен с добавяне на ребра. Хамилтоновостта също е монотонно свойство, защото, ако граф има Хамилтонов цикъл, този цикъл няма как да изчезне, ако добавим ребро. Ойлеровостта обаче не е монотонна: ако добавим едно ребро (x, y) към Ойлеров граф, то степените на x и y стават нечетни и графът престава да е Ойлеров.

AKR не е нито доказана, нито опровергана в продължение на десетилетия, което е свидетелство за нейната огромна трудност. AKR е доказана за частния случай, в който n е просто число, но това доказателство е изключително тежко и неочевидно. За допълнителна информация за него, вижте [110].

Част VI

Алгоритмична неподатливост

Лекция 14

NP-пълнота и NP-трудност.

Резюме: Въвеждаме понятието изчислителна неподатливост. Разглеждаме изчислителните задачи за разпознаване като формални езици, откъдето решаването на такава задача става разпознаване на език. Правим рекапитулация на машините на Turing и разглеждаме пример за работата на машина на Turing. Въвеждаме класа на сложност **P**. Въвеждаме недетерминирани машини на Turing по два еквивалентни начина и класа на сложност **NP**. Въвеждаме полиномиални (Karp) редукции и ги сравняваме с Turing редукции. Въвеждаме **NP**-пълнота и **NP**-трудност.

14.1 Въведение с пример и илюстрация

Професор Дълбоков е химик. Екипът на професор Дълбоков е направил двеста експеримента. Оказва се, че резултатите от някои експерименти са в рязко противоречие с резултатите от други експерименти. В реалността най-вероятно би имало степени на противоречие: някои двойки експерименти си противоречат много, други си противоречат не толкова много, а трети си противоречат малко. За целите на това изложение допусваме, че картината е черно-бяла: два експеримента или си противоречат напълно, или не си противоречат изобщо.

Дали екипът е на прага на велико откритие, или някои експерименти са направени небрежно? Преди всичко, професор Дълбоков иска да разбере кои експерименти са причинители на противоречията. Това означава експерименти, които, ако бъдат игнорирани, сред останалите няма противоречия.

Естествено, ако се игнорират всички експерименти, или всички без един, противоречия няма да има. Но това не е решение. Исква се да бъдат игнорирани колкото е възможно **по-малко** експерименти, така че противоречията да изчезнат.

Задачата се моделира с граф $G = (V, E)$, и то неориентиран, защото релацията на противоречие е симетрична. Върховете са експериментите, а ребрата са противоречията. Читателят може би е забелязал, че в езика на графите, това, което се търси, е минимално върхово покриване. Тази задача дефинирахме в Подсекция 12.8.1. Да си припомним дефиницията на задачата като оптимизационна задача.

Изч. Задача: MINIMUM VERTEX COVER

екземпляр: Неориентиран граф $G = (V, E)$.

решение: Подмножество $U \subset V$, такова че $\forall (u, v) \in E : (u \in U \text{ или } v \in U)$ и $|U|$ е минимално.

Наистина, за всяко върхово покриване U на G е вярно, че $G - U$ няма ребра. Тъй като ребрата са противоречията, ясно е, че ако игнорираме експериментите, съответни на кое е да е

върхово покриване, премахваме и противоречията. Ако върховото покриване е и минимално, то това е оптимално решение на задачата, която стои пред професора.

Как да намерим минимално върхово покриване в произволен граф? В Подсекция 12.8.1 видяхме как се намира минимално върхово покриване на дърво, но това не ни върши работа сега; никой не е казал, че графът на противоречията в експериментите е дърво.

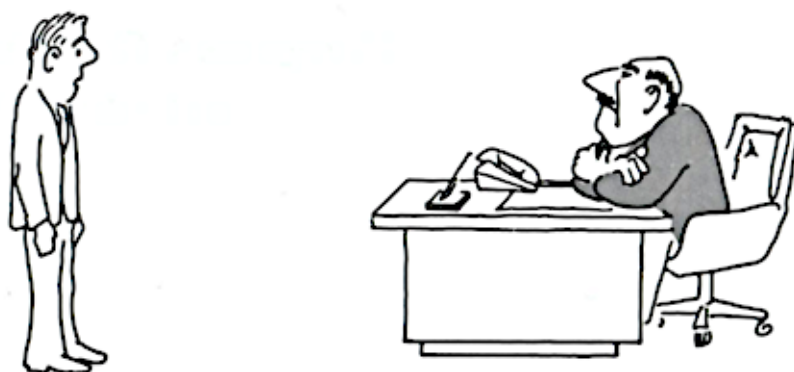
Колкото и да е странно, за тази просто звучаща, напълно естествена и очевидно полезна на практика задача, най-ефикасният известен алгоритъм е експоненциален. Нещо повече: тази задача е твърде близка до хиляди други задачи, за всяка от които не е известен по-бърз от експоненциален алгоритъм. Нещо повече:

- ако за поне една от тези задачи се докаже някаква суперполиномиална долна граница, от това ще следва, че нито една от тях не е решима в полиномиално време,
- но ако за поне една от тях се намери полиномиален алгоритъм, от това ще следва, че всяка от тях е решима в полиномиално време.

Усилията, хвърлени от компютърните учени в продължения на поне пет десетилетия в тази посока, са колосални. Постигнати са известни успехи, има значително по-добро разбиране за трудностите и мъчнотиите, но основната цел—или доказване на суперполиномиална долна граница, или намиране на ефикасен алгоритъм—остават все така недостижими.

Един от най-важните учебници за NP-пълнота и други видове алгоритмична неподатливост е култовият “Computers and Intractability: A Guide to the Theory of NP-Completeness” на Michael Garey и David Johnson [56]. Въпреки че е издаден през далечната 1979 г., той е интересен и актуален и днес. Да видим три илюстрации от този учебник (страници 2 и 3), които са с култов статус в средите на хората, занимаващи се с изчислителна сложност. На алгоритмичен дизайнер е възложено да разработи ефикасен алгоритъм за някаква изчислителна задача. Задачата обаче се оказва свръх силите на алгоритмичния дизайнер и той не успява да разработи желания алгоритъм. Оттук насетне са възможни три сценария.

1. Той просто се предава и отива да съобщи неприятната новина на шефа си. Не е необходимо да се уточнява кой е дизайнерът и кой е шефът – това е очевидно.



“I can't find an efficient algorithm, I guess I'm just too dumb.”

2. След като не успява да разработи алгоритъм, той успява да докаже, че за тази задача няма ефикасен алгоритъм. За разлика от първия сценарий, тук дизайнерът е в приподигнат дух заради постижението си. Очевидно никой няма да го накаже за това, че не е разработил алгоритъма, дори напротив, той е отбелязал точки, доказвайки, че няма алгоритъм с желаните качества.



"I can't find an efficient algorithm, because no such algorithm is possible!"

Този сценарий обаче няма нищо общо с реалността на NP-пълнотата. Никой не е доказал, че за задачи като VERTEX COVER и компания няма ефикасни алгоритми.

3. След като не успява да разработи алгоритъм, той успява да докаже, че задачата, за която търси ефикасен алгоритъм, в някакъв смисъл е много близка до огромен брой други задачи, за които изтъкнати специалисти в течение на много време не са успели да намерят ефикасни алгоритми. Естествено, това не е **доказателство**, че такива алгоритми няма! Това, че някакви хора са се мъчили да постигнат нещо и не са успели не доказва, че нещото е непостижимо. Но това е силна **улика**, че нещото е или непостижимо, или поне извънредно трудно; ерго, няма нищо срамно в неуспеха на "нашия човек" от илюстрацията. Езикът на тялото му в случая е нещо средно между първите две илюстрации – нито е умърлушен като на първата, нито е във висок боен дух като на втората.



"I can't find an efficient algorithm, but neither can all these famous people."

До ден днешен, третият сценарий от илюстрациите е реалността на NP-пълнотата. Най-доброто, което можем да направим в общия случай, поне на първо време, е да оправдаем алгоритмичния си неуспех с аналогични неуспехи на изтъкнати алгоритмици.

Тук трябва да се каже, че има доста начини да се "заобиколи" NP-пълнотата, ако конкретиката позволява, така че това не е последната дума по въпроса! На "нашия

човек” от илюстрацията предстои да опита да разработи апроксимиращ алгоритъм или параметризиран алгоритъм или нещо друго, но това е вторично и тези разработки биха били в сянката на основния резултат – задачата, която му е дадена, си остава нерешена.

14.2 Алгоритмична неподатливост

Основната тема на Лекция 14, Лекция 15, Лекция 16 и Лекция 17 е *алгоритмичната неподатливост*, или само *неподатливост*. Това е директен превод на английския термин *intractability*. Понятието е приложимо към задачи, а не към алгоритми. Понятието може да се ползва и неформално, но ние ще го формализираме. За целта ни трябва “ефикасен алгоритъм”.

Конвенция 15: Ефикасен алгоритъм

Казваме, че алгоритъм X е ефикасен, ако сложността по време на X е $O(n^k)$, за някаква положителна константа k .

Определение 117: Неподатлива задача

Казваме, че изчислителна задача P е *неподатлива*, ако за нея не съществува ефикасен алгоритъм.

Според Определение 117, задача, за която единственият известен алгоритъм е със сложност $\Theta(n^{100})$, е податлива, тъй като алгоритъмът е полиномиален и поради това е ефикасен. Това е безумно от практическа гледна точка, но да си припомним думите на Papadimitriou, цитирани на стр. 108:

Any attempt, in any field of mathematics, to capture an intuitive, real-life notion ... by a mathematical concept ... is bound to include certain undesirable specimens, while excluding others that arguably should be embraced.

И така, Конвенция 15 и Определение 117 остават в сила.

Ние вече видяхме една неподатлива задача: HALTING PROBLEM (Задача 8). Щом за нея няма алгоритъм, в частност за нея няма ефикасен алгоритъм. Може да обобщим, че алгоритмичната неизчислимост (*uncomputability*) е частен случай на алгоритмичната неподатливост (*intractability*). В тези лекции обаче не се интересуваме от неизчислими задачи; ще оставим неизчислимостта на логиците. Тук се интересуваме от неподатливостта на изчислими задачи, каквито са например VERTEX COVER и HAMILTONIAN CYCLE. За нас неподатливостта се корени в ограничеността на ресурсите, с които разполагат изчисленията в реалния свят, а не с принципната невъзможност за извършване на изчислението, без оглед на ресурсите, което е предмет на теорията на изчислимостта.

Според Garey и Johnson [56, стр. 11], има два вида неподатливост. Първата е тази на, например, VERTEX COVER и HAMILTONIAN CYCLE. Втората е неподатливостта на задачи от вида “Генерирай всички подмножества на множество” или “Генерирай всички пермутации на $\{1, \dots, n\}$ по дадено n ” или “Генерирай всички покриващи дървета на даден граф”. Втората неподатливост е коренно различна от първата и ние с нея няма да се занимаваме.

- При първия вид неподатливост, с която **ще се** занимаваме в тези лекции, решението е малко в размера на (кодирането на) екземпляра, като “малко” означава “не повече от полиномиално”. Примерно, при HAMILTONIAN CYCLE решението е с размер единица, защото задачата е задача за разпознаване. При този вид неподатливост е голямо времето за изчисляване на решението, въпреки че решението е малко.

Забележете, че при задачи като VERTEX COVER и HAMILTONIAN CYCLE има пълна неизвестност поначало. Какъв е размерът на минималното върхово покриване? Дали има Хамилтонов цикъл? Поначало ние не знаем, ако графът е произволен.

- При втория вид неподатливост, с която **няма да се** занимаваме в тези лекции, решението е голямо в размера на (кодирането на) екземпляра, като “голямо” означава “повече от полиномиално”. При този вид неподатливост е практически невъзможно да се генерира решението, понеже това генериране на (кодирането на) решението в явен вид иска ресурси (време и памет), с каквито не разполагаме.

Забележете, че при задачи като “Генерирай всички подмножества на дадено множество” или “Генерирай всички пермутации на $\{1, \dots, n\}$ по дадено n ”, неизвестност поначало няма. Знае се колко са подмножествата и освен се знае, абстрактно говорейки, кои са подмножествата. Знае се колко са пермутациите и се знае, поне на концептуално ниво, кои са пермутации. Ако искаме, можем да дефинираме наредба (нарича се *ранкиране*, на английски *ranking*, за подробности вижте [135]) на подмножествата и пермутациите и много бързо да изчислим кои точно елементи се намират в i -тото подмножество, за $1 \leq i \leq 2^n$, или кое е j -тото число в i -тата пермутация, за $1 \leq i \leq n!$ и $1 \leq j \leq n$. Ерго, дори да не разполагаме с решението като конкретика (степенното множество или множеството от пермутациите), нямаме никаква концептуална неяснота за него.

Предвид всичко това, възприемаме следната конвенция.

Конвенция 16

Разглеждаме само неподатливи задачи, чиито решения са с размер, не по-голям от полиномиален в размера на входа.

Допълнение 75: Накратко за историята на неподатливостта

Първите открития на някакви аналози на това, което днес наричаме неподатливост, са много отпреди времето на компютрите и Компютърната наука. Още през 1931 г. Kurt Gödel [58] показва—за ужас на голяма част от математическата общност—че има твърдения за естествените числа, които са недоказуеми **и** неопровергаеми във формални системи. Този основополагащ резултат слага кръст на десетилетни опити на математици като David Hilbert и Bertrand Russell да формализират и механизират (днес бихме казали, алгоритмизират) **цялата** математика.

Резултатът на Gödel скоро бива последван от други доказателства за невъзможност, измежду които е доказателството за алгоритмичната нерешимост на HALTING PROBLEM от Alan Turing през 1936 г. [151].

Забележете връзката между доказуемост и изчислимост! Първоначалната идея за механизиране (днес бихме казали, алгоритмизиране) на мисленето в математиката е много по-стара от Hilbert и Russell и може да се проследи [37] поне до Gottfried Leibniz, математик и философ от XVII век. Неговото желание е било да конструира системи за автоматично доказване на твърдения и генериране на верни твърдения, при дадени

аксиоми и правила за извод. Днешните изчислителни машини се ползват и за тази цел, но сравнително малко. Очевидно (и за съжаление) много повече изчислителна мощ се ползва в социалните мрежи, отколкото за алгоритмични доказателства.

Технологичният прогрес позволява след средата на XX век да се строят истински компютри. Това е огромна крачка отвъд абстрактните изчислителни модели, която променя света завинаги. Опитвайки се да конструират бързи алгоритми за оптимизационни задачи, компютърните учени се сблъскват с друг вид неподатливост, свързана с ресурсите (време и памет). Докато учените от епохата на абстрактното изчисление мислят по-скоро за съществуване, или несъществуване, на алгоритмични решения по принцип, през 50-те и 60-те години на XX век мнозина учени и инженери търсят алгоритми, които работят задоволително бързо върху истински компютри. Такива изследвания се правят и от двете страни на Желязната завеса, като западните учени, от една страна, и съветските учени, от друга страна, често не са наясно за постиженията на своите партньори (опоненти?) от другата страна на завесата. Добра обзорна статия за състоянието на съветската алгоритмика от тази епоха е статията на Трахтенброт [150].

В Западния свят, Hartmanis и Stearns [67] разглеждат неподатливост по отношение на ресурси и откриват йерархии на неподатливост, но изчислителните задачи са изкуствени и създадени само заради тези изследвания. Те показват, че за всяка функция $T(n)$ (колкото и бързо растяща да е) съществува изчислителна задача (задача за разпознаване на езици, по-точно), чието решение изисква поне $T(n)$ стъпки за всички достатъчно големи n . Те измислят и налагат термина “изчислителна сложност”, в оригинал “Computational Complexity”, и дефинират клас на сложност $\mathbf{DTIME}(T(n))$ спрямо дадена функция на сложност по време $T(n)$. Нещо повече, те откриват и доказват така наречената speed-up theorem, казваща, горе-долу, че ако дадена функция $f(n)$ е изчислима във време $T(n)$ върху машина на Turing M , то за всяка константа $k > 1$, функцията $f(n)$ е изчислима във време $T(n)/k$ от някаква машина на Turing M' . Този резултат се интерпретира така: мултипликативните константи нямат особено значение за сложността.

Класът на сложност $\mathbf{P}$ е дефиниран от Cobham [31], който предполага, че задачите, решими в полиномиално време, са точно задачите, които са ефикасно решими; накратко, “полиномиално време” и “ефикасност” са синоними. Това допускане е в сила и до ден днешен. Сравнете с Конвенция 15.

Edmonds [41] разглежда редукции между задачи, които са неподатливи, но естествени и полезни на практика: VERTEX COVER и INDEPENDENT SET към SET COVER.

Meyer и Stockmeyer [114] през 1972 г. доказват неподатливостта на една естествена изчислителна задача, свързана с регулярни изрази. “Регулярен израз с повдигане на квадрат” означава регулярен израз, който освен нормалните операции ползва и стрингово повдигане на квадрат $r^2 = rr$. Тази операция не прави регулярните изрази по-изразителни, но дава възможност да са експоненциално по-къси от регулярните изрази, които не я ползват.

Определение 118

Нека Σ е крайна азбука. Тогава

$$\mathbf{RSQ}(\Sigma) = \{r \mid r \text{ е регулярен израз с повдигане на квадрат над } \Sigma, \text{ такъв че } L(r) \neq \Sigma^*\}$$

Теорема 116: Експоненциална долна граница за $RSQ(\Sigma)$ (Theorem 2.1, [114])

Съществува крайна азбука Σ , такава че, ако M е произволна машина на Turing, разпознаваща $RSQ(\Sigma)$, то съществува константа $c > 1$, такава че M изисква памет (и оттам време) c^n върху поне един вход с размер n , за безкрайно много n .

Както ще стане ясно от изложението нататък, Теорема 116 не касае по никакъв начин NP-пълнотата. Теоремата казва, че задачата $RSQ(\Sigma)$ е пълна за класа на сложност **EXPSpace**, като $NP \subset \text{EXPSpace}$; **EXPSpace** е строго надмножество на **NP** и неговите пълни задачи са много по-трудни от пълните задачи на **NP**.

В началото на 70-те години има огромен прогрес в разбирането за неподатливостта. Днешната теория на NP-пълнотата започва през 1971 г. със статията “The Complexity of Theorem-Proving Procedures” на Stephen Cook [33]. Той акцентира ключовото значение на полиномиалните редукции, които “навръзват” привидно изключително различни изчислителни задачи в едно цяло. Понятието “редукция” е било известно в логиката много по-рано, но истинското значение на редукциите за този вид неподатливост е разкрито от Cook. Освен това, Cook показва, че недетерминизмът е ключов за разбирането на този вид неподатливост. Прочее, **NP** идва от **Nondeterministically Polynomial Time**. Освен това, Cook показва, че една конкретна, практически полезна и напълно естествена задача, а именно SAT, е пълна за задачите от множеството **NP**. И понятието за пълнота не е измислено от Cook, но той доказва, че SAT е NP-пълна задача. SAT е изключително изразителна задача, бивайки задача от логиката—в крайна сметка, логиката е направена, за да е изразителна—и се оказва подходящ “ключ” за “отключването” (показването на условна неподатливост) на огромен брой други задачи.

Стъпвайки на резултата на Cook, Richard Karp [89] показва през 1972 г. условната неподатливост на 21 други изчислителни задачи. Следва експлозия от изследвания и резултати в теорията на NP-пълнотата и разбирането за този вид неподатливост нараства неимоверно. Това разбиране е твърде непълно, предвид липсата на отговор на въпроса $P \stackrel{?}{=} NP$, но е несравнимо по-богато и задълбочено, отколкото в началото.

През 1971 г.—по същото време, по което Cook публикува семиналната си статия—в Москва, съветският учени Леонид Левин достига до същите изводи. Изследванията на Cook и Левин не са повлияни едни от други; това е напълно сигурно и е потвърдено от много участници в онези събития. По време на Студената война контактите между СССР и Западният свят са твърде ограничени. По думите на Трахтенброт [150], Левин докладва резултати още през 1971 г. на семинари в Москва и Ленинград. Те биват оценени, но съвсем не предизвикват сензацията, която предизвиква статията на Карп в САЩ през 1972 г. Чак през 1973 г. Левин публикува своята твърде лаконична статия [107], в която показва, че шест задачи, измежду която е и SAT, са универсални за множеството задачи, което днес наричаме **NP**. По мое мнение, статията на Левин [107] е езотерична: предназначена за изключително компетентни специалисти, които освен това познават добре контекст, който е необходим, за да се разбере статията, но не е изложен в явен вид в нея.

Във всеки случай, постижението на Левин е световно признато и често Теорема 126 се нарича “теорема на Cook–Левин”, а не просто “теорема на Cook”.

14.3 Задачите като формални езици

14.3.1 Всяка задача за разпознаване в някакъв смисъл е формален език

Да си припомним, че задача за разпознаване има еднобитово решение, което наричаме “отговор” (Определение 3). Ако задачата за разпознаване е P , множеството от екземплярите Π се разбива на множеството от ДА-екземплярите P_Y и множеството от НЕ-екземплярите P_N .

Не всяка задача е задача за разпознаване, а от особен интерес за нас са оптимизационните задачи (Определение 4). Да си припомним обаче Наблюдение 2: на всяка оптимизационна задача съответства задача за разпознаване, като наблюдението показва и как превръщаме оптимизационни задачи в съответните задачи за разпознаване. Ако функцията-цена w в оптимизационната задача се изчислява ефикасно—което винаги е вярно за задачите, които разглеждаме—то очевидно задачата във версия за разпознаване няма как да е алгоритмично по-трудна от оптимизационната версия: ако имаме бързо решение за оптимизационната версия, то автоматично ни дава решение за версията за разпознаване.

Последното е важно, понеже класическата теория на NP-пълнотата е за задачи за разпознаване. Но ние лесно може да обобщим нейните изводи за неподатливост и до съответните оптимизационни задачи, понеже те не може да са по-лесни.

Както отбелязахме в Наблюдение 5, на ниско ниво истинските компютри работят със стрингове, а не с числа. Функцията кодиране ни дава възможност да изобразяваме стринговете-кодирания в числа, и обратно. За машините на Turing е същото. Всяка машина на Turing—след малко ще видим формалната дефиниция—използва за “външна памет” стринг, записан на лента; ако казваме, че машината работи с числа или с графи, това е абстракция, при която някакви подстрингове на лентата са кодиранията на числата, или на елементите на графа.

В лекциите досега избягвахме да споменаваме кодирания изобщо. Приехме изчислителен модел (Допълнение 4), в който алгоритмите работят директно с числа или идентификатори на върхове на граф или каквито елементарни типове данни използваме, като по този начин се освободихме от необходимостта да мислим за кодирания на променливите. Сега обаче се налага да мислим за кодирания, поне концептуално, защото се готвим да използваме машини на Turing, а за тях въпросът с кодирането е ключов: ако дадена машина M решава дадена задача за графи и сменим кодирането на графа, ще се наложи да сменим и машината. В общия случай M не решава задачата при новото кодиране.

И така, ще разглеждаме двойки от задача за разпознаване $\Pi \subseteq \mathcal{I} \times \mathcal{S}$ (припомнете си Определение 1) и нейно кодиране $\mathcal{E} : \mathcal{I} \rightarrow \Sigma^*$, където

- “ $\mathcal{E}$ ” идва от “encoding”,
- $\mathcal{E}$ е инекция,
- $\mathcal{I}$ е множеството от екземплярите на Π ,
- Σ е крайната абзука на кодирането, като най-често $\Sigma = \{0, 1\}$.

$\mathcal{E}$ не е задължително биекция, защото може да не е сюрекция; тоест, може някои стрингове от Σ^* да не са валидни кодирания за екземпляри от $\mathcal{I}$. Но трябва да е инекция, за да бъде обратима! Да не е инекция е същото като пони един стринг от Σ^* да кодира два или повече екземпляра. Това е недопустимо.

Наблюдение 97

В текущия контекст, Σ^* се разбива на следните три множества:

$$\{x \in \Sigma^* \mid x \text{ кодира ДА-екземпляр на } P \text{ в } \mathcal{E}\}$$

$$\{x \in \Sigma^* \mid x \text{ кодира НЕ-екземпляр на } P \text{ в } \mathcal{E}\}$$

$$\{x \in \Sigma^* \mid x \text{ не е валидно кодиране на екземпляр на } P \text{ в } \mathcal{E}\}$$

Първото от тези множества е от особен интерес. В някакъв смисъл, ще идентифицираме P с него.

Определение 119: Език, съответен на задача за разпознаване

Нека P е задача за разпознаване и $\mathcal{E}$ е кодиране за нея с азбука Σ . Тогава *езикът, съответен на P и $\mathcal{E}$* , е

$$L(P, \mathcal{E}) = \{x \in \Sigma^* \mid x \text{ кодира ДА-екземпляр на } P \text{ в } \mathcal{E}\}$$

При някои задачи за разпознаване кодирането практически отсъства, в смисъл, че задачата е естествено задача за стрингове над Σ и всеки екземпляр съвпада с кодирането си. Такава задача е, примерно, ПАЛИНДРОМИ. Задачата е по отношение на някаква азбука Σ , която се подразбира. “Палиндром” означава стринг, който съвпада с реверсираното си копие (който се чете по един и същи начин отляво надясно и отдясно наляво).

Изч. Задача 60: Палиндром

екземпляр: Стринг $x \in \Sigma^*$.

въпрос: Дали x е палиндром?

В тази задача няма стрингове, които не кодират екземпляри. Ако разсъждаваме само за нея, няма смисъл да се притесняваме за кодирания. Всеки стринг от Σ^* е или ДА-екземпляр, или НЕ-екземпляр за нея.

В по-сложни задачи обаче, да кажем задачи върху графи, кодирането е от ключово значение, за да опишем точно големината на даден екземпляр. Тук става дума за истинската големина—дължината на стринга, който го кодира—която е от значение, ако изчислението се прави от машина на Turing. В тези по-сложни задачи неизбежно има стрингове, които са невалидни кодирания, така че би било грешка да кажем, че Σ^* се разбива на кодиранията на ДА-екземплярите и НЕ-екземплярите.

В резюме, задача за разпознаване плюс конкретно кодиране на нейните екземпляри в някакъв смисъл е задача за разпознаване на формален език. Колкото и неестествено да звучи, НАЙ-КЪС ПЪТ В ГРАФИ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ (Задача 5) плюс подходящо кодиране е задача за разпознаване на език! Стринговете, които кодират ДА-екземплярите на задачата, образуват езика. Това не може да бъде илюстрирано обаче, защото не сме казали какво е кодирането. Без него ние само знаем, че принципно ДА-екземплярите образуват език, но не можем да кажем за конкретен стринг дали кодира ДА-екземпляр, или не (не-то означава или кодира НЕ-екземпляр, или е невалидно кодиране).

Има добра причина да избягваме да дефинираме прецизно кодирания. Кодиранията са тежки и досадни. Помислете как бихте кодирали наредена петорка от граф (което всъщност означава **представяне** на граф, да кажем със списъци), тегловна функция, ID на връх, ID

на връх и число; такава петорка е екземпляр на задачата НАЙ-КЪС ПЪТ В ГРАФИ, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ. Кодирането трябва да е най-малкото инективно, тоест от стринга да е напълно ясно за кой граф, тегловна функция и така нататък, става дума; освен това трябва да е ефикасно изчислимо.

Понеже кодиранията са тежки и досадни, ние няма да ги правим експлицитно. Но ние ще знаем, че щом става дума за задача за разпознаване и машина на Turing за нея, **подходящо кодиране има**. Просто не ни се занимава да го дефинираме експлицитно и прецизно. Но какво е “подходящо кодиране”?

14.3.2 Подходящи кодираня на задачи за разпознаване

Дефиниция на “подходящо кодиране”, още може да кажем “разумно кодиране”, няма. Лесно обаче може да се даде екземпляр за неразумно кодиране. Ако задачата включва числа, неразумно би било числата да се кодират унарно. “Унарно” означава да се ползва само една буква, за да се опише магнитуда на число; примерно, само единици. При това положение няма избор, освен да сложим толкова букви от този вид, колкото е магнитудът на числото. Ако магнитудът е две, записваме “11”. Ако магнитудът е дванадесет, записваме “111111111111”. По понятни причини ще прескочим примера с магнитуд един милион. Както отбелязахме в предишни лекции, унарното кодиране раздува изкуствено големината на входа, и то експоненциално, спрямо по-икономични кодираня като бинарното. Чрез унарно кодиране можем да превърнем изкуствено експоненциален алгоритъм в полиномиален алгоритъм, понеже големината на входа при унарно кодиране е експоненциално по-голяма, а ние мерим сложността като функция на размера на входа.

Според Garey и Johnson [56], разумните кодираня удовлетворяват две изисквания (без това да е дефиниция на “разумно кодиране”).

- Първо, всеки две разумни кодираня са *полиномиално свързани*, на английски *polynomially related*. Ако Π е задача за разпознаване и $\mathcal{E}_1$ и $\mathcal{E}_2$ са нейни кодираня, няма значение дали с една и съща азбука или не, то $\mathcal{E}_1$ и $\mathcal{E}_2$ са полиномиално свързани, ако съществуват полиноми p_1 и p_2 , такива че за всеки екземпляр x на Π , ако σ_1 е кодирането на x чрез $\mathcal{E}_1$, а σ_2 е кодирането на x чрез $\mathcal{E}_2$, то $|\sigma_1| \leq p_1(|\sigma_2|)$ и $|\sigma_2| \leq p_2(|\sigma_1|)$. На прост български, разумните кодираня дават стрингове, които не са повече от полиномиално по-дълги един от друг.
- Второ, всяко разумно кодиране е *ефикасно декодируемо*, в оригинал *decodable*. Ако Π е задача за разпознаване и $\mathcal{E}$ е нейно кодиране, иска се за всеки стринг σ , който $\mathcal{E}$ дава, да може в полиномиално време да се реконструира обектът, който σ кодира. Това предотвратява кодираня, които изобразяват (някои) екземпляри в къси стрингове, от които обаче е изчислително (практически) невъзможно да се изработят описания на съответните екземпляри, с които да може да се работи.

Ето пример за кодиране на естествените числа, което не е ефикасно декодируемо. Примерът ползва машини на Turing (Определение 120). Нека $P(m, i)$ е предикатът

«съществува спираща машина на Turing M с точно m състояния, такава че $M(\epsilon) = 1^i$ »

Правим следните дефиниции.

$$\forall m \in \mathbb{N}^+ : BB(m) = \{i \in \mathbb{N}^+ \mid P(m, i)\}$$

$$BB_{\max} = \{\max BB(m) \mid m \in \mathbb{N}^+\}$$

Да разгледаме следното кодиране $\mathcal{E} : \mathbb{N} \rightarrow \{0, 1\}^*$. Да си припомним дефиницията на “ $\text{bin}(n)$ ” на стр. 316; $\text{bin}(n)$ е кодирането на естествено число n в двоична позиционна бройна система. Тогава, за всяко $n \in \mathbb{N}$,

- ♦ ако $n \notin \text{BB}_{\max}$, то $\mathcal{E}(n) = 1\text{bin}(n)$,
- ♦ ако $n \in \text{BB}_{\max}$, то $\mathcal{E}(n) = 0\sigma$, където $\text{bin}^{-1}(\sigma)$ е уникалното число m , такова че $P(m, n)$.

Декодирането на стринг от вида 0σ е дължината n на максималния стринг от единици, който може да бъде генериран от спираща машина на Turing с m състояния, където m се записва в двоична позиционна бройна система със σ . Но това е известната алгоритмично нерешима задача **BUSY BEAVER**.

Въпреки липсата на истинска дефиниция, всеки човек с познания и умения в областта има представа какво е “разумно кодиране”. Това ни позволява в изложението надолу за всяка задача, която разглеждаме, да не указваме точното кодиране, а да само да споменаваме, че има някакво резумно кодиране $\mathcal{E}$, и да ползваме името “ $\mathcal{E}$ ” без притеснения. Или дори да не споменаваме кодиране. Понякога говорим просто за езика $L(\Pi)$, където Π е задача за разпознаване.

14.4 Детерминирани машини на Turing

14.4.1 Фундамент

“Машина на Turing” е изключително важно понятие. Това е първият изчислителен модел, който е създаден въз основа на съображения от физическия свят. Идеята е да се моделира човек, който решава алгоритмично някаква задача в дискретни стъпки (дискретно време). Бидейки краен и ограничен, този човек във всеки момент е в точно едно състояние измежду краен брой състояния. Той обаче има достъп до неограничено много памет, която е линеаризирана (това, че в живота ползваме правоъгълни страници, за да си водим записки, е несъществено; двумерната страница може да се линеаризира ред по ред). Във всеки момент човекът разглежда точно една позиция в линейната памет. Някакво крайно управление (алгоритъм...) му казва какво да прави при всяка възможна комбинация от състояние и текуща буква: в какво състояние да мине, каква буква да сложи на нейно място, и коя от двете съседни позиции да стане текущата.

Колкото и да е странно, този привидно беден откъм възможности изчислителен модел се оказва универсален. Според [тезиса на Church](#), всичко, което е алгоритмично изчислимо, е изчислимо чрез машина на Turing. Това не е теорема, защото понятието “алгоритъм” е без дефиниция, а е хипотеза, която е универсално приета като истина. В реалния живот никой не строи машини на Turing, за да смята с тях. Тромавото последователно движение по лентата е несравнимо по-неефикасно от произволния достъп до клетките на паметта, с който сме свикнали при реалните компютри. Но именно ограничените възможности на машините на Turing ги прави полезно **абстрактно** средство за доказване на неподатливост.

Има много дефиниции на “машина на Turing”. Всички те са еквивалентни, разбира се. Оригиналната дефиниция е на самия Turing [151] от 1936 г., но широко приетите днес дефиниции се несъществено различни. Тук ще ползваме дефиницията от [56].

Определение 120: Детерминирана машина на Turing (MT)

Детерминирана машина на Turing, накратко машина на Turing или MT, е наредена седморка

$$(\Sigma, \Gamma, Q, q_0, q_y, q_n, \delta)$$

където

- Σ е крайна непразна азбука, наречена *входната азбука*,
- $\Gamma \supset \Sigma$ е крайна непразна азбука, наречена *лентовата азбука*, като буквата “ $\sqcup$ ”, тоест шпация, е в $\Gamma \setminus \Sigma$,
- Q е крайно множество от *състояния*,
- $q_0 \in Q$ е *началното състояние*,
- $q_y \in Q$, $q_y \neq q_0$, е *приемащото състояние*,
- $q_n \in Q$, $q_n \notin \{q_0, q_y\}$, е *отхвърлящото състояние*,
- $\delta : (Q \setminus \{q_y, q_n\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow, \downarrow\}$ е *функцията на преходите*.

Машината разполага с безкрайна в двете посоки лента, разбита на клетки, а във всяка клетка се съдържа точно една буква от Γ . За наше удобство, можем да мислим, че клетките са номерирани с целите числа $\dots, -1, 0, 1, \dots$, но машината не “вижда” тези номера и те са само за наше удобство.

Да въведем малко нотации. Обикновено бележим стрингове с “ $\mathbf{x}$ ” или “ σ ”. Ако $\mathbf{x}$ е непразен стринг, то x_1 е първата му буква и изобщо x_i е i -тата му буква. За празния стринг ползваме буквата ‘ ϵ ’. В началния момент, входът $\mathbf{x} \in \Sigma^*$ е записан така: x_1 в клетка 1, x_2 в клетка 2, $\dots$, x_n в клетка n , като $n = |\mathbf{x}|$. Всички останали клетки съдържат шпации $\sqcup$. Забележете, че $\sqcup \notin \Sigma$, така че входът не може да съдържа шпации. Входът обаче може да е празният стринг. Машината е снабдена с четящо-пишеща глава, която във всеки момент чете точно една от клетките. Тази клетка е *текущата клетка*. Съдържанието на текущата клетка се нарича *текущата буква*.

Във всеки момент от дискретното време, в което “живее” машината, тя е в точно едно състояние и точно една от клетките е текущата, така че и точно една от буквите е текущата. В началото машината е в състояние q_0 , а текущата клетка е клетка 1. В началото текущата буква е или x_1 , в случай, че входът $\mathbf{x}$ е непразен, или $\sqcup$, в противен случай. Във всеки следващ момент, ако текущото състояние не е q_y или q_n :

- машината минава в ново състояние (в частност може да остане в същото състояние),
- презаписва буквата от текущата клетка с нова буква (в частност може да запише същата буква),
- и
 - ♦ или премества главата наляво, което бележим с “ $\leftarrow$ ”,
 - ♦ или премества главата надясно, което бележим с “ $\rightarrow$ ”,
 - ♦ или остава главата на място, което бележим с “ $\downarrow$ ”.

съгласно функцията δ . По-точно, ако текущото състояние е $q \neq q_Y, q_N$ и текущата клетка съдържа буква a и $\delta(q, a) = (q', a', d)$, то машината минава в състояние q' , записва a' в текущата клетка и, ако $d = \leftarrow$, то главата се премества наляво, ако $d = \rightarrow$, то главата се премества надясно, а ако $d = \downarrow$, то главата остава на място.

Функцията δ е тотална, така че за всяка двойка от състояние (без q_Y или q_N) и буква машината “знае какво да прави”. Някои автори дефинират δ като частична функция и тогава машината спира, ако се стигне до двойка от текущо състояние и текуща буква, за които δ е недефинирана. За нас обаче δ е тотална и машината спира тстк се случи точно едно от следните две:

- текущото състояние стане q_Y , в който случай казваме, че машината *приема* входа x ,
- текущото състояние стане q_N , в който случай казваме, че машината *отхвърля* входа x .

Понякога е удобно да се мисли, че машината не спира, а “залепва” в състоянието q_Y или q_N и престава да еволюира съгласно δ . Разликата между тези две виждания е, че при първото виждане времето на машината спира при влизане в q_Y или q_N , а при второто виждане времето ѝ не спира никога, но тя престава да се променя.

Определение 121: Конфигурация на машина на Turing

Нека M е машина на Turing с вход x . Във всеки момент от работата на M , *конфигурацията на M* е наредената тройка от текущото състояние q , текущото съдържание на лентата y и текущата позиция на главата i , като $q \in Q$, $y \in \Sigma^*$ и $i \in \mathbb{Z}$. Накратко записваме конфигурацията така: $\langle q, y, i \rangle$.

Приемаща конфигурация е всяка конфигурация, в която състоянието е q_Y . *Отхвърляща конфигурация* е всяка конфигурация, в която състоянието е q_N .

Забележете, че конфигурацията не се състои просто от текущите състояние и буква. Текущите състояние и буква определят напълно какво ще се промени в машината в този момент, но не и как ще еволюира машината нататък във времето.

Тук може да се възрази, че лентата е безкрайна, така че текущото съдържание на лентата е безкрайно. Работата е там, че разглеждаме само обединението на

- клетките от входа
- и клетките, върху които машината е била; тоест, клетките, които са били текущи от началния момент досега.

За нас това е “съдържанието на лентата”. Всички останали клетки игнорираме. Някои автори дефинират, че машината не може да презаписва шпации $\sqcup$. Ако приемем тази конвенция, то текущото съдържание на лентата се състои от точно тези клетки, които не съдържат $\sqcup$. Очевидно текущото съдържание на лентата е непрекъснатата подредица от клетки, понеже машината не може да прескача клетки от лентата; ако текущата клетка е номер j , машината трябва да е “посетила” всички клетки между j и началната клетка 1.

Ясно е защо такава машина е детерминирана: при даден вход x , работата на машината е строго определена от δ . Последователността от конфигурации, през които ще мине машината, зависи от, и само от, входа и от функцията на преходите.

Въпреки че Turing използва безкрайна лента за своята машина в оригиналната статия [151], можем спокойно да минем без понятията безкрайна лента и шпации (съдържанието на клетките извън “съдържанието на лентата”). Вместо това, можем да дефинираме само *текущи*

string, който в началото е точно входът x . Този текущ string може обаче да нараства неограничено! При всеки опит на машината да “излезе” от него чрез $\leftarrow$ вляво от началото му или $\rightarrow$ вдясно от края му, към него (текущия string) се присъединява, вляво или вдясно, буквата ‘_’ като нова буква, която машината прочита. Ако възприемем това виждане, то съдържанието на лентата е точно текущият string и толкова.

Нека машината е наречена M . От общи съображения е ясно, че за даден вход x

- или M спира в q_Y , което бележим с “ $M(x) = q_Y$ ”,
- или M спира в q_N , което бележим с “ $M(x) = q_N$ ”,
- или M не спира никога, което бележим с “ $M(x) = \nearrow$ ”. Тук има две възможности.
 - ♦ Машината зацикля, което означава, че повтаря конфигурации. Очевидно е, че ако машината достигне повторно до конфигурация, в която вече е била, тя ще достигне тази конфигурация пак и пак и пак, до безкрай; щом машината е детерминирана, това е неизбежно.
 - ♦ Машината не повтаря конфигурация, но не спира. Най-простият начин да постигнем това е да ползваме само едно състояние, а именно началното q_0 . От него с която и да е буква оставаме в него, местейки главата надясно и записвайки каквото и да е. В такъв случай q_Y и q_N , дори да присъстват формално, биха били недостижими.

14.4.2 Съответствие между разпознаване на езици и решаване на задачи за разпознаване от машини на Turing

Определение 122: Езикът, разпознаван от машина на Turing

Нека M е МТ с входна азбука Σ . Езикът, разпознаван от M , е

$$\{x \in \Sigma^* \mid M(x) = q_Y\}$$

Този език означаваме с L_M .

Забележете, че не се иска машината да спира върху всички входове; тоест, да достига състояние q_N за всеки $x \in \Sigma^* \setminus L_M$. Ако x не е в езика, M може да го отхвърли чрез q_N , а може и да не спре никога.

Ако обаче искаме M да съответства на алгоритъм, M трябва да спира върху всеки вход. Определение 123 е направено предвид това съображение.

Определение 123: Решаване на задача за разпознаване от машина на Turing

Казваме, че МТ M решава задачата за разпознаване Π по отношение на кодирането $\mathcal{E}$, ако M спира върху всички stringове от входната азбука и $L_M = L(\Pi, \mathcal{E})$. Ако кодирането се подразбира, пишем просто $L_M = L(\Pi)$.

14.4.3 Пример за машина на Turing

Има смисъл да разгледаме пример за машина на Turing. Ще разгледаме машината на Turing M_{pal} , чийто език се състои точно от палиндромите над $\{0, 1\}$. Тази машина решава задачата

ПАЛИНДРОМИ, защото спира за всеки вход, така че тя приема палиндромите чрез q_Y и отхвърля не-палиндромите чрез q_N .

В някакъв смисъл, машината е идентична с функцията на преходите δ . Ако знаем δ и Σ , знаем всичко за машината. Γ и Q могат да бъдат изведени от δ . Функцията на преходите δ записваме в табличен вид: за всяка комбинация от състояние и буква дефинираме ново състояние, нова буква и движение на главата. Това е достатъчно, за да дефинираме M_{pal} . Таблица 14.1 съдържа функцията на преходите.

	0	1	$\sqcup$
q_0	$\langle r_0, \sqcup, \rightarrow \rangle$	$\langle r_1, \sqcup, \rightarrow \rangle$	$\langle q_Y, \sqcup, ! \rangle$
r_0	$\langle w_0, 0, \rightarrow \rangle$	$\langle w_0, 1, \rightarrow \rangle$	$\langle q_Y, \sqcup, ! \rangle$
w_0	$\langle w_0, 0, \rightarrow \rangle$	$\langle w_0, 1, \rightarrow \rangle$	$\langle s_0, \sqcup, \leftarrow \rangle$
s_0	$\langle t, \sqcup, \leftarrow \rangle$	$\langle q_N, \sqcup, ! \rangle$	$\langle s_0, \sqcup, ! \rangle$
t	$\langle t, 0, \leftarrow \rangle$	$\langle t, 1, \leftarrow \rangle$	$\langle q_0, \sqcup, \rightarrow \rangle$
r_1	$\langle w_1, 0, \rightarrow \rangle$	$\langle w_1, 1, \rightarrow \rangle$	$\langle q_Y, \sqcup, \leftarrow \rangle$
w_1	$\langle w_1, 0, \rightarrow \rangle$	$\langle w_1, 1, \rightarrow \rangle$	$\langle s_1, \sqcup, \leftarrow \rangle$
s_1	$\langle q_N, \sqcup, ! \rangle$	$\langle t, \sqcup, \leftarrow \rangle$	$\langle s_1, \sqcup, ! \rangle$

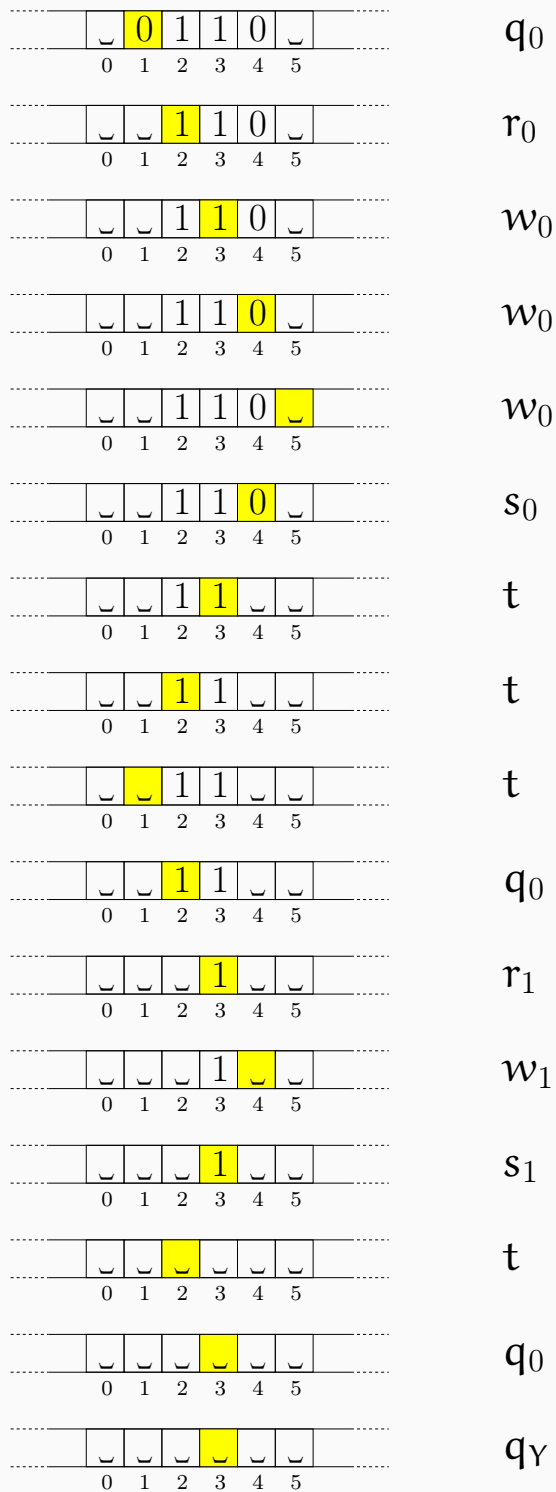
Таблица 14.1: Машина на Turing M_{pal} , която решава задачата ПАЛИНДРОМИ.

Ето как се чете таблицата. Започваме с първия ред, отговарящ на началното състояние q_0 . Ако машината е в q_0 и текущата буква е 0, минаваме в състояние r_0 , пишем $\sqcup$ и преместваме главата надясно. Ако машината е в q_0 и текущата буква е 1, минаваме в състояние r_1 , пишем $\sqcup$ и преместваме главата надясно. Ако машината е в q_0 и текущата буква е $\sqcup$, минаваме в състояние q_Y , с което приемаме стринга (празният стринг е палиндром по дефиниция). И така нататък. Ясно е как се чете цялата таблица.

В зелено са наредените тройки, в които състоянието е q_Y . В тях няма значение каква нова буква пишем и накъде местим главата, понеже в q_Y машината спира, приемайки. Аналогично, в червено са наредените тройки, в които състоянието е q_N . В тях също няма значение каква нова буква пишем и накъде местим главата, понеже в q_N машината спира, отхвърляйки. В сиво са невъзможни (недостижими) комбинации от състояние и буква; примерно, за тази машина, ако е в s_0 , няма как да “види” шпация, и така нататък. Няма значение какво е записано в сивите клетки – записано е нещо, колкото функцията да е тотална.

Работата на машината върху палиндрома 0110 е показана на Фигура 14.1. Приемаме, че времето тече отгоре надолу: най-горе е началната конфигурация със състояние q_0 , най-долу е финалната конфигурация със състояние q_Y . Във всяка конфигурация състоянието е написано вдясно, а съдържанието на лентата е вляво от него. Текущата буква е оцветена в жълто. Клетките на лентата са номерирани но, пак да кажем, това номериране е за по-голяма яснота при четене от човек. Машината не вижда тези номера.

Фигура 14.1 : Машина на Turing приема палиндром.



Ето формално доказателство за коректност на M_{pal} .

Теорема 117: Машината на Turing M_{pal} решава задачата Палиндроми върху $\{0, 1\}$

За всеки стринг $x \in \{0, 1\}^*$, ако M_{pal} е в състояние q_0 и главата на M_{pal} е върху първата буква на x^a , то

- ако x е палиндром, то $M_{\text{pal}}(x) = q_Y$,
- ако x не е палиндром, то $M_{\text{pal}}(x) = q_N$.

^aКакто вече се разбрахме, ако $x = \epsilon$, то първата буква на x е $\sqcup$.

Доказателство: Доказателството е със структурна индукция. За целта ще въведем индуктивна дефиниция на палиндромите. Няма да доказваме, че двете дефиниции са еквивалентни – това е очевидно.

Определение 124: Палиндром, индуктивна дефиниция

Разглеждаме стрингове над азбуката $\Sigma = \{0, 1\}$,

База. ϵ е палиндром, 0 е палиндром и 1 е палиндром.

Стъпка. Нека x е палиндром. Тогава $0x0$ е палиндром и $1x1$ е палиндром.

Нищо друго не е палиндром.

База: Да разгледаме трите стринга от базата на Определение 124. Ако $x = \epsilon$, то, от една страна x е палиндром, а от друга страна машината приема, понеже $\delta(q_0, \sqcup) = \langle q_Y, \sqcup, \downarrow \rangle$. Ако $x = 0$, то, от една страна x е палиндром, а от друга страна, машината от състояние q_0 отива в състояние r_0 и после в q_Y . Ако $x = 1$, то, от една страна x е палиндром, а от друга страна, машината от състояние q_0 отива в състояние r_1 и после в q_Y . ✓

Индуктивна стъпка: Да разгледаме работата на M_{pal} върху произволен стринг $x \in \Sigma^*$ с дължина $n \geq 2$. Следните случаи са взаимно изключващи се и изчерпателни.

- ❶ $x = 0\sigma 1$ за някой $\sigma \in \Sigma^*$. Тогава x не е палиндром съгласно Определение 124. Да разгледаме работата на $M_{\text{pal}}(x)$. Машината от състояние q_0 отива в състояние r_0 , изтривайки най-лявата буква на стринга, с което стрингът става $\sigma 1$. После в състояние w_0 достига първата шпация вдясно от $\sigma 1$, без да променя $\sigma 0$, после се връща една позиция наляво в състояние s_0 и после отхвърля, понеже най-дясната буква на $\sigma 1$ е 1, а $\delta(s_0, 1) = \langle q_N, \sqcup, \downarrow \rangle$.
- ❷ $x = 1\sigma 0$ за някой $\sigma \in \Sigma^*$. Тогава x не е палиндром съгласно Определение 124. Доказваме, че $M_{\text{pal}}(x)$ отхвърля начин, аналогичен на доказателството от ❶.
- ❸ $x = 0\sigma 0$ за някой $\sigma \in \Sigma^*$. Тогава x е палиндром тстк σ е палиндром, съгласно Определение 124.

Индуктивното предположение е, че $M_{\text{pal}}(\sigma)$ работи коректно, тоест, $M_{\text{pal}}(\sigma) = q_Y$, ако σ е палиндром, и $M_{\text{pal}}(\sigma) = q_N$, ако σ не е палиндром. Ще докажем, че $M_{\text{pal}}(x) = M_{\text{pal}}(\sigma)$.

Да разгледаме работата на $M_{\text{pal}}(x)$. Машината от състояние q_0 отива в състояние r_0 , изтривайки най-лявата буква на стринга, с което стрингът става $\sigma 0$. После в състояние w_0 достига първата шпация вдясно от $\sigma 0$ и после се връща една позиция на ляво в състояние s_0 . Сега текущата буква е 0. Тъй като $\delta(s_0, 0) = \langle t, \sqcup, \leftarrow \rangle$, машината изтрива

най-дясната буква на $\sigma 0$, с което стрингът става σ . После в състояние t връща главата до първата шпация вляво от σ и после влиза в състояние q_0 , понеже $\delta(t, \_) = \langle q_0, \_ , \rightarrow \rangle$.

В този момент главата на машината чете първата буква на σ , а машината е в състояние q_0 . Съгласно индуктивното предположение, машината спира в q_Y , ако σ е палиндром, или спира в q_N , ако σ не е палиндром.

Доказахме, че $M_{\text{pal}}(x) = M_{\text{pal}}(\sigma)$. От това и от факта, че x е палиндром тстк σ е палиндром следва, че $M_{\text{pal}}(x)$ работи коректно в случай ③.

④ $x = 1\sigma 1$ за някой $\sigma \in \Sigma^*$. Доказваме, че $M_{\text{pal}}(x)$ работи коректно по начин, напълно аналогичен на доказателството в ③. $\square$

С риск да се получи досадно повторение, ето разговорно описание на работата на машината върху стринг с дължина $n \geq 2$. Машината запомня каква е буквата в началото чрез състоянията r_0 или r_1 , но заменя началната буква с $\_$. От r_0 попада безусловно в w_0 , понеже не прочита шпация в r_0 . Аналогично, от r_1 безусловно попада в w_1 . И така, при стринг с дължина поне две, машината “помни” с каква буква, 0 или 1, е тръгнала от левия край чрез различните състояния w_0 и w_1 . В w_0 , както и в w_1 , машината просто придвижва главата до десния край (първата шпация), без да променя нищо (винаги записва същата буква, която е прочела). Попадайки върху шпация, машината “отскача наляво”, но чрез s_0 или s_1 тя продължава да помни коя буква се опитва да съвпадне.

В s_0 , ако открие 0, презаписва тази нула със шпация и по този начин двете нули (в левия и десния край) се оказват елиминирани и задачата се свежда до това дали остатъкът от стринга е палиндром. В s_0 , ако открие 1, машината отхвърля. Аналогично действа за s_1 .

Състоянието t служи само за да се върне главата до първата шпация вляво, без да се променя нищо. Когато в състояние t прочете шпация, тя е открила левия край на остатък от стринга, за който остава да се провери дали е палиндром, така че “отскача надясно”, застава в състояние q_0 и продължава аналогично. Сега стрингът от не-шпации е с дължина $n - 2$ и машината работи върху него.

14.4.4 Машини на Turing-изчислители на стрингови функции

Според обясненията досега, всяка машина на Turing M за всеки вход x , ако изобщо спре, прави едно от две неща: или приема x чрез q_Y , или отхвърля x чрез q_N . В някакъв смисъл, M изчислява частична или тотална[†] функция с домейн Σ^* и **двуелементен** кодомейн. Кодомейнът е двуелементен, понеже спиращите състояния са две.

Сега ще видим, че с несъществени промени в дефинициите, машините на Turing могат да изчисляват функции, чиито кодомейн е **безкраен**; а именно, безкрайно множество от стрингове. Нека машината е M . Заменяме двете спиращи състояния q_Y и q_N със само едно спиращо състояние q_h . При влизането в q_h , M спира. В този момент текущата клетка трябва да е клетка 1. За изход смятаме стринга, който се намира в този момент върху лентата в клетки номер 1, 2, ..., k , където $k + 1$ е минималният номер на клетка, която съдържа $\_$. Иначе казано, изходът е максималният подстринг, който започва в клетка 1, “върви” надясно и не съдържа шпации. Ако $k = 0$, очевидно клетка 1 съдържа шпация и изходът е ϵ . Забележете, че при спиране на M задължително има поне една клетка[‡] вдясно от клетка 1, съдържаща $\_$, така че изходът е добре дефиниран. M очевидно изчислява частична или тотална функция $f_M : \Sigma^* \rightarrow (\Gamma \setminus \{ \_ \})^*$. Ако M спира върху всички входове, функцията е тотална, а в противен случай е частична. Казваме, че M е *машина-изчислител* или само *изчислител*. За MT

[†]В зависимост от това дали има входове, върху които не спира, или спира върху всеки вход.

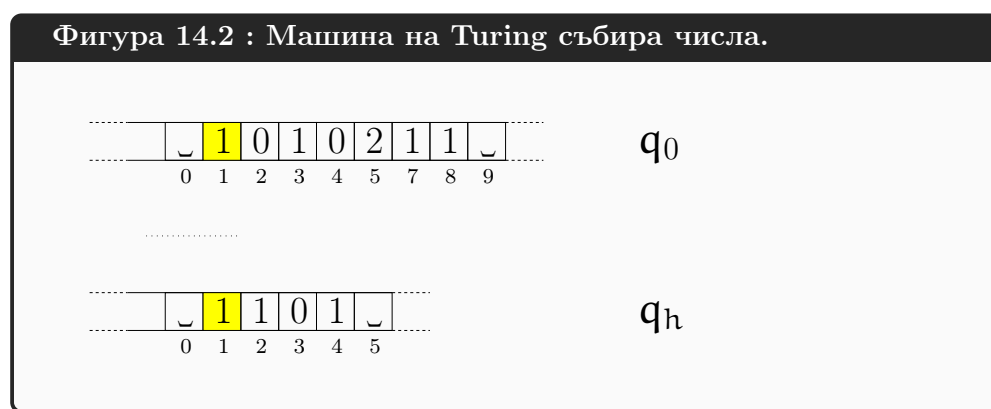
[‡]Всъщност, безброй много клетки.

съгласно Определение 120 използваме термина *машина-разпознавач* или само *разпознавач*; става дума за разпознавач на езици. Тъй можем да кодираме всеки мислим обект със стринг от $(\Gamma \setminus \{\sqcup\})^*$, то, съгласно тезиса на Church, всяка изчислима функция може да се реализира чрез машина-изчислител.

Машина-изчислител лесно симулира машина-разпознавач: изходът да е “1” симулира приемане чрез q_Y , а изходът да е “0” симулира отхвърляне чрез q_N .

Като съвсем прост пример, нека читателят сам/сама конструира МТ, която сумира две естествени числа. Входната азбука може да е $\Sigma = \{0, 1, 2\}$, като буквата 2 е разделител, който бележи границата между двете числа, които трябва да се сумират. Самите числа може да са написани в бинарна система, едното вляво, а другото вдясно от ‘2’. Изходът е бинарното кодиране на сумата на тези две числа. Като лентова азбука може да се ползва $\Gamma = \{0, 1, 2, \sqcup\}$.

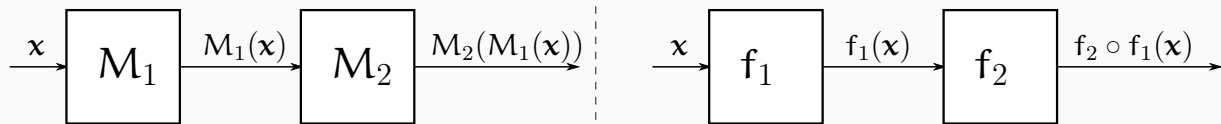
Примерно, нека входът е 1010211. Той кодира десет и три като числа за събиране. Изходът при този вход трябва да е 1101, което кодира тринадесет. Фигура 14.2 показва началната и крайната конфигурация. Детайлите по изчислението остават за читателя.



Забележете, че Определение 122 не е в сила за произволни машини-изчислители! За да говорим за езика L_M , разпознаван от машината на Turing M , трябва M да приема/отхвърля чрез q_Y/q_N , или ако спира чрез q_h , множеството от изходи да е двуелементно. Примерно, няма смисъл да говорим за езика, разпознаван от машината, която сумира естествени числа.

Дефинираме *композиция на машини на Turing*, използвайки машини-изчислители. Не особено формално, ако са дадени две машини M_1 и M_2 , да кажем с една и съща входна азбука Σ и една и съща лентова азбука $\Sigma \cup \{\sqcup\}$, *композицията на M_2 след M_1 върху вход x* е нова машина, която първо пуска (или симулира, ако предпочитате) M_1 с вход x . Когато M_1 спре, тази нова машина веднага пуска/симулира M_2 върху изхода, генериран от M_1 току-що. Ерго, M_1 и M_2 имат обща лента; те не я ползват едновременно, разбира се, а първо M_1 я ползва и спира, след което M_2 започва върху същата лента, като главата на M_2 в началото е точно върху клетка 1, там, където се намираше главата на M_1 при нейното (на M_1) спиране. Ако $M_1(x)$ бележи изхода на M_1 с вход x , то $M_2(M_1(x))$ бележи изхода на композицията на M_2 след M_1 върху вход x . Само композицията на машините, без да уточняваме входа, бележим с “ $M_2 \circ M_1$ ”.

Очевидно е, че ако функцията, изчислявана от M_1 , е f_1 , а функцията, изчислявана от M_2 , е f_2 , то $M_2 \circ M_1$ изчислява функцията $f_2 \circ f_1$. Фигура 14.3 илюстрира нашата представа за тази композиция. Вляво е показано как изходът на машината M_1 става вход на машината M_2 . Вдясно е показано същото нещо, но на ниво функции.

Фигура 14.3 : Композиция на машини-изчислители и на съответните функции.

$M_2(M_1(x))$: изходът на M_1 става вход на M_2 .

M_1 изчислява f_1 , M_2 изчислява f_2 ,
а $M_2 \circ M_1$ изчислява $f_2 \circ f_1$.

Възниква въпросът, какви изчислителни задачи решават машините-изчислители предвид класификацията в Подподсекция 1.1.2.2. За машините-разпознавачи е ясно, че решават задачи за разпознаване. Машините-изчислители решават задачи за търсене, а освен това решават и оптимизационни задачи. Да си припомним, че оптимизационните задачи са частен случай на задачите за търсене (Наблюдение 1).

14.4.5 Сложност по време на машина на Turing. Клас на сложност P.

Вече разполагаме с достатъчно средства, за да изразим прецизно сложността по време на машините на Turing. Забележете, че T_M от Определение 125 е напълно аналогична на T_A от Определение 18 за сложността по време на алгоритми. Разликата е в това, че големината на входа на алгоритъм се дефинира с доста условност, докато при машините на Turing е кристално ясно какво е големината на входа.

Определение 125: Сложност по време на машина на Turing

Нека M е МТ с входна азбука Σ , която спира върху всеки стринг над Σ . *Сложността по време на M* е функцията $T_M : \mathbb{N}^+ \rightarrow \mathbb{N}^+$:

$$T_M(n) = \max_{x \in \Sigma^n} \{k \mid M(x) \text{ спира след точно } k \text{ стъпки} \}$$

M се нарича *полиномиална машина на Turing*, ако съществува полином $p(n)$, такъв че $\forall n \in \mathbb{N}^+ : T_M(n) \leq p(n)$.

Алтернативно, M е полиномиална машина на Turing, ако съществува константа $c \geq 0$, такава че $\forall n \in \mathbb{N}^+ : T_M(n) = O(n^c)$.

Забележете, че машината M може да е разпознавач или изчислител. Това е без значение за Определение 125. Важното е за колко стъпки спира, а дали разпознава език или изчислява функция няма значение.

Определение 126: Клас на сложност, не особено формална дефиниция

Клас на сложност (на английски, *complexity class*) е множество изчислителни задачи със сходна сложност.

На ниво алгоритми, дефиницията на **P** е просто “задачите **за разпознаване**, за които има полиномиален алгоритъм”. На ниво машини на Turing, дефиницията е следната.

Определение 127: Клас на сложност P

Класът на сложност P , по отношение на дадена азбука Σ , е

$$P \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \text{съществува полиномиална машина на Turing } M, \text{ такава че } L = L_M\}$$

В Определение 127, машината трябва да е разпознавач.

Задача за разпознаване P се намира в P , ако съществува разумно кодиране $\mathcal{E}$ на P , такава че $L(P, \mathcal{E}) \in P$, което означава да има полиномиална МТ M , която решава P , кодирана чрез $\mathcal{E}$. Както вече видяхме в Подсекция 14.4.2, всяка МТ M , решаваща задача за разпознаване, има език L_M ; ерго, това, което се иска, е M да е полиномиална и $L(P, \mathcal{E}) = L_M$.

Добре известно е, че “полиномиалността” на машините на Turing е доста устойчиво, още може да кажем стабилно[†], свойство. Ако машина на Turing е полиномиална, на нея съответства алгоритъм, който е полиномиален, и обратното. Ерго, тромавият механизъм на последователно придвижване на главата по лентата на машината на Turing, в сравнение с “пъргавото скачане” от адрес на адрес на съответна RAM машина, не прави машината на Turing драматично по-неефикасна.

Да, машината на Turing е по-неефикасна. В примера с решаването на ПАЛИНДРОМИ (Таблица 14.1) това се вижда много ясно: машината съвпада една буква вляво с една буква вдясно, отново и отново, като за всяко съвпадане изминава пътя от единия до другия край и това води до квадратична сложност[‡], докато алгоритъм би скачал в константно време от адрес на адрес и би решил дали стрингът е палиндром в линейно време.

Но загубата на ефикасност при машините на Turing, спрямо алгоритмите, е само полиномиална, а не, да кажем, експоненциална. Формално доказателство на това излиза далече извън рамките на тези лекции. Подробно изложение на устойчивостта на полиномиалността на машините на Turing, както с една лента, така и с много ленти, има в учебника Computational Complexity на Papadimitriou [125].

14.4.6 Сложност по памет на машина на Turing.

В тези лекционни записки няма да въвеждаме и разглеждаме класове на сложност по памет. Подробно изложение на сложността по памет има в книгата Computational Complexity на Arora и Barak [6]. Тук само ще видим как се дефинира сложност по памет на машина на Turing.

Разглеждаме МТ с поне три ленти. Една от лентите е входната лента: върху нея е записан входът преди началото на работата. Тя е read-only. Друга лента е изходната лента: върху нея машината пише изхода и при спиране на работата изходът е това, което е записано на нея. Тази лента е write-only, като освен това нейната глава се мести само надясно. Това симулира писане в някакъв stream. Освен тези две ленти, машината има една или повече работни ленти, които са read-write и върху които главите се движат “нормално”.

Сложността по памет на машината върху даден вход е общият брой на клетките от **работните ленти**, които са използвани от машината при изчислението с този вход. С други думи, нито входът, нито изходът се отчитат при сложността по памет. Това е напълно аналогично на дефиницията на “сложност по памет” при алгоритмите (Подсекция 2.2.4).

[†]На английски терминът е “robust”; обратното е “fragile”.

[‡]Това не е аргумент, че всяка машина на Turing за решаване на ПАЛИНДРОМИ работи поне в квадратично време. Последното е вярно, но доказателството е нетривиално. Вижте [125, Problem 2.8.5, стр. 52].

14.5 Недетерминизъм и недетерминирани машини на Turing

Докато обикновените (детерминирани) машини на Turing са физически реализирани, изключвайки безкрайната лента, сега ще въведем способност на машините, наречена “недетерминизъм”, която няма никакъв аналог в реалния свят. Недетерминирани машини на Turing са с чисто теоретично значение. Има два еквивалентни начина да се дефинира недетерминирани машина на Turing. “НМТ” означава “недетерминирана машина на Turing”.

14.5.1 НМТ с размножаване

Що е то недетерминизъм? Първият начин да се дефинира НМТ е да се използва Определение 120 с единствената разлика, че сега δ е релация на преходите:

$$\delta \subseteq ((Q \setminus \{q_Y, q_N\}) \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \rightarrow, !\})$$

На прост български, това казва, че на една наредена двойка от състояние и буква съответстват, в общия случай, няколко възможности (може и нула) за “еволюция” на машината, които са наредени тройки от ново състояние, нова буква и движение на главата. Ще се разберем, че ако няма такива, машината *катастрофира* (на английски, *crashes*) и отхвърля входа.

Конфигурация на НМТ, приемаща конфигурация на НМТ и отхвърляща конфигурация на НМТ се дефинират точно като в Определение 121.

Как да интерпретираме възможността машината да еволюира по няколко различни начина едновременно? Какво означава да има няколко нови състояния и изобщо няколко нови конфигурации? Една смислена интерпретация е, че винаги, когато машината трябва да мине в $k > 1$ нови конфигурации, тя се “размножава”: тя изчезва, но се появяват k нови НМТ, всяка от които се оказва в точно една от тези k конфигурации. Всяко от тези копия “заживява свой живот” според δ . Всички тези копия са в едно и също дискретно време, но са различни в смисъл, че всяко си има свое състояние и своя лента със собствена позиция на главата. Тогава тези копия съществуват и работят едновременно, така че въвеждаме *паралелизъм*.

Забележете, че всяко от новите k копия е в своя конфигурация, но разликите между техните конфигурации са само в последния преход. Ерго, като цяло всяка от тези нови k машини “помни” основната част от историята на изчислението досега.

Лесно се вижда защо този изчислителен модел е нереалистичен: ако на всяка стъпка, всяко копие на машината се размножава, дори само на две, след m стъпки, ако не се е стигнало до спиране или катастрофа, ще има 2^m копия общо, работещи паралелно. Предвид факта, че броят на атомите в познатата Вселена се оценява на около 10^{80} , ясно е, че дори за m от порядъка на стотици няма как да се имплементират 2^m копия на недетерминираната машина. А принципът на Landauer (Допълнение 12) казва, че не може да осигурим достатъчно енергия на толкова много, работещи паралелно, машини.

Как недетерминирана машина приема и отхвърля. Да видим какво означава такава неограничено размножаваща се машина да приеме или отхвърли входа. При нормалните, детерминирани машини е ясно: приемането е достигане на приемаща конфигурация, а отхвърлянето е достигане на отхвърляща конфигурация. При размножаващата се машина обаче може в един и същи момент едно копие да достигне приемаща конфигурация, друго да достигне отхвърляща конфигурация или да катастрофира, а трето да е в не-спиращо състояние и да “иска” да продължава да работи. Разглеждаме машината като цяло; става дума за машината с всичките ѝ копия, а не за някое нейно отделно копие.

Определение 128: Приемане и отхвърляне от недетерминирана машина

Нека M е НМТ и x е неин вход.

- M приема x тстк **поне едно нейно копие** достигне до приемаща конфигурация.
- M отхвърля x тстк **всички нейни копия** достигнат до отхвърляща конфигурация или катастрофират, без никое копие да е достигнало до приемаща конфигурация.

И така, приемането е свързано с екзистенциалния квантор, а отхвърлянето, с универсалния; машината приема, ако **съществува** начин да приеме, и отхвърля, ако отхвърля **по всички** начини.

С оглед на това, можем да си представяме, че многото копия на машината, работещи едновременно в едно и също дискретно време, поддържат някаква комуникация помежду си;

- ако поне едно копие достигне приемаща конфигурация, всички останали “умират”, или поне престават да работят, и резултатът е приемане,
- но, ако няма копие в приемаща конфигурация и едно копие $\tilde{M}$ достигне отхвърляща конфигурация, то $\tilde{M}$ спира, обаче останалите копия продължават, стига да не са в отхвърляща конфигурация или катастрофирали.

Наблюдение 98: Асиметрия между приемането и отхвърлянето при недетерминизма

При недетерминизма има асиметрия между приемането и отхвърлянето. В някакъв смисъл приемането се фаворизира. При нормалните детерминирани машини приемането и отхвърлянето са напълно равнопоставени.

Дървовидност на изчислението при НМТ. Недетерминирани машини с размножаване изчисляват дървовидно във времето, което е принципно различно от линейния във времето начин, по който изчисляват детерминирани машини[†].

Определение 129: Дърво на изчислението на недетерминирана машина

Нека е дадена НМТ M и неин вход x . *Дърво на изчислението на $M(x)$* е ориентирано кореново дърво–арборесценция, чиито върхове са конфигурациите, а за всеки две конфигурации α и β , ребро (α, β) съществува тстк състоянието на α не е спиращо и една от новите конфигурации, които α поражда, е конфигурацията β . Коренът на дървото е стартовата конфигурация.

При детерминирани машини също можем да въведем “дърво на изчислението”, и то по точно същия начин, но това дърво при тях се оказва ориентиран път.

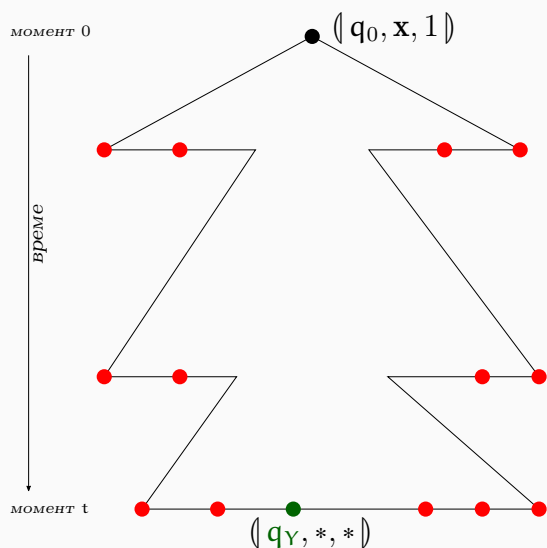
Дървото може да е крайно или безкрайно. Дървото на $M(x)$ е крайно тстк или M приема x или отхвърля x съгласно Определение 128. Това влече, че дървото на $M(x)$ е безкрайно, ако M не приема x , но и не го отхвърля в смисъл, че за всеки момент t от времето има поне едно останало работещо копие; тоест, в дървото има поне един безкраен път “надолу”.

[†]Както и алгоритмите, които са основния предмет на тези лекционни записки.

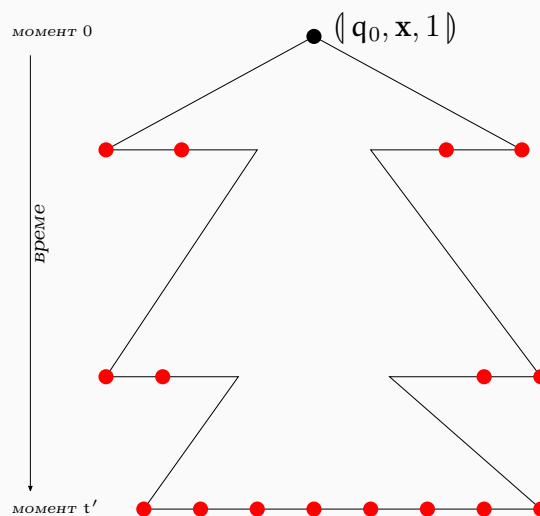
Да разглеждаме конкретен пример за работата на нетривиална НМТ, приемаща някакъв не съвсем тривиален вход и размножаваща се интензивно, би било прекалено. Фигура 14.1 показва пътя на изчислението на детерминирана МТ за сравнително проста задача и само 15 стъпки, и въпреки това е тегава за осмисляне. Представете си НМТ, която се размножава на две на всяка стъпка, във всяко копие и работи 15 стъпки – тя би имала 32 768 копия в края. Поради това ще разгледаме само абстрактен пример за приемане и отхвърляне от НМТ, показан на Фигура 14.4. И в двата случая машината започва от началната конфигурация $(q_0, x, 1)$.

- На подфигурата вляво, едно копие достига до приемаща конфигурация $(q_Y, *, *)$ (зелената точка) в момент t . В някакви предишни моменти копия са достигали до отхвърлящи конфигурации $(q_N, *, *)$ (червени точки) или са катастрофирали. В момента t може други копия да отхвърлят, да катастрофират, да приемат или да “искат” да работят още: това няма значение. Важното е, че поне едно копие е приело в този момент. Поради това машината приема входа x .
- На подфигурата вдясно, нито едно копие не е достигнало до приемаща конфигурация, а в момента t' всички копия вече или са вече достигнали до отхвърлящи конфигурации, или са катастрофирали, и няма нито едно работещо копие. Поради това машината отхвърля входа x .

Фигура 14.4 : Дървото на конфигурациите на недетерминирана машина на Turing.



НМТ приема при първото достигане на q_Y . Други копия може вече да са стигнали q_N , катастрофирали или да “искат” да работят.



НМТ отхвърля, ако всяко листо е в q_N или в катастрофа. За да отхвърли, не трябва да има работещи копия.

Допълнение 76: Backtracking и дървото на конфигурациите на НМТ

Дървото на конфигурациите на НМТ е доста подобно на *дървото на търсенето с връщане*. Търсене с връщане, на английски *backtracking*, е алгоритмична схема, която само споменахме в Секция 6.1. Ще разгледаме търсенето с връщане и дървото на търсене с връщане и после ще се върнем към недетерминирани машини на Turing.

Твърди се, че търсенето с връщане е именувано “backtracking” от математика Lehmer. Идеята е много естествена и интуитивна. Всеки от нас е изпълнявал търсене с връщане в някакъв контекст. По думите на Skiena [144, стр. 231]:

Backtracking is a systematic way to iterate through all possible configurations of a search space. ...

... we must generate each one possible configuration exactly once. ... We will model out combinatorial search solution as a vector $\mathbf{a} = (a_1, a_2, \dots, a_n)$...

At each step in the backtracking algorithm, we try to extend a given partial solution $\mathbf{a} = (a_1, a_2, \dots, a_k)$ by adding another element at the end. After extending it, we must test whether what we now have is a solution: if so, we should print it or count it. If not, we must check whether the partial solution is still potentially extendible to a complete solution.

Backtracking constructs a tree of partial solutions, where each vertex represents a partial solution. There is an edge from x to y if node y was created by advancing from x . The tree of partial solutions provides an alternative way to think about backtracking, for the process of constructing the solutions corresponds exactly to doing a depth-first traversal of the backtrack tree.

Да помислим за решение на HAMILTONIAN CYCLE. За тази задача не е известен ефикасен алгоритъм, така че да предложим какво да е ефективно алгоритмично решение. Става дума за обикновен, детерминиран алгоритъм.

- Има алгоритъм с пълно изчерпване^a. Той генерира всички $n!$ пермутации на върховете и за всяка пермутация проверява, а това отнема $\Omega(n)$ време, дали тя реализира Хамилтонов цикъл. Сложността по време е ужасна, дори ако съобразим, че множеството от пермутациите може да бъде разбито на $\frac{(n-1)!}{2}$ класа на еквивалентност, всеки с по $2n$ елемента, като всички пермутации от един клас отговарят на един единствен цикъл^b в графа, така че е достатъчно да извършим $\frac{(n-1)!}{2}$ проби.
- По-добър на практика, макар и пак със суперполиномиална сложност в най-лошия случай, е алгоритъм, работещ чрез търсене с връщане, на английски *backtracking*^c. Разглеждаме произволен връх u . Ако изобщо има Хамилтонов цикъл c , той съдържа u , като, от гледна точка на u , c “идва” от някой съсед на u , “минава” през u и “отива” към друг съсед на u . Ерго, можем да се опитаме да тръгнем от u нанякъде, и така нататък през всички върхове, за да се върнем до u , но без да повтаряме върхове. Това обаче трябва да стане по всички възможни начини, за да сме убедени, че Хамилтонов цикъл няма, в случай, че наистина няма.

На практика, предимството на търсенето с връщане пред пълното изчерпване е в това, че множеството, в което се търси с връщане (което Skiena нарича “search space”) може да е много по-малко от множеството от всички конфигурации. В

примера на Фигура 14.5, където графът има 8 върха, ако опитаме пълно изчерпване, трябва да генерираме $8! = 40\,320$ пермутации на върховете, или поне всички $\frac{7!}{2} = 2\,520$ потенциални цикли **без оглед на това, кои ребра присъстват и кои, не**. При търсенето с връщане не опитваме частични решения, за които в графа не са налице необходимите ребра. Иначе казано, търсенето с връщане е съобразено с конкретиката на графа, поради което дървото на търсенето с връщане, макар и огромно, в общия случай е значително по-малко от множеството на всички комбинаторни конфигурации, създадено без оглед на конкретния граф.

Да използваме търсене с връщане. Частичното решение (partial solution), за което говори Skiena, е редица без повтаряне от върхове, такъв че за всеки два съседни във редицата върха, в графа има ребро; първият елемент на редицата е винаги **u**. Такова частично решение отговаря на прост път с един край **u** в графа. Докато редицата не стане Хамилтонов цикъл (което означава да има ребро и между първия връх, който е **u**, и последния връх), не сме намерили решение. За всеки съсед **v** на последния връх, който не е в частичното решение: слагаме **v** в частичното решение като последен връх и правим същото с уголеменото частично решение. Ако няма такъв връх **v**, махаме последния връх от частичното решение и се връщаме “едно ниво нагоре”.

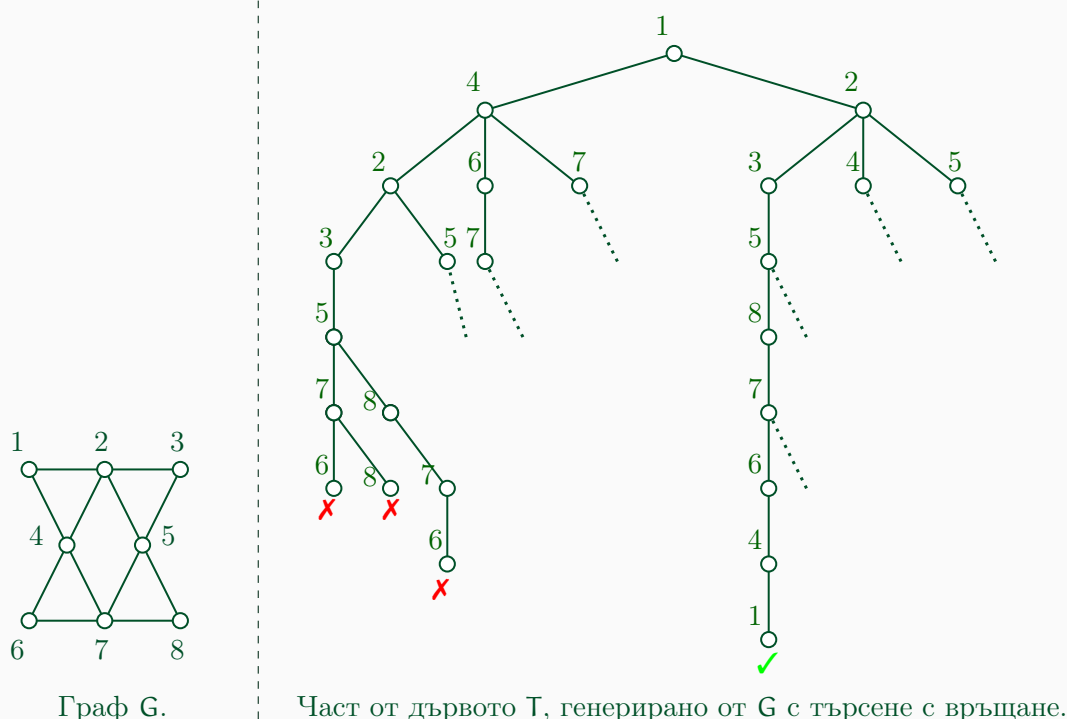
Идеята далечно напомня на DFS, но трябва да внимаваме с тази аналогия! Търсенето с връщане напомня на DFS само с това, че в първия момент, в който “забием” (не сме открили Хамилтонов цикъл, но от текущия връх не може да продължим никъде, понеже всичките му съседни са в частичното решение), се връщаме назад колкото може по-малко и опитваме отново. Съществена разлика с DFS е, че DFS не посещава връх повече от един път, докато търсенето с връщане може да посещава връх отново и отново, но по различни начини. Последното означава, че частичното решение е различно за всеки от тези начини.

Решението с търсене с връщане наистина реализира DFS, но не в оригиналния граф **G**, а в едно дърво **T**, което наричаме *дърво на търсенето с връщане*. **T** е дериват на **G**. Неговият размер е, в най-лошия случай, експоненциален или дори по-голям в размера на **G**. Фигура 14.5 показва граф **G** и само част от съответното му дърво **T**, в което DFS открива Хамилтонов цикъл. Цялото **T** е прекалено голямо, за да се нарисува прегледно, но показаната част от **T** е достатъчна, за да се види идеята. Да кажем, че дървото бива обхождано в preorder според фигурата. От корена 1 отиваме първо в 4, а от 4 първо в 2, Има само два избора: 3 и 5, понеже другите съседни на 2, а именно 1 и 4, са вече сложени в частичното решение. Да кажем, че отидем първо в 3. Оттам единствено може в 5. От 5 може само в 7 и 8. Нека първо е 7. Има два избора: 6 и 8. Да кажем, че първо е 6. Но в 6 “забиваме”, понеже няма продължение, а частичното решение 1, 4, 2, 3, 5, 7, 6 не е Хамилтонов цикъл. Връщаме се колкото може по-малко, тоест до 7. Опитваме 8 и с него също забиваме. Връщаме се на 5, опитваме 8, 7, 6 в този ред и пак забиваме – получихме Хамилтонов път, но не и цикъл.

И така нататък. В даден момент текущото частично решение ще се превърне в пълно решение-Хамилтонов цикъл, понеже графът е Хамилтонов. Едно пълно решение е показано с “✓” в листото.

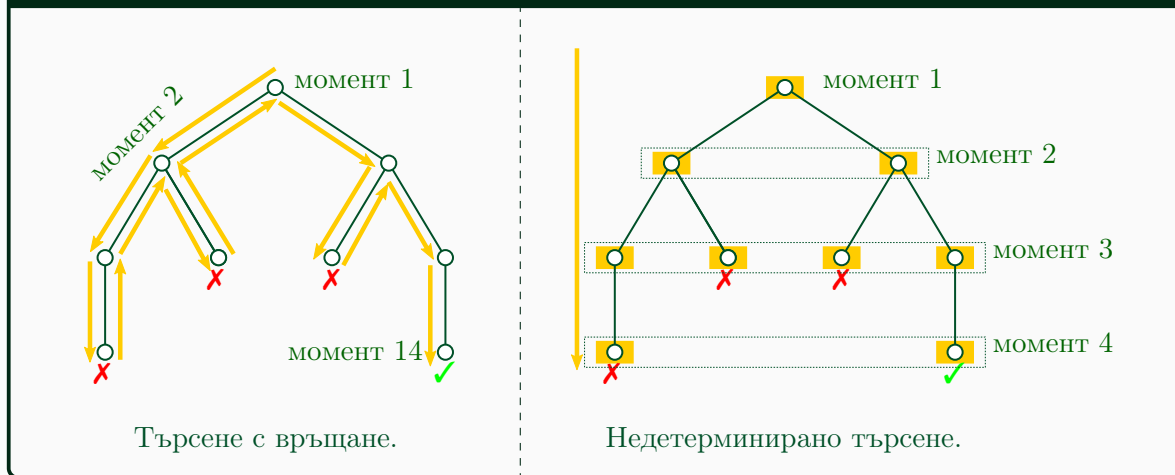
Едва ли е необходимо да се казва, че за всеки връх в дървото, съответното частично (или пълно) решение е пътят между този връх и корена 1.

Фигура 14.5 : Граф, в който търсим с връщане Хамилтонов цикъл.



След като видяхме как работи търсене с връщане за HAMILTONIAN CYCLE за конкретен граф и, най-важното, имаме добра представа какво е дървото на търсене с връщане, да се върнем към недетерминизма. Ето как може да се реши HAMILTONIAN CYCLE от недетерминиран алгоритъм. Недетерминиран алгоритъм **не е точно същото нещо** като НМТ, но за момента игнорираме това. Недетерминираният алгоритъм обхожда дървото T на търсенето с връщане, но не чрез DFS, а чрез нещо като BFS. Стартирайки в корена и вървейки към листата, при всяко не-листо той се размножава (размножава своите променливи, които му определят конфигурацията) в толкова копия, колкото са децата, и тези копия продължават аналогично. Копията се движат в нещо като фронт надолу, навън от корена, като във всеки момент всички копия са на една и съща дълбочина; ерго, целият фронт е еднакво отдалечен от корена винаги. Поради неограничения паралелизъм е възможно фронтът да напредва на единица разстояние за единица време. Ако дори едно копие достигне до q_Y , алгоритъмът приема. Ако копие достигне q_N или катастрофира, това не спира останалите копия, стига да имат продължение. Ако всички копия достигнат q_N или катастрофират, без да има достигане на q_Y , алгоритъмът отхвърля. Фигура 14.6 илюстрира принципната разлика между детерминираният търсене с връщане, реализиращо DFS в дървото на търсенето, и недетерминираният търсене, реализиращо BFS с неограничен паралелизъм с същото дърво. Първото е бавно, но е реализируемо. Второто е бързо, но не е реализируемо физически.

Фигура 14.6 : Търсене с връщане vs недетерминирано търсене.



Има една несъществена разлика между недетерминиран алгоритъм и НМТ. Недетерминираният алгоритъм може да се размножава в колкото си иска копия; в примера с HAMILTONIAN CYCLE, коренът на дървото на търсенето може да има до $n - 1$ деца, което означава, че недетерминираният алгоритъм се размножава в $n - 1$ копия. В контраст с това, НМТ може да се размножи само в константен брой копия, защото релацията на преходите δ е част от описанието на машината, което е предварително фиксирано. Иначе казано, дървото на конфигурациите на НМТ има само константна разклоненост, докато дървото на търсенето може да има разклоненост, която е някаква функция на n .

Това не е съществено. Един връх с разклоненост $f(n)$ в дървото, за каквато и да е функция f , може да бъде симулиран от някакво множество S от върхове, всеки с разклоненост $\Theta(1)$ по такъв начин, че общият брой ребра, излизащи надолу от S , да бъде $f(n)$. И така, дървото на конфигурациите на НМТ може да е по-голямо от дървото на търсенето с връщане, но и двете имат еднакъв брой отхвърлящи листа (маркирани с $\times$, ако дървото е на търсенето, или q_N или катастрофа, ако дървото е на конфигурациите на НМТ) и приемащи листа (маркирани с $\checkmark$, ако дървото е на търсенето, или q_Y , ако дървото е на конфигурациите на НМТ).

^aСпоменахме това понятие в Секция 6.1, където въведохме “алгоритмична схема”

^b**Потенциален** цикъл обаче! Може в графа този клас да не отговаря на цикъл поради липсващи ребра.

^cИ това понятие бе споменато в Секция 6.1.

Езици и задачи за разпознаване при НМТ. Определение 130 буквално повтаря Определение 122.

Определение 130: Езикът, разпознаван от недетерминирана машина на Turing

Нека M е НМТ с входна азбука Σ . Езикът, разпознаван от M , е

$$\{x \in \Sigma^* \mid M(x) = q_Y\}$$

Този език означаваме с L_M .

И тук е вярно, че не се иска машината да спира върху всички входове; тоест, да спира и за

входовете $x \in \Sigma^* \setminus L_M$. Ако x не е в езика, M може да го отхвърли по описания вече начин, а може и да не спре никога.

Определение 131 буквално повтаря Определение 123.

Определение 131: Решаване на задача за разпознаване от недетерминирана машина на Turing

Казваме, че НМТ M решава задачата за разпознаване P по отношение на кодирането $\mathcal{E}$, ако M спира върху всички стрингове от входната азбука и $L_M = L(P, \mathcal{E})$. Ако кодирането се подразбира, това става просто $L_M = L(P)$.

14.5.2 НМТ-верификатор

От формална гледна точка, дефинирането на НМТ чрез заменяне на функцията на преходите с релация на преходите е безукорно. Има обаче и друг начин да въведем НМТ, който в някои контексти е по-удобен за осмисляне на нещата. Преди да видим как се дефинира сложност по време на недетерминирана машина на Turing (Подсекция 14.5.5), ще разгледаме този алтернативен поглед на недетерминизма.

Алтернативната дефиниция е следната. Става дума за МТ, която работи в две *фази*. Нека машината е M , като тя спира върху всеки вход. Нека входът y е x и нека x е ДА-екземпляр на задачата за разпознаване, която машината решава. Това е важно: в момента разглеждаме работата върху ДА-екземпляр.

- Във Фаза 1, машината някак, магически, за само една стъпка, “отгатва” стринг $\sigma \in \Sigma^*$, който се нарича *сертификат*[†]. Сертификатът се оказва записан, и то само за единица време, в клетки $-1, -2, \dots, -m$ на лентата, където $m = |\sigma|$. Тъй като входът x е вдясно от клетка 0, няма опасност сертификатът да презапише част от входа. Терминът “отгатва” е неprecizen, но се използва, за да се подчертае функцията на σ . Precizният термин е “генерира”. Машината генерира σ за една стъпка и го записва по указания начин.
- Във Фаза 2, машината работи напълно детерминирано върху лентата, която съдържа както x , така и σ . Машината M има не релация на преходите като в Подсекция 14.5.1, а **функция** на преходите, точно като в Определение 120. Работейки детерминирано, машината достига състояние q_f , тоест приема входа (което е правилно, защото входът е ДА-екземпляр на задачата). Същественото е, че за детерминираното приемане на входа x машината (може да) ползва сертификата σ . За разлика от това, обикновената детерминирана МТ приема входа без помощта на сертификат.

Ясно е откъде идва името “сертификат”: σ **сертифицира**, тоест удостоверява, факта, че x е ДА-екземпляр.

Ако това е неясно, да видим пример. Да разгледаме пак HAMILTONIAN CYCLE и НМТ със сертификат за нея. Входът x е кодиране на граф $G = (V, E)$. Сертификатът σ е кодиране на пермутация $\langle v_{i_1}, \dots, v_{i_n} \rangle$ на върховете от V , която представлява Хамилтонов цикъл. Щом входът е ДА-екземпляр, то той кодира Хамилтонов граф и поне един Хамилтонов цикъл има. Ако читателят се пита, а как машината отгатва пермутация, представляваща именно Хамилтонов цикъл, отговорът е, че разглеждаме теоретичен модел, в който такава магия е позволена. И така, машината отгатва кодиране σ на Хамилтонов цикъл във Фаза 1.

[†]На английски често наричат този стринг *witness*.

Във Фаза 2 машината само проверява, работейки като **детерминирана** машина, че σ наистина кодира Хамилтонов цикъл в графа, кодиран от x . Проверката е лесна, поне на концептуално ниво: машината проверява, че всеки връх на графа се среща точно веднъж в σ (тоест, че $\{v_{i_1}, \dots, v_{i_n}\} = V$) и че в графа има ребра (v_{i_1}, v_{i_2}) , (v_{i_2}, v_{i_3}) , $\dots$, $(v_{i_{n-1}}, v_{i_n})$, (v_{i_n}, v_{i_1}) . Разбира се, ако трябва да направим истинска детерминирана машина на Turing, която да върши това, първо трябва да уточним представяне на графа (списъци на съседство или матрица на съседство) и после да измислим разумно кодиране. Без подробностите на кодирането е безсмислено да правим опит да конструираме машината. Дори да имаме смислено кодиране, подробностите на машината биха били много и много досадни. Тук просто се съгласяваме, че проверката на сертификата може да се направи в полиномиално време при добро желание.

Това не е единствената възможност за реализиране на НМТ със сертификат за HAMILTONIAN CYCLE, но сякаш е най-естествената.

НМТ-верификатор има смисъл само за ДА-екземпляри. В дефинирането на верификатора настояхме, че входът x кодира ДА-екземпляр. Възниква въпросът, а ако x кодира НЕ-екземпляр? Ако задачата е HAMILTONIAN CYCLE, но входът кодира граф, който не е Хамилтонов, как ще бъде отхвърлен от машината? Отговорът е, че за всеки сертификат σ , верификаторът, ползвайки σ , ще стигне до отхвърляне. Примерно, ако машината интерпретира сертификата като редица от върхове, реализираща Хамилтонов цикъл, в проверяващата фаза машината ще установи, че всъщност тази редица не е Хамилтонов цикъл (не съдържа всеки връх точно веднъж или съдържа всеки връх точно веднъж, но необходимите ребра, така щото тази редица да е Хамилтонов цикъл, не са налични в графа).

Асиметрията между приемането и отхвърлянето при НМТ-верификатор се състои в това, че за да приеме, трябва да има поне един сертификат, благодарение на който да приеме, а за да отхвърли, трябва да отхвърли при всеки предоставен сертификат. Ако не съблюдаваме последното, можем да конструираме верификатор, който приема всеки вход. Такъв верификатор би бил безсмислен, въпреки че би приемал всеки ДА-екземпляр; това, че би приемал и всеки НЕ-екземпляр, го обезсмисля.

В изложението надолу разглеждаме верификатори само върху ДА-екземпляри. Приемаме за очевидно, че конструирани от нас верификатори никога няма да приемат НЕ-екземплярите, без да разглеждаме в детайли какво би станало, ако верификатор получи входен стринг, кодиращ НЕ-екземпляр. Не случайно говорим за “верификатор”, а не за “решавач”. Да се реши задачата включва както приемане на ДА-екземплярите, така и отхвърляне на НЕ-екземплярите. При верификаторите ние разглеждаме само “половината” от това, а именно само приемането. Това си заслужава да се натърти.

Конвенция 17: Нашите верификатори не отхвърлят

Когато разглеждаме недетерминирани машини на Turing-верификатори, ги разглеждаме именно като верификатори: разглеждаме работата им по приемането на ДА-екземплярите, игнорирайки това, което биха правили върху НЕ-екземплярите.

14.5.3 Еквивалентност на двете дефиниции на НМТ

Двете дефиниции на НМТ—може дори да се каже “двете възможни визии за НМТ”—са еквивалентни, което ще докажем сега.

Теорема 118: Верификатор може да симулира ефикасно машина с размножаване

Нека е дадена НМТ с размножаване M съгласно Подсекция 14.5.1. Съществува еквивалентна НМТ-верификатор M' съгласно Подсекция 14.5.2, която симулира работата на M , и то с полиномиална загуба на ефикасност.

“С полиномиална загуба на ефикасност” означава, че при дадена M съществува полином $p_M(n)$, такъв че сложността по време на M' е ограничена от $p_M(T_M(n))$, където $T_M(n)$ е сложността по време на M (Определение 132). Помним, че M' е нормална детерминирана машина с изключение на това, че може да отгатва сертификати.

Доказателство: Ще разглеждаме само ДА-екземпляри съгласно Конвенция 17. Нека x е произволен ДА-екземпляр на задачата. Ще покажем, че има машина-верификатор M' , която приема x , симулирайки работата на M върху x . Работата на M върху x се характеризира напълно с дървото на конфигурациите T . Нека h е височината на T . Щом x е ДА-екземпляр, T има поне едно листо w , което е приемаща конфигурация. От корена, който е началната конфигурация, до w има уникален път p – това е свойство на дърветата. Е, машината-верификатор M' с вход x ще измине в някакъв смисъл само p . Магически генерираният сертификат σ има ролята да “навигира” M' по p от началната конфигурация до w .

Ето по-подробно обяснение. Нека δ е релацията на преходите на M . Верификаторът M' започва с магическото генериране на сертификат σ и после симулира M върху x , помагайки си с информация от сертификата. M прави преход от конфигурация в конфигурация (или дори в конфигурации, ако извършва размножаване) съгласно текущия символ и текущото състояние; тоест, съгласно δ . Симулаторът M' , естествено, разполага с δ и “чете” от нея, за да “разбере” какво да прави. Ключово е, че по време на цялата симулация, M' симулира едно единствено копие на M . В самото начало, M съществува в едно копие, чиято конфигурация е $(q_0, x, 1)$. Да разгледаме произволен момент t от “живота” на M , който не е последният. Да кажем, че M е в конфигурация a в този момент.

- Ако M минава в една единствена нова конфигурация b в момента $t + 1$, то симулаторът M' минава в съответна своя нова конфигурация. Това е тривиалният случай.

Имайте предвид, че времето, в което “живее” M' , е различно от времето, в което “живее” M , така че новата конфигурация на M' няма да се получи за единица време. За разлика от M , която се оказва в новата конфигурация b в момента $t + 1$, защото машините на Turing работят по този начин, M' трябва да прочете релевантната част от δ , за да “научи” в какво ново състояние минава M , какво записва M на лентата и накъде M мести главата си. Това прочитане и извършването на съответните действия от страна на M' за промяна на нейната (на M') конфигурация не може да станат за единица време. Но това не е съществено. Ако M' симулира M със само полиномиална загуба на ефикасност, доказателството е наред. А е очевидно, че загубата на ефикасност е само полиномиална.

- Нетривиалният случай е този, в който M се размножава, така че конфигурацията a от момента t поражда конфигурации $b_1, \dots, b_q$, за някакво $q \geq 1$, в момента $t + 1$. Симулаторът M' след прочитане на релевантната част от δ “научава”, че M се готви за размножаване. M' временно прекратява симулирането и “чете” от σ , за да разбере коя от $b_1, \dots, b_q$ е правилната конфигурация. “Правилната конфигурация” е тази b_i , която е след a по пътя p . А конфигурациите $b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_q$ са без значение за M' и тя ги игнорира. Нейната цел е да достигне w и така да приеме x .

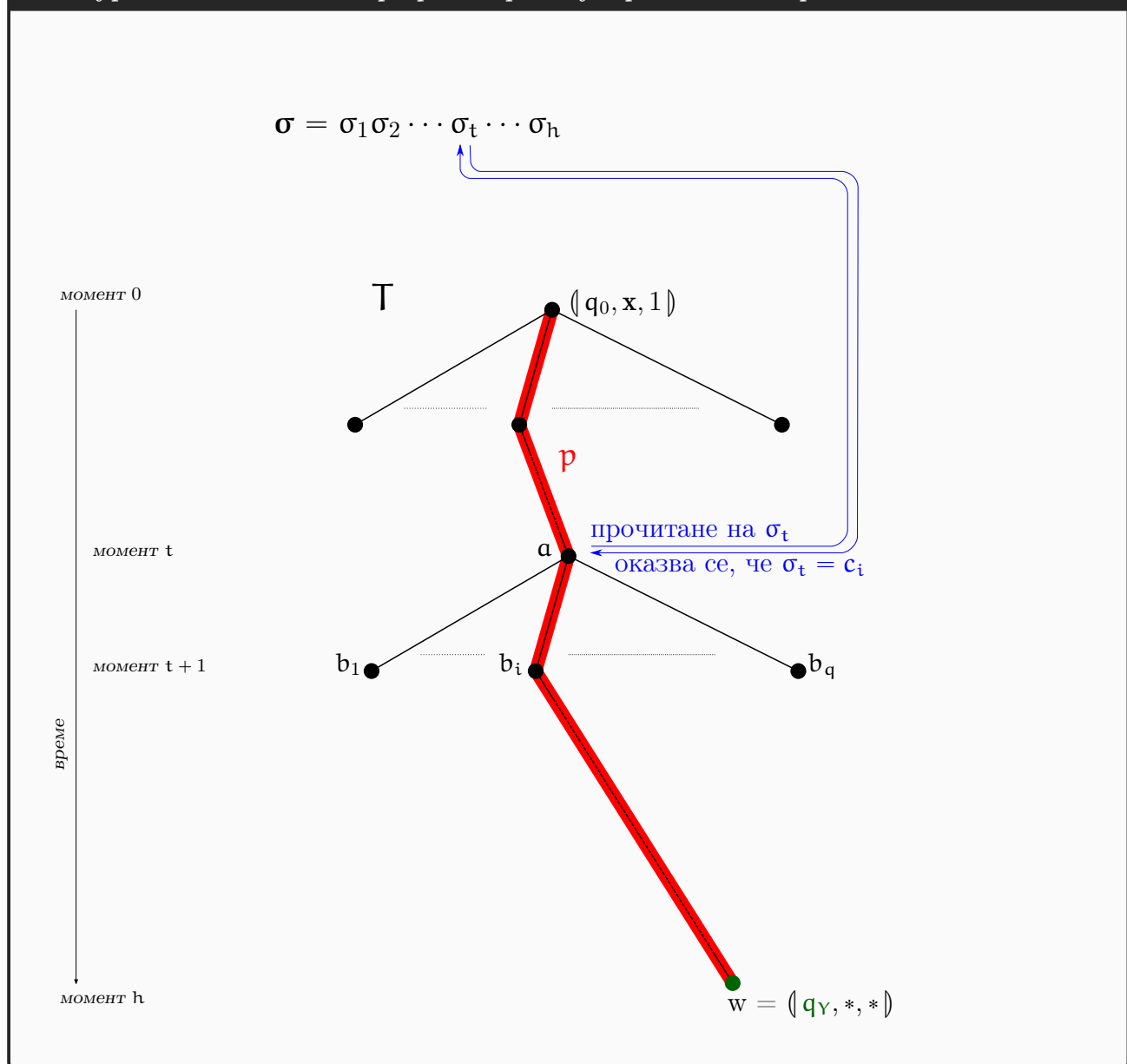
Какво четете M' от σ ? Нека разклонеността на T е ℓ и $\ell \geq 2$. Нека $\tilde{\Sigma} = \{c_1, \dots, c_\ell\}$ е подмножество на Σ . Сертификатът σ е стринг над $\tilde{\Sigma}$ с дължина h , като σ_i е номерът на правилната следваща конфигурация на M при прехода от момент j към момент $j + 1$, за $1 \leq j < h$. Тогава гореспоменатото четене от сертификата е просто прочитане на σ_t . σ_t е някое c_i от $\tilde{\Sigma}$. На буквата c_i съответства конфигурация b_i на M . Тогава симулаторът M' преминава в конфигурация, която съответства на конфигурацията b_i на симулираната машина.

И в този случай е очевидно, че загубата на ефикасност е само полиномиална, щом сертификатът има дължина h .

Би трябвало да е ясно в какъв смисъл сертификатът навигира M' . Всеки негов символ кодира уникалната следваща конфигурация на M по пътя към приемащата конфигурация w . Забележете, че $|p| = h$, така че сертификатът трябва да укаже “правилната посока” на M' най-много h пъти. $\square$

Фигура 14.7 илюстрира идеята на доказателството на Теорема 118. Имайте предвид, че дървото T е само за добиване на интуиция. M' не строи T , а само пътя p (и то имплицитно; по-точно е да се каже, че M' изминава p), който е част от T .

Фигура 14.7 : Машина-верификатор симулира машина с размножаване.

**Теорема 119: Машина с размножаване може да симулира ефикасно верификатор**

Нека е дадена НМТ-верификатор M съгласно Подсекция 14.5.2. Съществува еквивалентна НМТ с размножаване M' съгласно Подсекция 14.5.1, която симулира работата на M , и то с полиномиална загуба на ефикасност.

Доказателство: Дадена е НМТ-верификатор M . Ключовото наблюдение е свързано с четенето на букви от сертификата σ . Всяка буква на σ е от крайната азбука $\tilde{\Sigma} \subset \Sigma$, така че за всяка буква на σ има $|\tilde{\Sigma}|$ възможности, а M' може да симулира всички тези възможности едновременно, размножавайки се в $|\tilde{\Sigma}|$ нови копия.

Ако M не чете сертификата изобщо или сертификатът е празен, то тя е напълно детерминирана машина, използваща магията на недетерминизма. Съответно, M' не се размножава и има функция на преходите, а не релация на преходите, и също е напълно детерминирана. Нещо повече, тя е същата като M .

Да кажем, че M чете непразен сертификат. Размножаванията на M' съответстват точно

на четенията от сертификата от страна на M . При всяко прочитане на буква от сертификата от страна на M , машината M' се размножава в $|\tilde{\Sigma}|$ нови копия. Всяко от тях отговаря на една възможност за прочетената буква.

M работи върху някакъв вход x , който е ДА-екземпляр. Щом е ДА-екземпляр, M го приема. Щом M го приема, има сертификат σ , четенето на букви от който води до това, че M достига приемаща конфигурация w . Очевидно е, че в дървото T на конфигурациите на M' , има листо, чиято конфигурация съответства на w . Уникалният път в T от корена до w съответства точно на прочитането на сертификата от страна на M . Забележете, че T има разклоненост $|\tilde{\Sigma}|$ и, ако M' не катастрофира, T е съвършено дърво, в което пътищата от корена до листата отговарят на всички възможни сертификати с такава дължина (равна на височината).

Също така е очевидно, че загубата на ефикасност при тази симулация е само полиномиална. $\square$

14.5.4 Какви нови възможности ни дава недетерминизмът

Като ефективност на изчислението, недетерминизмът не ни дава нищо повече. Всичко, което може да бъде изчислено от НМТ, може да бъде изчислено от МТ, която не ползва никакви магии като неограничени размножавания в единица време или неясно как появяващи се сертификати, а методично генерира и обхожда с backtracking дървото на конфигурациите. Ако има поне една конфигурация с q_Y , детерминираната машина ще я открие и ще приеме. Ако всички листа са катастрофи или конфигурации с q_N , детерминираната машина ще отхвърли.

Недетерминизмът дава много по отношение на ефикасността на изчислението. Или поне така се смята в Компютърната наука. За съжаление, до ден днешен основните въпроси от неподатливостта нямат категорични отговори. За да видим какво (се смята, че) ни дава недетермизма обаче, трябва да въведем прецизно сложност по време на НМТ.

14.5.5 Сложност по време на НМТ. Клас на сложност NP.

Нека M е НМТ с размножаване и x е вход на M . Нека $M(x)$ спира. Броят на стъпките, за които M спира върху x , е точно височината на дървото на изчислението на $M(x)$: ако M приема x , това е броят стъпки до първото достигане на q_Y от някое копие, а ако M отхвърля x , това е броят стъпки до последното достигане на q_N или катастрофа.

Определение 132 буквално повтаря Определение 125.

Определение 132: Сложност по време на недетерминирана машина на Turing

Нека M е НМТ с входна азбука Σ , която спира върху всеки стринг над Σ . *Сложността по време на M* е функцията $T_M : \mathbb{N}^+ \rightarrow \mathbb{N}^+$:

$$T_M(n) = \max_{x \in \Sigma^n} \{k \mid M(x) \text{ спира след точно } k \text{ стъпки} \}$$

M се нарича *полиномиална недетерминирана машина на Turing*, ако съществува полином $p(n)$, такъв че $\forall n \in \mathbb{N}^+ : T_M(n) \leq p(n)$.

Определение 133: Клас на сложност NP

Класът на сложност **NP**, по отношение на дадена азбука Σ , е

$$\mathbf{NP} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \text{съществува полиномиална НМТ } M, \text{ такава че } L = L_M\}$$

Също както при класа **P** и детерминирани машини, задача за разпознаване P се намира в **NP**, ако съществува разумно кодиране ξ на P , такова че $L(P, \xi) \in \mathbf{NP}$, което означава да има недетерминирана полиномиална машина M , която решава P , кодирана чрез ξ .

Следното твърдение е прекалено очевидно, за да бъде наричано “теорема”.

Наблюдение 99: Всяка детерминирана МТ е и НМТ

Нека M е МТ. Тогава M е и НМТ. Може да мислим за M като за НМТ-верификатор, чиято първа фаза е празна: генерира празен сертификат или генерира някакъв сертификат, но не го чете; няма значение. Същественото е, че втората фаза на НМТ-верификатора е детерминирана и в този смисъл всяка МТ се явява втората фаза на НМТ.

Може да достигнем до същия извод, ако мислим за M като за НМТ с размножаване, чиято релация на преходите е функция. Това е напълно легитимно, понеже всяка функция е релация. Тогава M се явява НМТ с размножаване, която обаче не се размножава (иначе казано, на всяка стъпка от изчислението се размножава в точно едно копие) и дървото на конфигурациите ѝ е път.

Следствие 38: $P \subseteq \mathbf{NP}$

Щом всяка задача от **P** е в **NP**, то $P \subseteq \mathbf{NP}$.

Най-важният нерешен въпрос в теоретичната Компютърна наука е дали $\mathbf{NP} \subseteq P$. Иначе казано, $P \stackrel{?}{=} \mathbf{NP}$. Класът **P** се състои от лесните за решаване задачи, ако приемем, че “лесна за решаване задача” и “задача, за която има полиномиална МТ” са синоними. Класът **NP** се състои от лесните за проверяване задачи, предвид това, че за всеки ДА-екземпляр на такава задача, някоя полиномиална НМТ може да провери сертификат за него в полиномиално време. Въпросът $P \stackrel{?}{=} \mathbf{NP}$ е въпросът дали лесно за решаване е същото като лесно за проверяване. От общи съображения, човек би казал, че това не е така: примерно, много по-лесно е да проверим вече предложено коректно решение на трудна задача по математика и да се убедим, че е коректно, отколкото да решим задачата от нищото.

Както предстои да видим, измежду най-трудните задачи в класа **NP** има много задачи, които са с огромно практическо значение. Утвърдителен конструктивен[†] отговор на $P \stackrel{?}{=} \mathbf{NP}$ би дал на човечеството добро алгоритмично решение за всички задачи в **NP**, включително и въпросните най-трудни задачи. Отрицателен отговор би обезсмислил, поне в общия случай, търсенето на бързи алгоритми. За съжаление, досегашните опити за категоричен отговор на $P \stackrel{?}{=} \mathbf{NP}$ са безуспешни.

[†]“Конструктивен” тук означава “с алгоритъм”. Обратното на конструктивен е екзистенциален, което означава доказателство с чисто математически съображения, например принципа на Dirichlet, без намек за алгоритъм.

Допълнение 77: Защо $P \stackrel{?}{=} NP$ е толкова важен

И все пак, защо този въпрос е ключов? Заслужава си да се проследи малко от историята на математиката, имаща отношение към този въпрос. Както се каза в Допълнение 75, съществува отколен стремеж на математиците да се разбере дали мисленето може да се механизира и, ако да, да се създаде механизъм, с който да може, примерно, да се доказват или опровергават твърдения в математиката. Ако има такъв механизъм, то математикът, наместо да се мъчи да измисли доказателство или контрапример, може да опише интересното твърдение на някакъв изрядно формален, лишен от субективизъм език, да подаде това описание на въпросния механизъм, който (надяваме се, след не много време):

- или да върне изрядно формално, синтактично доказателство, почиващо на аксиоми или вече доказани твърдения, ако твърдението е вярно,
- или да върне някакво опровержение, ако твърдението е невярно.

Резултатите на Gödel [58] и Turing [151] слагат завинаги кръст на тези надежди.

Но хората не спират да се надяват. Щом не може да се генерират механично доказателства на произволни твърдения, дали пък не може да се генерират доказателства в частния случай, когато доказателствата са къси? Забележете, че алгоритмичната нерешимост на HALTING PROBLEM (Задача 8) почива върху това, че нямаме горна граница за времето за работа на програмата $\mathcal{P}$ върху входа $\mathcal{I}$. Ако в екземпляра на задачата има и естествено число k и въпросът е “Дали $\mathcal{P}$ с вход $\mathcal{I}$ терминира за не повече от k стъпки?”, задачата няма да е нерешима: просто симулираме $\mathcal{P}(\mathcal{I})$ за не повече от k стъпки и, ако открием, че спира, връщаме TRUE, в противен случай връщаме FALSE. Ерго, ако в задачата се дава и горна граница за броя на стъпките, алгоритмичната нерешимост изчезва.

Това се превежда в света на доказателствата на твърдения по следния начин. В общия случай няма как алгоритмично да установим дали дадено твърдение е истина или лъжа. Обаче много от важните теореми имат сравнително къси доказателства. Дали, ако е дадена и горна граница k за дължината на доказателството, има механизъм, който да открие доказателство с не по-голяма дължина, или коректно да върне, че такова няма? Този механизъм не трябва да работи с груба сила обаче! С груба сила може да се опита да генерираме всички стрингове с дължина $\leq k$ и сред тях да търсим желаното доказателство, но дори при само бинарно кодиране това означава от порядъка на 2^k стрингове в най-лошия случай. Хипотетичният алгоритъм, търсещ късо доказателство, трябва да прави нещо значително по-умно от систематично генериране на кандидат-доказателствата.

В добре известно писмо на Kurt Gödel до John von Neumann от 1965 г. се казва:

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n , allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F, n)$ be the number of steps the machine requires for this and let $\phi(n) = \max_F \psi(F, n)$. The question is how fast $\phi(n)$ grows for an optimal machine. One can show that $\phi(n) \geq k \cdot n$. If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental

work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. Since (1) it seems that $\phi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem; and (2) after all $\phi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to $\log N$ (or $(\log N)^2$).

На немски “entscheidung” означава “решение” или “заключение”, откъдето “entscheidungsproblem” означава буквално “задача за решаване”. Всяка задача в някакъв смисъл е за решаване, но тук се има предвид това, което наричаме “задача за разпознаване” – такава с булев отговор. Написано с главна буква, “Entscheidungsproblem” е прочутата трета задача на гениалния математик David Hilbert, представена на конференция през 1928 г. По думите на Davis [37, стр. 129]:

Hilbert had also sought explicit calculational procedures by means of which it would always be possible to determine, given some premises and a proposed conclusion, written in the notation of what has come to be called “first-order logic” whether Frege’s rules would enable that conclusion to be derived from those premises. The task of finding such procedures came to be known as Hilbert’s Entscheidungsproblem (literally: decision problem). Of course, systems of calculational procedures for solving specific problems were nothing new. Indeed the traditional mathematical curriculum has been largely made up of such calculational procedures, otherwise known as algorithms. We begin by learning algorithms for addition, subtraction, multiplication, and division of numbers, we move on to algorithms for manipulating algebraic expressions and solving equations, and, if we continue to calculus, we learn how to use the algorithms originally developed by Leibniz for that subject. However, Hilbert was asking for an algorithm of unprecedented scope. In principle, an algorithm for his Entscheidungsproblem would have reduced all human deductive reasoning to brute calculation.

Както знаем, Turing [151] през 1936 г. показва, че Entscheidungsproblem няма решение. Такъв алгоритъм няма, понеже, ако имаше, HALTING PROBLEM щеше да е решима. Gödel през 1956 г. много добре знае, че Entscheidungsproblem няма решение; в крайна сметка, резултатът на Turing от 1936 г. е вдъхновен от основополагащата статия на самия Gödel [58] от 1931 г. Това, което Gödel има предвид в писмото си до von Neumann, е по-скромна цел от тази на Hilbert с Entscheidungsproblem: да се търсят механично само къси доказателства. Но не с брутална сила в експоненциално (в дължината n на доказателството) време, а само в квадратично време.

Може би вече е видима връзката между хипотетичния алгоритъм, който има предвид Gödel, и $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. Да бъде проверено доказателство на математическо твърдение (записано в подходящ формализъм), е аналогично да това, да бъде проверено, че ДА-екземпляр на задача от $\mathbf{NP}$ наистина е ДА-екземпляр, като тази проверка се основава на сертификата. Да не разполагаме с доказателството в контекста на Gödel е същото като да не разполагаме със сертификата при задачата от $\mathbf{NP}$. Gödel пита дали можем

бързо да създадем доказателство, което е късо. Въпросът $P \stackrel{?}{=} NP$ е, дали можем да решим бързо задача от NP , ако не разполагаме със сертификат (или, ако NP се дефинира с размножаване, ако не ползваме магията на неограниченото размножаване). На фундаментално ниво, това е едно и също, ако под “бързо” разбираме “в полиномиално време”.

Да се върнем на $P \stackrel{?}{=} NP$, който въпрос е темата на това допълнение. Въпросът, повдигнат от Gödel до von Neumann, в някакъв смисъл е същият като $P \stackrel{?}{=} NP$. А това е изключително важен, ключов въпрос с философско значение. Наличието на алгоритъм, за какъвто говори Gödel, означава, че математическото мислене може да се механизира въпреки негативния отговор на Entscheidungsproblem. Несъществуването на такъв алгоритъм означава, че математическото мислене не може да се механизира изцяло; дори в някои конкретни случаи да има успешно прилагане на алгоритми за намиране на доказателства (теоремата за четирите цвята, например), като цяло математическото мислене е за хората, а не за машините. Бих си позволил да отида по-далече: ако $P = NP$, то хората наистина са машини, или поне може да се опишат адекватно алгоритмично, точно както твърдят материалистите. Ако обаче $P \neq NP$, то има нещо уникално и неповторимо в човешкия гений, което не може да бъде репликирано в машина или поне в алгоритъм. И така, $P \stackrel{?}{=} NP$ е въпрос и за това в какъв свят живеем и какви сме ние.

Разбира се, нещата са много по-сложни и нюансирани от това. С времето учените откриват, че важният въпрос не е с прост булев отговор. Примерно, може $P \neq NP$, но неподатливостта да е налична само при някакви изключителни, патологични примери на задачите от NP , а да има прилично бързи алгоритми, работещи върху почти всички примери. Ключово важна статия за нюансирани възможни отговори на $P \stackrel{?}{=} NP$ е статията на Russell Impagliazzo [81] от 1995 г. Той разглежда пет възможности, само най-простата (алгоритмично) от които е $P = NP$. Ако $P \neq NP$, има четири съществено различни възможности, само една от които (това, разбира се, е най-патологичната възможна разлика между P и NP) позволява истинска криптография с отворен ключ. В youtube са достъпни двете части от видеозапис на лекция на Russell Impagliazzo по този въпрос: [част 1](#) и [част 2](#).

14.6 Полиномиални редукции

Ние вече видяхме някакви редукции в Секция 13.3, но сега ще въведем понятието отново и много по-подробно. Основната идея е, че ако задача Π_1 се редуцира до задача Π_2 , то задача Π_1 е поне толкова трудна, колкото задача Π_2 ; задача Π_1 има някаква вътрешна, иманентна сложност, която не може да се изпари при редукцията. Ние винаги можем да направим нещата по-сложни, трансформирайки от задача в задача, но не можем да ги направим по-прости.

По думите на Garey и Johnson [56, стр. 34]:

If P differs from NP , ... all problems in $NP \setminus P$ are intractable. ...

Of course, until we can prove that $P \neq NP$, there is no hope of showing that any particular problem belongs to $NP \setminus P$. For this reason, the theory of NP -completeness focuses on proving results of the weaker form “if $P \neq NP$, then $\Pi \in NP \setminus P$ ”. ...

The key idea used in this conditional approach is that of polynomial transformation.

“Polynomial transformation” е синоним на полиномиална редукция.

14.6.1 Фундамент

Определение 134: Карп редукции между задачи

Нека P_1 и P_2 са задачи за разпознаване. *Карп редукция* от P_1 в P_2 е всяка функция $f : P_1 \rightarrow P_2$, такава че

- x е ДА-екземпляр на P_1 тстк $f(x)$ е ДА-екземпляр на P_2 и
- f е изчислима в полиномиално време.

f изобразява ДА-екземпляри на P_1 в ДА-екземпляри на P_2 и НЕ-екземпляри на P_1 в НЕ-екземпляри на P_2 , и освен това работи бързо, тоест в полиномиално време. Мислим за f като за полиномиален алгоритъм, чието множество от входовете е P_1 , а множеството от изходите е подмножество на P_2 . Послесното може да е и същинско подмножество; с други думи, не се иска f да е сюрекция. Не се иска също така f да е инекция.

Без изискването за изчислимост на f в полиномиално време, такава редукция се нарича *many-one reduction* на английски. Името идва от това, че f може да изобразява много екземпляри от P_1 в един екземпляр от P_2 . С други думи, може f да не е инекция[†]. Фактът, че има many-one reduction от P_1 в P_2 се бележи с “ $P_1 \leq_m P_2$ ”.

Many-one reduction, която е и изчислима в полиномиално време, е Карп редукция. Като синоним ще ползваме “полиномиална редукция” или дори само “редукция”, защото се подразбира, че е полиномиална. Ако има полиномиална редукция от P_1 до P_2 ще казваме, че P_1 *се свежда полиномиално до* P_2 .

Понякога е по-удобно да се разглеждат тези редукции като редукции между езици, а не между задачи за разпознаване. Съгласно Подсекция 14.4.2, това е същото.

Определение 135: Карп редукции между езици

Нека Σ е азбука и $L_1, L_2 \in \Sigma^*$. *Карп редукция* от L_1 в L_2 е всяка функция $f : \Sigma^* \rightarrow \Sigma^*$, такава че

- $\forall \sigma \in \Sigma^* : \sigma \in L_1 \leftrightarrow f(\sigma) \in L_2$,
- съществува полиномиална МТ, която изчислява f .

Забележете, че машината, която изчислява f , не е разпознавач.

Наблюдение 100 далечно прилича на Наблюдение 13, но забележете, че Наблюдение 13 касае **работните променливи** на алгоритъм, докато Наблюдение 100 касае **изхода** на алгоритъм (или изхода на МТ).

[†]За да е така, трябва или всички тези екземпляри от P_1 да са ДА-екземпляри и екземплярът от P_2 да е ДА-екземпляр, или всички тези екземпляри от P_1 да са НЕ-екземпляри и екземплярът от P_2 да е НЕ-екземпляр.

Наблюдение 100: Полиномиално ограничение по време влече полиномиално ограничение на размера на изхода

В контекста на Определение 134 или Определение 135, резултатът от редукцията е полиномиално ограничен. По-точно казано,

- в контекста на Определение 134, съществува полином p , такъв че за всеки екземпляр x на Π_1 , размерът на $f(x)$ не надхвърля $p(y)$, където y означава размера на x .
- в контекста на Определение 135, съществува полином p , такъв че за всеки $\sigma \in \Sigma^*$, $|f(\sigma)| \leq p(|\sigma|)$.

Обосновката е практически същата като обосновката на Наблюдение 13: за всеки записан символ от изхода, машината трябва да използва поне единица от дискретното си време.

Нотация 14: Бележим полиномиалните редукции с “ $\leq_p$ ”

Фактът, че Π_1 се свежда полиномиално до Π_2 , записваме с “ $\Pi_1 \leq_p \Pi_2$ ”.

Нотацията “ $\leq_p$ ” задава релация над задачите за разпознаване, която ще означаваме със същия символ.

Определение 136: $\leq_p$ като релация

Нека $\mathcal{D}$ е множеството от задачите за разпознаване. Релацията $\leq_p \subseteq \mathcal{D} \times \mathcal{D}$ се дефинира така: за всеки $\Pi_1, \Pi_2 \in \mathcal{D}$, $\Pi_1 \leq_p \Pi_2$ тстк Π_1 се свежда до Π_2 . Ще я наричаме *релацията на полиномиална сводимост*.

Практически същата релация може да въведем и над множеството от езиците над дадена азбука Σ : език L_1 е в релация с език L_2 тстк $L_1 \leq_p L_2$.

14.6.2 Основни свойства

За изложението върху редукции следните свойства на полиномите са ключово важни.

Наблюдение 101: Полиномите са затворени спрямо събиране, умножение и композиция

Нека $p_1(n)$ и $p_2(n)$ са полиноми. Тогава $p_1(n) + p_2(n)$, $p_1(n) \cdot p_2(n)$ и $p_1(p_2(n))$ са полиноми. Нещо повече, $p_1(n) + p_2(n)$ е от степен $\max\{\deg(p_1), \deg(p_2)\}$, $p_1(n) \cdot p_2(n)$ е от степен $\deg(p_1) + \deg(p_2)$ и $p_1(p_2(n))$ е от степен $\deg(p_1) \cdot \deg(p_2)$.

Примерно, ако $p_1(n) = n^2 + 2n + 7$ и $p_2(n) = 4n^4 + 4n^3 + 10n + 2$, то

- $p_1(n) + p_2(n)$ е $4n^4 + 4n^3 + n^2 + 12n + 9$, който е полином от степен $\max\{2, 4\} = 4$,
- $p_1(n) \cdot p_2(n)$ е $4n^6 + 12n^5 + 36n^4 + 38n^3 + 22n^2 + 74n + 14$, който е полином от степен $2 + 4 = 6$,
- $p_1(p_2(n))$ е $16n^8 + 32n^7 + 16n^6 + 80n^5 + 104n^4 + 24n^3 + 100n^2 + 60n + 15$, който е полином от степен $2 \cdot 4 = 8$.

Теорема 120: Полиномиалните редукции са транзитивни

$\leq_p$ е транзитивна.

Доказателство: Доказателството е по-кратко, ако редукциите са между езици (Определение 135), а не между задачи (Определение 134). Нека L_1 , L_2 и L_3 са езици над някаква азбука Σ . Нека $L_1 \leq_p L_2$ и $L_2 \leq_p L_3$. Ще докажем, че $L_1 \leq_p L_3$.

По определение,

- съществува функция $f_{1,2} : \Sigma^* \rightarrow \Sigma^*$, такава че $\forall x \in \Sigma^* : x \in L_1 \leftrightarrow f_{1,2}(x) \in L_2$ и $f_{1,2}$ е изчислима от полиномиална МТ $M_{1,2}$,
- съществува функция $f_{2,3} : \Sigma^* \rightarrow \Sigma^*$, такава че $\forall x \in \Sigma^* : x \in L_2 \leftrightarrow f_{2,3}(x) \in L_3$ и $f_{2,3}$ е изчислима от полиномиална МТ $M_{2,3}$.

Да разгледаме композицията $f_{2,3} \circ f_{1,2}$. Очевидно тя е функция с домейн и кодомейн Σ^* , като $\forall x \in \Sigma^* : x \in L_1 \leftrightarrow f_{2,3} \circ f_{1,2}(x) \in L_3$. Остава да покажем, че $f_{2,3} \circ f_{1,2}$ е изчислима от полиномиална МТ.

Очевидно композицията на машини $M_{2,3} \circ M_{1,2}$ изчислява $f_{2,3} \circ f_{1,2}$. Остава да покажем, че $M_{2,3} \circ M_{1,2}$ е полиномиална. За тази цел ще разгледаме работата на $M_{2,3} \circ M_{1,2}$ върху произволен $\sigma \in \Sigma^*$.

Щом $M_{1,2}$ е полиномиална, има полином p' , такъв че за всеки $x \in \Sigma^*$, $M_{1,2}(x)$ спира след най-много $p'(|x|)$ стъпки. Тогава, съгласно Наблюдение 100, размерът на $M_{1,2}(\sigma)$ е ограничен така:

$$|M_{1,2}(\sigma)| \leq p'(|\sigma|) \quad (14.1)$$

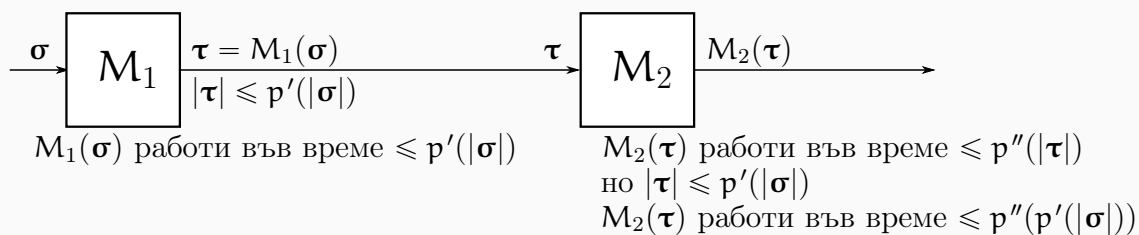
Щом $M_{2,3}$ е полиномиална, има полином p'' , такъв че за всеки $x \in \Sigma^*$, $M_{2,3}(x)$ спира след най-много $p''(|x|)$ стъпки. Тъй като пускаме $M_{2,3}$ върху $M_{1,2}(\sigma)$, който е стринг от Σ^* , има смисъл да разгледаме $M_{2,3}(M_{1,2}(\sigma))$. Очевидно, $M_{2,3}(M_{1,2}(\sigma))$ спира след най-много $p''(p'(|\sigma|))$ стъпки.

Ключовото наблюдение е, че $M_{2,3} \circ M_{1,2}(\sigma)$ работи във време, ограничено от $p'(|\sigma|) + p''(p'(|\sigma|))$. Ето защо.

- Първо пускаме $M_{1,2}$ върху σ , като времето за работа на $M_{1,2}(\sigma)$ е ограничено от $p'(|\sigma|)$. Изходът от $M_{1,2}(\sigma)$ е някакъв стринг τ , като $|\tau| \leq p'(|\sigma|)$ съгласно (14.1).
- После пускаме $M_{2,3}$ върху τ . Времето за работа на $M_{2,3}(\tau)$ е ограничено от $p''(|\tau|)$. Заклучаваме, че времето за работа на $M_{2,3}(\tau)$ е ограничено от $p''(p'(|\sigma|))$.

Фигура 14.8 илюстрира всичко това.

Фигура 14.8 : Композицията на МТ, илюстрираща композиция на редукции.



Съгласно Наблюдение 101, сумата и композицията на полиноми е полином. Заклучаваме, че $p'(|\sigma|) + p''(p'(|\sigma|))$ е полином на $|\sigma|$. Тогава $M_{2,3} \circ M_{1,2}(\sigma)$ е полиномиална МТ, изчисляваща $f_{2,3} \circ f_{1,2}$. Кое и трябваше да покажем. $\square$

“Преднаредба” дефинирахме в Допълнение 18. Преднаредба е релация, която е рефлексивна и транзитивна. Ако преднаредба е симетрична, то тя е релация на еквивалентност. Ако преднаредба е антисиметрична, то тя е частична наредба.

Преднаредбите имат *класове на еквивалентност*, но те не се дефинират по начина, по който сме ги дефинирали при релациите на еквивалентност по Дискретна математика. Един начин да се дефинира “клас на еквивалентност на преднаредба” е като в Определение 36: класовете на еквивалентност на преднаредба на максималните по включване подмножества на домейна, в които всеки елемент е в релация с всеки друг. Забележете, че клас на еквивалентност на релация на еквивалентност е частен случай на клас на еквивалентност на преднаредба, също както релация на еквивалентност е частен случай на преднаредба. Обратно казано, при релациите на еквивалентност класовете на еквивалентност са “напълно изолирани” един от друг, понеже няма как елемент от един клас да е в релация с елемент от друг клас. В контраст на това, при преднаредбите може елемент от един клас да е в релация с елемент от друг клас, но само при положение, че вторият не е в релация с първия (иначе биха били в един и същи клас). Също така забележете, че класовете на еквивалентност на частичните наредби са задължително едноелементни; това следва тривиално от антисиметричността.

Фактор-релация на преднаредба (Определение 36) е релация върху класовете на еквивалентност. Фактор-релацията задължително е антисиметрична; освен това е рефлексивна и транзитивна, което я прави частична наредба.

Следствие 39: Полиномиалните редукции задават преднаредба

$\leq_p$ е преднаредба.

Доказателство: Следва направо от Теорема 120, понеже рефлексивността е очевидна. $\square$

И така, полиномиалните редукции задават преднаредба върху множеството от задачите за разпознаване (или върху езиците, ако предпочитате).

Определение 137: Полиномиално еквивалентни задачи

За две задачи P_1 и P_2 казваме, че са *полиномиално еквивалентни*, ако $P_1 \leq_p P_2$ и $P_2 \leq_p P_1$. Ще отбелязваме този факт с “ $P_1 \equiv_p P_2$ ”.

Ако $P_1 \leq_p P_2$ и $P_2 \not\leq_p P_1$, ще казваме, че P_1 е *полиномиално строго по-малка* от P_2 , което ще записваме с “ $P_1 <_p P_2$ ”.

Ако $P_1 \equiv_p P_2$ или $P_1 <_p P_2$ или $P_2 <_p P_1$, казваме, че P_1 и P_2 са *полиномиално сравними*. В противен случай, те са *полиномиално несравними*.

В контекста на казаното горе за преднаредбите, P_1 и P_2 са полиномиално еквивалентни тстк те са от един и същи клас на еквивалентност на $\leq_p$. Ако обаче $P_1 <_p P_2$, то P_1 и P_2 са от различни класове на еквивалентност, които класове са сравними във фактор-релацията, като класът на P_1 предшества класа на P_2 . Ние не сме показали съществуването на полиномиално несравними задачи, но **ако** има такива задачи, **то** те са в класове на еквивалентност на $\leq_p$, които класове са несравними във фактор-релацията.

Започнахме Секция 14.6 с това, че ако P_1 се редуцира до P_2 , то P_1 е поне толкова трудна, колкото P_2 . Теорема 121 казва същото нещо, но е формулирана прецизно. Тук “лесна” е синоним на “принадлежи на **P**”. Теоремата е формулирана чрез езици, а не чрез задачи.

Теорема 121: Класът P и полиномиалните редукции

Нека L_1 и L_2 са езици над някаква азбука Σ . Нека $L_1 \leq_p L_2$. Тогава $L_2 \in P$ влече $L_1 \in P$.
Алтернативно, $L_1 \notin P$ влече $L_2 \notin P$.

Доказателство: Нека $f: \Sigma^* \rightarrow \Sigma^*$ е полиномиална редукция от L_1 в L_2 . Нека M_f е полиномиална МТ, изчисляваща f . Нека p_f е полином, такъв че за всеки $x \in \Sigma^*$, $M_f(x)$ работи във време, ограничено от $p_f(|x|)$.

Да допуснем, че $L_2 \in P$. Нека M_2 е полиномиална МТ, разпознаваща L_2 . Тогава съществува полином p_2 , такъв че за всеки $x \in \Sigma^*$, $M_2(x)$ работи във време, ограничено от $p_2(|x|)$.

Използвайки M_f и M_2 , конструираме полиномиална МТ M_1 , разпознаваща L_1 . M_1 е композицията $M_f \circ M_2$. Разглеждаме произволен $\sigma \in \Sigma^*$. Първо пускаме M_f с вход σ и после пускаме M_2 върху изхода на $M_f(\sigma)$. Като цяло, пускаме $M_2 \circ M_f(\sigma)$ и така симулираме $M_1(\sigma)$.

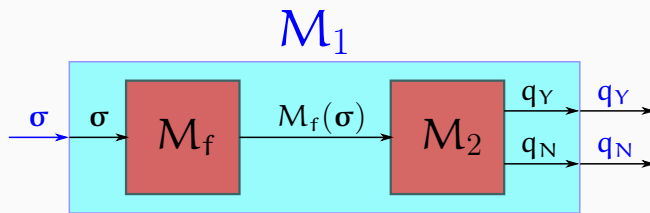
Но M_2 е машина-разпознавач, така че изходът от $M_2 \circ M_f(\sigma)$ е или q_Y , или q_N . По дефиниция, $\sigma \in L_1$ тстк $f(\sigma) \in L_2$, така че

- ако $M_2 \circ M_f(\sigma) = q_Y$, то $\sigma \in L_1$,
- ако $M_2 \circ M_f(\sigma) = q_N$, то $\sigma \notin L_1$.

Твърдим, че композицията $M_2 \circ M_f(x)$ работи във време, ограничено от $p_f(|x|) + p_2(p_f(|x|))$, което е някакъв полином. Това заключение правим по начин, напълно аналогичен на начина, по който направихме заключението в доказателството на Теорема 120. Покажахме, че $L_2 \in P$ влече $L_1 \in P$. Но това е същото като $L_1 \notin P$ влече $L_2 \notin P$ – тези две съждения са взаимно контрапозитивни. $\square$

Фигура 14.9 илюстрира нашата представа за полиномиалните редукции. $M_1(\sigma)$ се реализира чрез $M_2 \circ M_f(\sigma)$. От ключово значение е, че след като M_2 приеме чрез q_Y или отхвърли чрез q_N , **буквално** същото нещо става изход и на M_1 . С други думи, не е разрешено редукцията да използва изхода на M_2 , за да изчисли нещо от него и да върне това друго нещо. Изходът на M_1 просто повтаря изхода на M_2 .

Фигура 14.9 : Схема на полиномиална редукция $L_1 \leq_p L_2$.



“Минималният клас на еквивалентност на $\leq_p$ ” в заглавието на Теорема 122 означава следното. Тъй като $\leq_p$ е преднаредба, тя има класове на еквивалентност, които са елементите, върху които е дефинирана нейната фактор-релация, за която знаем, че е частична наредба. Тази частична наредба е (изброимо) безкрайна, но има точно един минимален елемент, който е класът P .

Теорема 122: P е минималният клас на еквивалентност на $\leq_p$

Нека P_1 , P_2 и P_3 са задачи за разпознаване, като $P_1, P_2 \in P$ и $P_3 \notin P$, като за P_3 знаем поне един ДА-екземпляр с константна големина и поне един НЕ-екземпляр с константна големина. Тогава $P_1 \equiv_p P_2$ и $P_1 <_p P_3$.

Доказателство: Това, че $P_1 \equiv_p P_2$, следва директно от Теорема 121. Ерго, задачите от P са в един и същи клас на еквивалентност на $\leq_p$.

Това, че $P_3 \not\leq_p P_1$, също следва директно от Теорема 121. Ерго, този клас на еквивалентност на $\leq_p$ се състои точно от задачите от P .

Да се убедим, че $P_1 \leq P_3$. Нека x е произволен екземпляр на P_1 . Щом $P_1 \in P$, задачата P_1 е решима в полиномиално време. Редукцията от P_1 в P_3 работи така: в полиномиално време изчисляваме дали x е ДА-екземпляр или НЕ-екземпляр на P_1 . Ако x е ДА-екземпляр на P_1 , конструираме някакъв ДА-екземпляр y на P_3 . Ако x е НЕ-екземпляр на P_1 , конструираме някакъв НЕ-екземпляр z на P_3 . Забележете, че не се опитваме да правим алгоритъм за P_3 , а само конструираме ДА-екземпляр или НЕ-екземпляр за нея, което винаги можем да направим, и то бързо, понеже разполагаме с поне един ДА-екземпляр с константна големина и поне един НЕ-екземпляр с константна големина за нея. $\square$

14.6.3 Примери за полиномиални редукции

Първо дефинираме една изключително популярна изчислителна задача, чието пълно име е TRAVELING SALESMAN PROBLEM, но е известна с краткото си име TSP. В редукцията ще ползваме задачата във варианта за разпознаване.

Изч. Задача 61: TSP, ОПТИМИЗАЦИОННА ВЕРСИЯ

екземпляр: Наредена двойка $\langle C, \text{dist} \rangle$, където C е множество от градове $C = \{c_1, \dots, c_n\}$, а dist е функция на разстоянията $d : C \times C \rightarrow \mathbb{N}^+$, такава че $\text{dist}(c_i, c_i) = 0$ за $1 \leq i \leq n$ и $\text{dist}(c_i, c_j) = \text{dist}(c_j, c_i)$ за $1 \leq i < j \leq n$.

въпрос: Пермутация π на градовете, такава че

$$\text{dist}(c_{\pi(1)}, c_{\pi(2)}) + \text{dist}(c_{\pi(2)}, c_{\pi(3)}) + \dots + \text{dist}(c_{\pi(n-1)}, c_{\pi(n)}) + \text{dist}(c_{\pi(1)}, c_{\pi(n)})$$

е минимално.

Изч. Задача 62: TSP, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Наредена тройка $\langle C, \text{dist}, k \rangle$, където C е множество от градове $C = \{c_1, \dots, c_n\}$, dist е функция на разстоянията $d : C \times C \rightarrow \mathbb{N}^+$, такава че $\text{dist}(c_i, c_i) = 0$ за $1 \leq i \leq n$ и $\text{dist}(c_i, c_j) = \text{dist}(c_j, c_i)$ за $1 \leq i < j \leq n$, а k е цяло положително число.

въпрос: Дали има пермутация π на градовете, такава че

$$\text{dist}(c_{\pi(1)}, c_{\pi(2)}) + \text{dist}(c_{\pi(2)}, c_{\pi(3)}) + \dots + \text{dist}(c_{\pi(n-1)}, c_{\pi(n)}) + \text{dist}(c_{\pi(1)}, c_{\pi(n)}) \leq k$$

Допълнение 78: За името “TRAVELING SALESMAN PROBLEM”

Очевидно става дума за неориентиран пълен тегловен граф на n върха. Случаите, в които $n = 1$ и $n = 2$, са безинтересни. Ако $n \geq 3$, в оптимизационния вариант се търси

Хамилтонов цикъл с минимално тегло, където теглото на цикъла е сумата от теглата на ребрата. Тегловната функция не е непременно метрика, понеже няма изискване неравенството на триъгълника да е изпълнено.

Името на задачата можеше да е МИНИМАЛЕН ТЕГЛОВЕН ХАМИЛТОНОВ ЦИКЪЛ, а “TSP” се ползва по исторически причини. В миналото наистина е имало търговски пътници, амбулантни търговци, които е трябвало да обиколят някакви градове, за да си продадат стоката, връщайки се там, откъдето са тръгнали. При това правейки възможно най-евтината обиколка. Това е причината в дефиницията на задачата да има “градове”.

От практическа гледна точка, задачата е формулирана по неестествен начин: при много градове и нормална пътна мрежа между тях е невъзможно да има директна връзка между всеки два града, която не минава през никой друг град и не пресича никое друго шосе между градове. Знаем, че пълният граф K_n не е планарен за $n \geq 5$, така че при пет или повече града пътната мрежа трябва да има доста тунели, подлези и надлези, така че да реализира истински пълен граф.

Трябва ли непременно да се избягва повторно минаване през градове? Ако мислим за житейската задача, очевидно не. Има примери, в които на търговския пътник е по-евтино да минава многократно през даден град, за да извърши обиколката. Ерго, настояването да не се повтарят градове в обиколката е, от практическа гледна точка, каприз, който може да увеличи цената.

Но приемаме, че задачата е дефинирана по този начин и толкова. Графът е пълен, така че при $n \geq 3$ има поне един Хамилтонов цикъл (всъщност, $\frac{1}{2}(n-1)!$ Хамилтонови цикли), така че множеството от допустимите решения е непразно.

През последните десетилетия пред очите ми името на задачата се промени от TRAVELING SALESMAN PROBLEM на TRAVELING SALESPERSON PROBLEM. Ясно е защо. За много от промените в полово-неутрална форма съм съгласен, примерно терминът “chairman” наистина утвърждава неверния стереотип, че председателят е непременно мъж, и е резонно да престане да се използва. Но професията traveling salesman вече не съществува. Амбулантни търговци, обикалящи градове с куфари и продаващи стока от врата на врата, вече (почти?) няма. В ерата на Интернет и онлайн търговия на какво ли не, този вид търговци са част от миналото. Не виждам смисъл да се опитваме да променяме миналото чрез промяна на езика, с който го описваме. Факт е, че всички или почти всички такива амбулантни търговци са били мъже, така че “salesman” не утвърждава никакъв неверен стереотип.

Редукция 1: HAMILTONIAN CYCLE $\leq_p$ TSP

Конструкция: Даден е екземпляр на HAMILTONIAN CYCLE, който е неориентиран граф $G = (V, E)$. Нека $V = \{v_1, \dots, v_n\}$. От G трябва да “изработим” екземпляр на TSP във вариант за разпознаване (Задача 62): трябва да конструираме множество S от градове, да конструираме разстояние между всеки два различни града, и да конструираме число k . Алгоритъмът, който извършва конструцията, трябва да е полиномиален в размера на G .

Има безброй начини да се направи редукцията, но сякаш най-простият е този. Първо, $S \leftarrow \{c_1, \dots, c_n\}$, където $c_1, \dots, c_n$ са някакви протоеlementи. Второ, за $1 \leq i < j \leq n$,

- ако $(v_i, v_j) \in E$, правим $\text{dist}(c_i, c_j) \leftarrow 1$,
- ако $(v_i, v_j) \notin E$, правим $\text{dist}(c_i, c_j) \leftarrow 2$.

Трето, правим $k \leftarrow n$.

Сега ще покажем, че редукцията е коректна в смисъл, че G е ДА-екземпляр на HAMILTONIAN CYCLE тстк $\langle C, \text{dist}, k \rangle$ е ДА-екземпляр на TSP. Това е доста очевидно, но ще направим подробно доказателство в двете посоки.

Първо допускаме, че G е ДА-екземпляр на HAMILTONIAN CYCLE. Тогава G има Хамилтонов цикъл $s = v_{q_1}, \dots, v_{q_n}$. Но очевидно $\text{dist}(c_{q_a}, c_{q_{a+1}}) = 1$ за $1 \leq a \leq n - 1$ и освен това $\text{dist}(c_{q_1}, c_{q_n}) = 1$. Тогава има пермутация π на градовете, каквато се “иска” в Задача 62 предвид това, че $k = n$. Пермутацията е $\pi(a) = q_a$, за $1 \leq a \leq n$.

Сега допускаме, че $\langle C, \text{dist}, k \rangle$ е ДА-екземпляр на TSP. Разглеждаме пермутацията π според дефиницията на Задача 62. Да допуснем, че има два ръзлични града c', c'' , които или са съседи в пермутацията, или са първия и последния в пермутацията, такива че съответните градове в графа не са съседи. Тогава $\text{dist}(c', c'') = 2$, откъдето сумата от разстоянията е строго по-голяма от n . Но тогава $\langle C, \text{dist}, k \rangle$ е НЕ-екземпляр на TSP. ⚡

Остава да покажем, че редукцията може да се извърши в полиномиално време. Входът на редукцията е графът G , който е екземпляр на HAMILTONIAN CYCLE. Твърдейки “редукцията може да се извърши в полиномиално време”, имаме предвид полиномиално в размера на G . На практика редукцията строи пълен тегловен граф, чиито върхове са върховете на G . Ребрата на новопостроения пълен граф са не повече от $\Theta(n^2)$. Теглата са колкото ребрата на новопостроения граф и всяко тегло е с константен размер, така че и теглата имат общ размер най-много $\Theta(n^2)$. И накрая редукцията конструира числото k със стойност n ; то има големина $\Theta(1)$. Ерго, редукцията строи обект с размер $O(n^2)$. Очевидно е, че тя може да се извърши във време $O(n^2)$ от обикновен алгоритъм. Ако се прави от МТ, константата в степенния показател може да е по-голяма, но сложността си остава полиномиална. $\square$

Редукцията на HAMILTONIAN CYCLE към TSP е класика. Тя е първият пример за редукция в учебника на Garey и Johnson [56, стр. 35-36], тъй като е много проста и естествена. Със сигурност има редукция и на TSP към HAMILTONIAN CYCLE, но тя би била много по-сложна; в нея трябва да кодираме числата от екземпляра на TSP в графа-екземпляр на HAMILTONIAN CYCLE, като този граф има Хамилтонов цикъл тстк числата задават ДА-екземпляр на TSP. Това изобщо не е тривиално.

Сега ще видим една редукция, която илюстрира факта, че можем да сведем най-простата възможна задача до най-сложната възможна задача. Както вече забелязахме, винаги можем да направим нещата по-сложни. Като най-проста задача да вземем ЧЕТНОСТ. Като най-сложна задача да вземем HALTING PROBLEM, която е алгоритмично нерешима, а алгоритмичната нерешимост е най-патологичната форма на алгоритмична сложност.

Изч. Задача: ЧЕТНОСТ

екземпляр: Естествено число n .

въпрос: Дали n е четно?

Редукция 2: ЧЕТНОСТ $\leq_p$ HALTING PROBLEM

Конструкция: Разглеждаме произволен екземпляр на ЧЕТНОСТ, който е едно естествено число n .

В константно време изчисляваме четността му.

- Ако n е четно, генерираме ДА-екземпляр на HALTING PROBLEM (Задача 8). Това всъщност е много лесно: трябва да генерираме наредена двойка от програма $\mathfrak{P}$ и неин вход $\mathfrak{I}$, така че $\mathfrak{P}$ спира върху вход $\mathfrak{I}$. Програмата $\mathfrak{P}$ е:

```
int main() {
    return 0; }
```

а $\mathfrak{I}$ е празният стринг. Очевидно $\mathfrak{P}$ с вход празния стринг спира.

- Ако n е нечетно, генерираме НЕ-екземпляр на HALTING PROBLEM. И това е много лесно: трябва да генерираме наредена двойка от програма $\mathfrak{P}$ и неин вход $\mathfrak{I}$, така че $\mathfrak{P}$ да не спира върху вход $\mathfrak{I}$. Напълно достатъчно е да генерираме програма, която зацикля, например $\mathfrak{P}$:

```
int main() {
    for(;;);
    return 0; }
```

Нека и сега $\mathfrak{I}$ е празният стринг. Очевидно $\mathfrak{P}$ с вход празния стринг спира не спира.

И така, $\text{ЧЕТНОСТ} \leq_p \text{HALTING PROBLEM}$. Естествено, $\text{HALTING PROBLEM} \not\leq_p \text{ЧЕТНОСТ}$, така че е изпълнено $\text{ЧЕТНОСТ} <_p \text{HALTING PROBLEM}$. $\square$

В източници от Интернет, често редуциите към HALTING PROBLEM се правят по следния начин. Пак разглеждаме свеждане на ЧЕТНОСТ към HALTING PROBLEM.

Ако n е четно, спри, в противен случай влез в безкраен цикъл.

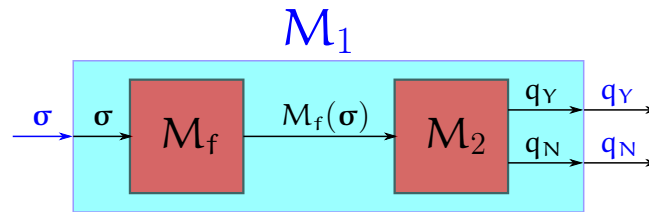
Да наречем тази програма Е2Н. Тя “улавя духа” на същината на нещата, но по формални причини не е редукция от ЧЕТНОСТ към HALTING PROBLEM. По определението, приложено в този контекст, редукцията е тотална функция, която изобразява четните числа в наредени двойки $(\mathfrak{P}, \mathfrak{I})$, където $\mathfrak{P}$ спира върху $\mathfrak{I}$, а нечетните числа в наредени двойки $(\mathfrak{P}, \mathfrak{I})$, където $\mathfrak{P}$ не спира върху $\mathfrak{I}$; освен това, тази тотална функция трябва да може да се реализира с полиномиален алгоритъм. Е2Н дори не реализира тотална функция, а само частична функция, понеже не спира върху нечетните числа. А върху четните числа не връща нищо, тоест, изходът е празният стринг, който не кодира $(\mathfrak{P}, \mathfrak{I})$, където $\mathfrak{P}$ спира върху $\mathfrak{I}$ в никое разумно кодиране.

Според автора, причината да се ползват конструкции като Е2Н за редукции към HALTING PROBLEM е бъркането на двете програми: програмата $\mathfrak{P}$, към която свеждаме, и програмата, осъществяваща редукцията (става дума за тоталната функция; имаме право да я наречем “програма”). Това са различни неща!

14.6.4 Карп редукции срещу Turing редукции

Да разгледаме отново общата схема на Карп редукция[†], показана на Фигура 14.9. Тъй като това е важно, ето отново схемата тук:

[†]“Карп редукция” и “полиномиална редукция” са синоними.



Тази схема не дава истинско решение в смисъл, че не казва точно как да имплементираме M_1 . Тя казва как да имплементираме M_1 , **ако вече** имаме имплементация на M_2 . Машината M_1 не е черна кутия, но M_2 е именно черна кутия – детайлите на нейната имплементация липсват. Общото време за работа върху даден вход не е ясно, защото не се знае колко бързо работи M_2 . Във всеки случай, ако M_2 е полиномиална, то и M_1 е полиномиална.

Ако човек, който не е запознат с тънкостите на теорията на NP-пълнотата трябва да имплементира M_1 , викайки M_2 като процедура, вероятно би подходил не като при Карп редукциите. При Карп редукциите, M_1 с вход σ първо генерира стринг $\tau = M_f(\sigma)$ (като $|\tau|$ е полиномиален в $|\sigma|$), подходящ за вход на M_2 , и вика M_2 само веднъж с вход τ и просто повтаря изхода на $M_2(\tau)$. Това изглежда ненужно рестриктивно по отношение на M_1 , въпреки че е ключово за NP-пълнотата. Ако се подхожда от общи съображения към изграждането на полиномиална редукция, защо M_1 да не се обръща към M_2 няколко пъти? Ако броят на тези обръщания е ограничен от полином в размера на входа, M_1 би била полиномиална, стига M_2 да е полиномиална. С други думи, за да бъде M_1 —работеща чрез викане на M_2 — полиномиална, нито е задължително викането на M_2 да е само едно, нито е задължително M_1 просто да повтаря изхода на M_2 .

И така, да допуснем, че не знаем Карп редукциите. Мислим за редукция, при която има многократни обръщания към M_2 , но не повече от полиномиален брой пъти, и въз основа на получените от M_2 отговори-битове се решава дали да приеме чрез q_Y или отхвърли чрез q_N . Всяко обръщане към M_2 се състои в генериране на някакъв стринг и предоставяне на този стринг като вход на M_2 . При това, **ако** M_2 е полиномиална МТ и генераторът на стринговете-входове на M_2 е полиномиален (което влече, че те са с полиномиален размер), **то** без съмнение имаме машина M_1 , която разпознава езика си в полиномиално време, ползвайки разпознавача M_2 като процедура. Ерго, няма как езикът на M_1 да не е в P , ако езикът на M_2 е в P , което е основна характеристика на редукциите. Човек би казал, че това е смислена дефиниция на полиномиална редукция, която дава доста по-голяма свобода при конструирането на M_1 .

Такива редукции има и те се наричат “Turing редукции”. Дебело подчертаваме, че те **не са** Карп редукции. В теорията, която ще разгледаме, понякога се искат непременно Карп редукции и е груба грешка в тези случаи да се ползват Turing редукции. Следната дефиниция на “Turing редукция” е от [56, стр. 111]. Забележете, че при нея става дума не за задачи за разпознаване, а за задачи за търсене: такава е най-общата дефиниция на “Turing редукция”.

Определение 138: Turing редукции между задачи за търсене

Нека P_1 и P_2 са задачи за търсене. *Полиномиална Turing редукция от P_1 в P_2* е алгоритъм $ALG_{1,2}$, който решава P_1 чрез викане на хипотетична процедура ALG_2 за решаване на P_2 , такива че, ако ALG_2 е полиномиален алгоритъм, то $ALG_{1,2}$ е полиномиален алгоритъм.

Фактът, че има такава редукция, бележим с “ $P_1 \leq_T P_2$ ”.

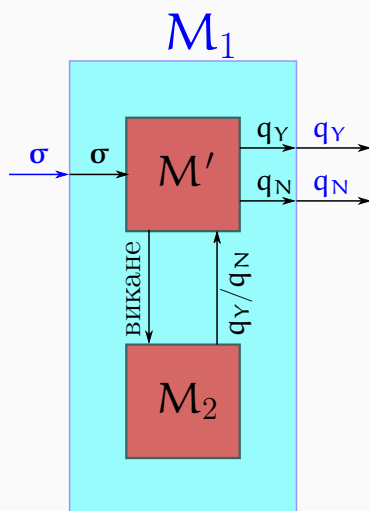
Терминът “хипотетична” идва оттам, че не се интересуваме от реализацията на ALG_2 , а гледаме на него като на черна кутия.

Еквивалентна дефиниция може да се направи не с алгоритми, а с полиномиални машини на Turing. Но те трябва да са не разпознавачи, а изчислители, понеже става дума за задачи за търсене.

Определение 138 изчерпва формалната страна на полиномиалните Turing редукции, що се отнася до същината на нещата. Има смисъл обаче да разгледаме една тяхна ограничена версия, защото на практика тя често се бърка с Карп редуциите. Turing редуциите са по-естествени и лесни за измисляне от Карп редуциите, и това важи особено силно за програмисти, които нямат опит в редуциите. Наистина, трябва известен опит, за да се избягва изкушението да се предлагат Turing редукции като решения на задачи, искащи Карп редукции.

Въпросната схема на ограничен вид полиномиални Turing редукции е показана на Фигура 14.10. Някаква МТ-разпознавач M_1 с език L_1 се реализира чрез МТ M' , която вика, може би многократно, МТ-разпознавач M_2 с език L_2 . M' е изчислител, що се отнася до виканията на M_2 , понеже за всяко викане генерира стринг, който подава като вход на M_2 , но в крайна сметка изходът от M' —това е окончателният отговор на въпроса дали $\sigma \in L_1$ —става чрез q_Y или q_N , така че M' като цяло е разпознавач. M' решава дали да приеме или да отхвърли даден вход на M_1 въз основа на отговорите, получени от M_2 . Ограничението на тази схема, по отношение на Определение 138, е в това, че и M_1 , и M_2 са разпознавачи, а не изчислители.

Фигура 14.10 : Схема на Turing редуция $L_1 \leq_T L_2$.



Има смисъл да кажем, че L_1 се свежда до L_2 , защото $L_1 \notin \mathbf{P}$ влече $L_2 \notin \mathbf{P}$, което е същината на редуцията: ако P_1 е трудна, то и P_2 трябва да е трудна. Но, отново да натъртим, това **не е** Карп редуция. Карп редуцията “изработва” един единствен вход на M_2 , докато при Turing редуцията може да се “изработват” много такива входове, стига техният брой да не е суперполиномиален (за да имаме право да говорим за **полиномиална** Turing редуция). При Карп редуциите (Фигура 14.9) крайният отговор се дава от M_2 , а M_f има за цел само да подготви подходящ вход за M_2 . При Turing редуциите крайният отговор се дава от M' , която е “мозъкът на операцията”.

За илюстрация на разликата между Карп и Turing редуциите, ще разгледаме Turing редуция и Карп редуция, и в двете посоки, между две много подобни задачи: HAMILTONIAN CYCLE (Задача 2) и HAMILTONIAN PATH.

Изч. Задача 63: HAMILTONIAN PATH**екземпляр:** Неориентиран граф $G = (V, E)$.**въпрос:** Дали има Хамилтонов път в G ?

Колкото и да си приличат, HAMILTONIAN CYCLE (Задача 2) и HAMILTONIAN PATH не са една и съща задача. Съществуват графи, например графът на Petersen, които имат Хамилтонов път, но не и Хамилтонов цикъл. От общи съображения е ясно, че те имат сходна сложност. Няма как едната да е трудна, а другата, лесна. Сега ще покажем това формално. БОО, да допуснем, че $n \geq 3$ и че става дума за свързани графи.

HAMILTONIAN CYCLE $\leq_T$ HAMILTONIAN PATH. Да помислим как HAMILTONIAN CYCLE се свежда по най-естествен начин към HAMILTONIAN PATH. Висящите върхове в даден граф имат много общо с Хамилтоновите пътища: очевидно е, че всеки висящ връх е край на всеки Хамилтонов път. Следователно, ако има два висящи върха u и v , за всеки Хамилтонов път е вярно, че краищата му са u и v [†]. И така, ако е даден граф $G = (V, E)$ като екземпляр на HAMILTONIAN CYCLE, можем да въведем два нови върха u и v —такива, които не са във V —и за всяко ребро $(x, y) \in E$ на G

1. да конструираме един нов граф $G' = (V \cup \{u, v\}, E \cup \{(x, u), (y, v)\})$
2. и да подадем този G' като вход на процедура, която отговаря коректно дали G' има Хамилтонов път.

Последователното конструиране на графите G' става от машината, която на Фигура 14.10 е наречена M' . Тя последователно изработва всеки от тях и с него вика M_2 , която чрез q_Y или q_N “казва” дали въпросният G' има или няма Хамилтонов път. Това е Turing редукация: конструират се последователно множество графи и всеки от тях е вход на хипотетична процедура, изчисляваща дали има Хамилтонов път или не.

Ще докажем, че G е Хамилтонов тстк поне в един от тези графи G' има Хамилтонов път.

В едната посока: Да допуснем, че G има Хамилтонов цикъл c . Но този цикъл има n ребра.

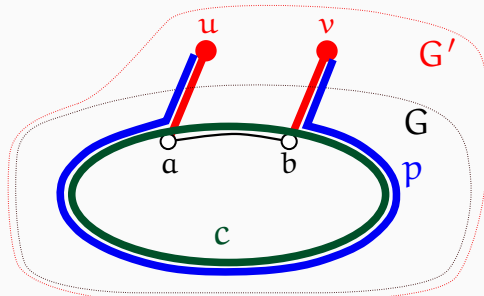
Нека $e = (a, b)$ е произволно ребро от c . Напълно очевидно е, че $G' = (V \cup \{u, v\}, E \cup \{(a, u), (b, v)\})$ има Хамилтонов път, който “тръгва” от u , минава през реброто (a, u) , след това “върви” точно по c от a до b и завършва през реброто (b, v) във v .

В другата посока: Да допуснем, че G' има Хамилтонов път p . Но крайните върхове на p са u и v и ребрата (u, a) и (b, v) са в p . Забелязваме, че p не съдържа реброто (a, b) , понеже G има поне още един връх z освен a и b , така че, ако (a, b) е в p , няма как p да “мине” и през z . Изтриваме u и v от G' и получаваме G . Но $p - u, v$ е Хамилтонов път $\hat{p}$ в G . Добавяме (a, b) към $\hat{p}$ и получаваме Хамилтонов цикъл c в G . $\square$

Фигура 14.11 илюстрира доказателството.

[†] Ако има повече от два висящи върха, Хамилтонови пътища няма изобщо, но това е без отношение към нашите разсъждения.

Фигура 14.11 : Turing редукция HAMILTONIAN CYCLE към HAMILTONIAN PATH.



Виканията на M_2 са $|E|$ на брой, откъдето заключаваме, че броят на виканията е не повече от полиномиален (в размера на входа). Така че това е полиномиална Turing редукция.

Ако искате, може да си представите тази Turing редукция много картинно. G е реализиран в пространството от парчета тел (ребрата му) и топчета (върховете му). Ние ще тестваме дали G е Хамилтонов като разполагаме с две парчета тел (отговарящи на ребрата с краища u и v), които последователно допираме до краищата на всяко ребро на G и за всяко такова допиране-тестване пускаме хипотетична процедура, която отговаря дали полученият по-голям граф има Хамилтонов път. Ако за поне един тест получим утвърдителен отговор, знаем, че оригиналният G е Хамилтонов съгласно горните съображения; ако обаче получим само негативни отговори, знаем от същите съображения, че оригиналният G не е Хамилтонов.

Но предложената редукция не е Карп редукция, понеже генерира много графи, а не един единствен. Наистина, при тази идея за свеждане се налага да се генерират много графи: ако ползваме само едно ребро (a, b) от G , за да изградим G' спрямо него (по гореописания начин) и да тестваме G' за наличие на Хамилтонов път, може да се провалим! Дори G да е Хамилтонов, ако (a, b) не е ребро от никой Хамилтонов цикъл, полученият G' няма да има Хамилтонов път (съгласно гореизложените съображения) и тестът на G' ще върне отрицателен отговор. Ерго, за да сработи тестването на само един G' по този начин, трябва някак да знаем, че (a, b) е ребро от Хамилтонов цикъл. А това няма как да знаем, или поне никой не знае как полиномиално време да се открие ребро, гарантирано принадлежащо на Хамилтонов цикъл (ако има такъв). Ето как може да се направи Карп редукция. Мнението на автора на записките е, че тя е значително по-неочевидна и нетривиална от Turing редукцията.

Редукция 3: HAMILTONIAN CYCLE $\leq_p$ HAMILTONIAN PATH

Конструкция: Разглеждаме произволен екземпляр на HAMILTONIAN CYCLE, който е граф $G = (V, E)$. БОО, $|V| \geq 3$. БОО, G няма срязващи върхове[†]. Разглеждаме произволен връх $x \in V$. Нека съседите на x са $y_1, \dots, y_k$, като $k \geq 2$.

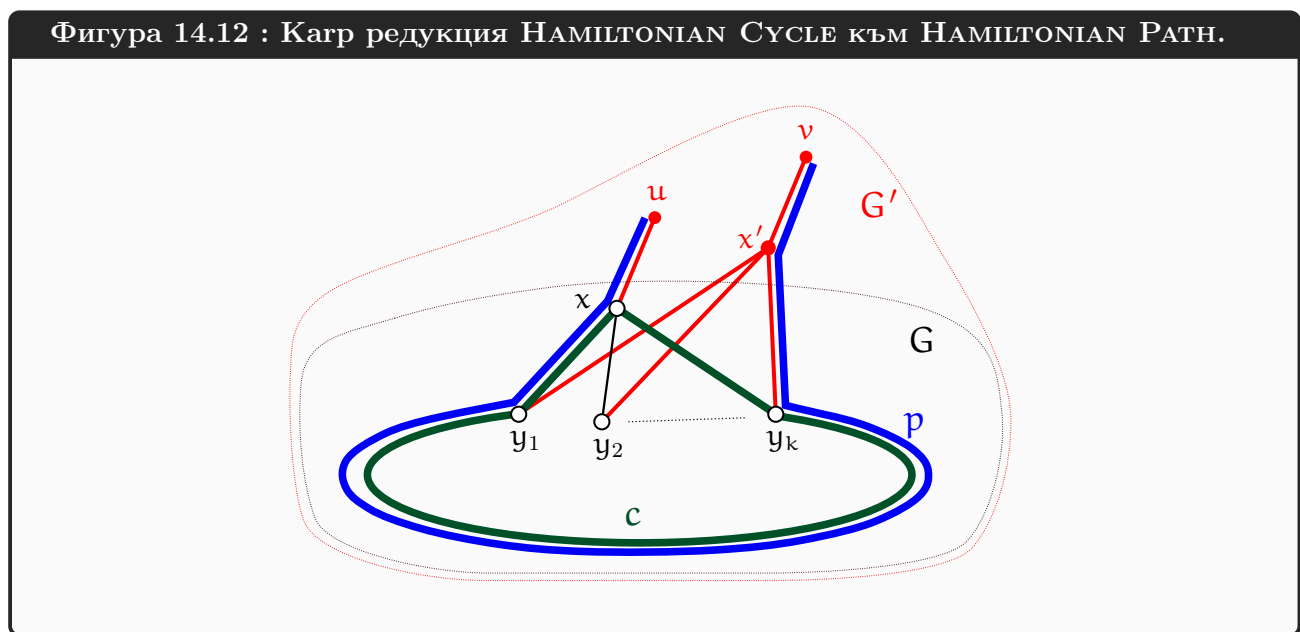
Добавяме един нов връх x' и го правим съсед точно на всеки от $y_1, \dots, y_k$. Добавяме още два нови върха u и v , като правим u съсед точно на x и v съсед точно на x' . Нека така увеличеният граф се нарича G' . Ще докажем, че G е Хамилтонов тстк G' има Хамилтонов път.

[†]Наистина е без ограничение на общността, защото в някакъв preprocessing, в полиномиално време може да се изчисли дали G има срязващи върхове. Ако има, то G е НЕ-екземпляр на HAMILTONIAN CYCLE по очевидни причини и тогава нашата Карп редукция конструира най-прост граф, в който няма Хамилтонов път, примерно два изолирани върха.

В едната посока: Да допуснем, че G има Хамилтонов цикъл c . По определение, c съдържа x . Очевидно c съдържа точно две от ребрата, инцидентни с x . БОО, нека тези ребра са (x, y_1) и (x, y_k) . Тогава в G' има Хамилтонов път, който “тръгва” от u , минава през реброто (u, x) , после през реброто (x, y_1) , след това “върви” точно по c от y_1 до y_k , минава през реброто (x', y_k) и завършва през реброто (x', v) във v .

В другата посока: Да допуснем, че G' има Хамилтонов път p . Но крайните върхове на p са u и v и ребрата (x, u) и (x', v) са в p . p съдържа точно едно ребро с краища x и някой от $y_1, \dots, y_k$; БОО, нека е реброто (x, y_1) . p съдържа точно едно ребро с краища x' и някой от $y_1, \dots, y_k$, който е различен от y_1 ; БОО, нека е реброто (x', y_k) . Изтриваме u , v и x' от G' и получаваме G . Но $p - u, v, x'$ е Хамилтонов път $\hat{p}$ в G . Забележете, че $\hat{p}$ съдържа реброто (x, y_1) , но не съдържа реброто (x', y_k) , понеже x' е изтрит. Също така $\hat{p}$ не съдържа реброто (x, y_k) , но по друга причина – то не присъства в p . Добавяме (x, y_k) към $\hat{p}$ и получаваме Хамилтонов цикъл c в G . $\square$

Фигура 14.12 илюстрира доказателството.



Редукциите—Turing и Karp—между същите две задачи в обратната посока са по-лесни, понеже наличието на Хамилтонов цикъл влече наличие на Хамилтонов път.

Има елементарна Turing редукция на HAMILTONIAN PATH към HAMILTONIAN CYCLE. Даден е граф $G = (V, E)$, екземпляр на HAMILTONIAN PATH. Отново приемаме БОО, че G е свързан и с поне 3 върха. Редукцията генерира следното множество S от графи.

1. Първо, самият G .
2. Второ, за всяка ненаредена двойка върхове x и y , които не са съседи, генерира графа $G' = (V, E \cup \{(x, y)\})$ и го слага в S .

След това редукцията пуска алгоритъм за Хамилтонов цикъл върху всеки граф от S . Ако поне един от тези графи се окаже Хамилтонов, то в G има Хамилтонов път по следната причина.

1. Ако самият G е Хамилтонов, то той тривиално съдържа Хамилтонов път.

2. Нека G не е Хамилтонов, но за някоя двойка несъседи x и y , графът $G' = (V, E \cup \{(x, y)\})$ е Хамилтонов. Тогава Хамилтоновия цикъл c в G' задължително съдържа реброто $e = (x, y)$, иначе G би бил Хамилтонов. Очевидно е, че $c - e$ представлява Хамилтонов път в G .

Карп редукцията също е лесна.

Редукция 4: $\text{HAMILTONIAN PATH} \leq_p \text{HAMILTONIAN CYCLE}$

Конструкция: Разглеждаме произволен екземпляр на HAMILTONIAN PATH , който е граф $G = (V, E)$. Добавяме нов връх u и правим u съсед на всеки връх в $G^\dagger$ и наричаме получения граф G' . Ще докажем, че G' е Хамилтонов тстк G съдържа Хамилтонов път.

В едната посока: Да допуснем, че G' има Хамилтонов цикъл c . По определение, c съдържа новия връх u , както и точно две ребра e_1 и e_2 , инцидентни с u . Очевидно $c - e_1 - e_2$ е Хамилтонов път в G .

В другата посока: Да допуснем, че G има Хамилтонов път p . Краищата на p са някакви върхове a и b от G . Тогава очевидно p плюс ребрата (a, u) и (b, u) плюс връх u е Хамилтонов цикъл в G' . $\square$

Забележка 3: Карп редукциите и Turing редукциите не са едно и също нещо

Доста хора бъркат Карп редукциите и Turing редукциите. Истината е, че всяка Карп редукция е Turing редукция, което е очевидно, но не всяка Turing редукция е Карп редукция. Turing редукциите са строго по-мощни. Това е показано в Допълнение 86: ако Карп и Turing редукциите бяха равномощни, нямаше да има разлика между класовете на сложност NP и coNP .

Накратко, може да има Turing редукция $P_1 \leq_T P_2$, но да няма Карп редукция $P_1 \leq_p P_2$.

14.7 NP-пълнота

Съществува представа, че NP се състои само от трудни задачи, но това съвсем не е така съгласно Определение 133. NP по дефиниция се състои от точно тези задачи за разпознаване, чиито Да-екземпляри могат да бъдат проверени бързо (в полиномиално време) при наличие на подходящ сертификат. Това включва както кошмарно трудни задачи като HAMILTONIAN PATH , така и суперелементарни задачи като ЧЕТНОСТ (дали дадено число е четно). ЧЕТНОСТ наистина е в NP , защото ЧЕТНОСТ очевидно е в P , а съгласно Следствие 38, $\text{P} \subseteq \text{NP}$.

Забележка

Съждението “Задача P е в NP ” казва малко за трудността на P : само това, че решенията ѝ може да се проверяват бързо. Да, има класове на сложност, които се намират високо над NP в йерархията на сложността и за които е известно, че са различни от NP , така че поне е ясно, че P не е в нито един от тях. Примерно, P не е алгоритмично нерешима.

Но от гледна точка на практическа решимост, “Задача P е в NP ” не казва нищо съществено. Може P да е тривиална, а може да е неподатлива.

[†]Връх като този u се нарича *универсален връх* – той е съсед на всеки друг връх.

NP-пълните задачи са, в някакъв смисъл, най-тежките задачи от NP; разговорно казано, те представляват “истинското NP”.

Определение 139: Клас на сложност NP-с

Нека P е задача за разпознаване. Казваме, че P е *NP-пълна*, ако

1. $P \in NP$ и
2. за всяка задача $P' \in NP$ е вярно, че $P' \leq_p P$.

Класът на сложност **NP-с** се състои точно от NP-пълните задачи.

На английски терминът е *NP-complete*.

Все още не сме доказали, че съществуват NP-пълни задачи, така че на този етап от изложението, **NP-с** може да е празното множество. Съществуването на NP-пълна задача ще докажем в Лекция 15.

Класът **NP-с**, който е централна тема в Лекция 14, Лекция 15, Лекция 16 и Лекция 17, е, в някакъв смисъл, призрак. За разлика от всички други области на математиката и теоретичната компютърна наука, които са солидни конструкции, солидно стъпили на други солидни конструкции, и така нататък чак до аксиомите, теорията на NP-пълнотата прилича на къща, построена върху плаващи пясъци. Причината е, че не знаем отговора на въпроса $P \stackrel{?}{=} NP$.

- Ако $P = NP$, то $P = NP = NP-с$, защото всеки две задачи от P се свеждат полиномиално една до друга. При това положение няма смисъл да разглеждаме **NP-с**. Не че Определение 139 става некоректно, но обектът, който то определя, се тривиализира.
- Ако $P \neq NP$, то $NP-с \subset NP$. В този случай има смисъл да разглеждаме множеството от най-тежките задачи в **NP**, което е **NP-с**.

В отсъствието на отговор на $P \stackrel{?}{=} NP$, ние строим теории, които може да колабират в нищото, ако се окаже, че $P = NP$. Както ще видим дори в тези записки нататък, доста съществени резултати са относителни в смисъл, че са формулирани като “Ако $P \neq NP$, то ...”. Наистина, мнозинството от изследователите вярват, че $P \neq NP$, но вярата не е достатъчна.

14.8 NP-трудност

Определение 140: Клас на сложност NP-h

Нека P е задача за разпознаване. Казваме, че P е *NP-трудна*, ако за всяка задача $P' \in NP$ е вярно, че $P' \leq_p P$. Класът на сложност **NP-h** се състои точно от NP-трудните задачи.

На английски терминът е *NP-hard*.

Очевидно задача е NP-пълна тстк тя е NP-трудна и принадлежи на NP. В теоретико-множествена нотация, $NP-с = NP-h \cap NP$. В тези записки по правило задачите, които са NP-трудни, са елементи на NP, но не винаги е така. HALTING PROBLEM е пример за задача, която е NP-трудна, но не в NP, откъдето не е и NP-пълна.

Теорема 123: HALTING PROBLEM $\in$ NP-h

HALTING PROBLEM е NP-трудна.

Доказателство: Да разгледаме произволна задача P' от множеството NP. Ще покажем, че $P' \leq_p$ HALTING PROBLEM.

Ние не знаем точно каква е P' , така че не може да използваме само неин екземпляр в редукцията. Щом $P' \in$ NP, съществува полиномиална НМТ M' , която решава P' . Допускаме, че разполагаме с M' , и то съвсем конкретно – с някакво подходящо описание (кодиране) на M' . Освен M' имаме и произволен екземпляр на P' , само че кодиран като стринг x с някакво подходящо кодиране.

От нас се иска следното. При дадени M' и x да конструираме двойка от програма и неин вход, такива че тази програма спира върху този вход тстк $M'(x) = q_Y$. И това конструиране да става във време, полиномиално в размера на кодирането на M' и $|x|$. В това се състои Карп редукцията.

Редукцията се състои от конструиране на МТ M , която **може** да симулира M' върху x . Тъй като $P' \in$ NP, за някакъв брой стъпки, или M' приема x , или M' отхвърля x , като това става по правилата на недетерминирани изчисления. Вярно е, че M' работи във време, което е най-много полиномиално в $|x|$, но това не е съществено. Същественото е, че M' или приема, или отхвърля x за краен брой стъпки.

M е детерминирана машина, която или би приела, или би отхвърлила x —ако бъде пусната да симулира $M'(x)$ —във време, което е най-много експоненциално по-голямо от времето за работа на $M'(x)$. При това отговорът на M би бил същият като на M' . Забележете, че нашата редукция **не включва симулирането** на M' от страна на M ! Ключовата дума е “**може**”. M **може** да симулира M' . Нашата редукция **включва само построяването** на M , което със сигурност може да стане във време, полиномиално в размера на (кодирането на) M' и x . Ако редукцията включваше симулирането на M' върху x , нямаше да е непременно полиномиална, поне според сегашното познание[†].

След края на хипотетичното симулиране на M' от страна на M (отново: това симулиране не е част от редукцията, но M е направена така, че **ако** бъде пусната да симулира M' , **то** тя ще получи коректния резултат), машината M разполага с информацията дали $M'(x) = q_Y$ или $M'(x) = q_N$. В края на редукцията, M прави още нещо: построява релевантен екземпляр на HALTING PROBLEM.

- Ако $M'(x) = q_Y$, M построява наредена двойка от програма $\mathfrak{P}$ и неин вход $\mathfrak{I}$, така че $\mathfrak{P}$ спира върху вход $\mathfrak{I}$. Това може да стане като в Редукция 2.
- Ако $M'(x) = q_N$, M построява наредена двойка от програма $\mathfrak{P}$ и неин вход $\mathfrak{I}$, така че $\mathfrak{P}$ не спира върху вход $\mathfrak{I}$. И това може да стане като в Редукция 2. $\square$

В заключение, NP-трудността е нещо като долна граница. Съждението “Задача P е NP-трудна” е същото като “Задача P е поне толкова трудна, колкото всяка задача от NP”. А NP-пълнотата е нещо като точна оценка за трудността на задачата, с точност до полиномиална редукция.

[†]Дали детерминирана МТ може да симулира само с полиномиална загуба на ефикасност недетерминирана МТ е същността на въпроса $P \stackrel{?}{=} NP$.

Лекция 15

SAT и теоремата на Cook.

Резюме: Правим рекапитулация на булевите функции и булевите формули. Въвеждаме валуации и удовлетворимост на булева формула. Правим рекапитулация на ДНФ и КНФ. Въвеждаме задачата SAT и разглеждаме примери за моделиращата сила на КНФ чрез свеждане на задачи до SAT. Доказваме, че SAT е **NP**-пълна. Разглеждаме идея за доказателство за **NP**-трудност чрез свеждане от задача, за която е известно, че е **NP**-трудна. Разглеждаме “географията” на класа **NP** според текущото човешко познание.

15.1 Синтаксис на булевите формули

Фиксираме изброимо безкрайно множество от булеви променливи $\mathcal{X} = \{x_1, x_2, \dots\}$. Колкото и педантично да звучи, различаваме името на булева променлива от самата булева променлива. Името на променливата е буква, макар от безкрайна азбука[†]. Буквите са фиксирани, непроменящи се обекти. В контраст с това, променливите са “кутийки”, във всяка от които се слага булева стойност; те не са фиксирани, иначе нямаше да са променливи.

На всяка променлива x_i съответстват два *литерала*: x_i , който е *положителният литерал* и който е името на променливата, и $\overline{x_i}$, който е *отрицателният литерал*. Визуалната разлика между $\boxed{x_i}$ в черно с нормален математически шрифт и $\boxed{x_i}$ в кафяво със шрифт monospace е удобна, за да различаваме дали става дума за променливата или нейното име. Обаче ще правим тази визуална разлика само в рамките на Секция 15.1. В изложението след това, от контекста трябва да е ясно дали $\boxed{x_i}$ означава променлива или литерал.

Азбуката Σ_v се състои от всички литерали над $\mathcal{X}$:

$$\Sigma_v = \{x_1, \overline{x_1}, x_2, \overline{x_2}, \dots\}$$

Нотация 15: $\text{Var}(\phi)$

Ако ϕ е булева формула (Определение 141), то с “ $\text{Var}(\phi)$ ” означаваме множеството от нейните променливи.

[†]Ако сме още по-педантични, името на променливата е стринг над крайната азбука $\{x, 0, 1, \dots, 9\}$, но чак толкова педантични няма да бъдем.

Определение 141: Синтаксис на булевите формули

Азбуката на булевите формули е $\Sigma_v \cup \{ (,), \neg, \wedge, \vee \}$.

- ❶ Всеки положителен литерал x_i е булева формула. $\text{Var}(x_i) = \{x_i\}$.
- ❷ Всеки отрицателен литерал $\overline{x_i}$ е булева формула. $\text{Var}(\overline{x_i}) = \{x_i\}$.
- ❸ Ако ϕ е булева формула, то $\neg\phi$ е булева формула. $\text{Var}(\neg\phi) = \text{Var}(\phi)$.
- ❹ Нека ϕ и ψ са булеви формули. Тогава $(\phi \wedge \psi)$ е булева формула и $\text{Var}((\phi \wedge \psi)) = \text{Var}(\phi) \cup \text{Var}(\psi)$.
- ❺ Нека ϕ и ψ са булеви формули. Тогава $(\phi \vee \psi)$ е булева формула и $\text{Var}((\phi \vee \psi)) = \text{Var}(\phi) \cup \text{Var}(\psi)$.

Означаваме множеството от всички булеви формули с Φ .

Забележете, че $\neg$, $\wedge$ и $\vee$ тук са просто букви. Когато разглеждаме синтактичната дефиниция, трябва да (се опитаме да) забравим техния общоприет смисъл, въпреки че го знаем много добре.

Ето примери за формули съгласно Определение 141:

$$\begin{array}{lll}
 \phi_1 = x_{14}, & \phi_2 = \overline{x_{14}}, & \phi_3 = \neg x_{14}, \\
 \phi_4 = \neg \overline{x_{14}}, & \phi_5 = \neg((x_2 \vee x_3) \vee x_{14}), & \phi_6 = \neg \neg \neg(x_1 \wedge x_2), \\
 \phi_7 = (\overline{x_1} \wedge (\overline{x_1} \wedge \overline{x_1})), & \phi_8 = ((\overline{x_1} \wedge \overline{x_1}) \wedge \overline{x_1}), & \phi_9 = (((\overline{x_1} \wedge \overline{x_1}) \wedge \overline{x_1}) \wedge \neg \overline{x_1})
 \end{array}$$

Формулите съгласно Определение 141 са тромави за четене. Освен това има известен нотационен излишък: може да се мине спокойно и без отрицателни литерали, ползвайки само негация ' $\neg$ '. Но в изложението нататък е доста удобно да се ползват положителни и отрицателни литерали, така че литералите остават.

15.2 Булеви функции

Булева функция е всяка функция от вида $f : \{0, 1\}^n \rightarrow \{0, 1\}$, където $n \in \mathbb{N}$. Числото n е *арността* на функцията, на английски *arity*. Всяка булева функция, по дефиниция, е множество от наредени двойки. Примерно, конюнкцията е $\{((0, 0), 0), ((0, 1), 0), ((1, 0), 0), ((1, 1), 1)\}$. Понякога е удобно да идентифицираме булева функция с нейния булев вектор. Това е доста по-компактно описание. Примерно, конюнкцията е 0001. Помним, че булев вектор представлява булева функция тстк дължината му ℓ е точна степен на двойката, като арността на булевата функция е $\log_2 \ell$.

15.3 Семантика на булевите формули. Удовлетворимост.

Определение 141 задава само синтаксиса на формулите. То казва кои стрингове над $\Sigma_v \cup \{ (,), \neg, \wedge, \vee \}$ са формули и кои не са. Преди да дефинираме семантиката ще дефинираме едно полезно понятие. Да си припомним, че $\mathcal{X}$ бе въведено като безкрайното множество от булеви променливи, от което ще вземаме променливи за нашите формули.

Определение 142: Валюация

За всяко $\mathcal{Y} \subseteq \mathcal{X}$, *валюация на $\mathcal{Y}$* е всяка функция $t : \mathcal{Y} \rightarrow \{0, 1\}$.

На английски терминът най-често е *truth assignment*. На български понякога се казва “булева оценка” вместо “валюация”.

Конвенция 18: Валюациите като булеви вектори

Нека $\mathcal{Y} \subseteq \mathcal{X}$ е крайно и $|\mathcal{Y}| = n$. Тогава $\mathcal{Y} = \{x_{i_1}, x_{i_2}, \dots, x_{i_n}\}$ за някое $\{i_1, i_2, \dots, i_n\} \subset \mathbb{N}$. БОО, нека $i_1 < i_2 < \dots < i_n$. Тогава всяка валюация t на $\mathcal{Y}$ може да бъде идентифицирана с n -мерен булев вектор, чийто k -ти елемент е $t(x_{i_k})$, за $1 \leq k \leq n$.

Предвид това, когато е удобно за изложението, ще мислим за валюация на крайно множество от булеви променливи като за булев вектор.

Нотация 16: $\text{Val}(\mathcal{Y})$

Ако $\mathcal{Y}$ е множество от булеви променливи, множеството от валюации на $\mathcal{Y}$ ще бележим с “ $\text{Val}(\mathcal{Y})$ ”. Ако $|\mathcal{Y}| = n \in \mathbb{N}$, в светлината на Конвенция 18 имаме право да кажем, че $\text{Val}(\mathcal{Y}) = \{0, 1\}^n$. Тогава всяка функция $f : \text{Val}(\mathcal{Y}) \rightarrow \{0, 1\}$ е булева функция с арност n .

Забележете, че $|\mathcal{Y}| = n$ влече $|\text{Val}(\mathcal{Y})| = 2^n$. Забележете също така, че “булева функция” и “валюация” съвсем не са синоними! Всяка булева функция представлява едно “раздаване”—или на нула, или на единица—на всяка валюация.

Допълнение 79: Удовлетворимост на всякакви формули с булев смисъл

Понятието удовлетворимост е приложимо към всяка формула, имаща променливи и имаща булев смисъл, а не само към булевите формули. Примерно, нека α и β са аритметични изрази над реалните числа. Щом са над реалните числа, семантиката на всеки от тях е реално число. Забележете, че формули като “ $\alpha = \beta$ ” или “ $\alpha \leq \beta$ ” са формули с булев смисъл: за всяко конкретно даване на стойности на променливите, целият израз е или истина, или лъжа.

Имаме право да говорим за удовлетворимост на формулите с булев смисъл. Всяка такава формула е удовлетворима тстк съществува даване на стойности на променливите, за което тя е истина. Като пример да разгледаме формулите

$$x + y = z$$

$$u + 1 \geq u$$

$$v + 1 \leq v$$

над реалните числа. Първата е удовлетворима, примерно за $x = \pi$, $y = \sqrt{2}$ и $z = \pi + \sqrt{2}$. Втората са удовлетворима, и то за всяко реално u . Третата не е удовлетворима.

Разбира се, безсмислено е да говорим за удовлетворимост на аритметичен израз α . Дори да има даване на реални стойности на променливите на α , за което α има стойност 0 или 1, това са само реални числа, а не булеви стойности.

В тези записки разглеждаме само удовлетворимост на булеви формули. Булевите формули съгласно Определение 141 не съдържат релационни символи като “=” или “ $\leq$ ”,

но въпреки това имат има булева интерпретация, както ще видим в Определение 143. Поради това има смисъл да говорим за удовлетворимост на булевите формули.

Определение 143: Семантика на булева формула под валюация. Удовлетворимост и фалшифицируемост.

Нека α е булева формула съгласно Определение 141. За всяка $t \in \text{Val}(\text{Var}(\alpha))$ дефинираме $\text{TA}(\alpha, t) \in \{0, 1\}$, която наричаме *семантиката на α под t* , по следния начин.^a

- Ако α е получена чрез ❶, то α е положителен литерал x_i . Тогава $\text{TA}(\alpha, t)$ е равно на $t(x_i)$.
- Ако α е получена чрез ❷, то α е отрицателен литерал $\overline{x_i}$. Тогава $\text{TA}(\alpha, t)$ има обратната стойност на $t(x_i)$.
- Ако α е получена чрез ❸, то α е “ $\neg\phi$ ” за някоя $\phi \in \Phi$. Тогава $\text{TA}(\alpha, t)$ има обратната стойност на $\text{TA}(\phi, t)$.
- Ако α е получена чрез ❹, то α е “ $(\phi \wedge \psi)$ ” за някои $\phi, \psi \in \Phi$. Тогава $\text{TA}(\alpha, t)$ има стойност 1 тстк и $\text{TA}(\phi, t|_{\text{Var}(\phi)})$, и $\text{TA}(\psi, t|_{\text{Var}(\psi)})$ имат стойност 1.
- Ако α е получена чрез ❺, то α е “ $(\phi \vee \psi)$ ” за някои $\phi, \psi \in \Phi$. Тогава $\text{TA}(\alpha, t)$ има стойност 1 тстк поне едната от $\text{TA}(\phi, t|_{\text{Var}(\phi)})$ и $\text{TA}(\psi, t|_{\text{Var}(\psi)})$ има стойност 1.

Казваме, че t *удовлетворява* α , ако $\text{TA}(\alpha, t) = 1$. Казваме, че α е *удовлетворима*, ако съществува валюация на нейните променливи, която я удовлетворява.

Дуално, t *фалшифицира* α , ако $\text{TA}(\alpha, t) = 0$. Казваме, че α е *фалшифицируема*, ако съществува валюация на нейните променливи, която я фалшифицира.

^a “TA” идва от Truth Assignment.

На английски е термините за “удовлетворява” и “удовлетворима” са съответно *satisfies* и *satisfiable*, а за “фалшифицира” и “фалшифицируема” са съответно *falsifies* и *falsifiable*.

Нотация 17: $t \models \phi$

Нотацията “ $t \models \phi$ ” е кратък запис за “ t удовлетворява ϕ ”.

Като пример, да разгледаме формулата $\alpha = (x_1 \wedge (\overline{x_2} \wedge x_3))$. Да разгледаме валюацията $t = 011$ на нейните променливи. Дали $t \models \alpha$? Прилагаме Определение 143. Тъй като α е от вида $(\phi \wedge \psi)$, t я удовлетворява тстк t удовлетворява и ϕ , и ψ . Но ϕ е x_1 . Дали $t|_{\text{Var}(\phi)}$ удовлетворява ϕ ; тоест, дали $t|_{\text{Var}(x_1)}$ удовлетворява x_1 ; тоест, дали $t|_{x_1}$ удовлетворява x_1 ? Това не е вярно, понеже $t(x_1) = 0$. Тогава $t \not\models \alpha$.

Щом t не удовлетворява α , t фалшифицира α .

Валюацията $t' = 101$ удовлетворява $\alpha = (x_1 \wedge (\overline{x_2} \wedge x_3))$. Щом t' удовлетворява α , t' не фалшифицира α .

Виждаме, че α е и удовлетворима, и фалшифицируема.

Пример за формула, която не е удовлетворима, е $((x_1 \wedge \overline{x_1}) \wedge (x_2 \vee x_3))$.

Определение 144: Семантика на булева формула.

Нека ϕ е произволна булева формула съгласно Определение 141. Семантиката на ϕ е булевата функция $f : \text{Val}(\text{Var}(\phi)) \rightarrow \{0, 1\}$, дефинирана така

$$\forall t \in \text{Val}(\text{Var}(\phi)) : f(t) = 1 \text{ тстк } t \models \phi$$

Ако ϕ е булева формула, такава че $\forall t \in \text{Val}(\text{Var}(\phi)) : t \models \phi$, казваме, че ϕ е *тавтология*. Още казваме, че ϕ е *валидна*. Прост пример за тавтология е $(x_1 \vee \overline{x_1})$. Формула е тавтология тстк е нефалшифицируема.

Ако ψ е булева формула, такава че $\forall t \in \text{Val}(\text{Var}(\phi)) : t \not\models \phi$, казваме, че ϕ е *противоречие*. Прост пример за противоречие е $((x_1 \wedge \overline{x_1}) \wedge (x_2 \vee x_3))$, която видяхме преди малко. Формула е противоречие тстк е неудовлетворима.

Наблюдение 102

Нека ϕ е булева формула. ϕ е противоречие тстк $\neg\phi$ е тавтология.

Булева формула да е тавтология е същото като семантиката ѝ да е константа единица. Булева формула да е противоречие е същото като семантиката ѝ да е константа нула. Тавтология и противоречие са две екстремни възможности, които обаче не представляват разбиване – има формули, които не са нито тавтологии, нито са противоречия, понеже има булеви функции, различни от константи. Примерно, $(x_1 \wedge (\overline{x_2} \wedge x_3))$ не е нито тавтология, нито е противоречие.

На пръв поглед е лесно да изчислим дали дадена формула ϕ е противоречие, тавтология или нито едно от двете: достатъчно е да изчислим $\text{TA}(\phi, t)$ за всяка $t \in \text{Val}(\text{Var}(\phi))$. Това е обаче е неефикасно, макар и ефективно, понеже броят на валюациите е експоненциален в броя на променливите. Както ще видим нататък, неефикасността на това изчисление е свързана директно с NP-пълнотата.

15.4 Конюнктивни и Дизюнктивни нормални форми

Тъй като формулите съгласно Определение 141 са тромави, на практика използваме други конвенции за записване на булеви формули. Формулите съгласно тях са несравнимо по-лесни за възприемане.

15.4.1 Дизюнктивна нормална форма

Конюнктивна клауза е всеки непразен стринг, който се състои от конкатенация на литерали, такива че всяко име на променлива се появява най-много веднъж—било като положителен, било като отрицателен литерал. Примери за конюнктивни клаузи са

$$x_1 x_3 x_4,$$

$$x_1 \overline{x_4},$$

$$x_1 x_2 x_3 x_4 x_5 x_6,$$

$$\overline{x_3} \text{ и}$$

$$x_2 \overline{x_5} \overline{x_6}$$

На практика е удобно да мислим за конюнктивните клаузи като за множества. По този начин отпадат евентуални проблеми, произтичащи от различни наредби на едни и същи литерали; ние искаме различно наредени, едни и същи, литерали да задават точно една клауза. Тогава тези клаузи стават

$$\begin{aligned} &\{x_1, x_3, x_4\} \\ &\{x_1, \overline{x_4}\}, \\ &\{x_1, x_2, x_3, x_4, x_5, x_6\}, \\ &\{\overline{x_3}\} \text{ и} \\ &\{x_2, \overline{x_5}, \overline{x_6}\} \end{aligned} \tag{15.1}$$

Определение 145: ДНФ

Дизюнктивна нормална форма, съкратено ДНФ, е формула, която се състои от една или повече различни конюнктивни клаузи, конкатенирани с буквата ‘ $\vee$ ’.

Примери за ДНФ са

$$\begin{aligned} &x_2 \overline{x_3} x_4 \text{ и} \\ &x_1 x_4 \vee x_2 \overline{x_5} \overline{x_6} \vee \overline{x_2} \overline{x_3} \end{aligned}$$

На практика е удобно да мислим за ДНФ като за множество от клаузи; ние искаме различно подредени, едни и същи, клаузи да задават точно една ДНФ. Тогава тези ДНФ стават съответно

$$\begin{aligned} &\{\{x_2, \overline{x_3}, x_4\}\} \text{ и} \\ &\{\{x_1, x_4\}, \{x_2 \overline{x_5}, \overline{x_6}\}, \{\overline{x_2}, \overline{x_3}\}\} \end{aligned} \tag{15.2}$$

ДНФ е частен случай на формула по Определение 141, ако

- мислим за всяка конюнктивна клауза

$$\lambda_1 \lambda_2 \lambda_3 \cdots \lambda_{k-1} \lambda_k$$

като за

$$((\cdots ((\lambda_1 \wedge \lambda_2) \wedge \lambda_3) \cdots \wedge \lambda_{k-1}) \wedge \lambda_k)$$

където $\lambda_1, \dots, \lambda_k$ са литералите

- мислим за цялата ДНФ като за

$$((\cdots ((\alpha_1 \vee \alpha_2) \vee \alpha_3) \cdots \vee \alpha_{t-1}) \vee \alpha_t)$$

където $\alpha_1, \dots, \alpha_t$ са конюнктивните клаузи.

Поради това Нотация 15, Определение 143 и Нотация 17 са приложими към ДНФ.

Наблюдение 103: Удовлетворимост на ДНФ

Нека ϕ е ДНФ и $t \in \text{Val}(\text{Var}(\phi))$. $t \models \phi$ тстк съществува конюнктивна клауза α на ϕ , такава че $t|_{\text{Var}(\alpha)} \models \alpha$. А $t|_{\text{Var}(\alpha)} \models \alpha$ тстк за всеки литерал λ на α е вярно, че $t|_{\text{Var}(\lambda)} \models \lambda$.

15.4.2 Конюнктивна нормална форма

Дизюнктивна клауза е всеки непразен стринг, който се състои от литерали, такива че всяко име на променлива се появява най-много веднъж—било като положителен, било като отрицателен литерал—конкатенирани със символа ‘ $\vee$ ’. Примери за дизюнктивни клаузи са

$$\begin{aligned} &x_1 \vee x_3 \vee x_4, \\ &x_1 \vee \overline{x_4}, \\ &x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6, \\ &\overline{x_3} \quad \text{и} \\ &x_2 \vee \overline{x_5} \vee \overline{x_6} \end{aligned}$$

На практика е удобно да мислим за дизюнктивните клаузи като за множества. По този начин отпадат евентуални проблеми, произтичащи от различни наредби на едни и същи литерали; ние искаме различно наредени, едни и същи, литерали да задават точно една клауза. Тогава тези клаузи стават

$$\begin{aligned} &\{x_1, x_3, x_4\} \\ &\{x_1, \overline{x_4}\}, \\ &\{x_1, x_2, x_3, x_4, x_5, x_6\}, \\ &\{\overline{x_3}\} \quad \text{и} \\ &\{x_2, \overline{x_5}, \overline{x_6}\} \end{aligned} \tag{15.3}$$

Недостатъкът на това представяне е контекстно зависимо. То има смисъл само ако предварително сме казали, че става дума за дизюнктивни клаузи. Забележете, че нищо не отличава (15.3) от (15.1), които означаваха конюнктивни клаузи.

Определение 146: КНФ

Конюнктивна нормална форма, съкратено КНФ, е формула, която се състои от конкатенация чрез буквата ‘ $\wedge$ ’ на една или повече различни дизюнктивни клаузи, всяка от тях е оградена от чифт скоби.

Допустимо е буквата ‘ $\wedge$ ’ между дизюнктивните клаузи да се пропуска. Примери за КНФ са

$$\begin{aligned} &(x_2 \vee \overline{x_3} \vee x_4) \quad \text{и} \\ &(x_1 \vee x_4 \vee x_2) \wedge (x_2 \vee \overline{x_5} \vee \overline{x_6}) \wedge (\overline{x_2} \vee \overline{x_3}) \end{aligned}$$

Втората КНФ може да се напише и като

$$(x_1 \vee x_4 \vee x_2)(x_2 \vee \overline{x_5} \vee \overline{x_6})(\overline{x_2} \vee \overline{x_3})$$

На практика е удобно да мислим за КНФ като за множество от клаузи; ние искаме различ-

но подредени, едни и същи, клаузи да задават точно една КНФ. Тогава тези КНФ стават съответно

$$\left\{ \{x_2, \overline{x_3}, x_4\} \right\} \text{ и } \left\{ \{x_1, x_4\}, \{x_2 \overline{x_5}, \overline{x_6}\}, \{\overline{x_2}, \overline{x_3}\} \right\} \quad (15.4)$$

Нищо не отличава (15.2) от (15.4). Само от контекста можем да знаем дали става дума за ДНФ или КНФ.

Също както ДНФ, и КНФ е частен случай на формула по Определение 141, ако

- мислим за всяка дизюнктивна клауза

$$\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \cdots \vee \lambda_{k-1} \vee \lambda_k$$

като за

$$((\cdots ((\lambda_1 \vee \lambda_2) \vee \lambda_3) \cdots \vee \lambda_{k-1}) \vee \lambda_k)$$

където $\lambda_1, \dots, \lambda_k$ са литералите

- мислим за цялата ДНФ като за

$$((\cdots ((\alpha_1 \wedge \alpha_2) \wedge \alpha_3) \cdots \wedge \alpha_{t-1}) \wedge \alpha_t)$$

където $\alpha_1, \dots, \alpha_t$ са дизюнктивните клаузи.

Поради това Нотация 15, Определение 143 и Нотация 17 са приложими към КНФ.

Сравнете Наблюдение 103 с Наблюдение 104. Те са дуални.

Наблюдение 104: Удовлетворимост на КНФ

Нека ϕ е КНФ и $t \in \text{Val}(\text{Var}(\phi))$. $t \models \phi$ тстк за всяка дизюнктивна клауза α на ϕ е вярно, че $t|_{\text{Var}(\alpha)} \models \alpha$. А $t|_{\text{Var}(\alpha)} \models \alpha$ тстк съществува поне един литерал λ в α , такъв че $t|_{\text{Var}(\lambda)} \models \lambda$.

15.4.3 Превръщане на формули в КНФ или ДНФ

Ето как можем да превърнем произволна булева формула α (съгласно Определение 141) в КНФ или ДНФ. Първо се премахваме всички негации ‘ $\neg$ ’, прилагайки, докато можем,

1. законите на De Morgan

$$\neg(\phi \wedge \psi) \equiv (\neg\phi \vee \neg\psi)$$

$$\neg(\phi \vee \psi) \equiv (\neg\phi \wedge \neg\psi)$$

така че всички символи негация ‘ $\neg$ ’ са само пред литерали, може би в дълги редици. Например, ако е дадена формулата

$$\neg\neg(\overline{x_1} \vee (\neg x_2 \wedge \neg \overline{x_3})) \quad (15.5)$$

правим

$$\begin{aligned}
 & \neg\neg(\overline{x_1} \vee (\neg x_2 \wedge \neg \overline{x_3})) \equiv \\
 & \neg(\neg \overline{x_1} \wedge \neg(\neg x_2 \wedge \neg \overline{x_3})) \equiv \\
 & \neg(\neg \overline{x_1} \wedge (\neg\neg x_2 \vee \neg\neg \overline{x_3})) \equiv \\
 & (\neg\neg \overline{x_1} \vee \neg(\neg\neg x_2 \vee \neg\neg \overline{x_3})) \equiv \\
 & (\neg\neg \overline{x_1} \vee (\neg\neg\neg x_2 \wedge \neg\neg\neg \overline{x_3}))
 \end{aligned}$$

2. закона за двойното отрицание $\neg\neg\phi \equiv \phi$, при което в нашия пример получаваме

$$(\overline{x_1} \vee (\neg x_2 \wedge \neg \overline{x_3}))$$

3. заменяне на всеки отрицателен литерал с негация $\neg\overline{x_i}$ със съответния положителен литерал x_i , а всеки положителен литерал с негация $\neg x_i$ със съответния отрицателен литерал $\overline{x_i}$; в нашия пример получаваме

$$(\overline{x_1} \vee (\overline{x_2} \wedge x_3)) \quad (15.6)$$

След като се освободим от негациите, правим следното.

- Ако искаме КНФ, прилагаме, докато можем, отляво надясно дистрибутивността на дизюнкцията спрямо конюнкцията

$$\begin{aligned}
 (\phi \vee (\psi \wedge \xi)) & \equiv ((\phi \vee \psi) \wedge (\phi \vee \xi)) \\
 ((\psi \wedge \xi) \vee \phi) & \equiv ((\psi \vee \phi) \wedge (\xi \vee \phi))
 \end{aligned}$$

В нашия пример, прилагаме това върху формулата от (15.6) и получаваме

$$(\overline{x_1} \vee (\overline{x_2} \wedge x_3)) \equiv ((\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3)) \quad (15.7)$$

което наистина е КНФ. Незначителна разлика с изискванията на Определение 146 е наличието на много скоби и ползването на “ $\wedge$ ”; съгласно Определение 146 бихме написали “ $(\overline{x_1} \vee \overline{x_2})(\overline{x_1} \vee x_3)$ ”.

- Ако искаме ДНФ, прилагаме, докато можем, отляво надясно дистрибутивността на конюнкцията спрямо дизюнкцията

$$\begin{aligned}
 (\phi \wedge (\psi \vee \xi)) & \equiv ((\phi \wedge \psi) \vee (\phi \wedge \xi)) \\
 ((\psi \vee \xi) \wedge \phi) & \equiv ((\psi \wedge \phi) \vee (\xi \wedge \phi))
 \end{aligned}$$

В нашия пример, формулата от (15.6) е почти в ДНФ, само че с прекалено много скоби и използване на “ $\wedge$ ”; съгласно Определение 145 бихме написали “ $\overline{x_1} \vee \overline{x_2} x_3$ ”.

Няма да даваме доказателство на следната теорема. Читателят лесно можа да направи прецизно доказателство със структурна индукция, доказвайки, че трансформациите от Подсекция 15.4.3 генерират формули, еквивалентни на дадената.

Теорема 124: Всяка булева формула може да се препише в ДНФ или КНФ

За всяка булева формула има еквивалентна формула в ДНФ и еквивалентна формула в КНФ.

За съжаление, или за щастие, трансформирането на булева формула в КНФ или ДНФ съгласно изброените правила не е ефикасно. То е ефективно, което и Теорема 124 казва, но може да доведе до експоненциално нарастване на дължината на формулата.

15.5 SATISFIABILITY (SAT) и моделиращата сила на КНФ

Логиката е изразителна. Нейните формули са замислени с оглед на това да могат да моделират адекватно аспекти на физическата или математическата реалност. В Секция 15.5 ще видим как няколко задачи за разпознаване се свеждат с Кагр редукции до една фундаментална задача от логиката, наречена SATISFIABILITY или накратко SAT, в която се пита дали дадена КНФ е удовлетворима. Разбира се, дизюнктивните нормални форми също са изразителни и имат моделираща мощ, но е факт, че по-често се изразяваме чрез конюнкции. Теоремата на Cook (Лекция 15) в частност използва КНФ. Така че дефинираме SAT именно чрез КНФ, въпреки че можехме да я дефинираме чрез произволна булева формула – съгласно Определение 143, удовлетворимост е понятие, приложимо към произволна булева формула.

Изч. Задача 64: SATISFIABILITY (SAT)

екземпляр: КНФ ϕ .

въпрос: Дали ϕ е удовлетворима?

Следният факт се ползва в много при моделиране чрез КНФ.

Лема 56

Нека ϕ и ψ са следните КНФ

$$\phi = \bigwedge_{1 \leq i < j \leq n} (\bar{x}_i \vee \bar{x}_j) \quad (15.8)$$

$$\psi = \left(\bigvee_{i=1}^n x_i \right) \wedge \bigwedge_{1 \leq i < j \leq n} (\bar{x}_i \vee \bar{x}_j) \quad (15.9)$$

ϕ е удовлетворима само от валюациите с не повече от една единица. ψ е удовлетворима само от валюациите с точно една единица.

Доказателство: Разглеждаме валюации на $\{x_1, \dots, x_n\}$. Първо да разгледаме валюацията $t = 00 \dots 0$. Очевидно t удовлетворява ϕ , но не удовлетворява ψ , понеже $\text{TA}((\bigvee_{i=1}^n x_i), t) = 0$.

Сега да разгледаме произволна валюация t' с точно една единица. Това означава, че $t'(x_s) = 1$ за някое $s \in \{1, \dots, n\}$ и $t'(x_k) = 0$ за всяко $k \neq s$. Тогава $\text{TA}((\bigvee_{i=1}^n x_i), t') = 1$. Освен това, за всеки i, j , такива че $1 \leq i < j \leq n$, $\text{TA}((\bar{x}_i \vee \bar{x}_j), t'|_{\{x_i, x_j\}}) = 1$. Ерго, за всяка дизюнктивна клауза α във ϕ или ψ е вярно, че $\text{TA}(\alpha, t'|_{\text{Var}(\alpha)}) = 1$.

Да разгледаме произволна валюация t'' с повече от една единица. Нека $t''(x_k) = t''(x_s) = 1$ за някои k и s , такива че $1 \leq k < s \leq n$. Но тогава $\text{TA}((\bar{x}_k \vee \bar{x}_s), t''|_{\{x_k, x_s\}}) = 0$, а клаузата $(\bar{x}_k \vee \bar{x}_s)$ се среща и във ϕ , и в ψ . $\square$

15.5.1 Моделиране на sudoku позиция чрез КНФ.

Sudoku е популярна занимателна игра (пъзел), в която в квадратна таблица 9×9 някои

квадратчета са празни, а в други има число от $\{1, \dots, 9\}$. На английски всички тези предварително разположени числа са *the clues*. Каретата са 9 на брой, всяко с размери 3×3 , и представляват разбиване на цялата 9×9 мрежа. Целта е празните полета да бъдат попълнени с числа от $\{1, \dots, 9\}$ по такъв начин, че във всеки хоризонтал всяко число от $\{1, \dots, 9\}$ да се среща точно веднъж, всеки вертикал всяко число от $\{1, \dots, 9\}$ да се среща точно веднъж и всяко каре всяко число от $\{1, \dots, 9\}$ да се среща точно веднъж.

Ето пример за Судоку: началната позиция е вляво, а решение за нея има вдясно. Каретата са очертани с удебелени стени. $\text{\LaTeX}$ кодът е на Roberto Bonvallet и е изложен за свободно ползване на texample.net.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Начална позиция

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Решение

По правило, судоку позициите, които се предлагат за решаване, имат уникални решения, но може да мислим, че просто е дадена позиция. Тя дори може да няма решение[†].

Ето няколко любопитни факта за судоку.

- Всички валидни позиции са около 6.67×10^{21} .
- За да има уникално решение, минималният брой предварително фиксирани числа (*clues*) е 17.
- Има очевидно обобщение, наречено n -судоку. То има мрежа с размери $n^2 \times n^2$, каретата са с размери $n \times n$, а числата за запълване са от $\{1, \dots, n\}$. Иска се след запълването, във всеки хоризонтал, вертикал и каре, всяко число от $\{1, \dots, n\}$ да се среща точно веднъж. Очевидно обикновеното судоку е 3-судоку. n -судоку, във вариант за разпознаване, е NP-пълна задача [68].

Да дефинираме судоку като изчислителна задача, и то във вариант за разпознаване. Става дума за класическото 3-судоку.

Изч. Задача 65: Судоку

екземпляр: Судоку позиция p .

въпрос: Има ли начин да се запълнят празните квадратчета в p според правилата на судоку?

Очевидно задачата може да се формулира като задача за търсене, примерно да се намери валидно запълване, да се изчисли броят на различните валидни запълвания или дори да се намерят всички валидни запълвания. Може да се формулира и като оптимизационна

[†]В другата крайност, ако в началната позиция няма сложено нито едно число, то всяко валидно запълване с 81 числа от $\{1, \dots, 9\}$ представлява решение.

задача – ако се въведе по смислен начин “тегло на запълване”, може да се търси запълване с минимално или максимално тегло. Но за целите на това изложение разглеждаме задачата във вариант за разпознаване.

Има лек потенциален проблем с дефиницията на СУДОКУ. Щом се има предвид 3-судоку, множеството от екземпляри е крайно. А, както бе казано още в Лекция 1, “разглеждаме само задачи с безкрайно много екземпляри”. Тук ще направим изключение. Изложението на редукцията, което използва 9 вместо n^2 , е по-лесно за възприемане.

Редукция 5: СУДОКУ $\leq_p$ SAT

Конструкция: Даден е екземпляр на СУДОКУ – това е 9×9 sudoku позиция, каквато описахме горе. Трябва да се конструира екземпляр на SAT, тоест КНФ, която е удовлетворима тстк позицията има решение. Иначе казано, иска се да моделираме дадена sudoku позиция чрез КНФ.

Решението е на две фази. В първата фаза само моделираме правилата на sudoku без оглед на конкретиката (фиксираните числа). Във втората фаза моделираме разполагането на фиксираните поначало числа.

Когато моделираме чрез булева формула, добре е първо да решим кои са булевите променливи и какво точно моделира всяка от тях чрез своите положителен и отрицателен литерал. Тъй като искаме не каква да е булева формула, а КНФ, после съставяме клаузите и сме готови: решението е просто конкатенацията на тези клаузи.

Най-общо казано, решеното sudoku се състои от числа от $\{1, \dots, 9\}$, разположени в 81 позиции съгласно някакви ограничения. КНФ е идеална за моделиране на ограничения, които са изразени чрез конюнкции. Судоку изисква, примерно, в клетка $(1, 1)$ да има число и в клетка $(1, 2)$ да има число и така нататък; тоест, ограниченията, дефиниращи задачата, са свързани с конюнкции.

Как обаче да моделираме естествени числа чрез булеви променливи, които има само две стойности? Налага се числата да се кодират: на едно число ще съответстват няколко булеви променливи. Тъй като числата са от $\{1, \dots, 9\}$ и 9 е малко, можем да си позволим да кодираме унарно. Това ще рече следното.

- За числото x в клетка $(1, 1)$ има девет булеви променливи. Ние ще направим клаузите така, че цялата КНФ да е удовлетворима от валуация t тстк t изобразява точно една от тези девет променливи в 1. Ако това е първата от тях, то $x = 1$. Ако това е втората от тях, то $x = 2$. И така нататък. Въпросните променливи са само за клетка $(1, 1)$.
- За числото y в клетка $(1, 2)$ също да има девет булеви променливи. И за тях е вярно, че цялата КНФ е удовлетворима от валуация t тстк t изобразява точно една от тях в 1. Въпросните променливи са само за клетка $(1, 2)$.
- И така нататък за останалите клетки на sudokuто.

И така, въвеждаме общо $9^3 = 729$ променливи, които наричаме $p_{i,j,k}$, където $1 \leq i, j, k \leq 9$. Смисълът им е следният: $p_{i,j,k} = 1$ тстк клетка (i, j) съдържа k .

Ограничителните условия (на английски терминът е *constraints*) са от правилата на играта:

1. във всеки ред, всяко число от $\{1, \dots, 9\}$ се среща точно веднъж,
2. във всяка колона, всяко число от $\{1, \dots, 9\}$ се среща точно веднъж,

3. във всяко каре, всяко число от $\{1, \dots, 9\}$ се среща точно веднъж.

Но “точно веднъж” не се моделира директно с дизюнкция, а нашите клаузи трябва да са дизюнктивни. Дизюнкцията е “поне едно”, а не “точно едно”. Поради това променяме условията така:

- ❶ във всеки ред, всяко число от $\{1, \dots, 9\}$ се среща поне веднъж,
- ❷ във всяка колона, всяко число от $\{1, \dots, 9\}$ се среща поне веднъж,
- ❸ във всяко каре, всяко число от $\{1, \dots, 9\}$ се среща поне веднъж.

Това звучи като друга игра, но след малко ще въведем още едно ограничение, което ще възстанови “судоковщината”. Сега да видим как ❶, ❷ и ❸ може да бъдат “преведени” на езика на дизюнктивните клаузи.

Да започнем с ❶, с първия ред. В променливите $p_{i,j,k}$, индексът i отговаря на реда. Щом става дума за първия ред, то $i = 1$. Мислим за променливите $p_{1,j,k}$, по всички j и k . Индексът k отговаря на числото. Щом **всяко** число от $\{1, \dots, 9\}$ се среща, ще използваме конюнкция по индекса k , за да моделираме този факт. Това, че се среща поне веднъж, е същото като да **съществува** колона, в която се среща; за целта използваме дизюнкция по индекса j . Следната КНФ ϕ_1 моделира ❶ за първия ред на sudokuто:

$$\begin{aligned} \phi_1 = & (p_{1,1,1} \vee p_{1,2,1} \vee p_{1,3,1} \vee \dots \vee p_{1,9,1}) \wedge \\ & (p_{1,1,2} \vee p_{1,2,2} \vee p_{1,3,2} \vee \dots \vee p_{1,9,2}) \wedge \\ & (p_{1,1,3} \vee p_{1,2,3} \vee p_{1,3,3} \vee \dots \vee p_{1,9,3}) \wedge \\ & \dots \wedge \\ & (p_{1,1,9} \vee p_{1,2,9} \vee p_{1,3,9} \vee \dots \vee p_{1,9,9}) \end{aligned} \quad (15.10)$$

Да разгледаме само първия ред $(p_{1,1,1} \vee p_{1,2,1} \vee p_{1,3,1} \vee \dots \vee p_{1,9,1})$ в записа на ϕ_1 . Нека t е произволна валуация, която удовлетворява крайната КНФ. За поне едно $j \in \{1, \dots, 9\}$ е изпълнено $t(p_{1,j,1}) = 1$. Но това се интерпретира така: “числото 1 се среща на ред 1”. Аналогично, вторият ред в записа на ϕ_1 гарантира, че 2 се среща на ред 1, третият ред в записа на ϕ_1 гарантира, че 3 се среща на ред 1, ..., деветият ред в записа на ϕ_1 гарантира, че 9 се среща на ред 1.

За втория ред на sudokuто аналогично конструираме

$$\begin{aligned} \phi_2 = & (p_{2,1,1} \vee p_{2,2,1} \vee p_{2,3,1} \vee \dots \vee p_{2,9,1}) \wedge \\ & (p_{2,1,2} \vee p_{2,2,2} \vee p_{2,3,2} \vee \dots \vee p_{2,9,2}) \wedge \\ & (p_{2,1,3} \vee p_{2,2,3} \vee p_{2,3,3} \vee \dots \vee p_{2,9,3}) \wedge \\ & \dots \wedge \\ & (p_{2,1,9} \vee p_{2,2,9} \vee p_{2,3,9} \vee \dots \vee p_{2,9,9}) \end{aligned}$$

И така нататък за останалите редове. След това конструираме $\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_9$. Накратко, заради ❶ конструираме формулата

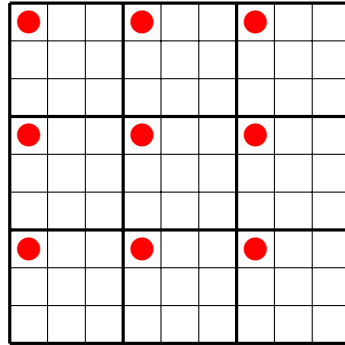
$$\phi = \bigwedge_{i=1}^9 \bigwedge_{k=1}^9 \left(\bigvee_{j=1}^9 p_{i,j,k} \right) \quad (15.11)$$

Имайте предвид, че (15.11) е компактен запис на доста голяма формула: тя има 81 клаузи, всяка с по 9 литерала. Да се опитваме да я запишем подробно е безсмислено.

Напълно аналогично, заради ❷ конструираме формулата

$$\psi = \bigwedge_{j=1}^9 \bigwedge_{k=1}^9 \left(\bigvee_{i=1}^9 p_{i,j,k} \right) \quad (15.12)$$

Да разгледаме ❸, което касае каретата. Можем да подходим така. Да кажем, че всяко каре има *лидер*, който е клетката горе вляво в карето. Лидерите са клетките (1, 1), (1, 4), (1, 7), (4, 1), (4, 4), (4, 7), (7, 1), (7, 4) и (7, 7). Ето илюстрацията на лидерите с червени точки.



Деветта лидери на карета

Всяка позиция в дадено каре се получава от позицията на съответния лидер с отместване или 0 или 1 или 2 по хоризонтал и отместване или 0 или 1 или 2 по вертикал. Това ни дава право да твърдим, че формулата, съответна на ❸, е

$$\xi = \bigwedge_{a \in \{1,4,7\}} \bigwedge_{b \in \{1,4,7\}} \bigwedge_{k=1}^9 \left(\bigvee_{x \in \{0,1,2\}} \bigvee_{y \in \{0,1,2\}} p_{a+x, b+y, k} \right) \quad (15.13)$$

Заклучаваме, че $\phi \wedge \psi \wedge \xi$ (от (15.11), (15.12) и (15.13)) е формула, която е удовлетворима тстк във всеки ред, всяка колона и всяко каре, всяко число се среща **поне** веднъж.

Но това не е достатъчно. Трябва да конструираме формула, която е удовлетворима тстк във всеки ред, всяка колона и всяко каре, всяко число се среща **точно** веднъж. Засега не сме постигнали това. Да видим защо.

Да разгледаме само ϕ_1 от (15.10). Както видяхме, нейните клаузи в съвкупността си гарантират, че всяко число се среща поне веднъж на ред 1. А числата са девет и позициите са девет. Как е възможно всяко число да се среща поне веднъж и да не е вярно, че всяко число се среща точно веднъж? Това означава поне едно число да се среща повече от веднъж. Но тогава има повече от девет числа на първия ред. Как е възможно това? За да стане ясно, че това е напълно възможно под ограниченията на $\phi \wedge \psi \wedge \xi$, трябва да **не мислим за попълване на sudoku позиция на ръка**.

Ако решаваме sudoku на ръка е абсурд да сложим повече от девет числа на първия ред, понеже това означава да сложим повече от едно число в някоя клетка. Никой човек няма да направи това, ако играе сериозно, защото клетка с няколко числа се набива на очи веднага. Ключовият факт е, че **клаузите не виждат мрежата на sudokuто**. Примерно, може $\phi \wedge \psi \wedge \xi$ да бъде удовлетворена от валуация t , такава че $t(p_{1,1,2}) = 1$ и $t(p_{1,1,3}) = 1$, което се интерпретира като “в клетка (1, 1) се намира и числото 2, и числото 3”[†]. Тази интерпретация я

[†]Нещо повече. ϕ се удовлетворява от валуация, в която всички $p_{1,j,k}$ са единици. Това се интерпретира като “всяко число се намира във всяка клетка на ред 1”.

правим ние обаче! Клаузите не се интересуват от нея. Ако ние като конструктори на клаузи допуснем възможността окончателната КНФ да е удовлетворима от валюация t , в която $t(p_{1,1,2}) = t(p_{1,1,3}) = 1$, то допускаме “решение”, според което в клетка $(1, 1)$ има и двойка, и тройка. Отговорността това да не се случи е наша.

Твърдим, че е достатъчно не допускаме в клетка да има повече от едно число. Тоест, че е достатъчно към $\phi \wedge \psi \wedge \xi$ да добавим с ‘ $\wedge$ ’ клаузи, предотвратяващи наличието на повече от едно число в клетка, за да имаме КНФ, удовлетворима тстк във всеки ред, колона и каре, всяко число се среща **точно** веднъж. Това твърдение ще приемем за очевидно.

За всеки $i, j \in \{1, \dots, 9\}$, конструираме следната КНФ

$$\gamma_{i,j} = \bigwedge_{1 \leq k < \ell \leq 9} (\overline{p_{i,j,k}} \vee \overline{p_{i,j,\ell}}) \quad (15.14)$$

Съгласно Лема 56, $\gamma_{i,j}$ е удовлетворима само от валюации с не повече от една единица. Това се интерпретира като “в клетка (i, j) има най-много едно число”.

Конструираме КНФ, свързвайки всички $\gamma_{i,j}$ с ‘ $\wedge$ ’. А именно,

$$\gamma = \bigwedge_{i=1}^9 \bigwedge_{j=1}^9 \bigwedge_{1 \leq k < \ell \leq 9} (\overline{p_{i,j,k}} \vee \overline{p_{i,j,\ell}}) \quad (15.15)$$

γ гарантира, че във всяка клетка се среща най-много едно число.

И така, ползвайки формулите от (15.11), (15.12), (15.13) и (15.15), построяваме КНФ

$$\phi \wedge \psi \wedge \xi \wedge \gamma \quad (15.16)$$

в първата фаза от решението. Тя е удовлетворима от и само от валюациите, според които във всеки ред, във всяка колона и във всяко каре, всяко число от $\{1, \dots, 9\}$ се среща точно веднъж. Това е КНФ със 729 променливи и точно $81 + 81 + 81 + 9 \times 9 \times 36 = 2925$ дизюнктивни клаузи. Тази КНФ точно описва правилата на sudoku: за всяка валюация t на променливите, $t \models \phi \wedge \psi \wedge \xi \wedge \gamma$ тстк t описва валидно записване на числа в клетките на sudoku.

Но $\phi \wedge \psi \wedge \xi \wedge \gamma$ моделира само правилата на sudoku. Тя не съдържа нищо, което отразява конкретиката на позицията, която е дадена. Прочее, тя има толкова удовлетворяващи валюации, колкото валидни позиции има играта по принцип; както вече бе отбелязано, те са около 6.67×10^{21} . Да се моделира конкретиката на дадената позиция е предмет на втората фаза от доказателството. За всяко число k , записано в клетка (i, j) , добавяме клауза с един литерал $(p_{i,j,k})$. Тези еднолитерални клаузи биват свързани с ‘ $\wedge$ ’ в окончателната КНФ. Очевидно, за да е вярно, че валюация t удовлетворява окончателната КНФ, за всяка клауза от вида (λ) , където λ е един литерал, трябва да е вярно, че $TA(\lambda, t|_{\text{Var}(\lambda)}) = 1$. С което доказателството за коректността на Редукция 5 приключи.

За да бъде редукцията Кагр редукция, трябва алгоритъмът, който я реализира и построява клаузите на КНФ, да работи в полиномиално време. Тук се натъкваме на проблема с крайния брой екземпляри на задачата (понеже разглеждаме 3-sudoku, а не n -sudoku). За крайно множество екземпляри, редукцията винаги има само константна сложност; дори редукцията да върши “алгоритмични безобразия” като да генерира (безсмислено) всички пермутации на променливите и после да ги игнорира, щом променливите са краен брой, сложността е константна.

Но нашата конструкция може да се обобщи за n -sudoku без проблеми, а там вече екземплярите не са с константна големина! Лесно се вижда, че ако екземплярът на n -sudoku

е кодиран унарно в смисъл, че за всяка клетка от мрежата (мрежата е $n^2 \times n^2$) се ползва поне единица памет, конструкцията е полиномиална. И това е край на доказателството на Редукция 5. $\square$

Като пример да разгледаме sudoku позицията на стр. 900. На позиция (1, 2) има 2, така че добавяме клауза $p_{1,2,2}$. На позиция (1, 4) има 5, така че добавяме клауза $p_{1,4,5}$. И така нататък, за всички фиксирани поначало числа. Решението за тази начална позиция е следната КНФ:

$$\phi \wedge \psi \wedge \xi \wedge \gamma \wedge (p_{1,2,2}) \wedge (p_{1,4,5}) \wedge \dots \wedge (p_{9,8,6})$$

Знаем, че позицията има решение (също показано на стр. 900, но вдясно и със синьо), откъдето следва, че формулата е удовлетворима.

Ако сега мислим за СУДОКУ като за задача за търсене, имаме право да кажем, че всяка удовлетворяваща валюация е тази КНФ задава едно решение. Най-вероятно решението е уникално, поради което има точно една удовлетворяваща валюация t , а именно

$$\begin{aligned} t(p_{1,1,1}) = 0, t(p_{1,1,2}) = 0, t(p_{1,1,3}) = 0, t(p_{1,1,4}) = 1, t(p_{1,1,5}) = 0, t(p_{1,1,6}) = 0, t(p_{1,1,7}) = 0, t(p_{1,1,8}) = 0, t(p_{1,1,9}) = 0, \\ t(p_{1,2,1}) = 0, t(p_{1,2,2}) = 1, t(p_{1,2,3}) = 0, t(p_{1,2,4}) = 0, t(p_{1,2,5}) = 0, t(p_{1,2,6}) = 0, t(p_{1,2,7}) = 0, t(p_{1,2,8}) = 0, t(p_{1,2,9}) = 0, \\ t(p_{1,3,1}) = 0, t(p_{1,3,2}) = 0, t(p_{1,3,3}) = 0, t(p_{1,3,4}) = 0, t(p_{1,3,5}) = 0, t(p_{1,3,6}) = 1, t(p_{1,3,7}) = 0, t(p_{1,3,8}) = 0, t(p_{1,3,9}) = 0, \\ \dots \\ t(p_{9,8,1}) = 0, t(p_{9,8,2}) = 0, t(p_{9,8,3}) = 0, t(p_{9,8,4}) = 0, t(p_{9,8,5}) = 0, t(p_{9,8,6}) = 1, t(p_{9,8,7}) = 0, t(p_{9,8,8}) = 0, t(p_{9,8,9}) = 0, \\ t(p_{9,9,1}) = 0, t(p_{9,9,2}) = 1, t(p_{9,9,3}) = 0, t(p_{9,9,4}) = 0, t(p_{9,9,5}) = 0, t(p_{9,9,6}) = 0, t(p_{9,9,7}) = 0, t(p_{9,9,8}) = 0, t(p_{9,9,9}) = 0 \end{aligned}$$

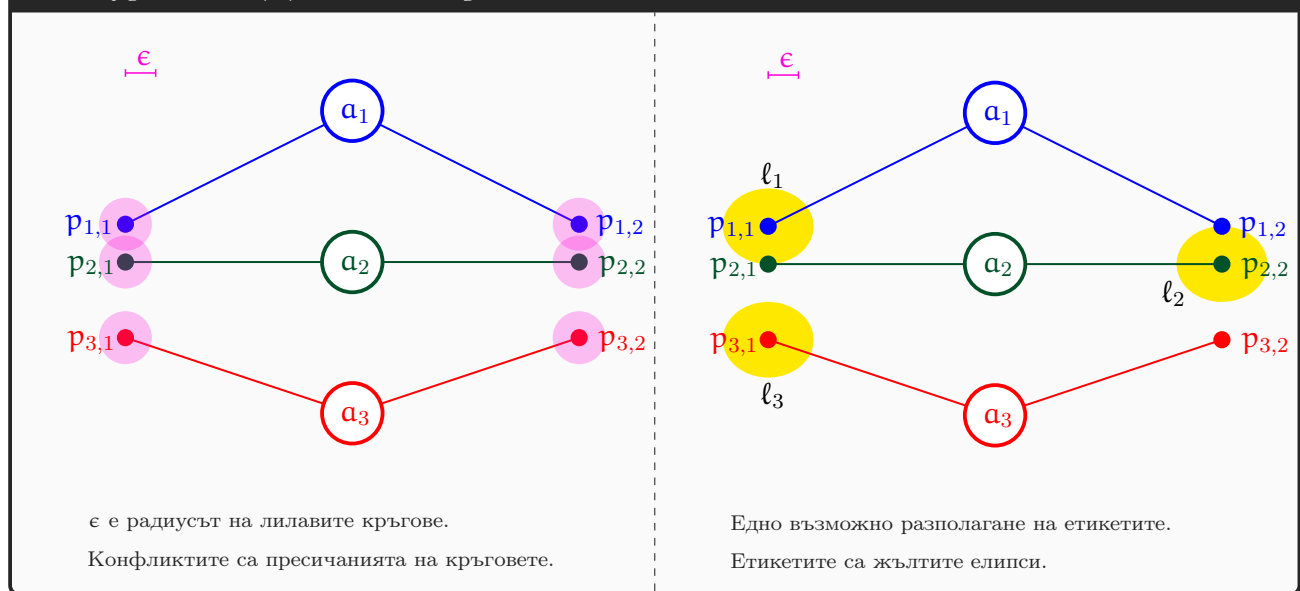
Единиците са написани в синьо, за да се отличават. Те са доста малко: само 81 на брой (за всяка клетка точно едно от съответните булеви променливи е единица). Останалите $729 - 81 = 648$ булеви променливи получават стойност нула от валюацията.

15.5.2 Моделиране на ограничено поставяне на етикети чрез КНФ.

Става дума за изчислителна задача, която на английски е известна като RESTRICTED LABEL PLACEMENT. Дадени са някакви обекти $a_1, \dots, a_n$, за които можем да мислим като за точки в равнината. Всеки обект a_i има етикет, наречен ℓ_i , за който са дадени краен брой, два по две различни, позиции $p_{i,1}, \dots, p_{i,t_i}$, на точно една от които той трябва да бъде сложен. Дадено е някакво минимално разстояние ϵ , такова че всеки два етикета трябва да се намират на поне ϵ един от друг. С други думи, някои двойки позиции на различни етикети са несъвместими в смисъл, че не е разрешено и двата етикета да се сложат на тях – биха били прекалено близо. Потенциалните конфликти са само между позициите – позиция и обект не може да са в конфликт, нито два обекта може да са в конфликт. Задачата е да се разположат всички етикети, но безконфликтно.

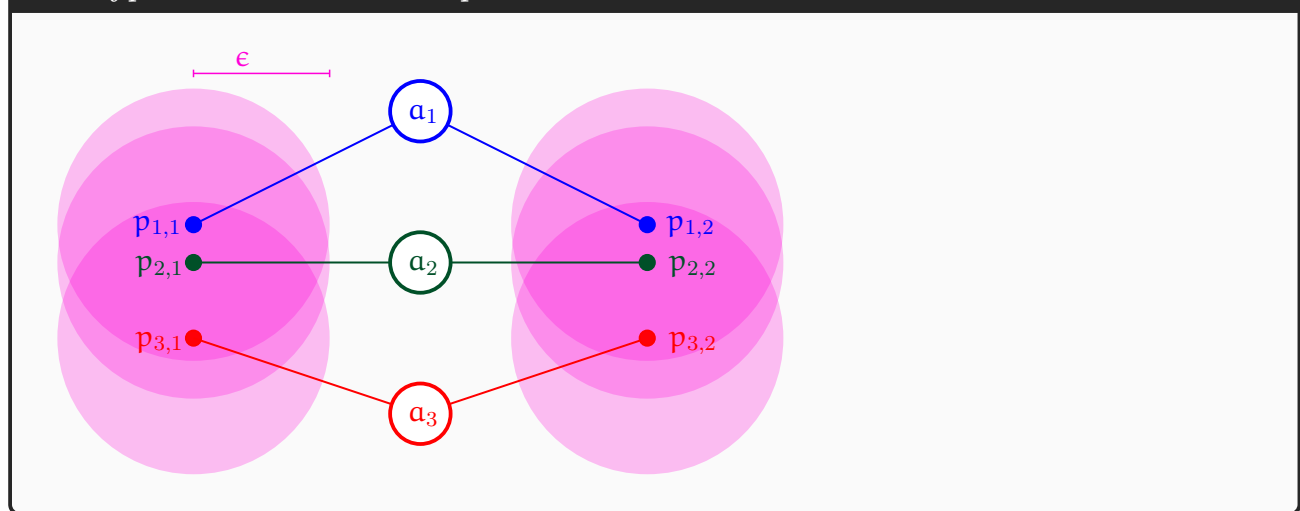
Фигура 15.1 илюстрира Да-екземпляр на RESTRICTED LABEL PLACEMENT. Вляво е показан екземплярът, а вдясно има решение. Обектите са a_1, a_2 и a_3 . Всеки обект a_i има точно две позиции $p_{i,1}$ и $p_{i,2}$, на точно една от които трябва да е неговият етикет ℓ_i . Съответствието между обекти и позиции е акцентирано с цветове. Чрез пресичанията на полупрозрачните кръгове с радиус ϵ са илюстрирани несъвместимостите между позиции: не може едновременно ℓ_1 да е на $p_{1,1}$ и ℓ_2 да е на $p_{2,1}$, и също така не може едновременно ℓ_1 да е на $p_{1,2}$ и ℓ_2 да е на $p_{2,2}$. Други несъвместимости няма. Едно възможно решение е ℓ_1 да е на $p_{1,1}$, ℓ_2 да е на $p_{2,2}$ и ℓ_3 да е на $p_{1,3}$.

Фигура 15.1 : Да-екземпляр на RESTRICTED LABEL PLACEMENT.



Фигура 15.2 илюстрира НЕ-екземпляр на RESTRICTED LABEL PLACEMENT. Тук позициите са същите, но ϵ е значително по-голямо, поради което всеки две от $p_{1,1}$, $p_{2,1}$ и $p_{3,1}$ са несъвместими, а също така и всеки две от $p_{2,1}$, $p_{2,2}$ и $p_{2,3}$ са несъвместими. Ерго, можем да сложим най-много два етикета, един на някоя от $p_{1,1}$, $p_{2,1}$ и $p_{3,1}$ и още един на някоя от $p_{2,1}$, $p_{2,2}$ и $p_{2,3}$, но няма как да сложим и трите етикета.

Фигура 15.2 : НЕ-екземпляр на RESTRICTED LABEL PLACEMENT.



Предложеното описание на задачата е доста многословно и има геометричен аспект. То служи за получаване на интуиция, но сега ще разгледаме формално описание, което не говори нито за обектите $a_1, \dots, a_n$, нито за етикетите $\ell_1, \dots, \ell_n$, нито за ϵ . “Оправданието” да въведем пестеливото формално описание е, че допускаме, че в някаква предварителна обработка сме изчислили кои двойки позиции $\{p_{i,a}, p_{j,b}\}$ са несъвместими, където $1 \leq i < j \leq n$, $a \in \{1, \dots, t_i\}$ и $b \in \{1, \dots, t_j\}$; ерго, в същинското решение имаме само позициите за всеки етикет, но без координати, а само като абстрактни елементи, и двойките от несъвместими позиции. Защо тези двойки позиции са несъвместими, за същинското решение няма значение.

Забележете, че има смисъл да допуснем, че $t_i \geq 2$ за всяко $i \in \{1, \dots, n\}$. Обратното би значело поне един обект a_i да има само една позиция $p_{i,1}$ за етикета си ℓ_i , което форсира

ℓ_i да отиде на $p_{i,1}$. В предварителната обработка можем да елиминираме такива обекти, евентуално ограничавайки позициите на други обекти.

Изч. Задача 66: RESTRICTED LABEL PLACEMENT

екземпляр: Непразни множества $P_1, \dots, P_n$ без общи елементи, като за $1 \leq i \leq n$: $P_i = \{p_{i,1}, \dots, p_{i,t_i}\}$ и $t_i \geq 2$. Множество $Q \subseteq \{\{p_{i,s}, p_{j,q}\} \mid 1 \leq i < j \leq n, 1 \leq s \leq t_i, 1 \leq q \leq t_j\}$.

решение: Множество S , съдържащо по точно един елемент от всяко P_i , като $\forall a, b \in S, a \neq b \rightarrow \{a, b\} \notin Q$.

Очевидно P_i съдържа позициите, на които може да бъде слаган етикетът ℓ_i , елементите на Q са конфликтните двойки позиции на различните етикети, а S задава някакво безконфликтно разполагане на етикетите, ако такова съществува. Ако такова не съществува, екземплярът няма решение.

Редукция 6: RESTRICTED LABEL PLACEMENT $\leq_p$ SAT

Конструкция: Ето как моделираме екземпляр на RESTRICTED LABEL PLACEMENT с КНФ. За всяко $i \in \{1, \dots, n\}$ въвеждаме t_i булеви променливи $b_{i,1}, \dots, b_{i,t_i}$, които отговарят на позициите $p_{i,1}, \dots, p_{i,t_i}$, асоциирани с обект a_i . Смисълът е очевидният: $b_{i,j}$ е 1 тстк етикет ℓ_i отива на позиция $p_{i,j}$.

А ето как конструираме клаузите. За всяка ненаредена двойка несъвместими позиции $\{p_{i,s}, p_{j,q}\}$ конструираме клауза $(\overline{b_{i,s}} \vee \overline{b_{j,q}})$. Смисълът от това трябва да е очевиден: $\overline{b_{i,s}} \vee \overline{b_{j,q}}$ се интерпретира като “не се слагат едновременно етикетът ℓ_i на позиция $p_{i,s}$ и етикетът ℓ_j на позиция $p_{j,q}$ ”, а по закона на De Morgan, $\overline{b_{i,s}} \vee \overline{b_{j,q}} \equiv \overline{b_{i,s} b_{j,q}}$. Използваме $(\overline{b_{i,s}} \vee \overline{b_{j,q}})$, понеже е дизюнктивна клауза. И, разбира се, свързваме всички такива клаузи с ‘ $\wedge$ ’. Като цяло, конструираме КНФ ϕ :

$$\phi = \bigwedge_{\{p_{i,s}, p_{j,q}\} \in Q} (\overline{b_{i,s}} \vee \overline{b_{j,q}}) \quad (15.17)$$

За екземплярите от Фигура 15.1 и Фигура 15.2, променливите са $b_{1,1}, b_{1,2}, b_{1,3}, b_{2,1}, b_{2,2}$ и $b_{2,3}$. Да разгледаме само екземпляр, илюстриран на Фигура 15.2. Съгласно казаното току-що, съгласно (15.17) конструираме следната КНФ

$$(\overline{b_{1,1}} \vee \overline{b_{2,1}}) \wedge (\overline{b_{1,1}} \vee \overline{b_{3,1}}) \wedge (\overline{b_{2,1}} \vee \overline{b_{3,1}}) \wedge (\overline{b_{1,2}} \vee \overline{b_{2,2}}) \wedge (\overline{b_{1,2}} \vee \overline{b_{3,2}}) \wedge (\overline{b_{2,2}} \vee \overline{b_{3,2}})$$

И веднага виждаме проблем: тази КНФ е удовлетворима, например от валуация, даваща нули на всички променливи, докато екземплярът, илюстриран на Фигура 15.2, е НЕ-екземпляр.

Работата е там, че тази КНФ не е достатъчна. Тя не е сбъркана; наистина тя предотвратява слагането на два етикета в две несъвместими позиции. Но тя е непълна с това, че не налага слагането на етикети изобщо. Очевидно валуацията $t(b_{i,j}) = 0$ за $i = 1, 2$ и $j = 1, 2, 3$ се интерпретира като “не се слагат никакви етикети”. Ако не се слагат етикети, няма и конфликти, но задачата иска да се слагат етикети.

И така, към ϕ от (15.17) трябва да добавим клаузи, налагащи слагането на етикети, по точно един етикет за всяко P_i , $1 \leq i \leq n$. Съгласно Лема 56, за всяко $i \in \{1, \dots, n\}$, следната КНФ

ψ_i е удовлетворима само от валуации с точно една единица:

$$\psi_i = \left(\bigvee_{j=1}^{t_i} p_{i,j} \right) \wedge \bigwedge_{1 \leq k < s \leq n} (\overline{p_{i,k}} \vee \overline{p_{i,s}}) \quad (15.18)$$

Конструираме КНФ ψ от всички тези формули

$$\psi = \bigwedge_{1 \leq i \leq n} \psi_i \quad (15.19)$$

Това е и крайт на конструкцията. В крайна сметка, конструираният екземпляр на SAT е $\phi \wedge \psi$, където ϕ е от (15.17), а ψ е от (15.19). Коректността показахме, а това, че конструкцията може да се извърши в полиномиално време, е очевидно.

За екземпляра на Фигура 15.2 конструираме следния екземпляр на SAT:

$$\underbrace{(\overline{b_{1,1}} \vee \overline{b_{2,1}}) \wedge (\overline{b_{1,1}} \vee \overline{b_{3,1}}) \wedge (\overline{b_{2,1}} \vee \overline{b_{3,1}}) \wedge (\overline{b_{1,2}} \vee \overline{b_{2,2}}) \wedge (\overline{b_{1,2}} \vee \overline{b_{3,2}}) \wedge (\overline{b_{2,2}} \vee \overline{b_{3,2}})}_{\phi} \wedge$$

$$\underbrace{(b_{1,1} \vee b_{1,2}) \wedge (\overline{b_{1,1}} \vee \overline{b_{1,2}}) \wedge (b_{2,1} \vee b_{2,2}) \wedge (\overline{b_{2,1}} \vee \overline{b_{2,2}}) \wedge (b_{3,1} \vee b_{3,2}) \wedge (\overline{b_{3,1}} \vee \overline{b_{3,2}})}_{\psi}$$

Нека читателят се убеди, че тя е неудовлетворима. □

15.5.3 Моделиране на наличие на Хамилтонов път чрез КНФ.

Ще приключим примерите с една лесна класическа редукция към SAT. Вече сме натрупали достатъчно интуиция и изложението ще е кратко.

Редукция 7: HAMILTONIAN PATH $\leq_p$ SAT

Конструкция: По даден екземпляр-граф $G = (V, E)$ конструираме КНФ ϕ , екземпляр на SAT, такава че ϕ е удовлетворима тстк G има Хамилтонов път. Да кажем, че $V = \{v_1, \dots, v_n\}$.

Булевите променливи са $b_{i,j}$ за $1 \leq i, j \leq n$. Смисълът е следният: $b_{i,j} = 1$ тстк v_j е i -ият поред връх в Хамилтонов път в G .

Сега да видим клаузите. Първо, имаме клаузи, гарантиращи, че всеки връх се появява поне веднъж в Хамилтоновия път. За $1 \leq j \leq n$, конструираме клауза

$$\phi_j = (b_{1,j} \vee b_{2,j} \vee \dots \vee b_{n,j})$$

Но v_j не може да се появява повече от веднъж в пътя, за $1 \leq j \leq n$. Съгласно Лема 56, това се гарантира чрез формулата

$$\psi_j = \bigwedge_{1 \leq i < k \leq n} (\overline{b_{i,j}} \vee \overline{b_{k,j}})$$

като такава формула се прави за всяко $j \in \{1, \dots, n\}$.

Второ, имаме клаузи, гарантиращи, че всяка позиция на пътя съдържа поне един връх. За $1 \leq i \leq n$, конструираме клауза

$$\xi_j = (b_{i,1} \vee b_{i,2} \vee \dots \vee b_{i,n})$$

Но позиция i на пътя не може да съдържа повече от един връх, за $1 \leq i \leq n$. Съгласно Лема 56, това се гарантира чрез формулата

$$\tau_j = \bigwedge_{1 \leq j < k \leq n} (\overline{b_{i,j}} \vee \overline{b_{i,k}})$$

като такава формула се прави за всяко $j \in \{1, \dots, n\}$.

Конструираме формулата

$$\alpha = \bigwedge_{j=1}^k \phi_j \wedge \bigwedge_{j=1}^k \psi_j \wedge \bigwedge_{j=1}^k \xi_j \wedge \bigwedge_{j=1}^k \tau_j$$

която моделира HAMILTONIAN PATH, но не е крайният отговор, защото не моделира G по никакъв начин. Тя е очевидно удовлетворима. Нещо повече, нейните удовлетворяващи валюации са $n!$. Според α , всяка пермутация на върховете е Хамилтонов път. α третира графа като пълен граф и “смята”, че между всеки два върха има ребро.

Сега ще моделираме и конкретиката на G . Трябва по някакъв начин да отразим “липсващите ребра”, понеже пътят не може да мине през тях. За всяка ненаредена двойка различни върхове $\{v_i, v_j\}$, които не са съседни в G , правим формула, гарантираща, че v_i и v_j не са съседни в Хамилтоновия път. Конкатенираме тези формули с ‘ $\wedge$ ’ и получаваме:

$$\beta = \bigwedge_{\substack{1 \leq i < j \leq n \\ (v_i, v_j) \notin E}} \bigwedge_{k=1}^{n-1} (\overline{b_{k,i}} \vee \overline{b_{k+1,j}}) \quad (15.20)$$

Окончателната формула е $\alpha \wedge \beta$.

Доказателството за коректност е лесно. В едната посока, ако G има Хамилтонов път, то има валюация на $\alpha \wedge \beta$, такава че всяка от дизюнктивните клаузи има поне един литерал, който е 1. В обратната посока, ако $\alpha \wedge \beta$ е удовлетворима, то има Хамилтонов път в G . Подробностите остават за читателя.

Това, че конструкцията може да се извърши във време, полиномиално в размера на G , е очевидно. $\square$

15.6 SAT е NP-пълна

Определение 139 не гарантира съществуването на NP-пълни задачи. Сега ще видим първите образци от този интересен вид.

Допълнение 80: BOUNDED HALTING PROBLEM е NP-пълна задача

В някакъв смисъл, каноничната NP-пълна задача е BOUNDED HALTING PROBLEM. Ето защо. Съществува прелюбопитна аналогия между алгоритмичната нерешимост в света на изчисленията без ограничение по време (теорията на изчислимостта), от една страна, и алгоритмичната неподатливост в света на изчисленията с ограничени ресурси (теорията на изчислителната сложност), от друга страна. Предвид тази аналогия,

- множеството от рекурсивните езици $\mathbf{R}$ от теорията на изчислимостта съответства на $\mathbf{P}$ от теорията на изчислителната сложност,

- а множеството от рекурсивно изброимите езици **RE** от теорията на изчислимостта съответства на **NP** от теорията на изчислителната сложност.

Каноничната алгоритмично нерешима задача е HALTING PROBLEM, която се явява пълна^a за **RE**. Съвсем естествено е за канонична неподатлива задача да вземем BOUNDED HALTING PROBLEM, която е очевидният аналог на HALTING PROBLEM в света на изчисленията с ограничени ресурси.

Изч. Задача 67: BOUNDED HALTING PROBLEM

екземпляр: НТМ M , вход x за M , естествено число k , написано унарно.

въпрос: Дали $M(x)$ спира за не повече от k стъпки?

Тук говорим за машина на Turing, а не за програма (сравнете със Задача 8). Искане се машината да е недетерминирана, защото, ако беше детерминирана, редуцията (която е детерминирана машина) можеше просто да я симулира за k стъпки и “да види” дали ще спре. А k е записано унарно, защото, ако беше записано ефикасно, примерно бинарно, редуцията, която включва изпълнението на k стъпки, щеше да е експоненциална, а не полиномиална.

Теорема 125: BOUNDED HALTING PROBLEM $\in$ NP-с

BOUNDED HALTING PROBLEM е **NP**-пълна.

Доказателство: Първо ще докажем, и то по два начина, че задачата е в **NP**. От Определение 133, ползващо Определение 130, знаем, че задача е в **NP** тстк нейните Да-екземпляри биват приемани от НТМ M' за полиномиално време. Забележете, че самата BOUNDED HALTING PROBLEM говори за НТМ, но това са две **различни** машини! Машината M' , която трябва да приема Да-екземплярите е различна от машината M , кодирана като стринг в някой Да-екземпляр.

Доказателство 1 за принадлежност към NP: Мислем за M' като за машина с размножаване. M' просто симулира работата на M върху Да-екземплярите на BOUNDED HALTING PROBLEM, приемайки веднага след момента, в който симулираната M приеме. А Да-екземплярите са точно тези, в които M приема x . Въпросната симулация работи в брой стъпки, не по-голям от полиномиален в k , понеже размерът на екземпляра на BOUNDED HALTING PROBLEM е по-голям от k . Това на свой ред е така, понеже k е записано унарно. И така, доказахме, че BOUNDED HALTING PROBLEM $\in$ **NP**.

Доказателство 2 за принадлежност към NP: Тъй като в тези лекционни записки при доказателствата за принадлежност към **NP** по правило ползваме другата дефиниция на НМТ, а именно на машина-верификатор (Подсекция 14.5.2), нека да видим и тук такава аргументация. Ще докажем, че BOUNDED HALTING PROBLEM $\in$ **NP**, мислейки за M' като за верификатор.

Всеки Да-екземпляр на BOUNDED HALTING PROBLEM е наредена тройка (M, x, k) , такава че M с вход x спира за най-много k стъпки. Да си представим M като недетерминирана машина с размножаване (Подсекция 14.5.1). Разглеждаме нейното дърво на изчислението T (Определение 129 при вход x). Коренът е стартовата конфигурация $(q_0, x, 1)$, останалите върхове са конфигурациите, които копията на машината генерират по време на работата, има поне едно листо-приемаща конфигурация $(q_f, *, *)$ и

това листо има дълбочина най-много k , понеже M приема x за най-много k стъпки. Оттук и височината на T е най-много k .

Конструираме M' по начин, много подобен на конструкцията от Теорема 118: разглеждаме пътя p в T от корена до листото-приемаща конфигурация и забелязваме, че сертификатът може да се ползва, за да “навигира” M' , така че да следва неотклонно p от корена до листото-приемаща конфигурация. Отново доказахме, че $\text{BOUNDED HALTING PROBLEM} \in \text{NP}$. Фигура 14.7 добре илюстрира идеята на това доказателство.

Сега да се убедим, че $\text{BOUNDED HALTING PROBLEM}$ е NP -трудна.

Редукция 8: $\forall P \in \text{NP} : P \leq_p \text{BOUNDED HALTING PROBLEM}$

Конструкция: Нека P е произволна задача от NP . Ще докажем, че за всяка задача $P \in \text{NP}$ е вярно, че $P \leq_p \text{BOUNDED HALTING PROBLEM}$. Какво знаем за P ? Само това, че съществува НТМ M'' и съществува полином $p(n)$, такива че M'' приема всеки ДА-екземпляр x на P във време, ограничено от $p(|x|)$. По даден екземпляр x на P , редукцията просто конструира $(M'', x, 1^{p(|x|)})$.

Коректността е очевидна: ако x е ДА-екземпляр на P и M'' е НТМ за P с ограничаващ сложността полином $p(n)$, по дефиниция $(M'', x, 1^{p(|x|)})$ е ДА-екземпляр на $\text{BOUNDED HALTING PROBLEM}$, и обратно.

Това, че редукцията иска само полиномиално време, също е очевидно: (кодирането на) M'' има константен размер, x е входът, а стрингът от единици $1^{p(|x|)}$ се генерира в полиномиално в размера на входа време. $\square$

Тази редукция е напълно тривиална за читател, интернализирал понятието полиномиална редукция и задачата $\text{BOUNDED HALTING PROBLEM}$. Обаче фактът, че $\text{BOUNDED HALTING PROBLEM} \in \text{NP-c}$ е напълно безполезен на практика за доказване за NP -пълнотата на други задачи. Задачата $\text{BOUNDED HALTING PROBLEM}$ просто няма моделираща сила. Редукцията в теоремата на Cook (Теорема 126) е ценна, понеже задача SAT има грамадна моделираща сила и нейната NP -пълнота, предвид Теорема 127, се явява ключ за доказване на NP -пълнотата на много други задачи.

Една странична забележка. Предвид ясните паралели между теорията на изчислимостта и теорията на изчислителната сложност, читателят може да се запита следното. В теорията на изчислимостта съществува средство, с помощта на което се доказва, че $\mathbf{R} \neq \mathbf{RE}$ и това средство е диагоналният метод. За повече информация, вижте [125, глава 3]. Защо не ползваме диагоналния метод, за да докажем, аналогичното твърдение $\mathbf{P} \neq \mathbf{NP}$. За съжаление, или за щастие, диагоналният метод не е приложим (при определени допускания), за да различим $\mathbf{P}$ от $\mathbf{NP}$ в теорията на изчислителната сложност [51] – факт, който е далеч отвъд обхвата на тези лекционни записки.

^aВсяка задача от $\mathbf{RE}$ се свежда до HALTING PROBLEM с редукция, която е изчислима и е many-one, но без изискването да е изчислима в полиномиално време.

Теорема 126: Теорема на Cook

$\text{SAT} \in \text{NP-c}$.

Доказателство: Първо ще докажем, че $\text{SAT} \in \text{NP}$. Нека е даден ДА-екземпляр на SAT ;

тоест, дадена е удовлетворима КНФ ϕ . НМТ-верификатор отгатва валуация t на ϕ , която е удовлетворяваща. Сертификатът е t . После верификаторът проверява, че във всяка клауза на ϕ има поне един литерал α , такъв че $t|_{\text{Var}(\alpha)} \models \alpha$. Това е достатъчно, за да бъде ϕ удовлетворима. Дължината на сертификата е полиномиална в размера на ϕ , и е очевидно, че проверката може да се извърши във време, полиномиално в размера на ϕ .

Това беше лесно. Сега ще докажем, че $\text{SAT} \in \text{NP-h}$, което не е лесно.

Редукция 9: $\forall P \in \text{NP} : P \leq_p \text{SAT}$

Конструкция: Разглеждаме произволна $P \in \text{NP}$. Мислим за P като за език, а за NP като множество от езици, над някаква азбука Σ . Задачите в NP са какви ли не: логически, графови, транспортни, оптимизационни, алгебрични и така нататък. Ние не знаем точно каква е NP и не искаме да губим общност, правейки допускания за природата ѝ. Единственото, което знаем за P , е, че съществуват НМТ-верификатор M и полином $p(n)$, такива че M приема всеки ДА-екземпляр x на P във време, ограничено от $p(|x|)$. Допускаме, че знаем $p(n)$ и че $p(n) \geq n, \forall n$. Допускаме, че имаме достъп до M , като знаем всичко за нея съгласно Определение 120: входната азбука на M е гореспоменатата Σ , но освен това знаем и лентовата азбука Γ , и множеството от състоянията Q , и стартовото състояние q_0 , и приемащото q_Y , и отхвърлящото q_N , и, най-важното, функцията на преходите δ . Тъй като M е верификатор, δ е функция на преходите, а недетерминизмът се състои в това, че като първа фаза от работата си M отгатва сертификат.

Без това допускане—че знаем M —няма как да направим доказателство, понеже не знаем нищо конкретно за природата на задачата P , която свеждаме до SAT.

Въпреки че мислим за P като за език, редукцията ще изобразява елементите на P в екземпляри на SAT, тоест във формули, а не в стрингове, които кодират формули. Това ще ни спести главоболията, свързани с кодиранията на формули.

Нека x е произволен стринг от Σ^* . Ще докажем, че съществува КНФ $\phi(x)$, която е удовлетворима тстк $x \in P$. Доказателството е конструктивно: ние ще покажем как може да се построи $\phi(x)$, и то във време, полиномиално в $|x|$.

$\phi(x)$ моделира приемащо изчисление на верификатора M върху вход x . И по-точно, $\phi(x)$ моделира конфигурациите на машината във всеки момент от изчислението – от началната конфигурация чак до приемащата конфигурация в края.

Нека $n = |x|$. Определение 121 казва, че конфигурация на МТ за даден момент е $\langle q, y, i \rangle$, където q е състоянието в този момент, y е съдържанието на лентата в този момент, а i е позицията на главата в този момент. q е елемент на крайното множество Q . Ключово наблюдение е, че ако x бива приет от НТМ M , чиято сложност по време $T_M(n)$ е ограничена от $p(n)$, то за всеки момент от работата на $M(x)$, y и i са ограничени от следните съображения.

- y се намира в подлентата между клетка $-p(n)$ и клетка $p(n) + 1$ включително. Аргументацията се корени в начина, по който машината движи главата си. Главата се движи последователно по лентата без опция да скача произволно от клетка в клетка. Казвайки “подлента”, имаме предвид непрекъснатата редица от лентови клетки.

За $p(n)$ стъпки, ако машината само движи главата наляво, текущата клетка може да е с най-малък номер $-p(n)$, но не и по-малък. Помним, че лентата е двустранна и клетките са номерирани с целите числа $\dots, -1, 0, 1, \dots$ и че в началото текущата клетка е 1.

Аналогично, за $p(n)$ стъпки, ако машината само движи главата надясно, текущата клетка може да е с най-голям номер $p(n) + 1$, но не по-голям.

Тези две съображения са несъвместими – няма как машината хем да мести главата само наляво, хем да мести главата само надясно. Ерго, ограничението, че y се намира някъде между клетка $-p(n)$ и клетка $p(n) + 1$ е вярно, но не е точно. Това е надценяване (*overestimate* на английски) на подлентата, в която може да е y . Но доказателството ни остава валидно дори при това надценяване. Важното е, че няма как машината да направи достъп до клетка наляво от $-p(n)$ и надясно от $p(n) + 1$. Формулата $\phi(x)$, която изграждаме, е достатъчно да моделира адекватно клетките на подлентата между $-p(n)$ и $p(n) + 1$ включително. Двете други подленти (те са безкрайни)—вляво от $-p(n)$ и вдясно от $p(n) + 1$ —са недостъпни за машина с ограничение $p(n)$ върху сложността и няма нужда да бъдат моделирани. В някакъв смисъл, тях ги няма.

- Напълно аналогично, за всеки момент от изчислението, за номера i на текущата клетка е вярно, че $-p(n) \leq i \leq p(n) + 1$.

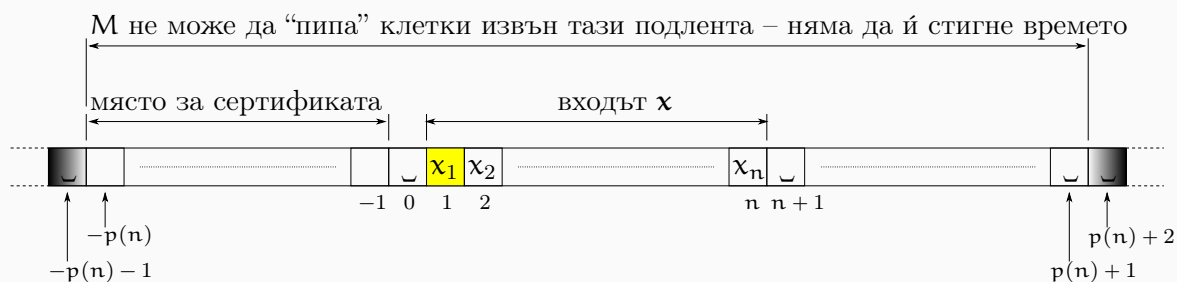
Заклучаваме, че конфигурациите, които трябва да бъдат моделирани за даден момент от работата на $M(x)$, са само полиномиално много в n . Оттук, общият брой на конфигурациите, които трябва да се моделират за цялата работа на работа на $M(x)$, е само полиномиален в n .

Фигура 15.3 илюстрира факта, че за всеки момент от работата на M върху x , “действието се развива” само в подлентата от клетка $-p(n)$ до клетка $p(n) + 1$ включително. Фигурата показва началното съдържание на лентата, при което

- входът x е разположен в подлентата от клетка 1 до клетка n включително,
- клетки $n + 1, \dots, p(n) + 1$ съдържат шпации,
- клетка 0 съдържа шпация, която служи за разделител между входа и сертификата,
- а съдържанието на клетки $-1, \dots, -p(n)$ не се уточнява; сертификатът σ е в подлентата от клетка -1 до клетка $-|\sigma|$, а всички клетки вдясно от него са със шпации, но не знаем колко е дълъг сертификатът и дали изобщо има сертификат (може и да е празен и тогава клетки $-1, \dots, -p(n)$ са със шпации); със сигурност $|\sigma| \leq p(n)$, в противен случай M не би имала възможност да прочете целия сертификат, понеже няма да ѝ стигне времето.

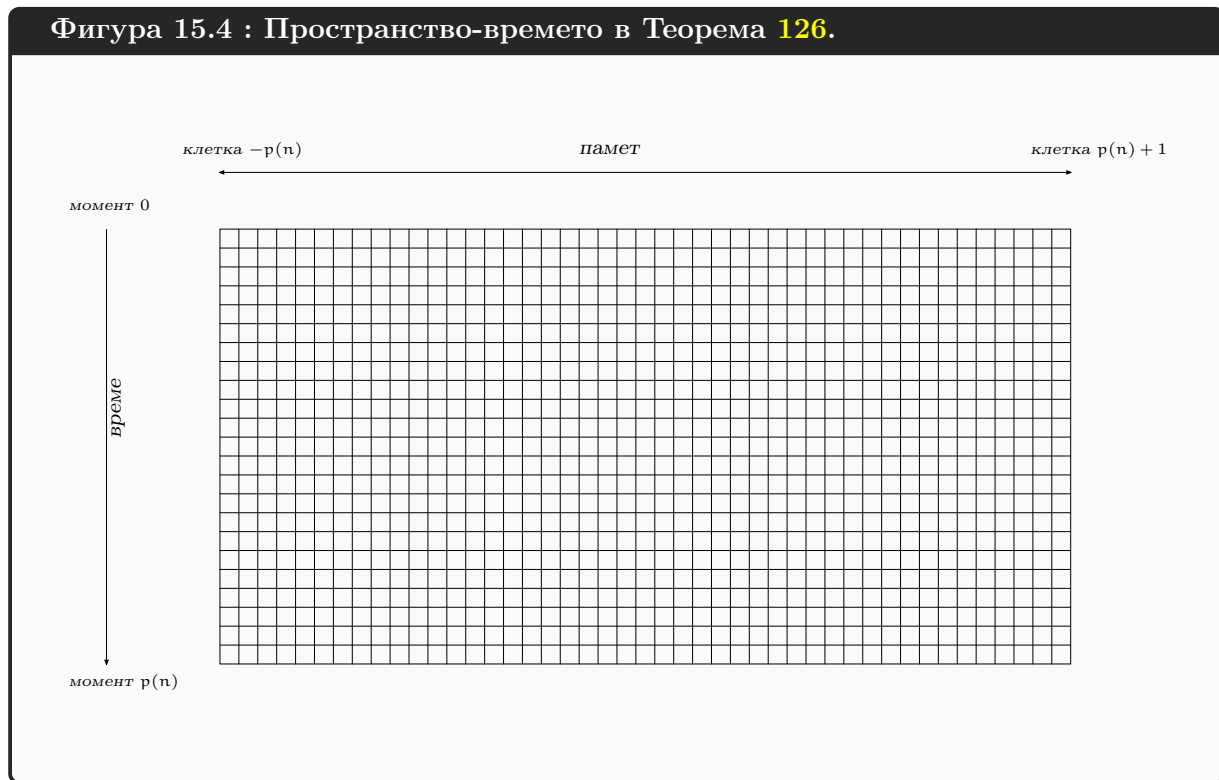
Фигурата освен това илюстрира факта, че подлентите вляво от $-p(n)$ и вдясно от $p(n) + 1$ остават недостъпни за M и можем спокойно да мислим, че ги няма.

Фигура 15.3 : Началното съдържание на лентата на $M(x)$ в Теорема 126.



Видяхме как $p(n)$ задава ограничение върху броя на клетките памет, която трябва да бъде моделирана чрез КНФ в редукцията. Естествено, имаме аналогично ограничение и за броя на моментите, които трябва да бъдат моделирани. Дискретното време, в което M приема x , започва в момент 0 и завършва най-много в момент $p(n)$. Следователно, налага се да моделираме най-много $p(n) + 1$ момента. За простота ще моделираме точно $p(n) + 1$ момента: постулираме, че ако машината достигне до приемаща конфигурация преди момент $p(n)$, тя “замръзва” в тази конфигурация и остава в нея до момент $p(n)$.

Подчертаваме, че всички тези моменти трябва да бъдат моделирани чрез **една** формула, понеже крайният резултат от редукцията е една КНФ. Заради това ще мислим за времето като за втора дименсия, ортогонална на дименсията на паметта (лентата). Фигура 15.4 илюстрира пространство-времето[†], което трябва да бъде моделирано.



Започваме да описваме редукцията. Нека множеството от състояния на M е

$$Q = \{q_0, q_1, q_2, q_3, \dots, q_r\}$$

където q_0 е стартовото състояние, q_1 е приемащото q_Y , q_2 е отхвърлящото q_N , а останалите са някакви работни състояния. Очевидно $|Q| = r + 1$. Нека лентовата азбука на M е

$$\Gamma = \{\gamma_0, \gamma_1, \gamma_2, \dots, \gamma_v\}$$

където γ_0 е шпацията ‘ $\sqcup$ ’, а останалите са някакви работни букви. Очевидно $|\Gamma| = v + 1$. Променливи, които редукцията ще ползва, са описани в Таблица 15.1. Те са доста и за удобство са групирани в три групи съобразно това, което моделират.

[†]Нещо подобно, но несравнимо по-сложно, се прави във физиката.

Група променливи	Как се менят индексите	Какво моделират
$S_{i,j}$	$i \in \{0, \dots, p(n)\}$ $j \in \{0, \dots, r\}$	В момент i , M е в състояние j
$H_{i,j}$	$i \in \{0, \dots, p(n)\}$ $j \in \{-p(n), \dots, p(n) + 1\}$	В момент i , текущата клетка е j
$T_{i,j,k}$	$i \in \{0, \dots, p(n)\}$ $j \in \{-p(n), \dots, p(n) + 1\}$ $k \in \{0, \dots, v\}$	В момент i , клетка j съдържа буква k

Таблица 15.1: Променливите в Редукция 9.

А ето какви клаузи конструираме с тези променливи. Клаузите са много и за яснота за групирани в седем групи, като клаузите във всяка група гарантират даден аспект от коректността на приемането на x от M .

Група 1: Клаузите от тази група гарантират, че във всеки момент от изчислението на $M(x)$, машината е в точно едно състояние (вижте Лема 56).

$$(S_{i,0} \vee S_{i,1} \vee \dots \vee S_{i,r}), \text{ за } 0 \leq i \leq p(n) \quad (15.21)$$

$$(\overline{S_{i,j}} \vee \overline{S_{i,k}}), \text{ за } 0 \leq i \leq p(n) \text{ и } 0 \leq j < k \leq r \quad (15.22)$$

Група 2: Клаузите от тази група гарантират, че във всеки момент от изчислението на $M(x)$, точно една клетка е текущата (вижте Лема 56).

$$(H_{i,-p(n)} \vee H_{i,-p(n)+1} \vee \dots \vee H_{i,p(n)+1}), \text{ за } 0 \leq i \leq p(n) \quad (15.23)$$

$$(\overline{H_{i,j}} \vee \overline{H_{i,k}}), \text{ за } 0 \leq i \leq p(n), \quad (15.24)$$

и $-p(n) \leq j < k \leq p(n) + 1$

Група 3: Клаузите от тази група гарантират, че във всеки момент от изчислението на $M(x)$, всяка клетка съдържа точно една буква (вижте Лема 56).

$$(T_{i,j,0} \vee T_{i,j,1} \vee \dots \vee T_{i,j,v}), \text{ за } 0 \leq i \leq p(n) \quad (15.25)$$

$$(\overline{T_{i,j,k}} \vee \overline{T_{i,j,\ell}}), \text{ за } 0 \leq i \leq p(n) \quad (15.26)$$

и $-p(n) \leq j < k \leq p(n) + 1$
и $0 \leq k < \ell \leq v$

Група 4: Клаузите от тази група гарантират, че в началния момент от изчислението на $M(x)$ налице е началната конфигурация. Тъй като тези клаузи налагат неща, те са с по един литерал; ерго, цялата КНФ, която редукцията конструира, може да бъде удовлетворена само от валуация t , такава че за всеки от тези литерали, да го наречем λ , е изпълнено $TA(\lambda, t|_{\text{Var}(\lambda)}) = 1$.

- Машината е в начално състояние $q_0 = 0$. Клаузата, която налага това, е

$$(S_{0,0}) \quad (15.27)$$

- Текущата клетка е 1. Клаузата, която налага това, е

$$(H_{0,1}) \quad (15.28)$$

- Входът е записан в клетки от 1 до n включително. На Фигура 15.3 отбелязахме буквите на входа като $'x_1', \dots, 'x_n'$, но това е нотацията за елементите на $\mathbf{x}$, която не казва нищо за букви от Γ . Нека $x_1 = \gamma_{j_1}, \dots, x_n = \gamma_{j_n}$, където $j_1, \dots, j_n \in \{1, \dots, \nu\}$ (нулата е изключена, защото входът не съдържа шпации). КНФ, която налага наличието на стринга $\gamma_{j_1}, \dots, \gamma_{j_n}$ в клетки 1, $\dots$, n , е

$$(T_{0,1,j_1}) \wedge (T_{0,2,j_2}) \wedge \dots \wedge (T_{0,n,j_n}) \quad (15.29)$$

- Клетки $n+1, \dots, p(n)+1$ съдържат шпации. КНФ, която налага това, е

$$(T_{0,n+1,0}) \wedge (T_{0,n+2,0}) \wedge \dots \wedge (T_{0,p(n)+1,0}) \quad (15.30)$$

Забележете нещо важно. Някоя от тези клаузи не моделира подлентата от $-p(n)$ до -1 . Там някъде се намира сертификатът (Фигура 15.3). И така, клаузите не се опитват да моделират сертификата по никакъв начин. В подлентата от $-p(n)$ до -1 може да има всичко, в смисъл, че клаузите налагат единствено всяка клетка на тази подлента да има точно една буква (това е заради клаузите от **Група 3**) в нулевия момент, но не налагат каква да е тази буква. По отношение на крайната КНФ, която редукцията конструира, за всяка удовлетворяваща валуация $\mathbf{t}$ е вярно, че $\mathbf{t}$ “раздава” някакви стойности на променливите $T_{0,-p(n),k}, T_{0,-p(n)+1,k}, \dots, T_{0,-1,k}$, където $k \in \{0, \dots, \nu\}$, което съответства на наличието някакви букви в подлентата от $-p(n)$ до -1 . Всяко съдържание на подлентата от $-p(n)$ до -1 в този смисъл е един сертификат, благодарение на който M приема $\mathbf{x}$.

Има една незначителна особеност. Типично, ние си представяме, че сертификатът не съдържа шпации. Съгласно това, ако сертификатът е σ , това означава той да е разположен в клетки $-1, -2, \dots, -|\sigma|$ в нулевия момент, така че в нулевия момент там няма шпации, а в нулевия момент от клетки $-|\sigma|-1$ до $-p(n)$ съдържанието е от шпации. От друга страна, нашите клаузи позволяват $-p(n)$ до -1 в нулевия момент да съдържа произволен стринг над Γ , включително стринг, изпъстрен с шпации. Това не е съществено – можем да дефинираме НТМ-верификатор дори със сертификат, съдържащ шпации.

Накратко, нашите клаузи не моделират подлентата от $-p(n)$ до -1 в нулевия момент. Там има някакъв стринг над Γ и той е сертификатът. Клаузите са удовлетворими тстк съществува съдържание подлентата от $-p(n)$ до -1 в нулевия момент, което води до приемащо изчисление, и толкова.

Група 5: Тази група има една единствена клауза, гарантираща, че в момент $p(n)$ машината е приела входа.

$$(S_{p(n),1}) \quad (15.31)$$

Група 6: Клаузите от тази група гарантират, че по време на работата на $M(\mathbf{x})$, при всеки преход от даден момент i към следващия момент $i+1$, съдържанието на всяка клетка, която не е текущата в момент i , не се променя. Без такова ограничение може на удовлетворяващата валуация $\mathbf{t}$ да съответства изчисление, при което произволни клетки произволно променят съдържанието си, а това не е позволено. Машината на Turing по определение има достъп до само една клетка в даден момент; останалите клетки просто помнят записаното в тях и не се променят. Разсъждаваме така:

за всеки момент без последния, за всяка позиция j , ако j не е текущата клетка, то в следващия момент съдържанието i е същото

Тази формулировка не е подходяща обаче, защото ползва “същото съдържание”, което звучи сякаш ще използваме равенство в клаузата, а дизюнктивните клаузи не могат да съдържат знак за равенство. Да опитаме така:

за всеки момент без последния, за всяка позиция j , за всяка буква k , ако текущата клетка не е j и съдържа буква k , то в следващия момент клетка j съдържа буква k

Това се превежда директно в

$$\overline{H_{i,j}} \wedge T_{i,j,k} \rightarrow T_{i+1,j,k}$$

Тази формула не е дизюнктивна клауза, но лесно намираме еквивалентна на нея дизюнктивна клауза

$$\begin{aligned} (\overline{H_{i,j}} \wedge T_{i,j,k}) \rightarrow T_{i+1,j,k} &\equiv \\ \overline{(\overline{H_{i,j}} \wedge T_{i,j,k})} \vee T_{i+1,j,k} &\equiv \\ \overline{\overline{H_{i,j}}} \vee \overline{T_{i,j,k}} \vee T_{i+1,j,k} &\equiv \\ H_{i,j} \vee \overline{T_{i,j,k}} \vee T_{i+1,j,k} \end{aligned}$$

И така, ето шестата група клаузи

$$\begin{aligned} (H_{i,j} \vee \overline{T_{i,j,k}} \vee T_{i+1,j,k}), \text{ за } 0 \leq i \leq p(n) - 1 \\ \text{и } -p(n) \leq j < k \leq p(n) + 1 \\ \text{и } 0 \leq k \leq v \end{aligned} \quad (15.32)$$

Група 7: Клаузите дотук не ползват нищо конкретно, касаещо машината M , освен полинома $p(n)$. Сега ще ползваме таблицата на преходите δ на M . Тази група клаузи гарантира, че преминаването от конфигурацията в даден момент към следващия става само съгласно δ . Тъй като с шестата група клаузи гарантирахме, че съдържанието на лентата, с изключение на текущата клетка, не може се променя от само себе си, то сега е достатъчно да гарантираме, че промяната в позицията на текущата клетка става съгласно δ .

Преминаването към нова конфигурация има три аспекта: ново състояние, нова буква и нова текуща клетка. За всяка комбинация от състояние k и текуща буква ℓ правим следното. Нека

$$\delta(q_k, \gamma_\ell) = (q_{k'}, \gamma_{\ell'}, d)$$

където k' е новото състояние, ℓ' е новата буква, а $d \in \{\leftarrow, \rightarrow, \downarrow\}$. Понеже моделираме позицията на главата, тоест, текущата клетка, с число, на ‘ $\leftarrow$ ’ съответства -1 , на ‘ $\rightarrow$ ’ съответства 1 и на ‘ $\downarrow$ ’ съответства 0 , така че $d \in \{-1, 0, 1\}$ за целите на конструкцията.

Първо да разгледаме новото състояние. Казваме така:

ако машината е в състояние q_k и текущата буква е γ_ℓ , то новото състояние е $q_{k'}$

Това не е достатъчно детайлно, понеже нямаме булеви променливи за текущите букви. Имаме булеви променливи за текущите позиции и булеви променливи за текущото съдържание на клетките на лентата. С оглед на това, казваме така

ако машината е в състояние q_k и текущата клетка е j и съдържанието на клетка j е буква γ_ℓ , то новото състояние е $q_{k'}$

По-подробно,

за всеки момент без последния, за всяка позиция j , за всяко състояние и за всяка буква, ако текущото състояние е q_k и текущата клетка е j и клетка j съдържа буква γ_ℓ , то новото състояние е $q_{k'}$

Това се превежда директно в

$$\begin{aligned} (S_{i,k} \wedge H_{i,j} \wedge T_{i,j,\ell}) \rightarrow S_{i+1,k'} &\equiv \\ \overline{(S_{i,k} \wedge H_{i,j} \wedge T_{i,j,\ell})} \vee S_{i+1,k'} &\equiv \\ \overline{S_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee S_{i+1,k'} \end{aligned}$$

И така, ето клаузите от седмата група, касаещи новото състояние.

$$\begin{aligned} (\overline{S_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee S_{i+1,k'}), \text{ за } 0 \leq i \leq p(n) - 1 & \quad (15.33) \\ \text{и } -p(n) \leq j < k \leq p(n) + 1 & \\ \text{и } 0 \leq k \leq r & \\ \text{и } 0 \leq \ell \leq v & \end{aligned}$$

Аналогично конструираме и клаузите от седмата група, касаещи новата буква и клаузите, касаещи новата текуща клетка.

$$\begin{aligned} (\overline{S_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee H_{i+1,j+d}), \text{ за } 0 \leq i \leq p(n) - 1 & \quad (15.34) \\ \text{и } -p(n) \leq j < k \leq p(n) + 1 & \\ \text{и } 0 \leq k \leq r & \\ \text{и } 0 \leq \ell \leq v & \end{aligned}$$

$$\begin{aligned} (\overline{S_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee T_{i+1,j,\ell'}), \text{ за } 0 \leq i \leq p(n) - 1 & \quad (15.35) \\ \text{и } -p(n) \leq j < k \leq p(n) + 1 & \\ \text{и } 0 \leq k \leq r & \\ \text{и } 0 \leq \ell \leq v & \end{aligned}$$

Има една особеност обаче. Ако състоянието q_k е измежду q_1 и q_2 (приемащото и отхвърлящото), то $k' = k$, $\ell' = \ell$ и $d = 0$, така че машината да “замръзне”.

И така, клаузите от (15.21), (15.22), (15.23), (15.24), (15.25), (15.26), (15.27), (15.28), (15.29), (15.30), (15.31), (15.32), (15.33), (15.34) и (15.35), конкатенирани чрез ‘ $\wedge$ ’, са КНФ, която е удовлетворима тстк машината-верификатор M приема входа x за не повече от $p(n)$ стъпки.

Това, че конструкцията може да се извърши за полиномиално време, би трябвало да е очевидно.

Това е краят на доказателството на Теорема 126. □

Ще приключим секцията с едно наблюдение. Теорема 126 казва, че удовлетворимостта на КНФ е неподатлива задача. КНФ и ДНФ са дуални и винаги се дефинират заедно. Обаче удовлетворимостта на ДНФ не е неподатлива, а е тривиална задача.

Наблюдение 105: Всяка ДНФ е удовлетворима

Всяка ДНФ е удовлетворима. По дефиниция (Определение 145), всяка ДНФ съдържа поне една конюнктивна клауза. На свой ред всяка конюнктивна клауза съдържа поне един литерал, като всичките ѝ литерали са на различни променливи, което я прави тривиално удовлетворима. Достатъчно е коя да е конюнктивна клауза да бъде удовлетворена, за да бъде удовлетворена цялата ДНФ. Ерго, никоя ДНФ не е противоречие.

Примерно, ако $\phi = x_1 \bar{x}_2 \vee x_2 x_3 x_9 \vee \dots$, всяка валюация t , такава че $t(x_1) = 1$ и $t(x_2) = 0$ удовлетворява ϕ .

Дуално, за ДНФ неподатливата задача е фалшифицируемост. Тази задача пък е тривиална за КНФ: щом има поне една конюнктивна клауза, КНФ не е тавтология. Примерно, $\phi = (x_1 \vee \bar{x}_2)(x_2 \vee x_3 \vee x_9) \dots$ не е тавтология, защото всяка валюация s , такава че $s(x_1) = 0$ и $s(x_2) = 1$, фалшифицира ϕ .

15.7 SAT е началото на редици от доказателства на NP-пълнота

Нека P е произволна задача от NP . Искаме да докажем, че P е NP -пълна. Ако доказателството е съгласно Определение 139, то е доста тежаво! Съгласно Определение 139, необходимо е да докажем, че всяка задача от NP се свежда до P с Карп редукция. Доказателството на Теорема 126, което току-що видяхме, беше точно такова и наистина беше тежаво. Но, ако **вече** сме доказали NP -пълнотата на някаква задача P' и ако докажем, че $P' \leq_p P$, то сме доказали, че NP -трудността на P . Теорема 127 ни дава право на това. Доказателствата, базирани на тази идея, по правило са много по-прости от “каноничните” доказателства, базирани на идеята всяка друга задача от NP се свежда до P с Карп редукция.

Теорема 127: $P \in NP \wedge P' \in NP\text{-с} \wedge P' \leq_p P \rightarrow P \in NP\text{-с}$

Нека $P, P' \in NP$. Ако $P' \leq_p P$ и $P' \in NP\text{-с}$, то $P \in NP\text{-с}$.

Доказателство: Тъй като по условие $P \in NP$, това, което остава да докажем, е, че за всяка задача $P'' \in NP$ е вярно, че $P'' \leq_p P$. Знаем, че $P' \in NP\text{-с}$. Това означава, че за всяка задача $P'' \in NP$ е вярно, че $P'' \leq_p P'$. Допускаме, че $P' \leq_p P$ и прилагаме Теорема 120. Заклучаваме, че за всяка задача $P'' \in NP$ е вярно, че $P'' \leq_p P$. $\square$

Следствие 40: Редици от доказателства на NP-пълнота, започващи със SAT

Теорема 126 и Теорема 127 позволяват следната схема за доказателства на NP -пълнота. За някаква задача P_1 доказваме, че $P_1 \in NP$ и $SAT \leq_p P_1$, от което следва, че P_1 е NP -пълна. За друга задача P_2 доказваме, че $P_2 \in NP$ и $P_1 \leq_p P_2$, от което следва, че P_2 е NP -пълна. За трета задача P_3 доказваме, че $P_3 \in NP$ и $P_2 \leq_p P_3$, от което следва, че P_3 е NP -пълна. И така нататък.

Всяка такава редица SAT, P_1, P_2, P_3 и така нататък от задачи наричаме *редица от доказателства на NP-пълнота, започваща със SAT*.

Безспорен факт е, че по отношение на дадена задача P , чиято NP -пълнота трябва да докажем, някои задачи са много по-удобни от други задачи в смисъл, че се свеждат до P със

значително по-проста и лесна за разбиране редукция. Ясно е, че всяка задача от **NP-с** се свежда до **P** с Karp редукция (стига да е вярно, че $P \in \text{NP-с}$), но за някои задачи описанието на тази редукция е просто и кратко, а за други задачи е сложно и тежко. Ако този факт не е очевиден в момента, със сигурност ще стане очевиден в Лекция 16.

Това е и причината да говорим за “редици от доказателства” в множествено число. Теоретично не е невъзможно да се конструира една единствена редица, започваща със SAT, а всяка задача, чието **NP**-пълнота е доказана чрез идеята на Теорема 127, да се намира някъде в тази редица. На практика е по-удобно доказателствата да бъдат извършени дървовидно, в антиарборесценция с корен SAT (Секция 16.1, Фигура 16.1).

Едно предупреждение за избягване на грешка, която е често срещана при опитите за доказване на **NP**-трудност.

Забележка 4: Внимание с посоката на редукцията!

В контекста на Теорема 127, ако докажем, че $P \leq_p P'$, не сме доказали нищо за **NP**-пълнотата на **P**. Ама наистина **НИЩО**.

Трябва да свеждаме известната задача P' към новата задача **P**, а не обратното. Редукцията в грешната посока е безполезна. Както знаем, винаги можем да направим нещата много по-сложни. В това се убедихме в Редукция 2.

15.8 Географията на NP

Както се каза много пъти дотук, преобладаващото, макар и недоказано, мнение сред специалистите е, че $P \neq \text{NP}$. Да допуснем за целите на тази секция, че това наистина е така. Очевидно тогава е вярно $P \cap \text{NP-с} = \emptyset$; обратното влече $P = \text{NP}$. Но **NP** не се разбива на **P** и **NP-с**! Ако $P \neq \text{NP}$, в **NP** има задачи, които не са нито в **P**, нито в **NP-с**. Можем да кажем, че те са прекалено трудни, за да са в **P**, но не са достатъчно трудни, за да са в **NP-с**. Теоремата на Ladner доказва, че такива задачи има. В оригиналната статия тази теорема е изразена чрез Karp редукции и Turing редукции (вижте [100, Theorem 1, стр. 158]). В учебника на Arora и Barak [6, Theorem 3.3, стр. 71] тази теорема е изразена по начин, който е близък до нашата терминология.

Теорема 128: Теорема на Ladner

Да допуснем, че $P \neq \text{NP}$. Тогава има език $L \in \text{NP} \setminus P$, който не е **NP**-пълнен. □

Доказателството е много сложно и е далеч отвъд обхвата на тези лекции. По същество, това доказателство е различно от оригиналното доказателство на Ladner [100, Theorem 1, стр. 158]; това доказателство първоначално е предложено от известния учен Russell Impagliazzo в непубликувана статия. Задачата, която се разглежда, е SAT_H , дефинирана по отношение на произволна функция $H: \mathbb{N} \rightarrow \mathbb{N}$. SAT_H е дефинирана като език: неговите стрингове се състоят от (кодиране на) КНФ с дължина n , последвани от една нула и стринг от единици, на брой $n^{H(n)}$:

$$\text{SAT}_H = \left\{ \psi 0 1^{n^{H(n)}} \mid \psi \in \text{SAT}, n = |\psi| \right\}$$

Тук ψ е стринг, тоест, кодиране на екземпляр на SAT. Въвежда се конкретна функция $H(n)$, която е много бавно растяща, като $H(n)$ не надхвърля $\lg \lg n$. Дефиницията на $H(n)$ е

сложна и използва изброяване на машините на Turing, но е напълно легитимна дефиниция, позволяваща, поне на теория, рекурсивно изчисляване на $H(n)$ за всяко n . По отношение на тази функция $H(n)$ се доказва, че ако $P \neq NP$, то $SAT_H \notin P$ и $SAT_H \notin NP-c$.

Това доказателство на теоремата на Ladner е известно като *proof by padding SAT*, защото екземплярите на SAT биват допълвани (*padded*) с единици. Това изкуствено сваля сложността, понеже екземплярите стават по-големи. Допълването е достатъчно голямо, така че задачата престава да е NP -пълна, но не е достатъчно, така че задачата да “влезе” в P .

Определение 147: Клас на сложност $NP-i$

Нека P е задача за разпознаване. Казваме, че P е NP -междинна, ако

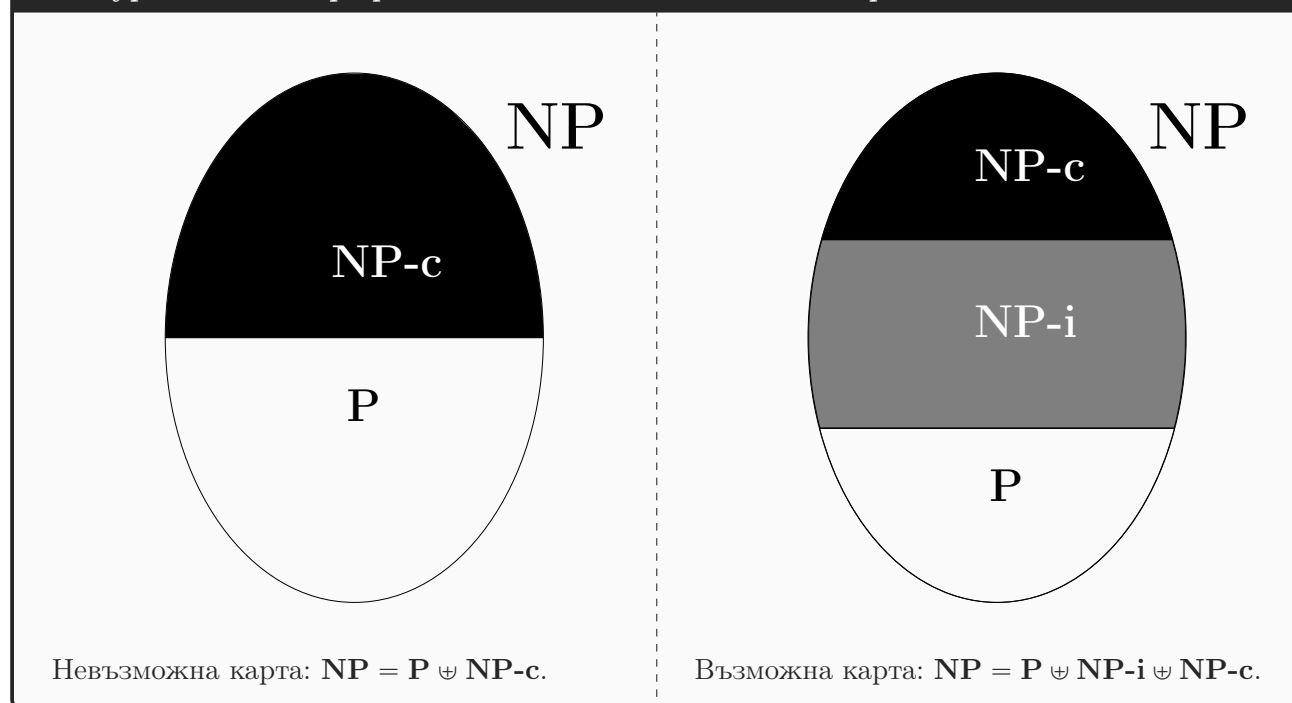
1. $P \in NP$ и
2. $P \notin P$ и $P \notin NP-c$.

Класът на сложност $NP-i$ се състои точно от NP -междинните задачи.

На английски терминът NP -междинна е *NP-intermediate*. В теоретико-множествена нотация, $NP-i = NP \setminus (P \cup NP-c)$. Теоремата на Ladner казва, че ако $P \neq NP$, то $NP-i \neq \emptyset$.

Фигура 15.5 показва вляво “географска карта” на NP , която е възможна от общи съображения, при допускането, че $P \neq NP$: NP се разбива на множеството от лесните задачи P и множеството от трудните задачи $NP-c$. Вдясно е показана картата, изменена в светлината на резултата на Ladner: между P и $NP-c$ се намира междинното множество от задачи $NP-i$. Да си припомним Нотация 13.

Фигура 15.5 : Географията на NP в светлината на теоремата на Ladner.



Езикът (или задачата) SAT_H е изкуствен и създаден само за целите на това доказателство. Arora и Barak казват [6, стр. 72]

Though the theorem shows the existence of some non- NP -complete language in $NP \setminus P$

if $NP \neq P$, this language seems somewhat contrived, and the proof has not been strengthened to yield a more natural language.

Учебникът е от 2009 г., но, доколкото е известно на автора на тези лекционни записки, и до ден днешен не е доказана принадлежността към **NP-i** на никоя естествена изчислителна задача. Има няколко кандидати за този клас, най-известните от които са GRAPH ISOMORPHISM[†] и FACTORING. И двете задачи са очевидно в **NP**. И двете изглеждат неподатливи – това са важни задачи с огромно практическо приложение и усилията за намиране на ефикасни алгоритми за тях са били колосални, но безрезултатни, което навежда на мисълта, че не са в **P**. И за двете не е показано да са **NP**-пълни. Съществуват сериозни основания да се смята, че не са **NP**-пълни: по думите на Garey и Johnson [56, стр. 155–156]:

*Researchers who have attempted to prove that GRAPH ISOMORPHISM is **NP**-complete have noted that its nature is much more constrained than that of a typical **NP**-complete problem, such as SUBGRAPH ISOMORPHISM. **NP**-completeness proofs seem to require a bit of leeway; if the desired structure X (subset, permutation, schedule, etc.) exists, it should still exist even if certain aspects of the instance are locally altered. For example, a function f will be an isomorphism between a graph H and a subgraph of a graph G even if we add edges to G or delete edges not in the image of f . However, if f is an isomorphism between H and G itself, then any change in G must be reflected by a corresponding change in h , or else f will no longer be an isomorphism. In other words, proofs of **NP**-completeness seem to require a certain amount of redundancy in the target problem, a redundancy that GRAPH ISOMORPHISM lacks. Unfortunately, this lack of redundancy does not seem to be much of a help in designing a polynomial time algorithm for GRAPH ISOMORPHISM either, so perhaps it belongs to **NP-i**.*

Всичко това навежда на мисълта, че те са елементи от мистериозния клас **NP-i**. За съжаление, и тук се налага да се задоволим със спекулации и вярване, защото истински прецизно доказателство или опровержение, засега, няма. Ето дефиниции на тези задачи.

Изч. Задача 68: GRAPH ISOMORPHISM

екземпляр: Неориентирани графи G_1 и G_2 .

въпрос: Дали G_1 и G_2 са изоморфни?

Изч. Задача 69: FACTORING, ВЕРСИЯ ЗА ТЪРСЕНЕ

екземпляр: Естествено число n , записано бинарно.

решение: Факторизация на n в прости множители.

Това, разбира се, не е задача за разпознаване, поради което не може да става дума да е в **P** или **NP**. Но в следния вариант тя става задача за разпознаване и в този вариант е кандидат за **NP-i**.

Изч. Задача 70: FACTORING, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Естествени числа n и k , записани бинарно.

въпрос: Дали n има прост фактор, ненадхвърлящ k ?

[†] Да не се бърка GRAPH ISOMORPHISM със задачата SUBGRAPH ISOMORPHISM, която е известна **NP**-пълна задача, както ще видим в Лекция 16.

Съществува много по-силен и впечатляващ резултат от теоремата на Ladner: при допускането, че $\mathbf{P} \neq \mathbf{NP}$, съществува цяла безкрайна йерархия от класове на сложност в $\mathbf{NP}$, чийто минимален елемент е $\mathbf{P}$ и чийто максимален елемент е $\mathbf{NP-c}$, в която всеки по-нисък клас се съдържа в по-високия клас, но задачите от по-високия клас не се свеждат до задачи от по-ниския с Karp редукции [138].

Завършваме, подчертавайки, че всички тези теоретични резултати са в сила при допускането, че $\mathbf{P} \neq \mathbf{NP}$. Ако се окаже, че $\mathbf{P} = \mathbf{NP}$, тези теории колабират – при това положение, Karp редукциите са прекалено мощни и спрямо тях, класът $\mathbf{NP}$ няма структура.

Лекция 16

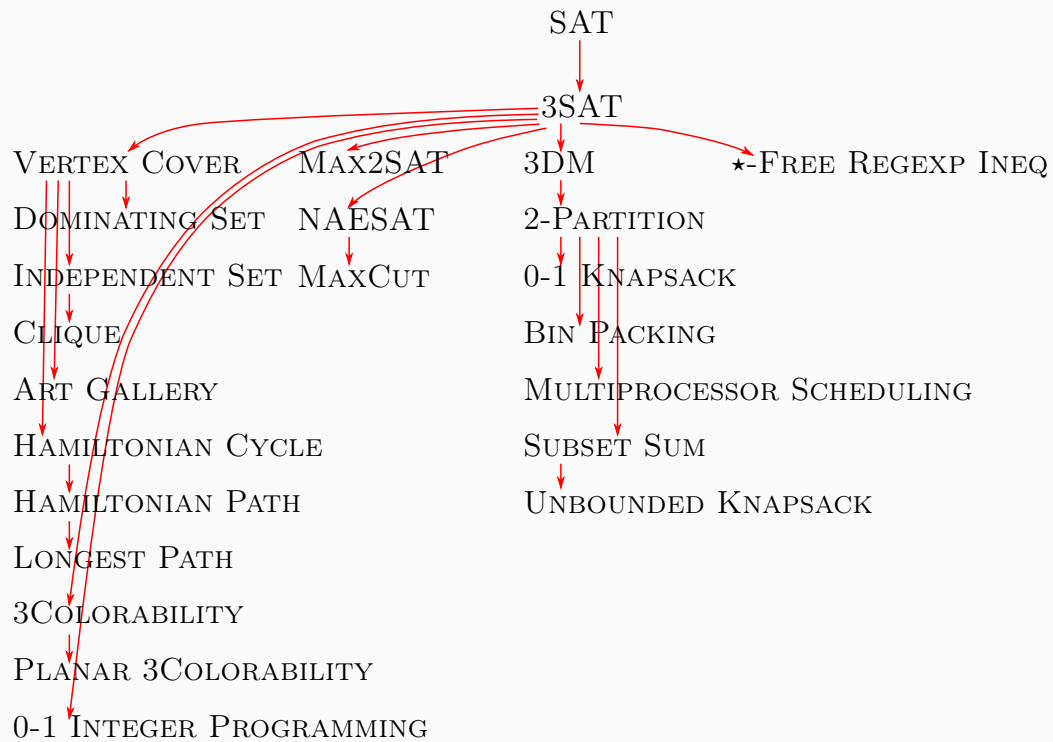
Основни NP-пълни задачи.

Резюме: Доказваме, че задачата 3SAT е NP-пълна. Стъпвайки на този резултат, доказваме NP-пълнотата на множество задачи от областта на теорията на графите и оптимизацията.

16.1 Дървото на доказателствата на NP-пълнота.

В Секция 15.7 казахме, че само за SAT ще направим доказателство за NP-пълнота съгласно Определение 139. Следващите доказателствата за NP-пълнотата ще се базират на Теорема 127. Тези доказателства ще бъдат в редици, започващи със SAT, съгласно Следствие 40. Съвкупността от тези редици е дърво с корен SAT. Фигура 16.1 показва това дърво. Горната част на дървото е от [56, Глава 3.1, стр. 46–47].

Фигура 16.1 : Дървовидната структура на доказателствата за NP-трудност.



16.2 Редукциите.

Тази секция съдържа само редукциите, поради което тук е доказана само NP-трудността на задачите. За всяка от задачите от тази секция е очевидно, че принадлежи на NP, така че това дори не го споменаваме повече.

16.2.1 3SAT, MAX2SAT и NAESAT

3SAT $\in$ NP-с. На практика е удобно е да използваме една ограничена версия на SAT, която също е NP-пълна. Ще използваме нея, за да свеждаме до други задачи, защото нейните клаузи са с фиксиран брой литерали.

Изч. Задача 71: 3SAT

екземпляр: КНФ ϕ , в която всяка дизюнктивна клауза има точно три литерала.

въпрос: Дали ϕ е удовлетворима?

Ето два екземпляра на 3SAT: ϕ_1 и ϕ_2 .

$$\phi_1 = (x \vee y \vee z)$$

$$\phi_2 = (x_1 \vee \overline{x_2} \vee x_3)(\overline{x_1} \vee \overline{x_2} \vee x_4)(x_1 \vee x_3 \vee x_4)$$

Забележете, че литералите трябва да са **точно** три, а не най-много три.

Редукция 10: $\text{SAT} \leq_p \text{3SAT}$

Конструкция: Нека е даден екземпляр ϕ на SAT. ϕ е КНФ над някакви променливи $x_1, \dots, x_n$ и с някакви клаузи $\alpha_1, \dots, \alpha_m$. Във време, полиномиално в размера на ϕ , ще конструираме КНФ ψ , екземпляр на 3SAT, такава че ϕ е удовлетворима тстк ψ е удовлетворима.

Конструкцията заменя всяка клауза α_i с КНФ β_i , състояща се от една или повече клаузи, всяка от които има точно три литерала. Окончателната ψ е $\psi = \beta_1 \wedge \dots \wedge \beta_m$. Конструкцията е лесна за разбиране, защото е локална, в смисъл че при заменянето на α_i с β_i се въвеждат и ползват нови променливи, които не се появяват в някои други β_j . Поради това и верификацията е лесна – достатъчно е да се покаже, че всяка α_i е удовлетворима тстк съответната β_i е удовлетворима.

Разглеждаме произволна клауза α_i . Следните възможности са изчерпателни и взаимно изключващи се.

Случай 1: α_i има точно един литерал. Тогава $\alpha_i = (\lambda)$ за някакъв литерал λ над променливите $x_1, \dots, x_n$. Въвеждаме две нови променливи $y_{i,1}$ и $y_{i,2}$, които няма да ползваме за никоя друга α_j . Конструираме

$$\beta_i = (\lambda \vee y_{i,1} \vee y_{i,2}) \wedge (\lambda \vee y_{i,1} \vee \overline{y_{i,2}}) \wedge (\lambda \vee \overline{y_{i,1}} \vee y_{i,2}) \wedge (\lambda \vee \overline{y_{i,1}} \vee \overline{y_{i,2}})$$

Твърдим, че α_i е удовлетворима тстк β_i е удовлетворима. В едната посока, нека α_i е удовлетворима от валюация t на променливата на нейния литерал λ . Това е същото като $\text{TA}(\alpha_i, t|_{\text{Var}(\lambda)}) = 1$. Но тогава за всяка валюация $t' \in \text{Val}(\text{Var}(\beta_i))$, такава че $\text{TA}(\lambda, t'|_{\text{Var}(\lambda)}) = 1$ е вярно, че $t' \models \beta_i$, понеже λ се среща във всяка клауза на β_i . Ерго, β_i е удовлетворима. Забележете, че $t'(y_{i,1})$ и $t'(y_{i,2})$ нямат значение.

В обратната посока, нека β_i е удовлетворима от валюация t' на $\text{Var}(\beta_i)$. Тогава във всяка от четирите клаузи на β_i има поне един литерал μ , такъв че $\text{TA}(\mu, t'|_{\text{Var}(\mu)}) = 1$. Ще покажем, че този литерал може да е само λ . Наистина, за всяка от четирите възможности за $t'|_{\{y_{i,1}, y_{i,2}\}}$ е вярно, че в поне една клауза[†] на β_i , е вярно, че $\text{TA}(\mu_1, t'|_{\text{Var}(\mu_1)}) = \text{TA}(\mu_2, t'|_{\mu_2}) = 0$, където μ_1 и μ_2 са литералите, съответни на $y_{i,1}$ и $y_{i,2}$. И така, β_i може да бъде удовлетворена само “през” λ . Но тогава

$$t' \models \beta_i \rightarrow t'|_{\text{Var}(\lambda)} \models \alpha_i$$

Случай 2: α_i има точно два литерала. Тогава $\alpha_i = (\lambda \vee \mu)$ за някакви литерали λ и μ над различни променливи от $x_1, \dots, x_n$. Въвеждаме една нова променлива y_i , която няма да ползваме за никоя друга α_j . Конструираме

$$\beta_i = (\lambda \vee \mu \vee y_i) \wedge (\lambda \vee \mu \vee \overline{y_i})$$

Твърдим, че α_i е удовлетворима тстк β_i е удовлетворима. Доказателството е аналогично на доказателството в предния случай.

Случай 3: α_i има точно три литерала. В този случай просто $\beta_i = \alpha_i$. Замяната на α_i с β_i , за която споменахме горе, е тривиалната замяна, при която α_i се заменя със себе си. Тук няма какво да се доказва за коректността.

Случай 4: α_i има повече от три литерала. Да кажем, че $\alpha_i = (\lambda_1 \vee \dots \vee \lambda_k)$, където $k \geq 4$. Въвеждаме $k - 3$ нови променливи $y_{i,1}, \dots, y_{i,k-3}$, които няма да ползваме за никоя друга α_j . Конструираме

$$\begin{aligned} \beta_i = & (\lambda_1 \vee \lambda_2 \vee y_{i,1}) \wedge \\ & (\overline{y_{i,1}} \vee \lambda_3 \vee y_{i,2}) \wedge \\ & (\overline{y_{i,2}} \vee \lambda_4 \vee y_{i,3}) \wedge \\ & \dots \\ & (\overline{y_{i,j-1}} \vee \lambda_{j+1} \vee y_{i,j}) \wedge \\ & (\overline{y_{i,j}} \vee \lambda_{j+2} \vee y_{i,j+1}) \wedge \\ & (\overline{y_{i,j+1}} \vee \lambda_{j+3} \vee y_{i,j+2}) \wedge \\ & \dots \\ & (\overline{y_{i,k-4}} \vee \lambda_{k-2} \vee y_{i,k-3}) \wedge \\ & (\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k) \end{aligned}$$

Твърдим, че α_i е удовлетворима тстк β_i е удовлетворима.

- В едната посока, нека α_i е удовлетворима. Тогава съществува $t \in \text{Val}(\text{Var}(\alpha_i))$, такава че $t \models \alpha_i$, което е същото като за поне един литерал λ_ℓ на α_i да е вярно $\text{TA}(\lambda_\ell, t|_{\text{Var}(\lambda_\ell)}) = 1$. Разглеждаме три подслучая.

Случай 4.1: $\ell \in \{1, 2\}$. Тогава β_i се удовлетворява от валюацията

$$t' \in \text{Val}(\text{Var}(\{\lambda_1, \dots, \lambda_k\}) \cup \{y_{i,1}, \dots, y_{i,k-3}\})$$

[†] Ако трябва да сме прецизни, **точно** една клауза на β_i , но това е без значение.

такава че

$$\begin{aligned} t' \upharpoonright_{\text{Var}(\{\lambda_1, \dots, \lambda_k\})} &= t \\ t'(y_{i,j}) &= 0, \text{ за } 1 \leq j \leq k-3 \end{aligned}$$

Веднага се вижда, че под t' , във всяка клауза на β_i има поне един литерал, “през” който клаузата може да бъде удовлетворена: в първата клауза, това е λ_1 или λ_2 , а във всяка от останалите клаузи това е отрицателният $\overline{y_{i,j}}$.

Случай 4.2: $\ell \in \{3, \dots, k-2\}$. Тогава β_i се удовлетворява от валюацията

$$t' \in \text{Val}(\text{Var}(\{\lambda_1, \dots, \lambda_k\}) \cup \{y_{i,1}, \dots, y_{i,k-3}\})$$

такава че

$$\begin{aligned} t' \upharpoonright_{\text{Var}(\{\lambda_1, \dots, \lambda_k\})} &= t \\ t'(y_{i,j}) &= 1, \text{ за } 1 \leq j \leq \ell-2 \\ t'(y_{i,j}) &= 0, \text{ за } \ell-1 \leq j \leq k-3 \end{aligned}$$

Ще покажем, че във всяка клауза на β_i има поне един литерал, “през” който тази клауза може да бъде удовлетворена. В клаузата $(\overline{y_{i,\ell-2}} \vee \lambda_\ell \vee y_{i,\ell-2})$ това е самият λ_ℓ , в клаузите “отгоре” това е положителният литерал $y_{i,j}$, а в клаузите “отдолу” това е отрицателният $\overline{y_{i,j}}$. В следния израз, литералите μ , за които $\text{TA}(\mu, t' \upharpoonright_{\text{Var}(\mu)}) = 1$, са в зелен цвят, а литералите μ , за които $\text{TA}(\mu, t' \upharpoonright_{\text{Var}(\mu)}) = 0$ които имат стойност 0 със сигурност, са в червен цвят. Останалите литерали—тези, за чиято стойност не сме сигурни, а те всъщност са останалите λ_j —са в черно.

$$\begin{aligned} \beta_i = & (\lambda_1 \vee \lambda_2 \vee y_{i,1}) \wedge \\ & (\overline{y_{i,1}} \vee \lambda_3 \vee y_{i,2}) \wedge \\ & (\overline{y_{i,2}} \vee \lambda_4 \vee y_{i,3}) \wedge \\ & \dots \\ & (\overline{y_{i,\ell-3}} \vee \lambda_{\ell-1} \vee y_{i,\ell-2}) \wedge \\ & (\overline{y_{i,\ell-2}} \vee \lambda_\ell \vee y_{i,\ell-1}) \wedge \\ & (\overline{y_{i,\ell-1}} \vee \lambda_{\ell+1} \vee y_{i,\ell}) \wedge \\ & \dots \\ & (\overline{y_{i,k-4}} \vee \lambda_{k-2} \vee y_{i,k-3}) \wedge \\ & (\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k) \end{aligned}$$

Случай 4.3: $\ell \in \{k-1, k\}$. Тогава β_i се удовлетворява от валюацията

$$t' \in \text{Val}(\text{Var}(\{\lambda_1, \dots, \lambda_k\}) \cup \{y_{i,1}, \dots, y_{i,k-3}\})$$

такава че

$$\begin{aligned} t' \upharpoonright_{\text{Var}(\{\lambda_1, \dots, \lambda_k\})} &= t \\ t'(y_{i,j}) &= 1, \text{ за } 1 \leq j \leq k-3 \end{aligned}$$

Веднага се вижда, че под t' , във всяка клауза на β_i има поне един литерал със стойност 1: в последната клауза, това е λ_{k-1} или λ_k , а във всяка от останалите клаузи това е положителният $y_{i,j}$.

- В другата посока, нека β_i е удовлетворима от валуация t' . Ще покажем, че тогава за поне един литерал λ_j е вярно, че $TA(\lambda_j, t'|_{\text{Var}(\lambda_j)}) = 1$. Да допуснем противното: за всяко $j \in \{0, \dots, k\}$ е вярно, че $TA(\lambda_j, t'|_{\text{Var}(\lambda_j)}) = 0$.

Разглеждаме първата клауза на β_i

$$(\lambda_1 \vee \lambda_2 \vee y_{i,1})$$

Щом е $TA(\lambda_1, t'|_{\text{Var}(\lambda_1)}) = 0$ и $TA(\lambda_2, t'|_{\text{Var}(\lambda_2)}) = 0$, трябва да е вярно, че $TA(y_{i,1}, t'|_{\{y_{i,1}\}}) = 1$, така че $t'(y_{i,1}) = 1$.

Разглеждаме втората клауза на β_i

$$(\overline{y_{i,1}} \vee \lambda_3 \vee y_{i,2})$$

Щом $t'(y_{i,1}) = 1$, то $TA(\overline{y_{i,1}}, t'|_{\{y_{i,1}\}}) = 0$. Освен това $TA(\lambda_3, t'|_{\text{Var}(\lambda_3)}) = 0$. Тогава трябва да е вярно, че $TA(y_{i,2}, t'|_{\text{Var}(y_{i,2})}) = 1$, така че $t'(y_{i,2}) = 1$.

Разглеждаме третата клауза на β_i

$$(\overline{y_{i,2}} \vee \lambda_4 \vee y_{i,3})$$

Щом $t'(y_{i,2}) = 1$, то $TA(\overline{y_{i,2}}, t'|_{\{y_{i,2}\}}) = 0$. Освен това $TA(\lambda_4, t'|_{\text{Var}(\lambda_4)}) = 0$. Тогава трябва да е вярно, че $TA(y_{i,3}, t'|_{\{y_{i,3}\}}) = 1$, така че $t'(y_{i,3}) = 1$.

И така нататък – показваме, че за $j \in \{1, \dots, k-3\}$, $t'(y_{i,j}) = 1$. Да разгледаме последната клауза на β_i

$$(\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k)$$

Забелязваме, че $TA(\overline{y_{i,k-3}}, t'|_{\{y_{i,k-3}\}}) = TA(\lambda_{k-1}, t'|_{\text{Var}(\lambda_{k-1})}) = TA(\lambda_k, t'|_{\text{Var}(\lambda_k)}) = 0$, така че

$$TA((\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k), t'|_{\{y_{i,k-3}\} \cup \text{Var}(\lambda_{k-1}) \cup \text{Var}(\lambda_k)}) = 0$$

Тогава $t' \not\models \beta_i$. ✗

Показахме, че за поне един литерал λ_j е вярно, че $TA(\lambda_j, t'|_{\text{Var}(\lambda_j)}) = 1$. Тогава $t'|_{\text{Var}(\{\lambda_1, \dots, \lambda_k\})} \models \alpha_i$. Показахме, че ако β_i е удовлетворима, то α_i е удовлетворима.

С това изчерпахме възможните случаи. Показахме, че ϕ е удовлетворима тстк ψ е удовлетворима.

Фактът, че ψ може да бъде конструирана във време, полиномиално в размера на ϕ , е очевиден. □

Ето малък пример. Разглеждаме следната КНФ-екземпляр на SAT:

$$(x_1)(x_2 \vee \overline{x_3})(x_4 \vee \overline{x_5} \vee x_6)(\overline{x_1} \vee \overline{x_2} \vee x_7 \vee x_8 \vee \overline{x_9})$$

Съгласно конструкцията, която току-що видяхме, на нея съответства следната КНФ-екземпляр на 3SAT:

$$\begin{aligned} & (x_1 \vee y_1 \vee y_2)(x_1 \vee y_1 \vee \overline{y_2})(x_1 \vee \overline{y_1} \vee y_2)(x_1 \vee \overline{y_1} \vee \overline{y_2}) \\ & (x_2 \vee \overline{x_3} \vee z)(x_2 \vee \overline{x_3} \vee \overline{z}) \\ & (x_4 \vee \overline{x_5} \vee x_6) \\ & (\overline{x_1} \vee \overline{x_2} \vee w_1)(\overline{w_1} \vee x_7 \vee w_2)(\overline{w_2} \vee x_8 \vee \overline{x_9}) \end{aligned}$$

И двете КНФ са удовлетворими.

Допълнение 81: 2SAT $\in$ P

Задачата 2SAT се дефинира по очевидния начин.

Изч. Задача 72: 2SAT

екземпляр: КНФ ϕ , в която всяка дизюнктивна клауза има точно два литерала.

въпрос: Дали ϕ е удовлетворима?

Колкото и да е странно, 2SAT е ефикасно решима. При допускането, че $P \stackrel{?}{=} NP$, има цяла пропаст между сложността на 2SAT и сложността на 3SAT, въпреки че 2 и 3 са съседни цели числа. Сложността скача драстично, когато някакъв параметър в дефиницията на задачата от 2 става 3. Ще видим още примери за този любопитен феномен в теорията на изчислителната сложност.

- Задачата 3COLORABILITY е NP-пълна (Редукция 19), докато 2COLORABILITY е ефикасно решима чрез BFS (Подсекция 8.4.3) или DFS. И в двата случая, това, което позволява ефикасно решение при параметър 2, е, че избор на някаква стойност на някакъв елемент налага другата стойност—а тя е само една—върху друг елемент, което на свой ред форсира стойност върху трети елемент и така нататък. При оцветяването на графи, ако разполагаме само с 2 цвята, изборът на цвят на някой връх форсира другия цвят за неговите съседи, и така нататък. При 2SAT ефикасният алгоритъм не е толкова прост за обясняване, но възможността за ефикасен алгоритъм идва оттам, че във всяка клауза трябва да има литерал със стойност 1 и, ако единият не е такъв, то другият трябва да е такъв, иначе цялата КНФ не може да бъде удовлетворена; това на свой ред налага неща за други клаузи, и така нататък. В някакъв смисъл, ефикасното решение идва от дихотомията “при даден избор на стойност тук, има само един избор там”.

От друга страна, ако параметърът е 3 или повече, избор на стойност за някакъв елемент оставя 2 или повече възможности за други избори, което води до експоненциално нарастване на възможностите.

- Задачата 3-DIMENSIONAL MATCHING за намиране на перфектно съчетание в 3-униформен триделен хиперграф е NP-пълна, докато задачата СЪЧЕТАНИЕ за намиране на перфектно съчетание в обикновен граф (това е 2-униформен хиперграф), който е двуделен, е в P (Подсекция 16.2.7.4). И тук преходът от 2 към 3 води до качествен скок в сложността на задачата.

Разбира се, няма стопроцентово правило “параметър 2 означава алгоритмична податливост”. Задачата MAX2SAT (Задача 73) се явява обобщение на 2SAT и във версията

си за разпознаване е **NP**-пълна (Редукция 11).

Да видим ефикасно решение на 2SAT. То е открито от Aspvall, Plass и Tarjan и публикувано през 1978 г. [7]. Следното понятие е ключово.

Определение 148: Implication graph

Нека е дадена формула ϕ , която е екземпляр на 2SAT. Нека $\text{Var}(\phi) = \{x_1, \dots, x_n\}$. *Импликационният граф*, на английски *the implication graph*, на ϕ е ориентиранят граф $G = (V, E)$, където

$$V = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$$

$$E = \{(u, v) \mid \phi \text{ има клауза, еквивалентна на } (\neg u, v) \text{ или } (u, \neg v)\}$$

На прост български, върховете на графа са литералите над тези променливи (дори някои литерали да не се появяват във ϕ , те са върхове на графа), а всяка клауза поражда точно две ребра и други ребра няма. Ребрата се пораждат от клаузите по следния начин.

- Нека $\alpha = (x_i \vee x_j)$ е клауза на ϕ . Но $\alpha \equiv (\neg \bar{x}_i \vee x_j)$ и $\alpha \equiv (\neg \bar{x}_j \vee x_i)$. Виждаме, че α поражда ребрата $(\bar{x}_i, x_j)$ и $(\bar{x}_j, x_i)$ в графа.
- Нека $\alpha = (\bar{x}_i \vee x_j)$ е клауза на ϕ . Но $\alpha \equiv (\neg x_i \vee \bar{x}_j)$. Виждаме, че α поражда ребрата (x_i, x_j) и $(\bar{x}_j, \bar{x}_i)$ в графа.
- Нека $\alpha = (\bar{x}_i \vee \bar{x}_j)$ е клауза на ϕ . Виждаме, че α поражда ребрата $(x_i, \bar{x}_j)$ и $(x_j, \bar{x}_i)$ в графа.

Името “implication graph” идва оттам, че всяка дизюнкция $p \vee q$ е еквивалентна на импликацията $\neg p \rightarrow q$, но също така и на импликацията $\neg q \rightarrow p$. Тези две импликации са еквивалентни помежду си, естествено. Ерго, импликационният граф съдържа, под формата на ориентирани ребра, точно тези импликации (над литералите), които са еквивалентни на дизюнктивните клаузи. Ребрата на графа са два пъти повече от клаузите, понеже на всяка клауза съответстват точно две ребра.

Ако допускахме клаузи от вида $(x \vee \bar{x})$, в импликационния граф щяхме да имаме съответни ребра-примки $(\bar{x}, \bar{x})$ и (x, x) . Ако допускахме клаузи от вида $(x \vee x)$, в импликационния граф щяхме да имаме две еднакви съответни ребра $(\bar{x}, x)$ ^a. Но ние не допускаме такива клаузи, така че импликационният граф нито има примки, нито е мултиграф.

Импликационният граф има някаква симетрия – ако обърнем “поляритета” на литералите и посоката на ребрата, ще получим същия граф. Такъв граф на английски се нарича *skew symmetric*. В рамките на Допълнение 81 ще наричаме такъв граф *анти-симетричен*.

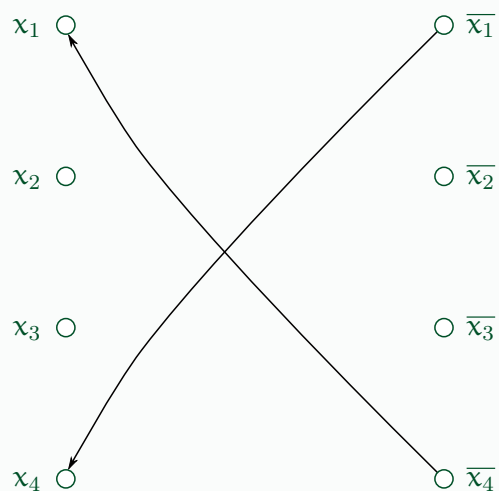
Да видим пример. Нека

$$\phi_1 = (x_1 \vee x_4)(\bar{x}_2 \vee x_4)(x_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_4)(\bar{x}_2 \vee \bar{x}_3)(x_2 \vee \bar{x}_4) \quad (16.1)$$

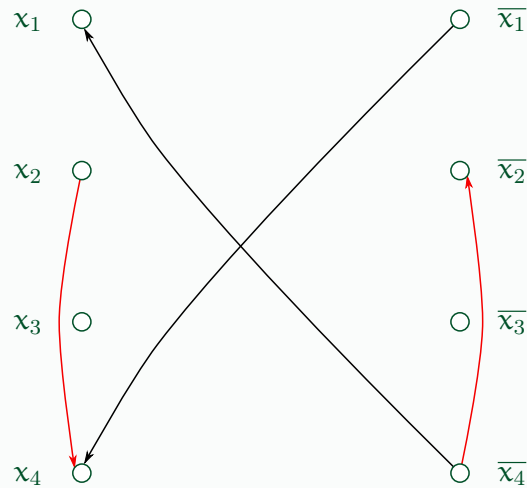
Променливите са $x_1, \dots, x_4$, което означава, че импликационният граф има следните осем върха.



Да разгледаме клаузите. Има клауза $(x_1 \vee x_4)$ и заради нея слагаме ребра $(\overline{x_1}, x_4)$ и $(\overline{x_4}, x_1)$.



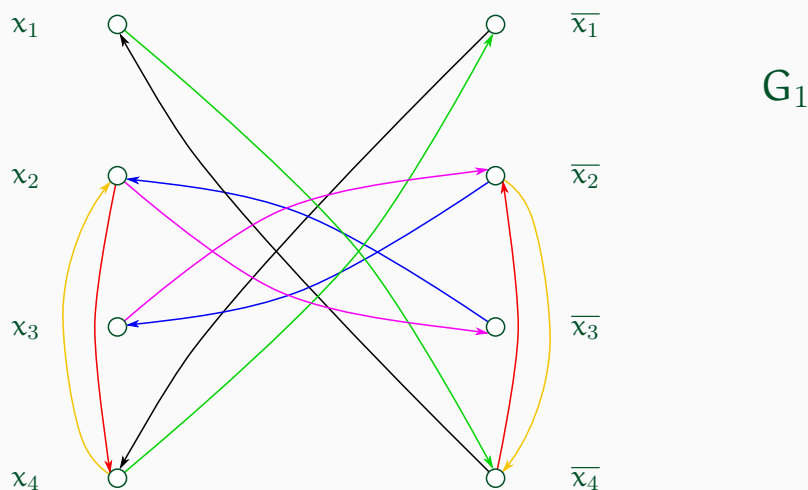
Има клауза $(\overline{x_2} \vee x_4)$ и заради нея слагаме ребра (x_2, x_4) и $(\overline{x_4}, \overline{x_2})$.



И така нататък. Целият граф G_1 е показан на Фигура 16.2, заедно с формулата.

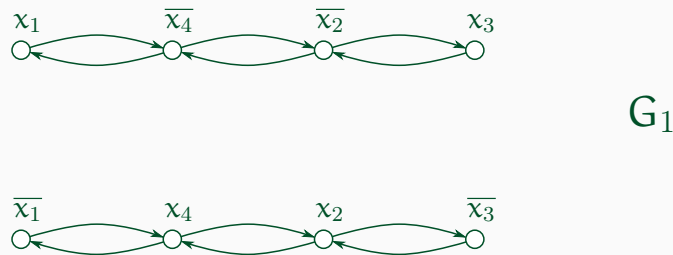
Фигура 16.2 : 2SAT формулата ϕ_1 и нейният импликационен граф G_1 .

$$\phi_1 = (x_1 \vee x_4)(\overline{x_2} \vee x_4)(x_2 \vee x_3)(\overline{x_1} \vee \overline{x_4})(\overline{x_2} \vee \overline{x_3})(x_2 \vee \overline{x_4})$$



С цветове са подчертани двойките ребра, отговарящи на една и съща клауза. Фигура 16.3 показва графа от Фигура 16.2, но нарисуван по-прегледно.

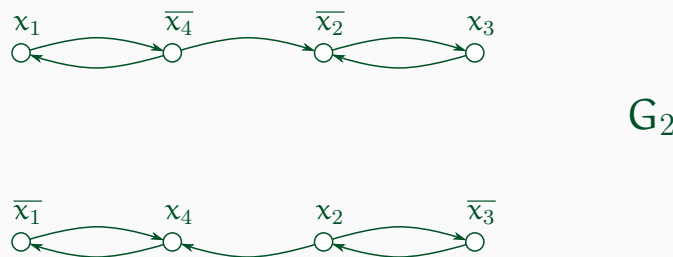
Фигура 16.3 : Графът G_1 от Фигура 16.2, нарисуван по-прегледно.



Графът G_1 от Фигура 16.3 не е добра илюстрация на антисиметричността на графа, понеже при него само обръщането на поляритета на литералите води до същия граф G_1 , както и само обръщане на посоката на ребрата води до същия граф G_1 . Но ако премахнем една клауза от ϕ_1 от (16.1), да кажем $(x_2 \vee \bar{x}_4)$, ще получим КНФ ϕ_2 , чийто импликационен граф ще наречем G_2 . За G_2 е вярно, че трябва и да обърнем поляритета на литералите, и да обърнем посоката на ребрата, за да получим същия граф. Вижте Фигура 16.4.

Фигура 16.4 : Графът от Фигура 16.2 след махане на $(x_2 \vee \bar{x}_4)$ от ϕ_1 .

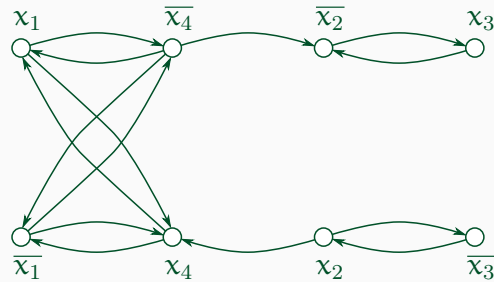
$$\phi_2 = (x_1 \vee x_4)(\bar{x}_2 \vee x_4)(x_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_4)(\bar{x}_2 \vee \bar{x}_3)$$



Ще казваме, че импликационен граф е *удачен*, ако няма променлива, чиито два литерала са силно свързани. Импликационните графи на Фигура 16.3 и Фигура 16.4 са удачни. Фигура 16.5 показва неудачен импликационен граф G_3 . Той е импликационният граф на формулата $\phi_3 = \phi_2(\bar{x}_1 \vee x_4)(x_1 \vee \bar{x}_4)$.

Фигура 16.5 : Неудачен импликационен граф.

$$\phi_3 = (x_1 \vee x_4)(\bar{x}_2 \vee x_4)(x_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_4)(\bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_4)(x_1 \vee \bar{x}_4)$$

 G_3

Забележете, че в удачните G_1 и G_2 , силно свързаните компоненти се групират по двойки, като във всяка двойка, литералите са на едни и същи променливи, но са (литералите) с обратна полярност. Ето как изглеждат групиранията в контекста на класовете на еквивалентност на релацията $\text{SC}()$:

$$\begin{aligned} \text{SCC}(G_1) &= \underbrace{\{\{x_1, \bar{x}_2, x_3, \bar{x}_4\}, \{\bar{x}_1, x_2, \bar{x}_3, x_4\}\}}_{\text{една двойка}} \\ \text{SCC}(G_2) &= \underbrace{\{\{x_1, \bar{x}_4\}, \{\bar{x}_1, x_4\}\}}_{\text{една двойка}} \underbrace{\{\{\bar{x}_2, x_3\}, \{x_2, \bar{x}_3\}\}}_{\text{друга двойка}} \end{aligned}$$

В контраст на това,

$$\text{SCC}(G_3) = \{\{x_1, \bar{x}_1, x_4, \bar{x}_4\}, \{\bar{x}_2, x_3\}, \{x_2, \bar{x}_3\}\}$$

и такова групиране на силно свързаните компоненти (или, ако предпочитате, класовете на еквивалентност) по двойки няма. Причината не е, че броят на силно свързаните компоненти е нечетен. Сами се убедете, че ако конкатенираме ϕ_3 с “ $(x_2 \vee \bar{x}_3)(\bar{x}_2 \vee x_3)$ ”, съответният импликационен граф ще има точно две силно свързани компоненти, като класовете на еквивалентност ще са

$$\{\{x_1, \bar{x}_1, x_4, \bar{x}_4\}, \{x_2, \bar{x}_2, x_3, \bar{x}_3\}\}$$

Въпреки че броят е четен, не можем да групираме тези множества по двойки, така че във всяка двойка литералите са на едни и същи променливи, но са с обратна полярност. Дали можем или не можем да направим такова групиране по двойки на класовете на еквивалентност зависи само от това, дали графът е удачен, или не.

Наблюдение 106

Импликационен граф G е удачен тстк съществува групиране на елементите на $\text{SCC}(G)$ по двойки, такова че за всяка двойка е вярно, че елементите на единия елемент се получават от елементите на другия елемент чрез обръщане на поляритета. Това позволява да дефинираме по най-естествения начин—ако G е удачен—за всеки елемент на $\text{SCC}(G)$ неговият съответен елемент; респективно, за всяка силно свързана компонента на G , нейната съответна силно свързана компонента.

Примерно, на класа на еквивалентност $\{x_1, \bar{x}_2, x_3, \bar{x}_4\}$ от $\text{SCC}(G_1)$, съответният клас е $\{\bar{x}_1, x_2, \bar{x}_3, x_4\}$; в контекста на силно свързаните компоненти на G_1 , едната и другата са взаимно съответни. По отношение на G_2 , класовете $\{x_1, \bar{x}_4\}$ и $\{\bar{x}_1, x_4\}$ са взаимно съответни; в контекста на силно свързаните компоненти на G_2 , силно свързаните компоненти, индуцирани от тези класове, са взаимно съответни.

Лема 57

Нека $X = \{x_1, \dots, x_n\}$ е множество от булеви променливи и ϕ е КНФ над X с по точно два литерала в клауза. За всяка валюация t на X е вярно, че $t \models \phi$ тстк съществува ребро $(a, b) \in E$, такова че $\text{TA}(a, t|_{\text{Var}(a)}) = 1$ и $\text{TA}(b, t|_{\text{Var}(b)}) = 0$.

Доказателство: Първо забелязваме, че в графа има ребро (a, b) тстк ϕ съдържа клауза α , еквивалентна на $(\neg a \vee b)$.

Странична забележка: не е коректно да се каже “ ϕ съдържа клаузата $(\bar{a} \vee b)$ ”, понеже a и b са литерали, а не променливи. Ако a е отрицателен литерал $\bar{x}_i$ за някое $x_i \in X$, то ϕ съдържа не $(\bar{x}_i, b)$, а просто (x_i, b) . И не е коректно да се каже “ ϕ съдържа клауза $(\neg a \vee b)$ ”, понеже буквата ‘ $\neg$ ’ не е в азбуката на КНФ. Край на забележката.

Да допуснем, че съществува ребро $(a, b) \in E$, такова че $\text{TA}(a, t|_{\text{Var}(a)}) = 1$ и $\text{TA}(b, t|_{\text{Var}(b)}) = 0$. Тогава за клаузата α , еквивалентна на $(\neg a \vee b)$, е вярно, че $\text{TA}(\alpha, t|_{\text{Var}(\alpha)}) = 0$. Тогава $t \not\models \phi$.

Да допуснем, че за всяко ребро $(a, b) \in E$ е вярно точно едно от следните. Нека α е клаузата във ϕ , съответстваща на (a, b) в графа; както вече отбелязахме, $\alpha \equiv (\neg a \vee b)$.

- $\text{TA}(a, t|_{\text{Var}(a)}) = 0$ и $\text{TA}(b, t|_{\text{Var}(b)}) = 0$. Тогава $\text{TA}(\alpha, t|_{\text{Var}(\alpha)}) = 1$ и $t \models \phi$.
- $\text{TA}(a, t|_{\text{Var}(a)}) = 0$ и $\text{TA}(b, t|_{\text{Var}(b)}) = 1$. Тогава $\text{TA}(\alpha, t|_{\text{Var}(\alpha)}) = 1$ и $t \models \phi$.
- $\text{TA}(a, t|_{\text{Var}(a)}) = 1$ и $\text{TA}(b, t|_{\text{Var}(b)}) = 1$. Тогава $\text{TA}(\alpha, t|_{\text{Var}(\alpha)}) = 1$ и $t \models \phi$. □

Теорема 129: Екземпляр на 2SAT е удовлетворим тстк импл. граф е удачен

Нека ϕ е екземпляр на 2SAT. ϕ е удовлетворима тстк импликационният ѝ граф е удачен.

Доказателство, 1: В едната посока, нека ϕ е удовлетворима. Да допуснем, че съществува $x \in \text{Var}(\phi)$, такова че в G съществуват пътища p и q , такива че $x \xrightarrow{p} \bar{x}$ и $\bar{x} \xrightarrow{q} x$. Щом ϕ е удовлетворима, съществува удовлетворяваща валюация t на $\text{Var}(\phi)$. Да разгледаме възможностите за $t(x)$ поотделно.

- Да допуснем, че $t(x) = 1$. Тогава $TA(x, t|_{\{x\}}) = 1$ и $TA(\bar{x}, t|_{\{x\}}) = 0$. Да разгледаме p . Валюацията t асоциира всеки връх на p или с 0, или с 1. Началният връх x има стойност 1, а крайният връх $\bar{x}$ има стойност 0. Очевидно съществува ребро (u, v) в p , такова че $TA(u, t|_{Var(u)}) = 1$, а $TA(v, t|_{Var(v)}) = 0$. Съгласно Лема 57, $t \neq \phi$. ⚡
- Да допуснем, че $t(x) = 0$. Разглеждаме пътя q от $\bar{x}$ до x . Аналогично на предишния случай, заключаваме, че в q има ребро (u, v) , такова че $TA(u, t|_{Var(u)}) = 1$ и $TA(v, t|_{Var(v)}) = 0$. Съгласно Лема 57, $t \neq \phi$. ⚡

Доказателство, 2: В другата посока, нека не съществува променлива x , чиито положителен и отрицателен литерал са силно свързани в G . Следният алгоритъм строи удовлетворяваща валюация t на $Var(\phi)$.

ALG2SAT(Удачен импликационен граф G на 2SAT формула ϕ)

```

1  построяваме  $G/\mathcal{SC}(G)$ 
2  сортираме  $G/\mathcal{SC}(G)$  топологически
3  foreach  $H \in \mathcal{SCC}(G)$  в реда на топологическото сортиране
4      foreach  $\lambda \in H$ 
5          (*  $\lambda$  е литерал на  $\phi$  *)
6          нека  $x$  е променливата на  $\lambda$ 
7          if  $t(x)$  не е дефинирана
8              if  $\lambda$  е положителен литерал
9                   $t(x) \leftarrow 0$ 
10             else
11                  $t(x) \leftarrow 1$ 
12  return  $t$ 

```

Ще докажем коректността на алгоритъма ALG2SAT. Както знаем от Наблюдение 56, $G/\mathcal{SC}(G)$ е даг, така че топологическото му сортиране на ред 2 е добре дефинирано.

Трябва да докажем, че на ред 12, t е валюация на променливите на ϕ , удовлетворяваща ϕ . Това, че t е валюация на променливите на ϕ , е очевидно: всяка променлива се среща точно два пъти като литерал-върх на G (положителен и отрицателен), а от друга страна всеки литерал в даден момент е съдържанието на променливата λ на ред 4. Ерго, всяка променлива бива “обработена” и получава стойност 0 (ред 9) или 1 (ред 11) съгласно поляритета на първия си, в реда на топо-сортирането, литерал.

Остава да докажем, че t е удовлетворяваща валюация. Неформално, това е така, понеже G е удачен и силно свързаните му компоненти са групирани по двойки на взаимно съответствие (Наблюдение 106). След топо-сортирането на фактор-графа, за всяка двойка взаимно съответни елементи на $\mathcal{SCC}(G)$ е вярно, че единият елемент, да речем Y , е преди другия, да речем Z , в топо-сортирането. В главния цикъл (редове 3–11), променливите на литералите от Y получават такива стойности, че литералите му да са нули. Като автоматично следствие, литералите на Z стават единици още преди H да е получил стойност Z на ред 3.

- Тъй като графът е удачен, Y не съдържа положителен и отрицателен литерал на една и съща променлива, така че в рамките на Y няма опит за даване и на нула, и на единица на една и съща променлива. Ерго, в рамките на Y , всичко е наред.
- От това, че литералите на Y са станали нули, не може да се получи ребро (u, v) , такова че u е литерал от Y , v е литерал не от Y , $TA(u, t|_{\text{Var}(u)}) = 1$ и $TA(v, t|_{\text{Var}(v)}) = 0$. Причината е, че фактор-графът е топо-сортиран и всяко ребро с начало литерал от Y и край, който не е от Y , води надясно от Y ; ерго,
 - ♦ или $TA(v, t|_{\text{Var}(v)})$ не е дефинирана в момента, в който променливата на u получава своята стойност,
 - ♦ или v е вече е “направена” единица от това, че на нейната променлива е дадена такава стойност, че литералът, еквивалентен на $\neg v$, който вече е бил “обработен”, е бил “направен” да е нула.

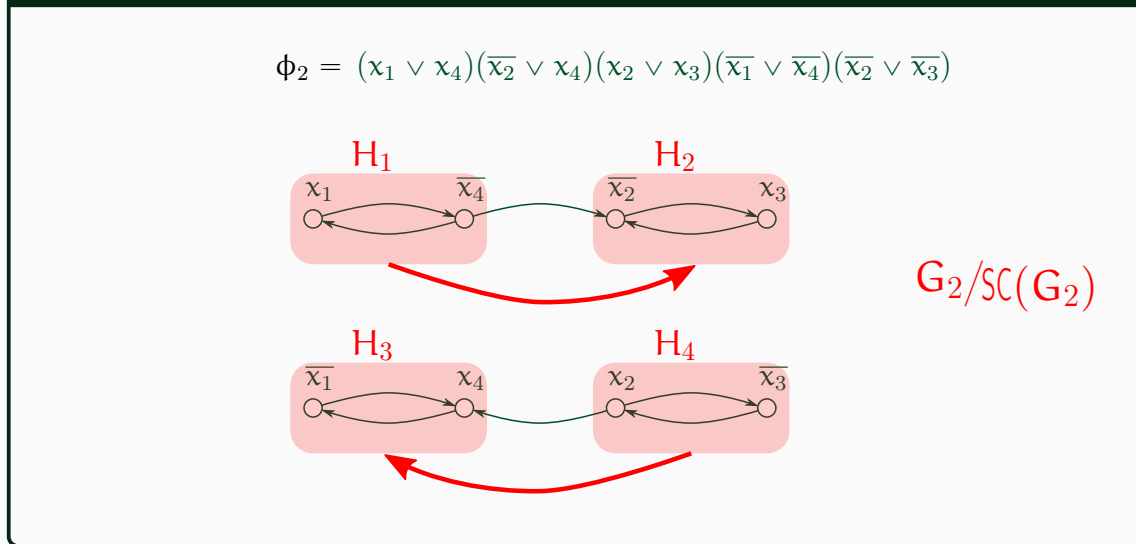
И в двата случая няма как стигнем до ситуацията $TA(u, t|_{\text{Var}(u)}) = 1$ и $TA(v, t|_{\text{Var}(v)}) = 0$, която е “гибелна” за удовлетворяването съгласно Лема 57.

Формалната верификация на ALG2SAT остава за читателя. □

Да видим пример за работата на ALG2SAT. Нека входът е графът G_2 от Фигура 16.4. Той е импликационният граф на $\phi_2 = (x_1 \vee x_4)(\bar{x}_2 \vee x_4)(x_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_4)(\bar{x}_2 \vee \bar{x}_3)$. На Фигура 16.6 е показан фактор-графът му

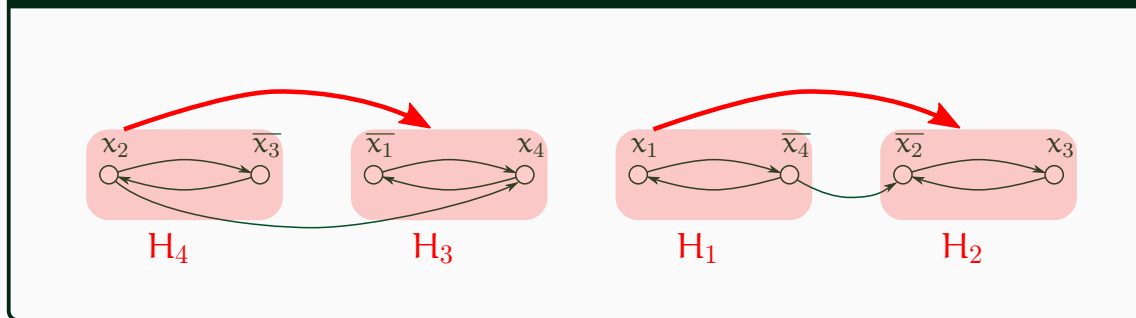
$$G_2/SC(G_2) = (\{H_1, H_2, H_3, H_4\}, \{(H_1, H_2), (H_4, H_3)\})$$

Фигура 16.6 : Фактор-графът на графа от Фигура 16.4.

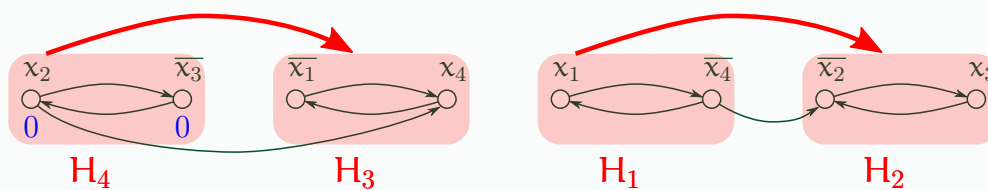


Да топо-сортираме фактор-графа. Фигура 16.7 показва възможен резултат.

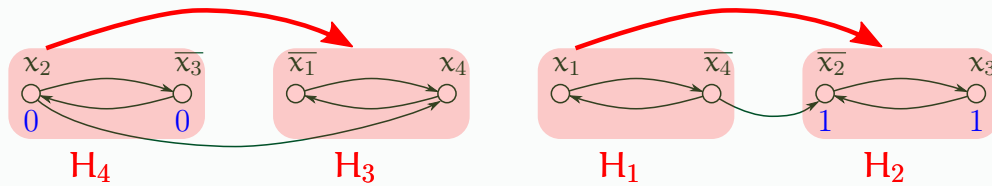
Фигура 16.7 : Топо-сортиране на фактор-графа от Фигура 16.6.



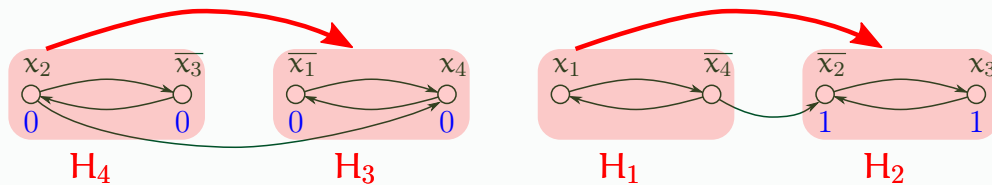
При първата итерация на главния цикъл (редове 3–11), променливата H получава стойност H_1 . Алгоритъмът “нулира” и двата литерала x_2 и $\bar{x}_3$:



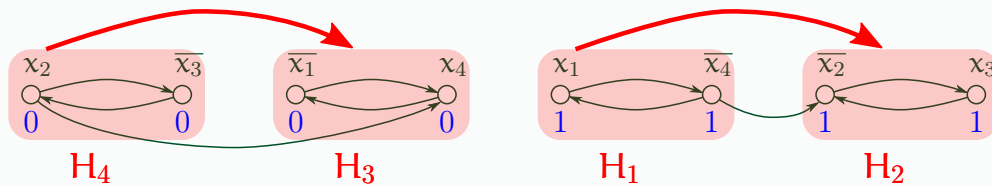
Това става с $t(x_2) \leftarrow 0$ и $t(x_3) \leftarrow 1$. Забележете, че H_2 е силно свързаната компонента, съответна на H_4 , така че даването на стойности нула на литералите на H_4 дава стойности единица на литералите в H_2 :



При втората итерация на главния цикъл (редове 3–11), променливата H получава стойност H_3 . Алгоритъмът “нулира” и двата литерала $\bar{x}_1$ и x_4 :



Това става с $t(x_1) \leftarrow 1$ и $t(x_4) \leftarrow 0$. Забележете, че H_1 е силно свързаната компонента, съответна на H_3 , така че даването на стойности нула на литералите на H_3 дава стойности единица на литералите в H_1 :



Алгоритъмът изпълнява още две итерации на главния цикъл, но при тях не се случва нищо, понеже променливите от литералите на H_1 и H_2 вече имат стойности. Алгоритъмът връща $t = \langle 1, 0, 1, 0 \rangle$. Това е удовлетворяваща валюация за ϕ_2 . Ето как става удовлетворяването. Във всяка клауза има поне един “удовлетворен литерал”, който е в син цвят.

$$\phi_2 = (x_1 \vee x_4)(\bar{x}_2 \vee x_4)(x_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_4)(\bar{x}_2 \vee \bar{x}_3)$$

И накрая отбелязваме, че конструктивното доказателство на Теорема 129 ни дава възможност не просто да изчислим дали дадена 2SAT формула е удовлетворима, а и, ако е удовлетворима, да изчислим една удовлетворяваща валюация. Нещо повече: ALG2SAT има линейна сложност по време, което е очевидно предвид линейните сложности на

алгоритмите за построяване на фактор-графа и топо-сортирането му. Ерго, можем да получим удовлетворяваща валюация, стига да има такава, във време, което е асимптотично оптимално.

^aАко настояваме импликационният граф да не е мултиграф и допускаме клаузи от вида $(x \vee x)$, няма да е вярно, че ребрата са два пъти колкото клаузите, понеже тогава $(x \vee x)$ поражда точно едно ребро.

MAX2SAT $\in$ NP-с. Задачата MAX2SAT е естествено обобщение на 2SAT. За разлика от 2SAT, тя е неподатлива. На прост български, задачата е в екземпляр на 2SAT да се намерят максимален брой клаузи, които са удовлетворими едновременно. Във версия за разпознаване, задачата е NP-пълна.

Изн. Задача 73: MAX2SAT, ОПТИМИЗАЦИОННА ВЕРСИЯ

екземпляр: КНФ ϕ , в която всяка дизюнктивна клауза има точно два литерала.

решение: КНФ ψ , чието множество от клаузи е подмножество на множеството от клаузите на ϕ , която е удовлетворима и има максимален брой клаузи.

Изн. Задача 74: MAX2SAT, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: КНФ ϕ , в която всяка дизюнктивна клауза има точно два литерала, естествено число k .

Въпрос: Дали съществува КНФ ψ , чието множество от клаузи е подмножество на множеството от клаузите на ϕ , която е удовлетворима и има поне k клаузи?

Естествено, имаме предвид MAX2SAT във версия за разпознаване.

Редукция 11: $3SAT \leq_p MAX2SAT$

Конструкция: Тази редукция има една несъществена особеност: в екземпляра на MAX2SAT се допускат клаузи с точно един литерал като “ (x) ”. С други думи, формулировката на задачата е “всяка клауза има **най-много** два литерала”. Или, ако предпочитате, всяка клауза има точно два литерала, но се допускат клаузи от вида “ $(x \vee x)$ ”. Възможността за клаузи с по един литерал наистина е несъществена: ако не допускаме клаузи с еднакви литерали и държим на **точно** два литерала в клауза, можем да преобразуваме всяка (x) в $(x \vee z)(x \vee \bar{z})$, където z е нова променлива. Също както в Редукция 10, това е еквивалентно преобразуване.

Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

където

$$\phi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee \lambda_{3,i})$$

Ще конструираме съответен екземпляр $\langle \psi, k \rangle$ на MAX2SAT. За всяка клауза $\phi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee \lambda_{3,i})$ на ϕ конструираме КНФ ψ_i , такава че:

$$\psi_i \equiv (\lambda_{1,i})(\lambda_{2,i})(\lambda_{3,i})(z_i)(\neg\lambda_{1,i} \vee \neg\lambda_{2,i})(\neg\lambda_{1,i} \vee \neg\lambda_{3,i})(\neg\lambda_{2,i} \vee \neg\lambda_{3,i})(\lambda_{1,i} \vee \bar{z}_i)(\lambda_{2,i} \vee \bar{z}_i)(\lambda_{3,i} \vee \bar{z}_i)$$

където z_i е нова променлива, която няма да ползваме в никоя друга ψ_j . Желаната КНФ ψ е

$$\psi = \bigvee_{i=1}^m \psi_i$$

а k е $7m$. Забележете, че ψ има точно $10m$ клаузи.

Ще докажем, че ϕ е ДА-екземпляр на 3SAT тстк $\langle \psi, k \rangle$ е ДА-екземпляр на MAX2SAT. Следните помощни твърдения са ключови.

Лема 58

Нека β е следната КНФ с десет клаузи:

$$\beta = \underbrace{(x_1)(x_2)(x_3)(z)}_{\text{група 1}} \underbrace{(\bar{x}_1 \vee \bar{x}_2)(\bar{x}_1 \vee \bar{x}_3)(\bar{x}_2 \vee \bar{x}_3)}_{\text{група 2}} \underbrace{(x_1 \vee \bar{z})(x_2 \vee \bar{z})(x_3 \vee \bar{z})}_{\text{група 3}}$$

Не съществува $t \in \text{Val}(\text{Var}(\beta))$, такава че t удовлетворява повече от седем клаузи на β . Съществуват валюации, които удовлетворяват точно седем клаузи, но за всяка такава валюация t , за поне едно $i \in \{1, 2, 3\}$ е изпълнено $t(x_i) = 1$.

Доказателство: Това, че всички десет клаузи не могат да бъдат удовлетворени едновременно, е очевидно. Сега да видим защо седем е точната горна граница за броя на удовлетворените клаузи и защо тя е достижима тстк поне едно x_i получава стойност 1. Аргументацията е с изчерпателно разглеждане на случаи и подслучаи. Забележете, че клаузите са симетрични по отношение на x_1 , x_2 и x_3 , така е достатъчно да се разгледат бройките на единици върху тези променливи.

1. Ако направим $t(x_1) = t(x_2) = t(x_3) = 1$, губим група 2.
 - Ако направим $t(z) = 1$, седемте клаузи от групи 1 и 3 са удовлетворени. ✓
 - Ако направим $t(z) = 0$, клаузата (z) от група 1 не е удовлетворена, а група 3 е удовлетворена, така че са удовлетворени точно шест клаузи.
2. Ако направим $t(x_1) = t(x_2) = 1$ и $t(x_3) = 0$, губим клаузите (x_3) и $(\bar{x}_1 \vee \bar{x}_2)$, така че имаме пет удовлетворени клаузи от групи 1 и 2.
 - Ако направим $t(z) = 1$, не губим повече от група 1, но губим $(x_3 \vee \bar{z})$ от група 3, така че удовлетворяваме точно седем клаузи. ✓
 - Ако направим $t(z) = 0$, губим клаузата (z) от група 1, но не губим нищо от група 3, така че удовлетворяваме точно седем клаузи. ✓
3. Ако направим $t(x_1) = 1$ и $t(x_2) = t(x_3) = 0$, губим клаузите (x_2) и (x_3) от група 1. От група 2 не губим нищо. Дотук имаме четири удовлетворени клаузи.
 - Ако направим $t(z) = 1$, имаме още една удовлетворена клауза в група 1, но губим $(x_2 \vee \bar{z})$ и $(x_3 \vee \bar{z})$ от група 3, така че удовлетворяваме точно шест клаузи.
 - Ако направим $t(z) = 0$, губим клаузата (z) от група 1, но не губим нищо от група 3, така че удовлетворяваме точно седем клаузи. ✓
4. Ако направим $t(x_1) = t(x_2) = t(x_3) = 0$, губим клаузите (x_1) , (x_2) и (x_3) от група 1.

- Ако направим $t(z) = 1$, не губим повече от група 1, но губим цялата група 3 с нейните три клаузи, така че не можем да удовлетворим седем клаузи.
- Ако направим $t(z) = 0$, губим и клаузата (z) от група 1 и пак не можем да удовлетворим седем клаузи. $\square$

Следствие 41

Нека x_1, x_2, x_3 и z са булеви променливи, а λ_i е литерал над x_i , за $1 \leq i \leq 3$. Нека β е КНФ с десет клаузи, еквивалентна на

$$\beta = (\lambda_1)(\lambda_2)(\lambda_3)(z)(\neg\lambda_1 \vee \neg\lambda_2)(\neg\lambda_1 \vee \neg\lambda_3)(\neg\lambda_2 \vee \neg\lambda_3)(\lambda_1 \vee \bar{z})(\lambda_2 \vee \bar{z})(\lambda_3 \vee \bar{z})$$

Седем е точната горна граница за броя на удовлетворените клаузи на β от произволна валюация на $\{x_1, x_2, x_3, z\}$. Нещо повече: тази горна граница е достижима само от валюация $t \in \text{Val}(\text{Var}(\beta))$, за която съществува $i \in \{1, 2, 3\}$, такова че $\text{TA}(\lambda_i, t|_{\text{Var}(\lambda_i)}) = 1$. $\square$

Ето обосновката на коректността.

- Да допуснем, че ϕ е удовлетворима. Разглеждаме произволна удовлетворяваща $t \in \text{Val}(\text{Var}(\phi))$.

Разглеждаме произволно $i \in \{1, \dots, m\}$. Нека $t_i = t|_{\text{Var}(\phi_i)}$. Щом $t \models \phi$, за някакво непразно $X \subseteq \{\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i}\}$ е вярно, че t удовлетворява точно елементите на X . Тогава е вярно, че t_i удовлетворява точно елементите на X .

И така, имаме клауза $\phi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee \lambda_{3,i})$ и валюация t_i на нейните променливи, която валюацията удовлетворява точно някакво непразно подмножество X на нейните литерали. От Следствие 41 знаем, че по отношение на съответната КНФ ψ_i , за всяко непразно $Y \subseteq \{\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i}\}$ съществува валюация на променливите на ψ_i , удовлетворяваща седем от десетте клаузи на ψ_i и литералите-елементи на Y . Тогава за нашето доказателство вземаме такава $t'_i \in \text{Val}(\text{Var}(\psi_i))$, че $t'_i|_{\text{Var}(\phi_i)} = t_i$, а $t'_i(z_i)$ има стойност, гарантираща, че точно седем клаузи на ψ_i биват удовлетворени от t'_i .

Относно $t'_i(z_i)$: забележете, че в доказателството на Лема 58, в случаи 1, 2 и 3, когато седем клаузи са удовлетворими, $t(z)$ не е фиксирано на 0 или 1! В случай 1, $t(z)$ трябва да е 1, в случай 2 е все едно дали е 0 или 1, а в случай 3, $t(z)$ трябва да е 0. Така че, когато построяваме $t'_i(z_i)$, дали ще дадем стойност 0 или 1 зависи от това, колко от ламбдите са удовлетворени от t_i .

Разглеждаме валюацията $t' \in \text{Val}(\text{Var}(\psi))$, която се образува от комбинирането на $t'_1, \dots, t'_m$. По-формално казано, $t'|_{\text{Var}(\psi_i)} = t'_i$, за $1 \leq i \leq m$. Очевидно t' удовлетворява точно $7m$ клаузи на ψ . Тогава $\langle \psi, 7m \rangle$ е ДА-екземпляр на MAX2SAT.

- Да допуснем, че ϕ е неудовлетворима. Да разгледаме произволна валюация $t \in \text{Val}(\text{Var}(\phi))$. Съществува клауза ϕ_i , която за всеки литерал $\lambda_{j,i}$ е вярно, че $\text{TA}(\lambda_{j,i}, t|_{\text{Var}(\lambda_{j,i})}) = 0$. Съгласно Следствие 41, за всяка валюация на $\text{Var}(\psi_i)$ е вярно, че тя удовлетворява не повече от шест клаузи на ψ_i . Заклучаваме, че както валюацията $t|_{\text{Var}(\phi_i)} \cup \{(z_i, 0)\}$, така и валюацията $t|_{\text{Var}(\phi_i)} \cup \{(z_i, 1)\}$ удовлетворяват не повече от шест клаузи на ψ_i .

Пак съгласно Следствие 41, за всяко $j \in \{1, \dots, m\} \setminus \{i\}$, всяка валюация $t_j \in \text{Val}(\text{Var}(\psi_j))$ удовлетворява най-много седем от десетте клаузи на ψ_j . Заклучаваме, че t удовлетворява по-малко от $7m$ клаузи на ψ . Тогава $\langle \psi, 7m \rangle$ е НЕ-екземпляр на MAX2SAT.

А това, че редукцията може да се извърши в полиномиално време, е очевидно. $\square$

NAESAT $\in$ **NP-с**. Доколкото е известно на автора на лекционните записки, за разлика от SAT, която е има огромни приложения в практиката, задачата NAESAT няма кой знае какви директни приложения в практиката, но е полезно средство в теорията за доказване на неподатливост на други задачи.

Определение 149: NAE-удовлетворимост

Нека ϕ е КНФ с поне два литерала във всяка клауза. Казваме, че ϕ е NAE-удовлетворима, ако съществува $t \in \text{Val}(\text{Var}(\phi))$, такава че за всяка клауза α на ϕ е вярно, че

- α съдържа литерал λ , такъв че $\text{TA}(\lambda, t|_{\text{Var}(\lambda)}) = 1$ и
- α съдържа литерал μ , такъв че $\text{TA}(\mu, t|_{\text{Var}(\mu)}) = 0$.

На английски терминът е *NAE-satisfiable*. “NAE” идва от “not all equal”. Пример за КНФ, която е NAE-удовлетворима, е

$$\phi_1 = (x_1 \vee x_2)$$

например от валюацията $\langle 1, 0 \rangle$. Пример за КНФ, която не е NAE-удовлетворима, е

$$\phi_1 = (x_1 \vee x_2)(x_1 \vee \overline{x_2})$$

Ако $t(x_1) = 0$, трябва $t(x_2)$ да е такава, че за всеки литерал λ на x_2 да е изпълнено $\text{TA}(\lambda, t|_{\{x_2\}}) = 1$, което е невъзможно. Ако $t(x_1) = 1$, трябва $t(x_2)$ да е такава, че за всеки литерал λ на x_2 да е изпълнено $\text{TA}(\lambda, t|_{\{x_2\}}) = 0$, което е невъзможно.

Съответната изчислителна задача е неподатлива, ако всяка клауза има поне три литерала. БОО, нека са точно три.

Изч. Задача 75: NAESAT

екземпляр: КНФ ϕ , в която всяка дизюнктивна клауза има точно три литерала.

Въпрос: Дали ϕ е NAE-удовлетворима?

Естествено, имаме предвид NAESAT във версия за разпознаване.

Редукция 12: $3\text{SAT} \leq_p \text{NAESAT}$

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

където

$$\phi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee \lambda_{3,i})$$

Ще конструираме съответен екземпляр ψ на NAESAT. За всяка клауза $\phi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee \lambda_{3,i})$ на ϕ , построяваме три нови променливи x_i , y_i и z_i , които няма да се ползват другаде, и построяваме КНФ

$$\psi_i = (\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

Желаната КНФ ψ е

$$\psi = \bigvee_{i=1}^m \psi_i$$

Ще докажем, че ϕ е ДА-екземпляр на 3SAT тстк $\langle \psi, k \rangle$ е ДА-екземпляр на NAESAT. В едната посока, нека ϕ е удовлетворима. Разглеждаме произволна валюация $t \in \text{Val}(\text{Var}(\phi))$, която удовлетворява ϕ . Разглеждаме произволно $i \in \{1, \dots, m\}$. Предвид симетричното участие на $\lambda_{1,i}$ и $\lambda_{2,i}$ в ψ_i , следните възможности са изчерпателни.

- $\text{TA}(\lambda_{1,i}, t|_{\text{Var}(\lambda_{1,i})}) = \text{TA}(\lambda_{2,i}, t|_{\text{Var}(\lambda_{2,i})}) = 1$. Всяка валюация $t'_i \in \text{Val}(\text{Var}(\psi_i))$, където $t(x_i) = t(y_i) = 0$ NAE-удовлетворява ψ_i . Казваме “всяка”, понеже стойностите на $t(\text{Var}(\lambda_{3,i}))$ и $t(z_i)$ са без значение. Ето как става NAE-удовлетворяването; литералите със стойност 1 са в зелено, литералите със стойност 0 са в червено, а в черно са тези с произволна стойност.

$$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

- $\text{TA}(\lambda_{1,i}, t|_{\text{Var}(\lambda_{1,i})}) = 1$ и $\text{TA}(\lambda_{2,i}, t|_{\text{Var}(\lambda_{2,i})}) = 0$. Правим $t(x_i) = t(y_i) = 0$. Отново ψ_i може да бъде NAE-удовлетворена.

$$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

- $\text{TA}(\lambda_{1,i}, t|_{\text{Var}(\lambda_{1,i})}) = \text{TA}(\lambda_{2,i}, t|_{\text{Var}(\lambda_{2,i})}) = 0$. Правим $t(x_i) = t(y_i) = 1$. Отново ψ_i може да бъде NAE-удовлетворена.

$$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

В другата посока, нека ψ е NAE-удовлетворима. Разглеждаме произволна валюация $t' \in \text{Val}(\text{Var}(\psi))$, която NAE-удовлетворява ψ . Разглеждаме произволно $i \in \{1, \dots, m\}$.

- Ако за поне един литерал $\lambda_{j,i}$, където $j \in \{1, 2, 3\}$, е вярно, че $\text{TA}(\lambda_{j,i}, t'|_{\text{Var}(\lambda_{j,i})}) = 1$, очевидно ϕ_i е удовлетворена.
- Да допуснем, че $\text{TA}(\lambda_{1,i}, t'|_{\text{Var}(\lambda_{1,i})}) = \text{TA}(\lambda_{2,i}, t'|_{\text{Var}(\lambda_{2,i})}) = \text{TA}(\lambda_{3,i}, t'|_{\text{Var}(\lambda_{3,i})}) = 0$. За да бъде ψ_i NAE-удовлетворена, трябва $t'(x_i) = t'(y_i) = 1$:

$$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

Но тогава ψ_i е NAE-удовлетворена и от валюацията $\overline{t'}$, която е обратната на t' :

$$\forall w \in \text{Var}(\psi) : \overline{t'}(w) = \neg t'(w)$$

На прост български, обръщаме стойностите на t' и получаваме валюация, която също NAE-удовлетворява цялата ψ . За да се убедим в това, разглеждаме само какво става в ψ_i :

$$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

Но тогава $\overline{t'}|_{\text{Var}(\phi)} \models \phi$.

Това, че конструкцията може да се извърши в полиномиално време, е очевидно. Малка оптимизация е да не ползваме отделни z_i , а само една нова променлива z , която да присъства във всяка ψ_i . $\square$

16.2.2 MAXCUT

Тегло на срез в граф е броят на ребрата, прекосяващи среза (Определение 90). Обобщаваме тези понятия за неориентирани мултиграфи по естествения начин – при наличие на паралелни ребра между два върха u и v в различни дялове на среза, добавяме броя на тези паралелни ребра към теглото на среза. Като оптимизационна задача, MAXCUT е: в даден неориентиран мултиграф граф да се намери срез с максимално тегло. Задачата има изненадващи приложения в области като VLSI design и статистическата физика [10]. За съжаление, или за щастие, задачата е неподатлива, както ще видим тук.

Заслужава да се отбележи, че задачата MINCUT за намиране на срез с минимално тегло е решима в полиномиално време чрез алгоритъм за намиране на максимален поток в граф. Също така си заслужава да се отбележи, че MAXCUT е решима в полиномиално време върху планарните графи, макар и от сложен за разбиране и имплементиране алгоритъм [65].

Изч. Задача 76: MAXCUT, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран мултиграф G , естествено число k .

въпрос: Дали в G има срез с тегло $\geq k$?

Редукция 13: $NAESAT \leq_p MAXCUT$

Конструкция: Нека е даден екземпляр ϕ на NAESAT. Ще конструираме мултиграф $G = (V, E, f)$ и число k , такива че ϕ е NAE-удовлетворима тстк G има срез с тегло $\geq k$. Нека $Var(\phi) = \{x_1, \dots, x_n\}$ и ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

където

$$\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$$

Тогава

$$V = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$$

Забележете, че всички литерали на променливите присъстват във V , независимо от това, че ϕ може да не съдържа и двата литерала на някоя променлива.

Нека ℓ_i е общият брой на срещанията на литералите x_i и $\bar{x}_i$ във ϕ , за всяко $i \in \{1, \dots, n\}$. Очевидно

$$\sum_{i=1}^n \ell_i = 3m \tag{16.2}$$

тъй като всяка клауза има точно три литерала.

Тук за пръв път редуцираме към графова задача и за пръв път имаме работа с *приспособление*; на английски е *gadget*. Приспособление, или джаджа, е част на графа, към който редуцираме. Тази част се среща многократно и моделира някакъв съществен аспект на обекта, от който свеждаме. Под “част на графа” имаме предвид не непременно свързана компонента, а някакъв подграф. Приспособление в тази редукция е K_3 . Ще наричаме такива подграфи *триъгълници*.

Ребрата се конструират на два етапа.

Етап 1. Първо, за всяка клауза $\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$ слагаме три нови ребра $e_{1,j}$, $e_{2,j}$ и $e_{3,j}$ в E , като

$$f(e_{1,j}) = \{\lambda_{1,j}, \lambda_{2,j}\}$$

$$f(e_{2,j}) = \{\lambda_{1,j}, \lambda_{3,j}\}$$

$$f(e_{3,j}) = \{\lambda_{2,j}, \lambda_{3,j}\}$$

С това се появява един триъгълник , чиито върхове са литералите на ϕ_j .

Забележете, че по този начин може да се появят паралелни ребра в графа. Примерно, **всяко** срещане на двата литерала x_1 и $\bar{x}_2$ в клауза на ϕ води до поява на **ново** ребро с краища x_1 и $\bar{x}_2$ в графа.

Етап 2. Второ, за всяка променлива x_i слагаме сноп, наречен E_i , от ℓ_i на брой паралелни ребра с краища върховете x_i и $\bar{x}_i$.

Числото k е $5m$.

Ето обосновка на коректността. В едната посока, да допуснем, че в така построения граф има срез $\{V_1, V_2\}$ с тегло $\geq 5m$. Ще докажем, че ϕ е NAE-удовлетворима.

Лема 59

В текущия контекст имаме право да допуснем БОО, че за всяка променлива е вярно, че двата ѝ литерала са в различни дялове на среза.

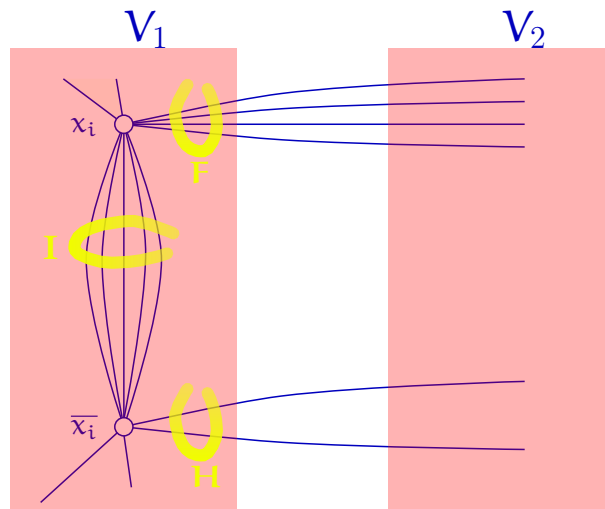
Доказателство: Нека за някое $i \in \{1, \dots, n\}$ е вярно, че и двата литерала x_i и $\bar{x}_i$ са в единия дял; БОО, нека са във V_1 . Нека

$$I = \{e \in E \mid f(e) = \{x_i, \bar{x}_i\}\}$$

$$F = \{e \in E \mid \exists \mu \in V_2 : (f(e) = \{x_i, \mu\})\}$$

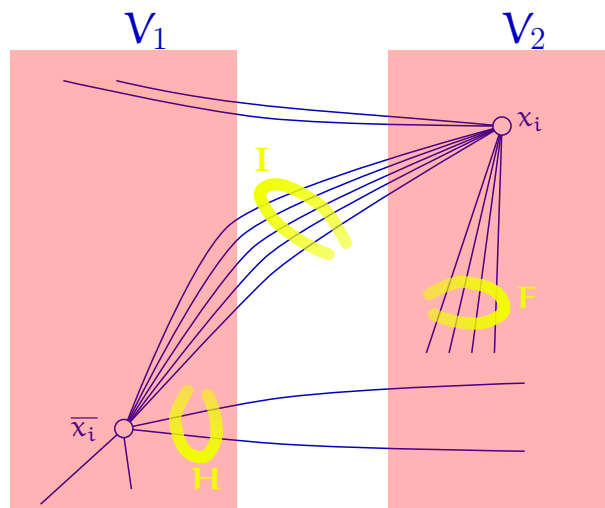
$$H = \{e \in E \mid \exists \mu \in V_2 : (f(e) = \{\bar{x}_i, \mu\})\}$$

Щом x_i и $\bar{x}_i$ са във V_1 , ребрата от I не допринасят за теглото на среза. Ребрата, инцидентни с x_i или $\bar{x}_i$ и допринасящи за теглото на среза, са точно ребрата от $F \cup H$, на брой $|F| + |H|$. Следната фигура илюстрира това.



Нека x_i се среща в p клаузи на ϕ , а $\bar{x}_i$ се среща в q клаузи на ϕ . По дефиницията на КНФ, x_i и $\bar{x}_i$ не са заедно в нито една клауза, така че става дума за $p + q$ различни клаузи; ерго, $p + q = \ell_i$. Всяко срещане на x_i във ϕ поражда в графа две ребра, инцидентни с x_i , понеже във всяка клауза, x_i е заедно с два други литерала. Аналогично, всяко срещане на $\bar{x}_i$ във ϕ поражда в графа две ребра, инцидентни с $\bar{x}_i$. Имайки предвид това, че x_i и $\bar{x}_i$ не се срещат заедно в клауза, заключаваме, че в графа има $2p + 2q$ ребра, инцидентни с точно единия от x_i и $\bar{x}_i$.

Ясно е, че $|F| \leq 2p$, понеже е възможно ребра, инцидентни с x_i , да не са във F . Аналогично, $|H| \leq 2q$, понеже е възможно ребра, инцидентни с $\bar{x}_i$, да не са в H . Заключаваме, че $|F| + |H| \leq 2\ell_i$, така че сумарният принос на x_i и $\bar{x}_i$ за теглото на среза не надхвърля $2\ell_i$. Поне единото от $|F|$ и $|H|$ не надхвърля ℓ_i . БоО, нека $|F|$ не надхвърля ℓ_i . Прехвърляме x_i във V_2 , без да променяме нищо друго. Да видим това как се отразява на теглото на среза.



Новото тегло на среза е поне колкото старото тегло минус $|F|$ плюс $|I|$. Но $|F| \leq \ell_i$, а $|I| = \ell_i$. Възможно е да е дори повече заради наличието на ребра, инцидентни с x_i , чийто друг край е във V_1 . Заключаваме, че теглото на среза или нараства, или остава същото.

И така, прехвърляйки в другия дял подходящ връх от двойка противоположни литерали, намиращи се поначало в един дял, теглото продължава да е $\geq 5m$, а броят на двойките противоположни литерали в един и същи дял намалява. Ясно е, че със серия от подходящи прехвърляния ще разместим литералите така, че всяка литералите от всяка противоположна двойка са в различни дялове, като теглото на получения срез ще е $\geq 5m$. $\square$

Съгласно Лема 59, БОО допускаме, че за всяка двойка противоположни литерали-върхове, единият е в единия дял, а другият е в другия дял. Всяка $\{x_i, \bar{x}_i\}$ е свързана със сноп от ℓ_i ребра, които са прекосяващи срез а ребра при текущите допускания. Тогава имаме $\sum_{i=1}^n \ell_i$ прекосяващи срез а, свързващи противоположни литерали, ребра, а $\sum_{i=1}^n \ell_i = 3m$ (16.2).

Останалите $\geq 2m$ прекосяващи срез а ребра трябва да са ребра от триъгълниците, построени в **Етап 1** от конструкцията. Ключовото наблюдение е, че всеки триъгълник допринася или 2 за теглото на срез а, ако той има върхове и в двата дяла, или 0, ако върховете му са в единия дял. Ерго, трябва да има поне m триъгълника с върхове и в двата дяла. Но в **Етап 1** от конструкцията правим точно m триъгълника, по един за всяка клауза. Заклучаваме, че всеки триъгълник има поне един връх във V_1 и поне един връх във V_2 .

Читателят сигурно вече се досеща, че принадлежността на литералите или към V_1 , или към V_2 моделира NAE-удовлетворимост. Да кажем, тези във V_1 имат стойности 1, а тези във V_2 имат стойности 0. Подробно казано, построяваме следната $t \in \text{Val}(\text{Var}(\phi))$. За всяка x_i , $t(x_i)$ е такава, че

- ако литералът x_i е във V_1 , то $t(x_i) = 1$,
- а ако литералът $\bar{x}_i$ е във V_1 , то $t(x_i) = 0$.

Помним, че в нито един дял не се съдържа противоположни литерали на една променлива, така че не може да има опит $t(x_i)$ да е хем 1, хем 0. Фактът, че всеки триъгълник има върхове и в двата дяла в графа, се превежда така във формулата ϕ : всяка клауза на ϕ има литерал със стойност 0 и друг литерал със стойност 1.

В обратната посока, аргументът е огледален: ако ϕ е NAE-удовлетворена от валюация t , разбиваме литералите на такива със стойности 0 и такива със стойности 1. Тогава в графа има срез с тегло $\geq 5m$, понеже противоположните литерали се оказват в противоположни дялове и това дава $3m$ прекосяващи ребра, а освен това всеки триъгълник се оказва с върхове и в двата дяла, откъдето идват още $2m$ прекосяващи ребра.

Това, че конструкцията може да се извърши в полиномиално време, е очевидно. $\square$

Ето пример. Да разгледаме следната КНФ ϕ , която често ще ползваме за пример:

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_2} \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{\phi_3} \underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_4} \quad (16.3)$$

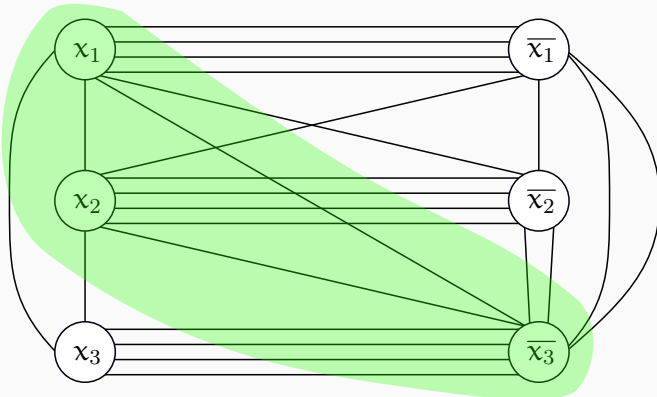
ϕ е NAE-удовлетворена от валюацията $\langle 1, 1, 0 \rangle$:

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Фигура 16.8 показва съответния мултиграф и срез с тегло $5 \times 4 = 20$. Единият дял на срез а е маркиран чрез зелен фон. Неговите литерали са тези, чиято стойност е 1 при валюацията $\langle 1, 1, 0 \rangle$. Другият дял е от другите три върха – това са литералите, чиято стойност е 0 при тази валюация.

Фигура 16.8 : Пример за $\text{NAESAT} \leq_p \text{MAXCUT}$.

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$



16.2.3 VERTEX COVER

В Лекция 12 видяхме задачата VERTEX COVER в оптимизационна версия: Задача 51. Във версия за разпознаване, задачата е следната.

Изм. Задача 77: VERTEX COVER, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G , естествено число k .

въпрос: Дали G има върхово покриване U , такова че $|U| \leq k$?

В тази лекция имаме предвид задачата само във версия за разпознаване. В тази версия, задачата е NP-пълна.

Редукция 14: $3\text{SAT} \leq_p \text{VERTEX COVER}$

Конструкция: Нека е даден екземпляр ϕ на 3SAT, като клаузите на ϕ са $\phi_1, \dots, \phi_m$ и всяка клауза има точно три литерала. Нека $\text{Var}(\phi) = \{x_1, \dots, x_n\}$. Да кажем, че за $j \in \{1, \dots, m\}$, $\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$. По дефиниция, $\lambda_{1,j}$, $\lambda_{2,j}$ и $\lambda_{3,j}$ са литерали на различни променливи. Във време, полиномиално в размера на ϕ , ще конструираме екземпляр $\langle G = (V, E), k \rangle$ на VERTEX COVER, където G е граф, а k е число, такъв че G има върхово покриване с мощност $\leq k$ тстк ϕ е удовлетворима.

Множеството от върховете е

$$V = \{u_i^+ \mid 1 \leq i \leq n\} \cup \{u_i^- \mid 1 \leq i \leq n\} \cup \{z_{i,j} \mid i \in \{1, 2, 3\}, j \in \{1, \dots, m\}\}$$

Всяка двойка върхове u_i^+ и u_i^- съответства на променливата x_i . Върховете $z_{i,j}$ съответстват на клаузите, като връх $z_{i,j}$ съответства на $\lambda_{i,j}$.

Множеството от ребрата E е

$$E = \{(u_i^+, u_i^-) \mid 1 \leq i \leq n\} \cup \{(z_{i,j}, z_{k,j}) \mid 1 \leq i < \ell \leq 3, 1 \leq j \leq m\} \cup \bigcup_{i=1}^m E'_i$$

където, за $j \in \{1, \dots, m\}$,

$$E'_j = \{(z_{1,j}, a_j), (z_{2,j}, b_j), (z_{3,j}, c_j)\}$$

където

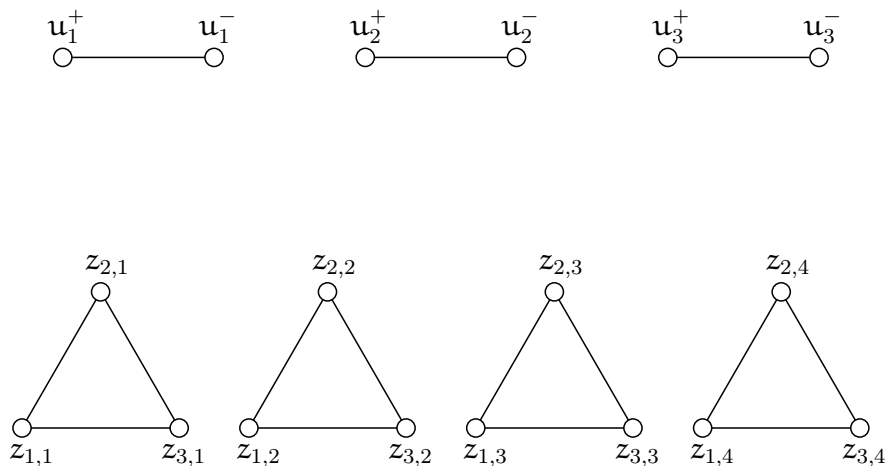
- ако $\lambda_{1,j}$ е положителен литерал x_p , то $a_j = u_p^+$, а ако $\lambda_{1,j}$ е отрицателен литерал $\overline{x_p}$, то $a_j = u_p^-$,
- ако $\lambda_{2,j}$ е положителен литерал x_q , то $b_j = u_q^+$, а ако $\lambda_{2,j}$ е отрицателен литерал $\overline{x_q}$, то $a_j = u_q^-$,
- ако $\lambda_{3,j}$ е положителен литерал x_s , то $a_j = u_s^+$, а ако $\lambda_{3,j}$ е отрицателен литерал $\overline{x_s}$, то $a_j = u_s^-$.

За удобство казваме, че ребрата от $\bigcup_{i=1}^m E'_i$ са *сините ребра*, а останалите ребра са *черните ребра*.

Преди да довършим конструкцията (все още не сме определили k), да илюстрираме с пример нещата дотук. Да разгледаме отново Φ от (16.3):

$$\Phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2} \underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3} \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4}$$

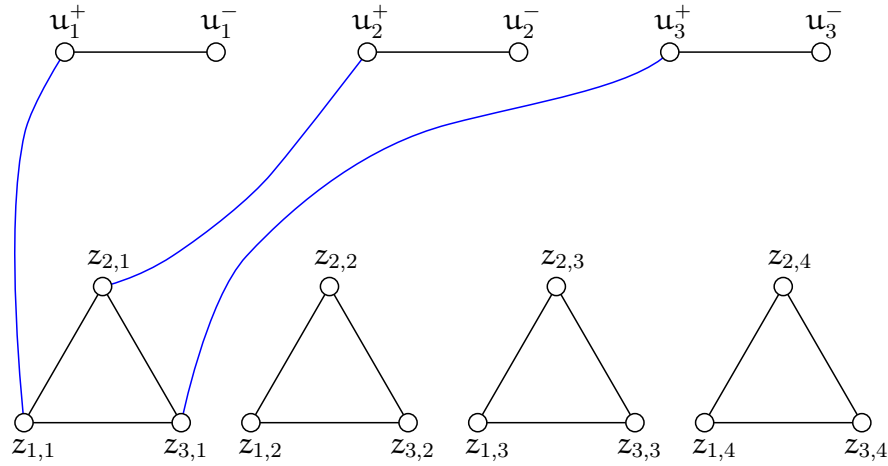
Променливите са x_1 , x_2 и x_3 и има четири клаузи $\phi_1, \dots, \phi_4$. Конструираме върхове u_1^+ , u_1^- , u_2^+ , u_2^- , u_3^+ и u_3^- , съответни на променливите, като конструираме черни ребра (u_1^+, u_1^-) , (u_2^+, u_2^-) и (u_3^+, u_3^-) . Освен това, за $1 \leq j \leq 4$, конструираме върхове $z_{1,j}$, $z_{2,j}$, $z_{3,j}$, с черно ребро между всеки два от тях. Това са 3-кликите, съответни на клаузите. Игнорирайки ребрата от $\bigcup_{i=1}^4 E'_i$ (сините ребра), картината е такава:



Да конструираме ребрата от $\bigcup_{i=1}^4 E'_i$, които нарекохме “сините ребра”. Да започнем с E'_1 . То се състои от три ребра:

- ребро $(z_{1,1}, u_1^+)$ заради първия литерал във ϕ_1 , а именно x_1 ,
- ребро $(z_{2,1}, u_2^+)$ заради втория литерал във ϕ_1 , а именно x_2 ,
- и ребро $(z_{3,1}, u_3^+)$ заради третия литерал във ϕ_1 , а именно x_3 .

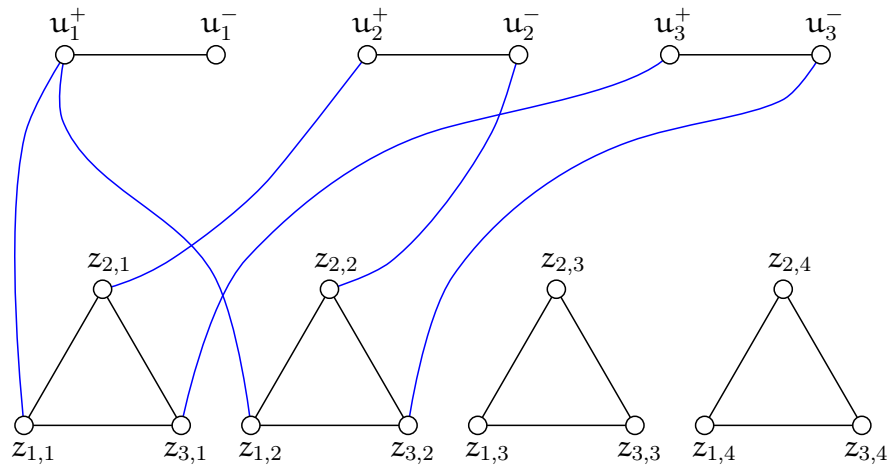
Добавяме ребрата от E'_1 към предната илюстрация.



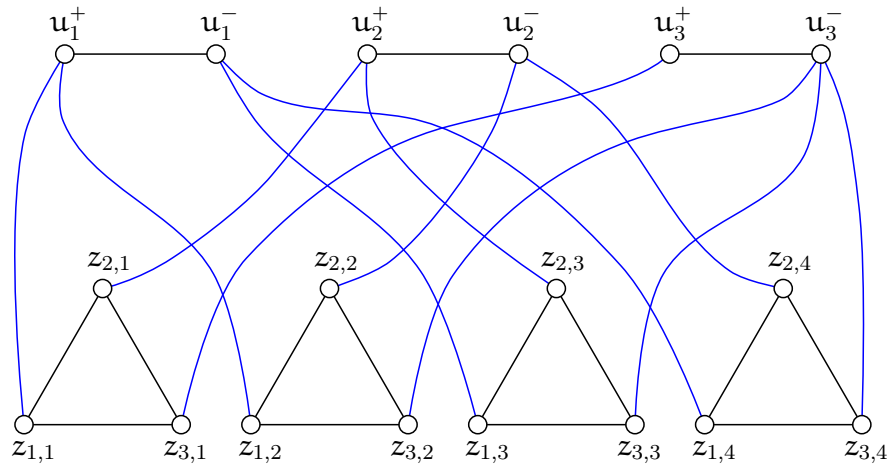
Да конструираме E'_2 . То се състои от три ребра:

- ребро $(z_{1,2}, u_1^+)$ заради първия литерал във ϕ_1 , а именно x_1 ,
- ребро $(z_{2,2}, u_2^-)$ заради втория литерал във ϕ_1 , а именно $\bar{x}_2$,
- и ребро $(z_{3,2}, u_3^-)$ заради третия литерал във ϕ_1 , а именно $\bar{x}_3$.

Добавяме ребрата от E'_2 към предната илюстрация.



По аналогичен начин добавяме и сините ребра, чиито един край е от третата или четвъртата 3-клика.



Би трябвало да е ясно какъв е смисълът на това. За всяка 3-клика “долу”, за всеки неин връх се слага точно едно синьо ребро между този връх и единия край някое от ребрата от вида (u_i^+, u_i^-) “горе”, като това синьо ребро моделира появата на променливата x_i в клаузата, съответна на въпросната 3-клика. Всяко ребро от вида (u_i^+, u_i^-) има един положителен край и един отрицателен край, като имената на върховете са такива, за да указват какви краища са. Положителните върхове “горе” моделират положителните литерали в клаузите, а отрицателните върхове “горе” моделират отрицателните литерали в клаузите.

Забележете, че точно сините ребра моделират особеностите на формулата ϕ . Черните ребра се конструират само въз основа на m и n , без оглед на конкретиката на ϕ .

За да довършим конструирането, трябва да кажем и колко е k . Числото k е $2m + n$. Забележете, че стойността $2m + n$ е точна долна граница за k дори в отсъствието на сините ребра! За да бъдат покрити само черните ребра, покриването трябва да съдържа поне $2m + n$ върха, понеже за всяко черно ребро “горе” трябва поне единия му край да е в покриването, а за всяка 3-клика “долу” трябва поне два от върховете ѝ да са в покриването. Ерго, не може да има покриване с по-малко от $2m + n$ върха.

Но в отсъствието на сините ребра имаме голяма свобода за това, точно кои $2m + n$ върха да вземем за покриването: за всяко ребро “горе” можем да вземем кой да е от краищата му, а за всяка 3-клика “долу” можем да вземем кои да е два от трите ѝ върха.

Сега да разгледаме и сините ребра. Те също трябва да бъдат покрити. Ключовото наблюдение е, че покриването на сините ребра може да стане само от въпросните $2m + n$ върха, необходими (и достатъчни) за покриването на черните ребра. Това налага някаква координация между вземането на върхове “горе” и върхове “долу”.

Наблюдение 107: Заради сините ребра трябва координация между “горе” и “долу”

За всяко $j \in \{1, \dots, m\}$, ние **не вземаме** точно един от върховете на 3-кликата $\{z_{1,j}, z_{2,j}, z_{3,j}\}$ в покриването. БОО, нека $z_{1,j}$ не е в покриването. Тогава синьото ребро, инцидентно със $z_{1,j}$, може да бъде покрито само от **другия** си край.

Нека $\mathcal{U}$ е произволно върхово покриване на G с мощност $2m + n$. Нека

$$\hat{\mathcal{U}} = \mathcal{U} \cap (\{u_i^+ \mid 1 \leq i \leq n\} \cup \{u_i^- \mid 1 \leq i \leq n\})$$

На прост български, $\hat{\mathcal{U}}$ са “горните” върхове в покриването. Както вече забелязахме, $|\hat{\mathcal{U}}| = n$ и всеки връх в $\hat{\mathcal{U}}$ е точно единият от u_i^+ и u_i^- , по всички $i \in \{1, \dots, n\}$.

Във всяка 3-клика “долу” има точно един връх, който не е в $\mathcal{U}$. Да наречем съвкупността от тези върхове “долу”, *важните върхове*. Съгласно Наблюдение 107, всеки важен връх е съсед на точно един връх от $\hat{\mathcal{U}}$. По този начин за всяка клика “долу” има точно един съответен връх “горе”, който е в $\hat{\mathcal{U}}$. Това съответствие задава функция h с домейн множеството от 3-кликите и кодомейн $\hat{\mathcal{U}}$. В общия случай, h очевидно не е инекция, но също така не е сюрекция, понеже е възможно върхове от $\hat{\mathcal{U}}$ да не са съседи на важни върхове[†].

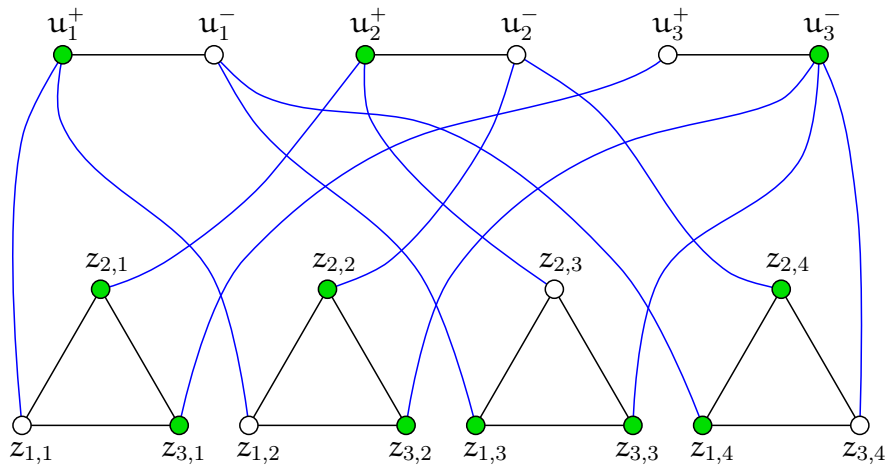
Да разгледаме следната валюация t на $\text{Var}(\phi)$: за всяко $i \in \{1, \dots, n\}$, ако $u_i^+ \in \hat{\mathcal{U}}$, то $t(x_i) = 1$, а ако $u_i^- \in \hat{\mathcal{U}}$, то $t(x_i) = 0$. Тъй като точно единият от $\{u_i^+, u_i^-\}$ е в $\hat{\mathcal{U}}$, валюацията е добре дефинирана – не може $t(x_i)$ да е хем 1, хем 0, и не може $t(x_i)$ да е недефинирано. Да се убедим, че t е удовлетворяваща валюация. За тази цел ползваме h , функцията от 3-кликите в $\hat{\mathcal{U}}$. За всяко $j \in \{1, \dots, m\}$, $h(\{z_{1,j}, z_{2,j}, z_{3,j}\})$ съответства на литерала, от който клаузата ϕ_j бива удовлетворена:

- ако $h(\{z_{1,j}, z_{2,j}, z_{3,j}\}) = u_i^+$, за някое $i \in \{1, \dots, n\}$, то ϕ_j съдържа положителния литерал x_i и, тъй като $t(x_i) = 1$, вярно е, че $\text{TA}(\phi_j, t) = 1$.
- ако $h(\{z_{1,j}, z_{2,j}, z_{3,j}\}) = u_i^-$, за някое $i \in \{1, \dots, n\}$, то ϕ_j съдържа отрицателния литерал $\bar{x}_i$ и, тъй като $t(x_i) = 0$, вярно е, че $\text{TA}(\phi_j, t) = 1$.

Докажем, че всяко върхово покриване на графа с мощност $2m + n$ задава удовлетворяваща валюация на ϕ . В обратната посока, доказателството е огледално.

Ето илюстрация с примера от (16.3). Формулата е удовлетворима, примерно от валюацията $t = \langle 1, 1, 0 \rangle$. Съответно графът има върхово покриване с мощност $2m + n = 2 \times 4 + 3 = 11$. Върховете в покриването са оцветени в зелено. Във формулата, във всяка клауза точно един литерал е оцветен в зелено – това е литерал, чрез който клаузата бива удовлетворена.

$$\phi = (x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

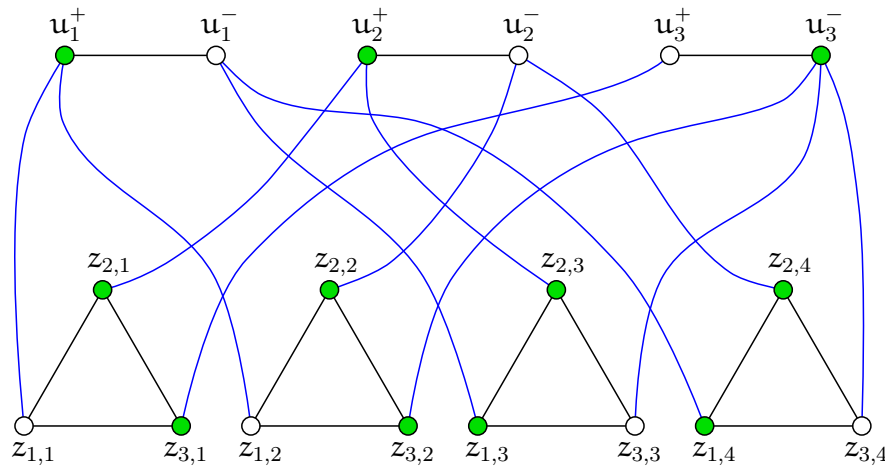


Забележете, че клауза може да бъде удовлетворена чрез повече от един литерал при дадена валюация. Примерно, щом $t(x_2) = 1$ и $t(x_3) = 0$, третата клауза се удовлетворява както чрез положителния литерал x_2 , така и чрез отрицателния литерал $\bar{x}_3$. Но в графа точно един връх

[†]Представете си КНФ, в която един и същи литерал, да кажем x_1 , се намира във всяка клауза. Тогава за $j \in \{1, \dots, m\}$, j -тата 3-клика има точно един връх, БОО нека е $z_{1,j}$, съседен на u_1^+ . Тогава има покриване на целия граф, което съдържа u_1^+ и не съдържа никой $z_{1,j}$. Тогава върховете $z_{1,j}$ са точно важните върхове и функцията h изобразява всяка 3-клика в u_1^+ . Очевидно h не е сюрекция.

в съответната трета отляво надясно 3-клика е извън покриването; това е връх $z_{2,3}$. Неговият съсед в покриването е u_2^+ и поради това в третата клауза зеленият литерал е положителният x_2 , а не отрицателният $\bar{x}_3$. Отрицателният $\bar{x}_3$ щеше да е зеленият литерал в третата клауза, ако бяхме избрали $z_{3,3}$ да не е в покриването. Можем ли да направим това? Да, разбира се, ето.

$$\phi = (x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$



Покажахме с пример, че изборът на валюация определя еднозначно кои върхове “горе” влизат в покриването, но “долу” имаме известна свобода. Ако по отношение на дадена валюация можем да удовлетворим някоя клауза с повече от един литерала, то имаме повече от един избор за това, кой връх от съответната ѝ 3-клика да не влезе в покриването. Но винаги точно един връх от 3-клика е извън покриването – това се налага от факта, че $k = 2m + n$.

Наблюдение 108

В контекста на редукцията $3SAT \leq_p VERTEX COVER$, няма противоречие между следните факти.

- Дефиницията на “удовлетворяваща валюация” иска поне един литерал във всяка клауза да има стойност 1.
- Върховото покриване не включва точно един връх от всяка 3-клика.

Противоречие няма. Същественото е, че за всеки невключен в покриването връх “долу” има съсед в покриването “горе”. А ако при тази валюация някоя клауза се удовлетворява от няколко литерала, то има същият брой начини да изберем кой връх от съответната 3-клика да не включваме в покриването.

И накрая само отбелязваме, че очевидно редукцията може да се извърши в полиномиално време. □

16.2.4 INDEPENDENT SET, CLIQUE и DOMINATING SET

След като показвахме неподатливостта на VERTEX COVER, с лекота ще покажем неподатливостта на INDEPENDENT SET и CLIQUE. Малко по-триково е свеждането на VERTEX COVER

до DOMINATING SET. Оптимизационните версии на INDEPENDENT SET и DOMINATING SET задачи видяхме в Лекция 12: Задача 52 и Задача 55. Ето трите задачи във версия за разпознаване.

Изч. Задача 78: INDEPENDENT SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G , естествено число k .

въпрос: Дали G има антиклика U , такава че $|U| \geq k$?

Изч. Задача 79: CLIQUE, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G , естествено число k .

въпрос: Дали G има клика U , такава че $|U| \geq k$?

Изч. Задача 80: DOMINATING SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G , естествено число k .

въпрос: Дали G има доминиращо множество U , такава че $|U| \leq k$?

Редукция 15: VERTEX COVER $\leq_p$ INDEPENDENT SET

Конструкция: Даден е произволен екземпляр $\langle G, k \rangle$ на VERTEX COVER. Построяваме екземпляр $\langle G, n - k \rangle$ на INDEPENDENT SET. Твърдим, че $\langle G, k \rangle$ е ДА-екземпляр на VERTEX COVER тстк $\langle G, n - k \rangle$ е ДА-екземпляр на INDEPENDENT SET. Но това следва веднага от Теорема 105. $\square$

Редукция 16: INDEPENDENT SET $\leq_p$ CLIQUE

Конструкция: Даден е произволен екземпляр $\langle G, k \rangle$ на INDEPENDENT SET. Построяваме екземпляр $\langle \bar{G}, k \rangle$ на CLIQUE. Твърдим, че $\langle G, k \rangle$ е ДА-екземпляр на INDEPENDENT SET тстк $\langle \bar{G}, k \rangle$ е ДА-екземпляр на CLIQUE. Но това следва веднага от факта, че за всяко $U \subseteq V(G)$ е вярно, че U е антиклика в G тстк U е клика в $\bar{G}$. $\square$

Редукция 17: VERTEX COVER $\leq_p$ DOMINATING SET

Конструкция: Задачите VERTEX COVER и DOMINATING SET са близки, но не са на практика-една-и-съща-задача, какъвто е случаят с VERTEX COVER и INDEPENDENT SET, или INDEPENDENT SET и CLIQUE.

По даден екземпляр $\langle G = (V, E), k \rangle$ на VERTEX COVER, конструираме екземпляр $\langle G' = (V', E'), k \rangle$ на DOMINATING SET, такива че G има върхово покриване с мощност $\leq k$ тстк G' има доминиращо множество с мощност $\leq k$.

Нека $V = \{v_1, \dots, v_n\}$ и $E = \{e_1, \dots, e_m\}$. Нека f е свързващата функция на G , в смисъл, че за всяко ребро e_i , $f(e_i)$ е множеството от двата му края. Конструираме m на брой нови

върхове $z_1, \dots, z_m$: такива, които не присъстват във V . Конструираме

$$V' = V \cup \{z_1, \dots, z_m\}$$

$$\forall j \in \{1, \dots, m\} : E'_j = \{(u, z_j), (v, z_j)\}, \text{ където } f(e_j) = \{u, v\}$$

$$E' = \bigcup_{j=1}^m E'_j$$

На прост български, добавяме нови върхове точно колкото са ребрата на G , и “дублираме” всяко ребро e_j с две нови ребра с общ край, като общият им край е точно един от новите върхове, а останалите им два края съвпадат с краищата на e_j .

Коректност: Нека $\langle G = (V, E), k \rangle$ е ДА-екземпляр на VERTEX COVER. БОО, нека U няма изолирани върхове. Нека U е върхово покриване на G с мощност k . Ще докажем, че U е доминиращо множество на G' . V' се разбива на V и $\{z_1, \dots, z_m\}$.

- За всеки връх $v \in V$ е вярно, че $v \in U$ или v има съсед $w \in V$, такъв че $w \in U$ – обратното влече, че има ребро $e \in E$, такова че $f(e) \cap U = \emptyset$. На свой ред това влече, че U не е върхово покриване на G .

Това е същността на доказателството на Теорема 106.

- За всеки връх z_j , $1 \leq j \leq m$, е вярно, че z_j има съсед в U – обратното влече, че $f(e_j) \cap U = \emptyset$. На свой ред това влече, че U не е върхово покриване на G .

Доказахме, че всеки връх на G' е в U или има съсед в U . Тогава U е доминиращо множество за G' .

В обратната посока, нека $\langle G' = (V', E'), k \rangle$ е ДА-екземпляр на DOMINATING SET. Нека U е доминиращо множество на G' с мощност k . Тук обаче не е вярно, че U или дори $U \cap V$ непременно е върхово покриване на G . Лесно може да се измисли пример, в който $U \subseteq \{z_1, \dots, z_m\}$; ерго, $U \cap V = \emptyset$, а $\emptyset$ няма как да е покриващо множество на G , който няма изолирани върхове. Да разгледаме множеството $\tilde{E} = \{(u, v) \in E \mid u \notin U \text{ и } v \notin U\}$.

Наблюдение 109

В текущия контекст е вярно, че за всяко ребро $e_j \in \tilde{E}$, върхът $z_j \in U$. В противен случай z_j не може да бъде доминиран, защото той самият не е в U и няма съсед в U . Ключовият факт е, че единствените съседи на z_j са върховете от $f(e_j)$.

Ребрата от $\tilde{E}$ са причината $U \cap V$ да не е върхово покриване на G : всяко от тях е “непокрыто” и от двата края. Ако $\tilde{E} = \emptyset$, $U \cap V$ е върхово покриване на G и доказателството е готово. Да си представим изпълнението на следния алгоритъм в текущия контекст.

ALG-DS-VC()

```

1  while  $\tilde{E} \neq \emptyset$  do
2       $e_j$  е произволно ребро от  $\tilde{E}$ 
3       $\tilde{E} \leftarrow \tilde{E} \setminus \{e_j\}$ 
4       $U \leftarrow U \setminus \{z_j\}$ 
5      if  $f(e_j) \cap U = \emptyset$ 
6           $x$  е произволен връх от  $f(e_j)$ 
7           $U \leftarrow U \cup \{x\}$ 
8  return
```

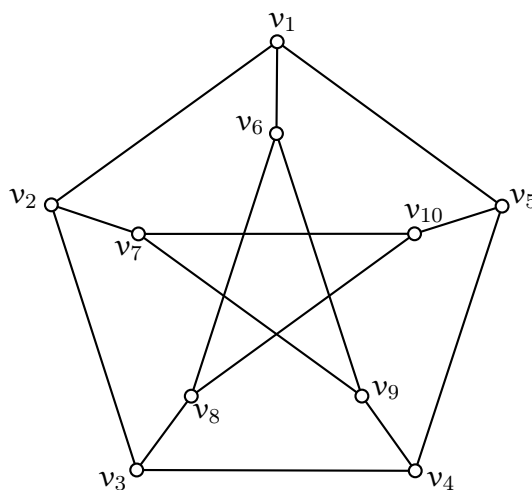
Инвариант за цикъла (редове 1–7) е:

При всяко достигане на ред 1, $\mathcal{U}$ е покриващо множество за $G - \tilde{E}$ и $|\mathcal{U}| \leq k$.

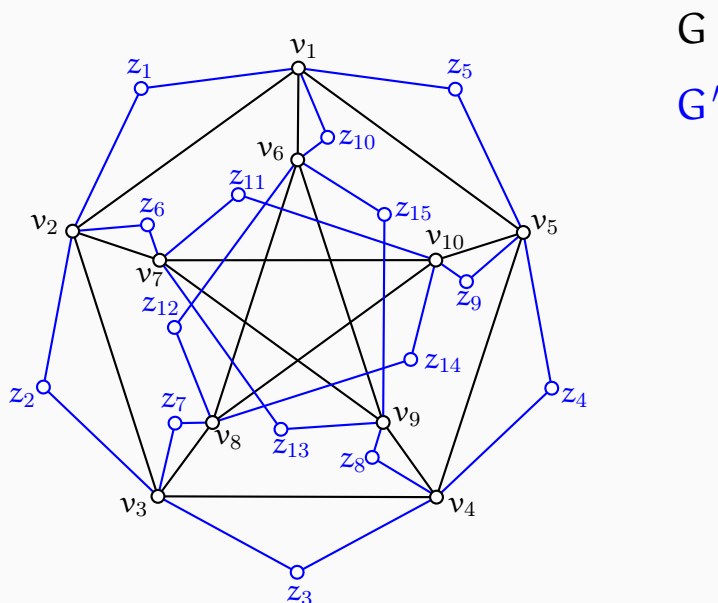
Доказателството е тривиално и остава за читателя. Доказателството използва Наблюдение 109, което обосновава присъствието на z_j в $\mathcal{U}$ преди изпълнението на ред 4. Забележете, че проверката дали $f(e_j) \cap \mathcal{U} = \emptyset$ на ред 5 има смисъл! Наистина, поначало $\tilde{E}$ са ребра, чиито краища не са в $\mathcal{U}$, обаче е възможно при предишна итерация на цикъла на алгоритъма, единият край на e_j да е бил сложен в $\mathcal{U}$.

При термилирането на алгоритъма е изпълнено $\tilde{E} = \emptyset$. Съгласно инварианта, $\mathcal{U}$ е покриващо множество на G , и освен това $|\mathcal{U}| \leq k$. $\square$

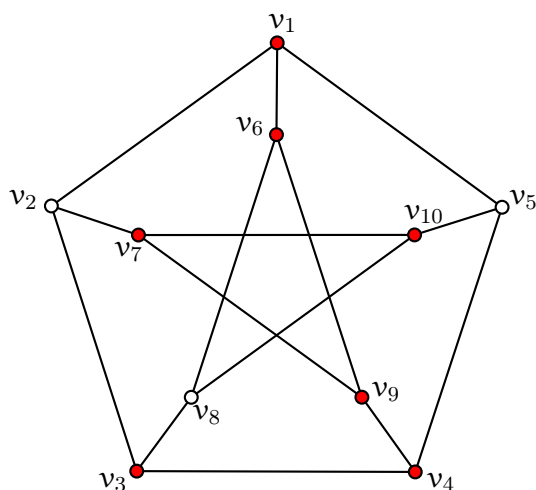
За илюстрация ще използваме графа на Petersen:



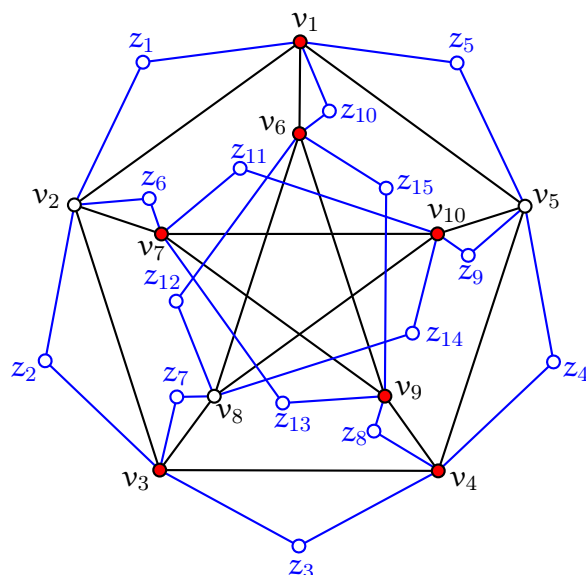
Фигура 16.9 показва конструкцията върху графа на Petersen. G от екземпляра на VERTEX COVER е в черно, а G' от екземпляра на DOMINATING SET се получава от G с добавяне на сините върхове и ребра.

Фигура 16.9 : G и G' от $\text{VERTEX COVER} \leq_p \text{DOMINATING SET}$.

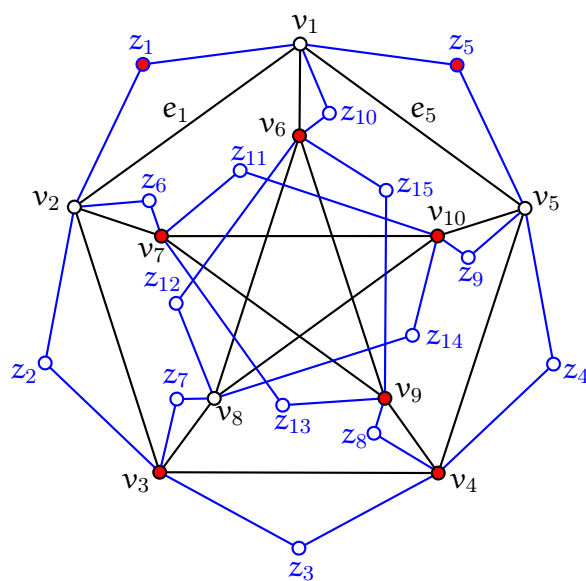
Ето върхово покриване на G със седем върха (нарисувани са в червено). Както знаем (вижте разсъжденията на стр. 704), това не е оптимално, понеже $\tau(G) = 6$. Но няма изискване екземплярът на задачата, от която свеждаме, да е оптимален. Важното е, че $\langle G, 7 \rangle$, където G е графът на Petersen, е ДА-екземпляр на VERTEX COVER .



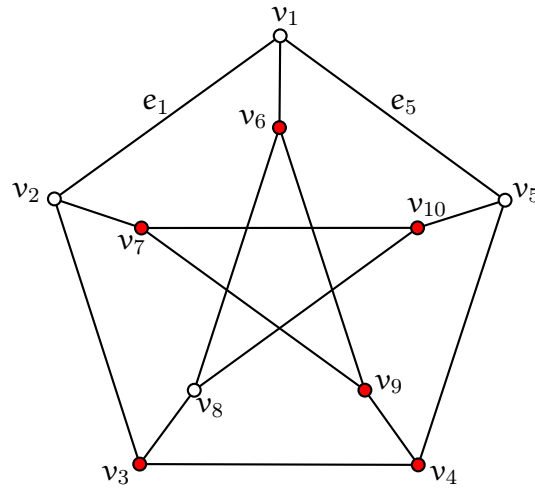
В графа G' , същите върхове са доминиращо множество:



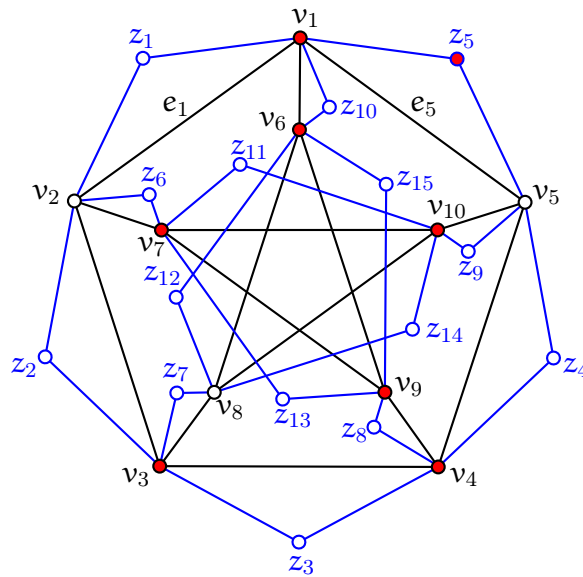
В обратната посока, да разгледаме доминиране на G' от осем върха:



Сечението на червените върхове и върховете на G не е върхово покриване на G . Има две проблемни ребра на G , а именно e_1 и e_5 , които са непокрити и от двете страни. Множеството $\tilde{E}$ съдържа точно тях. Това се вижда по-ясно, ако разгледаме само G плюс върховете от доминиращото множество, които са и върхове на G :

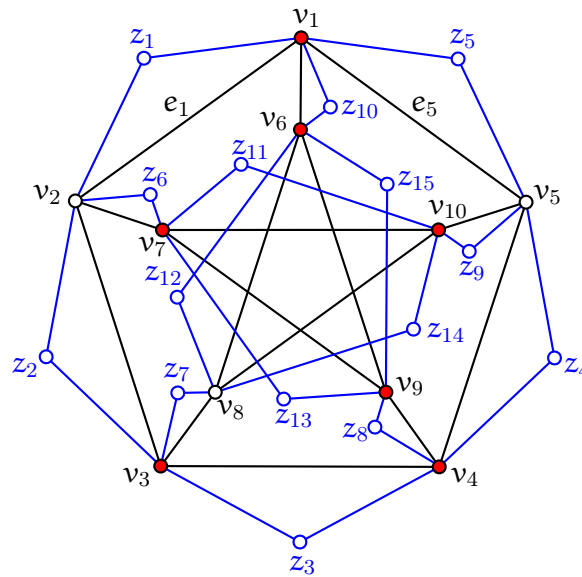


За да се справим с проблема, изпълняваме алгоритъма. Нека първо разгледаме e_1 на ред 2. Махаме e_1 от $\tilde{E}$, махаме z_1 от $\mathcal{U}$ и вкарваме в $\mathcal{U}$ единият от съседите му; да кажем, v_1 :



Странична забележка: след всяко такова “прехвърляне” на връх, $\mathcal{U}$ остава доминиращо множество на G' . В инварианта на алгоритъма не включихме това съждение, понеже там само искахме да покажем, че $\mathcal{U}$ след термилирането е покриване на G .

Реброто e_5 вече не е проблемно, но, ако следваме стриктно алгоритъма, трябва да изпълним още една итерация, понеже $\tilde{E} = \{e_5\}$ е непразно. Махаме e_5 от $\tilde{E}$ и махаме z_5 от $\mathcal{U}$. Но $f(e_5) \cap \mathcal{U}$ в този момент е $\{v_1\}$, така че редове 6 и 7 не се изпълняват. Алгоритъмът приключва, като $\mathcal{U}$ покрива G .



Виждаме, че $\langle G, 8 \rangle$ е ДА-екземпляр на VERTEX COVER, което е следствие от факта, че $\langle G', 8 \rangle$ е ДА-екземпляр на DOMINATING SET.

След като видяхме и пример за работата на редукцията, едно предупреждение. То може и да е излишно, но може и да не е. Следната “редукция” е **некоректна**: по даден екземпляр $\langle G, k \rangle$ на VERTEX COVER конструираме екземпляр $\langle G, k \rangle$ на DOMINATING SET (на практика нищо не се конструира, защото екземплярите съвпадат). “Обосновката” е Теорема 106: за всеки граф G без изолирани върхове е вярно, че всяко върхово покриване е и доминиращо множество. Тъй като БОО можем да допуснем, че G няма изолирани върхове, то, вследствие на теоремата, ако G има върхово покриване с мощност $\leq k$, то G задължително има доминиращо множество с мощност $\leq k$.

Това е вярно, но не е достатъчно. Редукцията трябва да осигурява “тогава и само тогава, когато”! Ако G от екземпляра на VERTEX COVER няма върхово покриване с размер $\leq k$, то графът, произведен от редукцията, трябва да няма доминиращо множество с мощност според редукцията. И ето защо тази “редукция” е некоректна: възможно е $\langle G, k \rangle$ да е НЕ-екземпляр на VERTEX COVER и все пак $\langle G, k \rangle$ да е ДА-екземпляр на DOMINATING SET.

16.2.5 0-1 INTEGER PROGRAMMING

Фундамент: линейно и целочислено програмиране. *Линейно програмиране* (на английски е *Linear Programming* или LP) е дял от приложната математика, който се занимава с определен вид оптимизационни задачи. Всяка от тези задачи може да се моделира чрез линейна функция f на n променливи $f = \sum_{k=1}^n c_k x_k$, наречена *целева функция* (*objective function* на английски), където $c_1, \dots, c_n$ са реални константи, а $x_1, \dots, x_n$ са реалните променливи.

Дадени са m ограничения

$$\begin{aligned} \sum_{k=1}^n a_{1,k} x_k &\leq b_1 \\ \sum_{k=1}^n a_{2,k} x_k &\leq b_2 \\ &\dots \\ \sum_{k=1}^n a_{m,k} x_k &\leq b_m \end{aligned}$$

където $a_{i,k}$ са реални константи за $1 \leq i \leq m$ и $1 \leq k \leq n$, а $b_1, \dots, b_m$ също са реални константи. Има и още ограничения: $x_k \geq 0$ за $1 \leq k \leq n$. Допустимо решение е всеки вектор $\mathbf{x} \in \mathbb{R}^n$, за който ограниченията са удовлетворени. Задачата е максимизационна: да се намери допустимо решение, за което целевата функция достига максимална стойност. На пръв поглед задачата е континуална, понеже в общия случай множеството от допустимите решения е неизброимо безкрайно, но поради факта, че целевата функция е линейна и ограниченията са линейни неравенства, екстремалните допустими решения—измежду които е и оптималното—са краен брой, макар и потенциално грамаден [126, стр. 5–6].

Огромен брой житейски и математически задачи може да се формулират като задачи за линейно програмиране [126], [62]. Известни са ефикасни алгоритми за линейното програмиране, като симплекс методът е най-популярният от тях.

Понякога решенията, които ни интересуват, са целочислени поради природата на обектите, които моделираме. Например, фирма за превоз на стоки иска да максимизира печалбата си от някакъв вид превози. Напълно възможно е симплекс методът да изчисли, че оптимално решение се получава при използване на 9.39 камиона. Не става дума за грешка в моделирането или в пресмятането, просто линейното програмиране не отчита неделимостта на камионите. Задачи, искащи целочислени решения, са предмет на целочисленото програмиране (на английски е *Integer Linear Programming* или ILP). По-академично казано, допустимите решения в ILP са елементи на $\mathbb{Z}^n$.

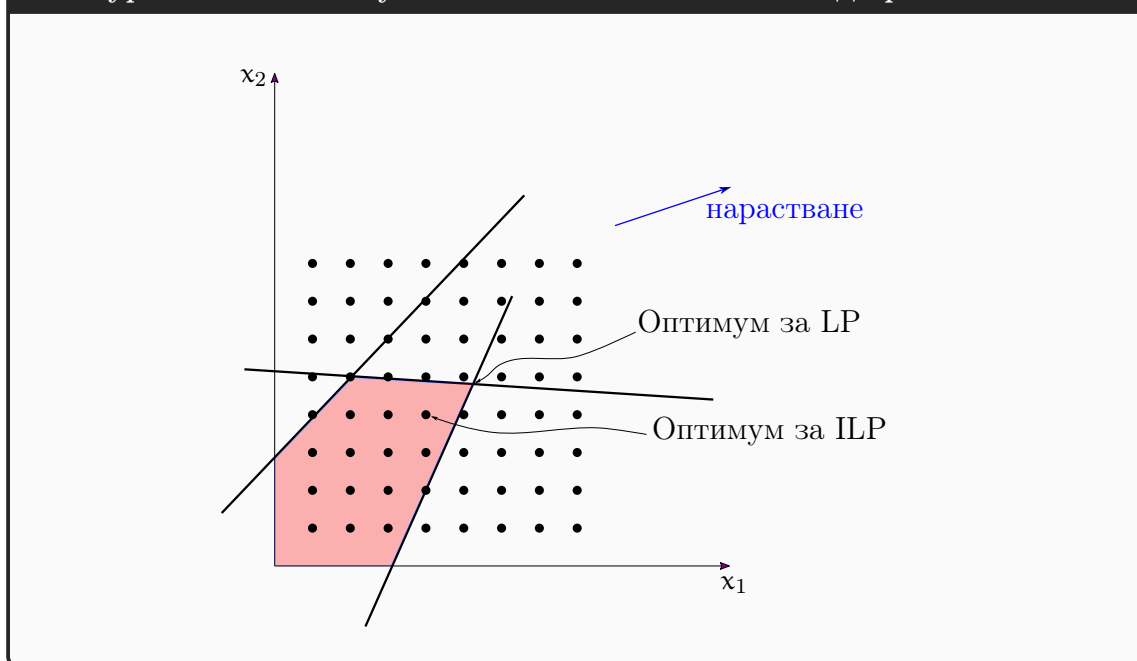
На пръв поглед може да използваме подходящ алгоритъм за линейно програмиране и после да закръглим $x_1, \dots, x_n$ до най-близките цели числа. Тази идея се разисква подробно в [126, стр. 307–308]. Понякога тя работи, особено ако става дума за големи дробни числа, които са “не особено чувствителни” към закръгляване. Понякога обаче тази идея не работи. По думите на Papadimitriou и Steiglitz:

There are, however, very serious limitations to this approach. Most applications of ILP are not just LP's with solutions that happen to come in quanta of one unit. Most frequently, an ILP is the result of a conscious use of the integer constraint to model combinatorial constraints or nonlinearities of different sorts. These ILP's are, by their very nature, not susceptible to the rounding approach, essentially because rounding defeats the purpose of ILP formulation, and it is as hard to perform as solving the original combinatorial problem from scratch.

Фигура 16.10 илюстрира това, което Papadimitriou и Steiglitz искат да кажат. Прочее, тя практически повтаря Figure 13-1 в [126, стр. 308]. Променливите са само две: x_1 и x_2 . Ако x_1 и x_2 са реални положителни променливи, множеството от заеманите от тях стойности е целият първи квадрант $\mathbb{R}^+ \times \mathbb{R}^+$. Ако са цели положителни числа, това множество е дискретното пространство $\mathbb{Z}^+ \times \mathbb{Z}^+$, което е показано с едри точки на фигурата. Множеството

от допустимите стойности е в червено – то е сечението на трите релевантни полуравнини спрямо трите показани прави. Целевата функция е такава, че синият вектор (посоката на нарастването) е перпендикулярен на $c_1x_1 + c_2x_2 = \alpha$, $\alpha = \text{const}$.

Фигура 16.10 : Оптимумът на ILP не е опт. на LP след приближение.



Оптималното решение на континуалната задача е посоченият връх на многоъгълника-пространство на допустимите решения. Оптималното решение на дискретната задача също е посочено, но то е сравнително далече от оптималното решение на континуалната задача в смисъл, че има четири точки от дискретното пространство, които са по-близо (до оптималното решение на континуалната задача) от него. Ерго, ако апроксимираме решението на континуалната задача, може да получим решение, което е извън множеството от допустимите решения.

Булево програмиране. Ако “целочислено програмиране” е задачата за намиране на оптимум от целочислени променливи, може да наречем “булево програмиране” частния случай, в който променливите x_i вземат стойности от $\{0, 1\}$. В такъв случай всяко x_i моделира включване изцяло или не-включване изобщо на нещо. Тази доста ограничена версия на задачата е неподатлива, което влече неподатливостта на целочисленото програмиране.

Ето и формалната дефиниция на задачата. Това е задача за разпознаване, а не оптимизационна задача, така че не се опитваме да максимизираме целева функция.

Изч. Задача 81: 0-1 INTEGER PROGRAMMING

екземпляр: Нека $x_1, \dots, x_n$ са булеви променливи. Дадени са m неравенства

$$\sum_{k=1}^n a_{1,k} x_k \leq b_1$$

$$\sum_{k=1}^n a_{2,k} x_k \leq b_2$$

...

$$\sum_{k=1}^n a_{m,k} x_k \leq b_m$$

където $a_{i,k}$ са цели константи за $1 \leq i \leq m$ и $1 \leq k \leq n$ и $b_1, \dots, b_m$ също са цели константи.

въпрос: Дали съществува булева функция $h: \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$, такава че всички ограничения са удовлетворени при нейните стойности на променливите?

Редукция 18: 3SAT $\leq_p$ 0-1 INTEGER PROGRAMMING

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

Нека $\text{Var}(\phi) = \{x_1, \dots, x_n\}$. Конструираме съответен екземпляр ψ на 0-1 INTEGER PROGRAMMING. Множеството от променливите на ψ е същото, а именно $\{x_1, \dots, x_n\}$. Конструираме $m \times n$ булеви константи $a_{i,k}$ за $i \in \{1, \dots, m\}$ и $k \in \{1, \dots, n\}$ така:

$$a_{i,k} = \begin{cases} 0, & \text{ако литерал на променливата } x_i \text{ не се среща в клаузата } \phi_k \\ 1, & \text{ако литералът } \bar{x}_i \text{ се среща в клаузата } \phi_k \\ -1, & \text{ако литералът } x_i \text{ се среща в клаузата } \phi_k \end{cases}$$

За $k \in \{1, \dots, m\}$ конструираме b_k така:

$$b_k = -1 + \sum_{i=1}^n \max\{0, a_{i,k}\}$$

На прост български, b_k е -1 плюс броя на отрицателните литерали в клауза ϕ_k .

Първо да видим пример за конструкцията. Да разгледаме отново ϕ от (16.3):

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_2} \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{\phi_3} \underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_4}$$

Променливите са x_1, x_2 и x_3 и има четири клаузи $\phi_1, \dots, \phi_4$. Генерираме следната система

от неравенства:

$$\begin{aligned} -x_1 - x_2 - x_3 &\leq -1 + 0 \\ -x_1 + x_2 + x_3 &\leq -1 + 2 \\ +x_1 - x_2 + x_3 &\leq -1 + 2 \\ +x_1 + x_2 + x_3 &\leq -1 + 3 \end{aligned}$$

Да умножим по -1 неравенствата:

$$\begin{aligned} +x_1 + x_2 + x_3 &\geq +1 - 0 \\ +x_1 - x_2 - x_3 &\geq +1 - 2 \\ -x_1 + x_2 - x_3 &\geq +1 - 2 \\ -x_1 - x_2 - x_3 &\geq +1 - 3 \end{aligned}$$

Да прехвърлим във всяко неравенство минус-единиците вдясно в лявата страна с обратен знак:

$$\begin{aligned} x_1 + x_2 + x_3 &\geq 1 \\ x_1 + (1 - x_2) + (1 - x_3) &\geq 1 \\ (1 - x_1) + x_2 + (1 - x_3) &\geq 1 \\ (1 - x_1) + (1 - x_2) + (1 - x_3) &\geq 1 \end{aligned} \tag{16.4}$$

ϕ е удовлетворима, например от валюацията $t = \langle 1, 1, 0 \rangle$. Съответно, неравенствата се удовлетворяват от тези стойности на променливите:

$$\begin{aligned} 1 + 1 + 0 &\geq 1 \quad \checkmark \\ 1 + (1 - 1) + (1 - 0) &\geq 1 \quad \checkmark \\ (1 - 1) + 1 + (1 - 0) &\geq 1 \quad \checkmark \\ (1 - 1) + (1 - 1) + (1 - 0) &\geq 1 \quad \checkmark \end{aligned}$$

Няма да правим подробно формално доказателство за коректност, понеже нещата са твърде очевидни. Ясно е, че неравенствата на ψ може да бъдат преписани като в (16.4). Ако ги напишем в този вид, коректността наистина е очевидна. Лявата страна на всяко неравенство е сума от n събираеми, $n-3$ от които са “чисти нули”[†] и само 3 [‡] от тях са от вида x_i или $(1-x_i)$. Предвид факта, че променливите са булеви, очевидно всяко събираемо е неотрицателно. За да бъде в сила неравенството, поне едно събираемо трябва да е 1.

- Да допуснем, че ϕ е ДА-екземпляр на 3SAT. Тогава има валюация $t \in \text{Val}(\text{Var}(\phi))$, такава че във всяка клауза α на ϕ има поне един литерал λ , такъв че $\text{TA}(\lambda, t) = 1$. Ако $\lambda = x_i$, то $t(x_i) = 1$. Но тогава в неравенството, съответно на α , вляво има събираемо x_i , така че това неравенство е в сила. Ако $\lambda = \overline{x_i}$, то $t(x_i) = 0$. Но тогава в неравенството, съответно на α , вляво има събираемо $(1 - x_i)$, така че това неравенство е в сила.
- Да допуснем, че ϕ е НЕ-екземпляр на 3SAT. Тогава за всяка валюация $t \in \text{Val}(\text{Var}(\phi))$ съществува клауза α на ϕ , такава че за всеки литерал $\lambda \in \alpha$ е вярно, че $\text{TA}(\lambda, t) = 0$.

[†]В 16.4 “чисти нули” няма, понеже примерът е малък. “Чисти нули” се появяват, когато $n > 3$.

[‡]Защото редуцираме от 3SAT.

Нека α е произволна клауза на ϕ . Да разгледаме неравенството, съответно на α . То има $n - 3$ “чисти нули” и още 3 събираеми. За всяко от тях, ако е от вида x_i , то $x_i = 0$, а ако е от вида $(1 - x_i)$, то $x_i = 1$: това е необходимо, за да не бъде удовлетворена α от t . Но тогава всички n събираеми вляво в неравенството са нули, поради което то не е в сила. Ерго, поне едно неравенство в ψ не е в сила. $\square$

16.2.6 3-COLORABILITY, k-COLORABILITY и PLANAR 3-COLORABILITY

3-COLORABILITY $\in$ NP-с. Да си припомним 3-COLORABILITY.

Изм. Задача 82: 3-COLORABILITY

екземпляр: Неориентиран граф G .

въпрос: Дали G е върхово оцветим с ≤ 3 цвята?

Редукция 19: $3SAT \leq_p 3-COLORABILITY$

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

където

$$\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$$

Нека $\text{Var}(\phi) = \{x_1, \dots, x_n\}$. Ще конструираме граф-екземпляр $G = (V, E)$ на 3-COLORABILITY, такъв че G е оцветим с три цвята тстк ϕ е удовлетворима.

Първо, конструираме множество от $2n$ върха $V_1 = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. Чрез тях конструираме множество E_1 от n ребра, нито две от които нямат общ край:

$$E_1 = \{(x_i, \bar{x}_i) \mid 1 \leq i \leq n\}$$

Добавяме три нови върха z_1, z_2 и z_3 и конструираме множество от ребра E_2 :

$$E_2 = \{(x_i, z_1) \mid 1 \leq i \leq n\} \cup \{(\bar{x}_i, z_1) \mid 1 \leq i \leq n\} \cup \{(z_1, z_2), (z_1, z_3), (z_2, z_3)\}$$

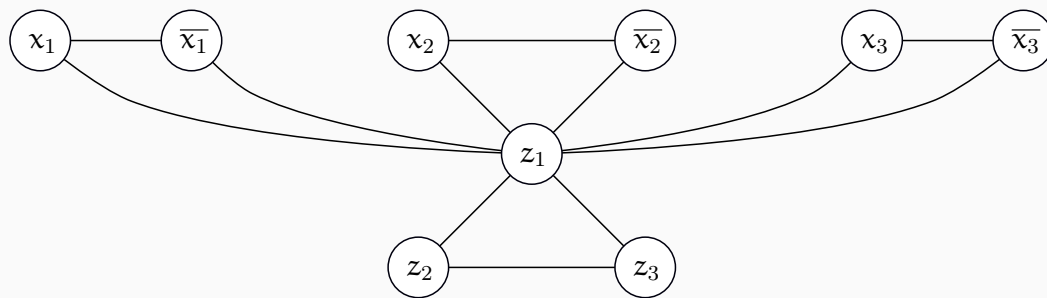
По този начин получаваме $n + 1$ на брой триъгълници с един общ връх z_1 . Това са приспособления от един вид, които конструкцията ползва. Преди да видим другия вид приспособления, да илюстрираме с пример конструкцията дотук. Да разгледаме отново ϕ от (16.3) като пример:

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_2} \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{\phi_3} \underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_4}$$

Фигура 16.11 показва четирите приспособления-триъгълници за този екземпляр.

Фигура 16.11 : Първият вид у-ва в редукцията $3SAT \leq_p 3-COLORABILITY$.

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

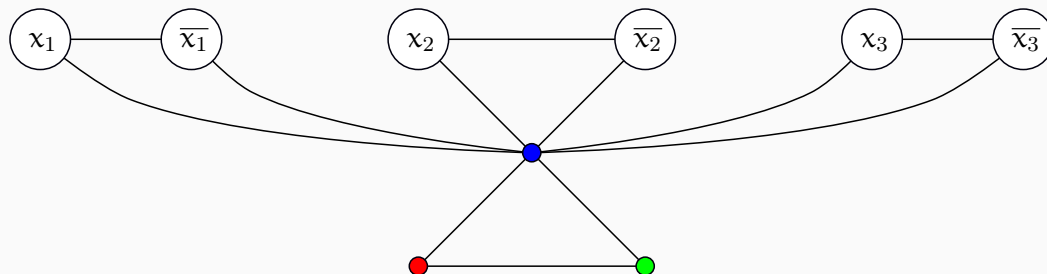


Да въведем трите цвята. Те не са част от конструкцията, но има смисъл да се въведат сега, за да е ясно какво се опитваме да постигнем. За целите на това изложение цветовете са ИСТИНА, ЛЪЖА и НЕУТРАЛЕН. Има смисъл да въведем ИСТИНА и ЛЪЖА, понеже свеждаме от 3SAT. Третият цвят НЕУТРАЛЕН присъства, понеже свеждаме към оцветимост с **три** цвята. За да е налице оцветимост с три цвята, за всеки триъгълник трябва всеки връх да е в точно един от трите цвята. На фигурите ще кодираме ИСТИНА със зелено, ЛЪЖА с червено и НЕУТРАЛЕН със синьо.

Читателят може би се досеща, че цветовете ИСТИНА и ЛЪЖА са за върховете от V_1 , понеже те отговарят на литерали, а литералите имат стойности 1 или 0 при дадена валуация. Тогава връх z_1 , който е съсед на всички върхове от V_1 , е НЕУТРАЛЕН, което влече, че точно единият от z_2 и z_3 е ИСТИНА, а другият е ЛЪЖА. Да кажем, че z_2 е ЛЪЖА, което прави z_3 да е ИСТИНА. Фигура 16.12 показва това частично раздаване на истини и лъжи.

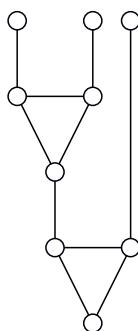
Фигура 16.12 : Частично 3-оцветяване на екземпляра от Фигура 16.11.

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \bar{x}_2 \vee \bar{x}_3)(\bar{x}_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$



Другият вид приспособления се състои от два триъгълника плюс едно ребро между два от

върховете им плюс още три ребра с висящи върхове-краища, както е показано на следната фигура.

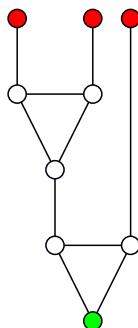


Тези приспособления се наричат *OR-gates*. Върхът от степен 2 (най-долу) се нарича *изходът*, а трите висящи върхове (най-горе) се наричат *входовете*. Сега ще стане ясно защо. Дефинираме, че оцветяване на OR-gate е *подходящо*, ако входовете и изходът получават само ИСТИНА или ЛЪЖА; тоест, никога НЕУТРАЛЕН. Очевидно това има смисъл, ако OR-gate-ът трябва да симулира логически елемент в двузначна логика.

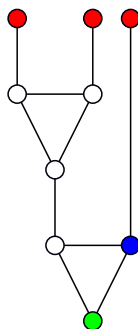
Лема 60

В текущия контекст, за всяко подходящо оцветяване на OR-gate, изхода е оцветен в ИСТИНА тстк поне единият вход е оцветен в ИСТИНА.

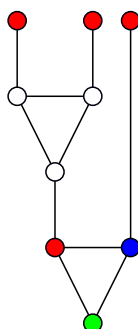
Доказателство: В едната посока, ще покажем, че е невъзможно изходът да е истина, а и трите входа да са лъжа. Да допуснем, че това е възможно.



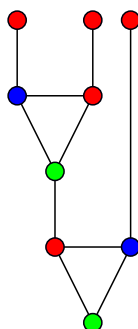
Върхът, който е между истината и лъжата, трябва да е неутрален:



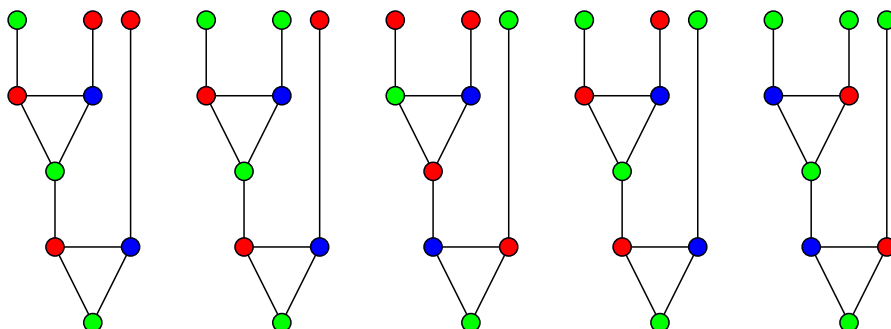
Тогава третият връх от долният триъгълник трябва да е лъжа:



При това положение, няма как да оцветим трите върха на горния триъгълник в истина, лъжа и неутрален, без да допуснем ребро с два края лъжа. Примерно:



В другата посока, нека поне един от входовете е истина. Тогава приспособлението може да бъде 3-оцветено по подходящ начин:



С което доказателството приключи. □

Да довършим конструкцията. За всяка клауза $\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$ правим по един OR-gate, след което:

1. Идентифицираме единия му вход с върха $\lambda_{1,j} \in V_1$, втория му вход с върха $\lambda_{2,j} \in V_1$ и третия му вход с върха $\lambda_{3,j} \in V_1$. Привидно, входовете не са симетрични, но е лесно да се убедим, че са симетрични по отношение на конструкцията – няма значение кой вход на OR-gate-а с кой връх-литерал се идентифицира, стига това да става по биективен начин.

2. Слагаме две нови ребра, единият край на всяко от които е изходът на OR-gate-a, а другите два края са z_1 и z_2 .

Обосновката на коректността е много лесна, ако си спомним, че z_1 е в цвят НЕУТРАЛЕН, а z_2 е в цвят ЛЪЖА. Това налага изходът на всеки OR-gate да е в цвят ИСТИНА. Ясно е, че всеки връх от V_1 е или истина, или лъжа, заради триъгълниците с два върха от V_1 и трети връх z_1 , който е неутрален. Съгласно Лема 60, графът е 3-оцветим тстк във всяка клауза има поне един литерал, който е истина. Обосновахме конструкцията.

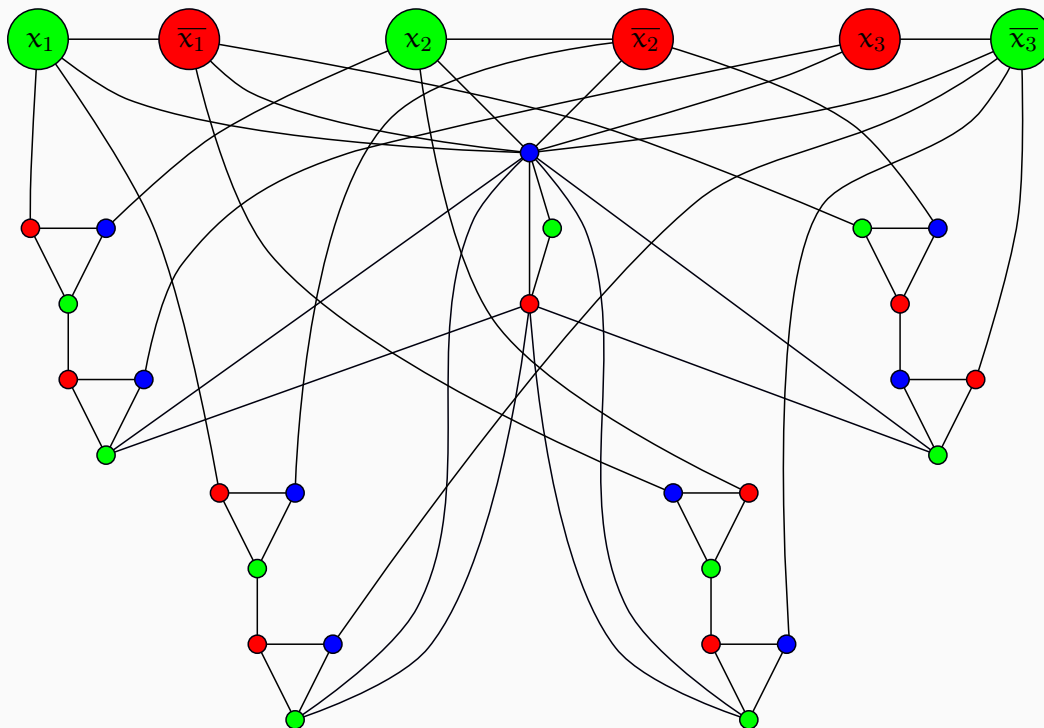
А това, че конструкцията може да се извърши в полиномиално време, е очевидно. $\square$

Забележете, че имената на цветовете—ИСТИНА, ЛЪЖА и НЕУТРАЛЕН—са само за удобство на читателя, който или която трябва да се убеди в коректността на конструкцията. В действителност, по дадена 3SAT КНФ ние правим граф, чийто върхове се изобразяват върху 3-елементно множество без конфликти върху ребрата тстк КНФ е удовлетворима. Това е всичко съществено. Имената на цветовете са подбрани по този начин и казваме, че връх z_1 е в цвят НЕУТРАЛЕН, само за да можем после да кажем, че поне един връх от всяко ребро от E_1 е в цвят ИСТИНА. Кое то, в света на КНФ, се транслира като “всяка клауза е удовлетворена”.

Фигура 16.13 показва цялата конструкция за ϕ от (16.3). Формулата е удовлетворима от валуацията $t = \langle 1, 1, 0 \rangle$ и на фигурата е показано съответно 3-оцветяване.

Фигура 16.13 : Да-екземпляр на 3SAT и съотв. Да-екземпляр на 3-COLORABILITY.

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$



k -COLORABILITY $\in$ NP-с. Да си припомним k -COLORABILITY.

Изч. Задача 83: k -COLORABILITY

екземпляр: Неориентиран граф G , естествено число k .

въпрос: Дали G е върхово оцветим с $\leq k$ цвята?

Редукция 20: 3 -COLORABILITY $\leq_p$ k -COLORABILITY

Конструкция: Това е най-лесният вид редукция. Всеки екземпляр на 3 -COLORABILITY е и (почти) екземпляр на k -COLORABILITY. Даден е граф G , който е екземпляр на 3 -COLORABILITY. Редукцията генерира наредена двойка $\langle G, 3 \rangle$, която е екземпляр на k -COLORABILITY. Очевидно е, че G е ДА-екземпляр на 3 -COLORABILITY тстк $\langle G, 3 \rangle$ е ДА-екземпляр на k -COLORABILITY, така че практически няма какво да се обосновава. $\square$

Аналогично, всеки екземпляр на 3 SAT е и екземпляр на SAT. Буквално. Без “почти”. Обаче редукцията $SAT \leq_p 3SAT$ (Редукция 10) не беше съвсем тривиална. Направихме подробно формално доказателство, за да я обосновем. Причината е посоката на редукцията. **Ако** поначало бяхме показали в Теорема 126, че $3SAT$, а не SAT , е NP-пълна, **то** доказателството за NP-пълнотата на SAT щеше да е много лесно, понеже редукцията $3SAT \leq_p SAT$ е толкова тривиална, колкото 3 -COLORABILITY $\leq_p$ k -COLORABILITY.

PLANAR 3-COLORABILITY $\in$ NP-с. Задачата е 3 -COLORABILITY, но върху планарни графи. Тъй като планарните графи са строго подмножество на графите, Редукция 19 не ни дава директно желанния резултат.

Ако не е ясно защо, помислете за това: дърветата също са строго подмножество на графите и всяко дърво е 2-оцветимо, откъдето е и 3-оцветимо; ерго, задачата за 3-оцветяване на дърветата не просто е в P , а е решима в константно време от алгоритъм, който винаги връща ДА. И така, от това, че сме доказали NP-пълнотата на някаква задача върху графите по принцип, не следва автоматично доказателство, че тя остава NP-пълна и върху някакви ограничени графи. Може да е така, можа да не е така. Трябва да се изследва.

Изч. Задача 84: PLANAR 3-COLORABILITY

екземпляр: Неориентиран планарен граф G .

въпрос: Дали G е върхово оцветим с ≤ 3 цвята?

Редукция 21: 3 -COLORABILITY $\leq_p$ PLANAR 3-COLORABILITY

Конструкция: Даден е екземпляр на 3 -COLORABILITY; тоест, граф $G = (V, E)$ с n върха. Ще конструираме екземпляр на PLANAR 3-COLORABILITY, тоест, планарен граф, който е 3-оцветим тстк G е 3-оцветим и чийто размер е най-много полиномиално по-голям от размера на G .

Възможно е G да не е планарен. Конструираме планарно вписване $\mathcal{G} = (V, \mathcal{E})$ на G , като обаче допускаме пресичания на планарни ребра във вътрешни точки. Ако G не е планарен, пресичанията на планарни ребра във вътрешни точки са неизбежни. Настояваме

- всеки две планарни ребра да имат най-много една обща точка,

- ако две планарни ребра имат обща точка, тя или да е крайна и за двете, или да е вътрешна и за двете, и
- всяка точка да принадлежи на не повече от две планарни ребра.

Това може да се постигне лесно, ако

1. планарните върхове се разположат като върхове на регулярен n -ъгълник,
2. всяко планарно ребро се реализира като отсечка, свързваща двойка планарни върхове,
3. за всяка обща вътрешна точка на повече от две планарни ребра, техните краища да се преместят (малко) така, че да се получат $\binom{k}{2}$ общи вътрешни точки, всяка принадлежаща на точно 2 от тези ребра; приемаме за очевидно, че това може да се направи.

И така, две планарни ребра на $\mathcal{G}$ или не се пресичат изобщо, или се пресичат в точка, която е край и на двете (и е планарен връх), или се пресичат в точка, която е вътрешна и за двете. Разглеждаме само вътрешните пресечни точки. Именуваме ги $p_1, \dots, p_t$. Нека $P = \{p_1, \dots, p_t\}$. Добре известно е, че $t \leq \binom{n}{4}$ и тази граница е точна. Същественото за нашата конструкция е, че t е ограничено от полином на n .

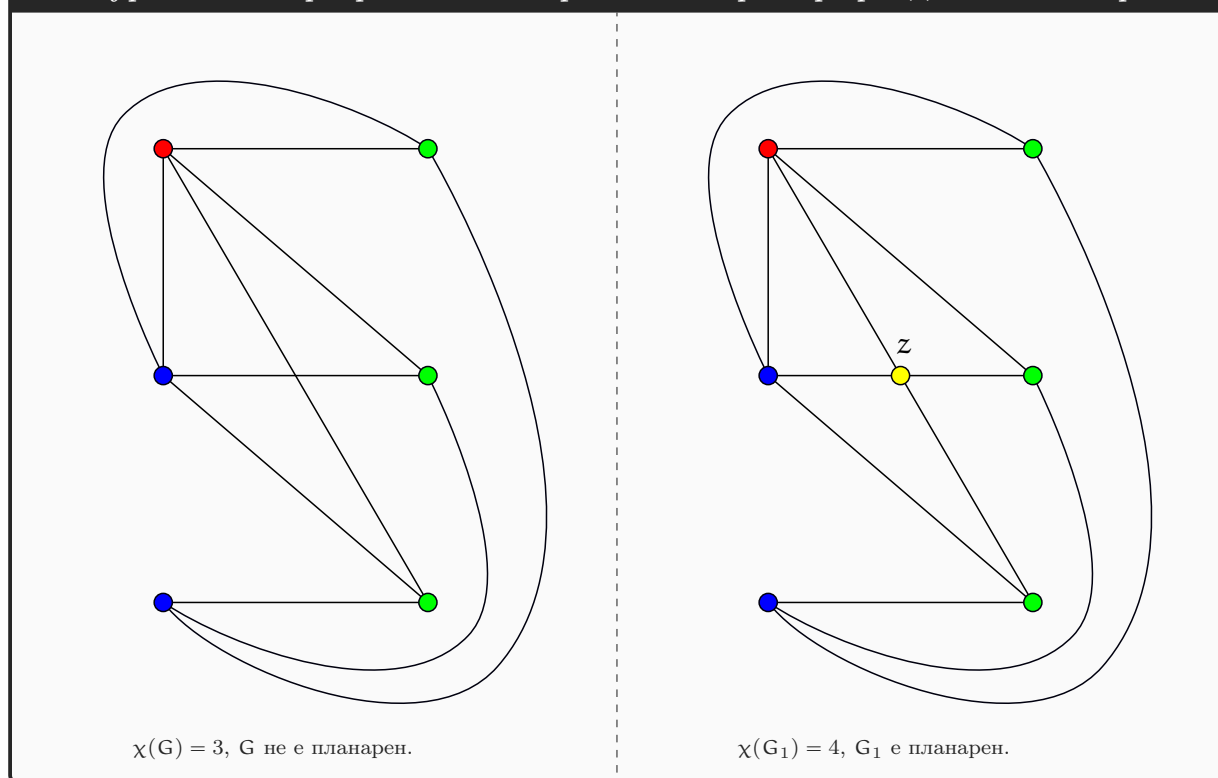
Първото нещо, което ни хрумва, е да направим всяка p_j нов планарен връх. Така конструираме ново планарно вписване $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$.

- $\mathcal{V}_1 = \mathcal{V} \cup \{p_1, \dots, p_t\}$.
- Всяко ребро на $\mathcal{E}$ бива подразделено на $k + 1$ ребра от k -те точки измежду $p_1, \dots, p_t$, които му принадлежат, където $k \geq 0$. $\mathcal{E}_1$ се състои точно от така получените ребра.

Очевидно $\mathcal{G}_1$ е планарно вписване, в което никои две планарни ребра не се пресичат във вътрешни точки. По този начин $\mathcal{G}_1$ е истинско планарно вписване, за разлика от $\mathcal{G}$, в което може да има пресичания на планарни ребра не в краищата. Нека G_1 е графът, който съответства на $\mathcal{G}_1$. Очевидно G_1 е планарен и е с размер, не повече от полиномиално по-голям от размера на G .

За съжаление, G_1 не е графът, към който можем да сведем в конструкцията. Пример за това има на Фигура 16.14. G е 3-оцветим и не е планарен, понеже съдържа $K_{3,3}$ като подграф. Показаното планарно вписване е с точно едно пресичане на планарни ребра във вътрешна точка. Ако тази точка стане нов връх z , който подразделя две ребра на G в четири ребра и полученият граф е G_1 , то $\chi(G_1) = 4$.

Фигура 16.14 : Превр. на непланарен в планарен граф с добавяне на връх.



Допълнение 82: Намиране на хроматичното число чрез Maple (TM)

Хроматичното число на граф може да бъде изчислено с Maple (TM), стига графът да е достатъчно малък. Да вземем за пример G_1 от Фигура 16.14, който е достатъчно малък, така че да може да бъде оцветен оптимално от алгоритъм с брутална сила. Командата на Maple (TM) за целта е **ChromaticNumber**. Преди това трябва да построим графа, което става с командата **Graph**. А преди да я използваме, трябва да заредим пакета **GraphTheory**.

```
C:\Users\Minko>"C:\Program Files\Maple 2018\bin.X86_64_WINDOWS\cmacle"
| \~ / |      Maple 2018 (X86 64 WINDOWS)
._ | \ | / | _ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2018
 \  MAPLE  /   All rights reserved. Maple is a trademark of
 <_ _ _ _ _>   Waterloo Maple Inc.
 |              Type ? for help.
> with(GraphTheory):
> G1 := Graph({{u1, u2}, {u1, v1}, {u1, v2}, {u1, z}, {u2, v\
> 1}, {u2, v3}, {u2, z}, {u3, v1}, {u3, v2}, {u3, v3}, {v2, z}, {v3, z}}):
> ChromaticNumber(G2, col2);

4

> col2;

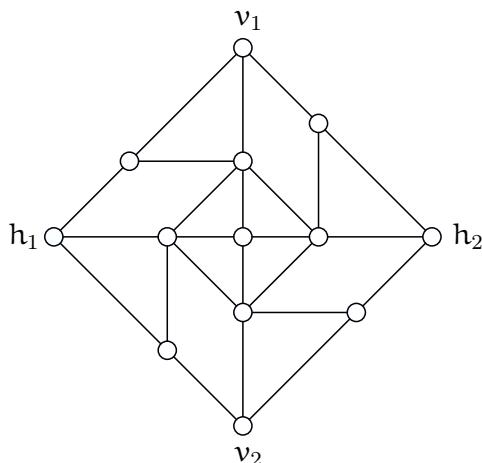
[[u1, v3], [v1, z], [u2, v2], [u3]]

> quit
memory used=2.9MB, alloc=8.3MB, time=0.14
```

Описанието на графа трябва да е очевидно ясно. Това са ребрата на графа G_1 от Фигура 16.14, като трите неименувани върха вляво са u_1 , u_2 и u_3 , а тези вдясно са

v_1 , v_2 и v_3 . Командата `ChromaticNumber` получава като втори аргумент списък (тип данни `list`), в който записва някое оптимално оцветяване. В случая списъкът е `col2`. Съдържанието му виждаме, като просто напишем името му като команда.

И така, полученият граф G_1 може да има по-голямо хроматично число от това на G . За това ще трансформираме G по друг начин в планарен граф G_2 , който начин запазва 3-оцветимостта. За целта въвеждаме това приспособление:



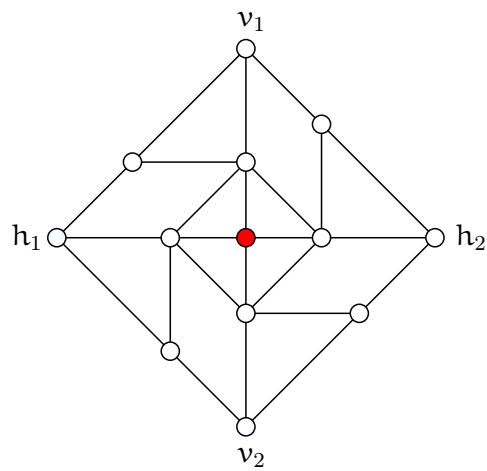
Приспособлението е планарен граф с 13 върха и 24 ребра. Четирите върха с имена на илюстрацията са *краищата* на приспособлението. Те се групират в две двойки на *вертикалните краища* v_1 и v_2 , и *хоризонталните краища* h_1 и h_2 . Останалите върхове остават анонимни. Ето важните за нашата конструкция свойства на приспособлението.

Факт 1 Хроматичното число на приспособлението е 3.

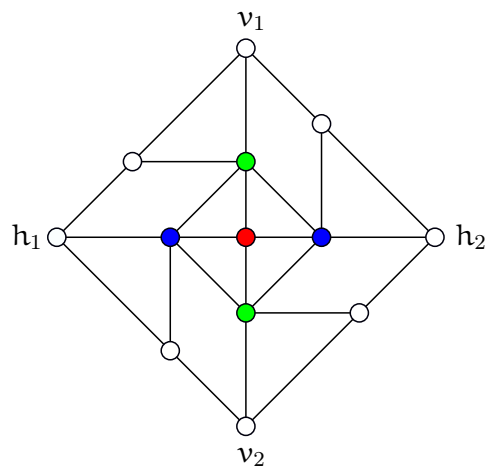
Факт 2 За всяко 3-оцветяване на приспособлението, вертикалните краища са в един и същи цвят c_v . Хоризонталните краища също са в един и същи цвят c_h , който може да съвпада или да не съвпада с c_v .

Факт 3 Всяко оцветяване на вертикалните краища в един и същи цвят c_v и на хоризонталните краища в един и същи цвят c_h еднозначно определя цветовете на анонимните върхове, ако искаме да имаме 3-оцветяване на приспособлението. Това остава в сила независимо от това дали $c_v = c_h$ или $c_v \neq c_h$.

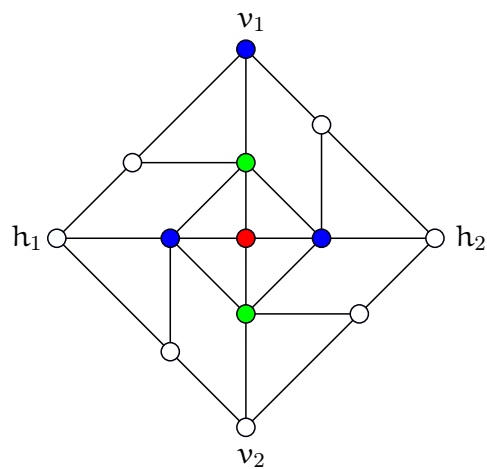
Ето конструктивно доказателство и на трите факта. Да използваме червен, зелен и син цвят. БОО, нека върхът в центъра е червен:



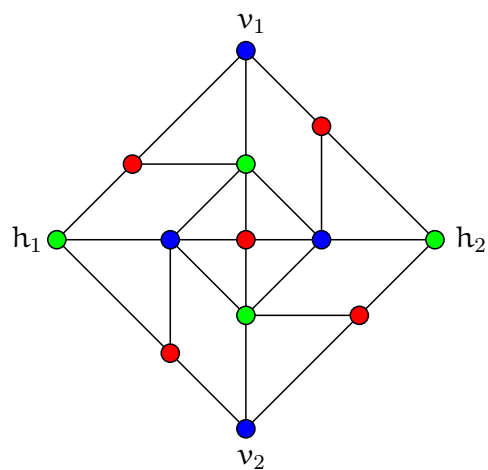
Това форсира син и зелен цвят върху съседите му, които алтернират по цикъла, който образуват. Да кажем:



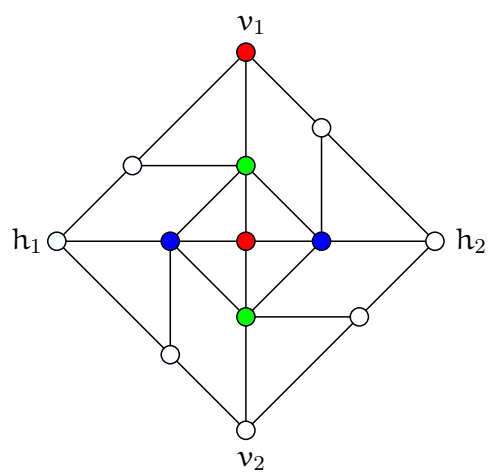
За v_1 има избор между синьо и червено. Първо да допуснем, че v_1 е син.



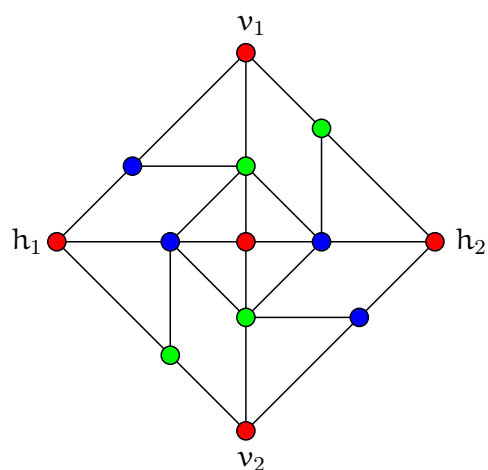
Това форсира оцветяването на останалите върхове.



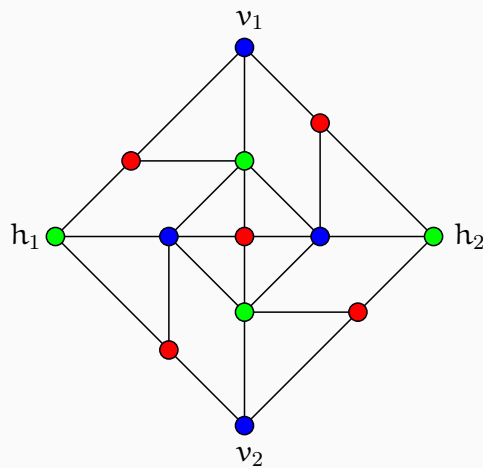
Сега да допуснем, че v_1 е червен.



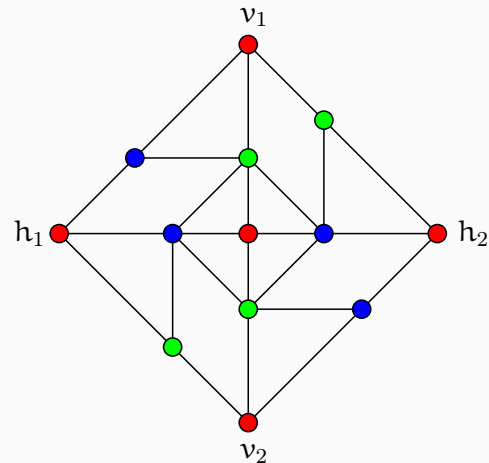
Това също форсира оцветяването на останалите върхове.



Фигура 16.15 показва двете възможни, с точност до симетрия, 3-оцветявания на приспособлението, ако централният връх е червен.

Фигура 16.15 : 3-оцветяванията на джаджата от 3-COLOR. $\leq_p$ PLANAR 3-COLOR.

Вертикалните и хоризонталните краища са в различни цветове.



Краищата са в един и същи цвят.

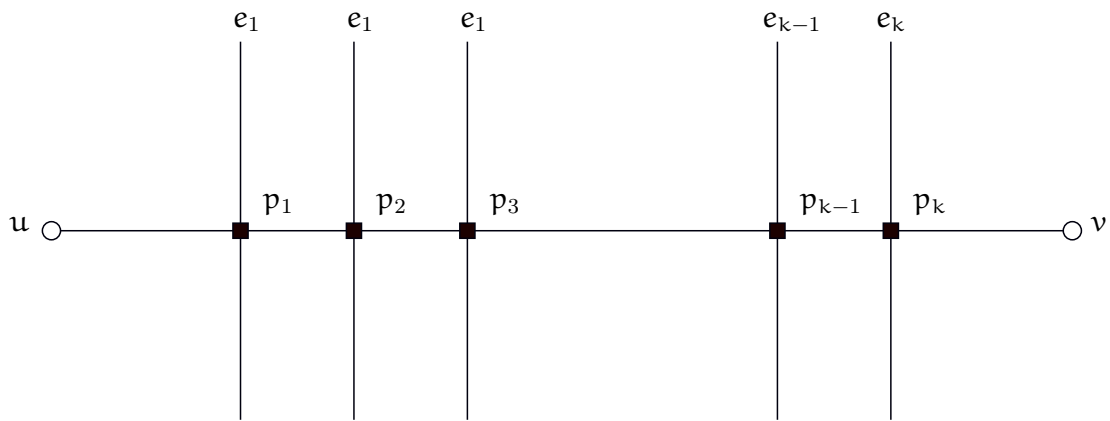
Ето обяснение на идеята на конструкцията, използваща приспособленията. За $j \in \{1, \dots, t\}$, заменяме всяка p_j плюс достатъчно малка околност около нея с (планарното вписване на) едно приспособление D_j ето така:



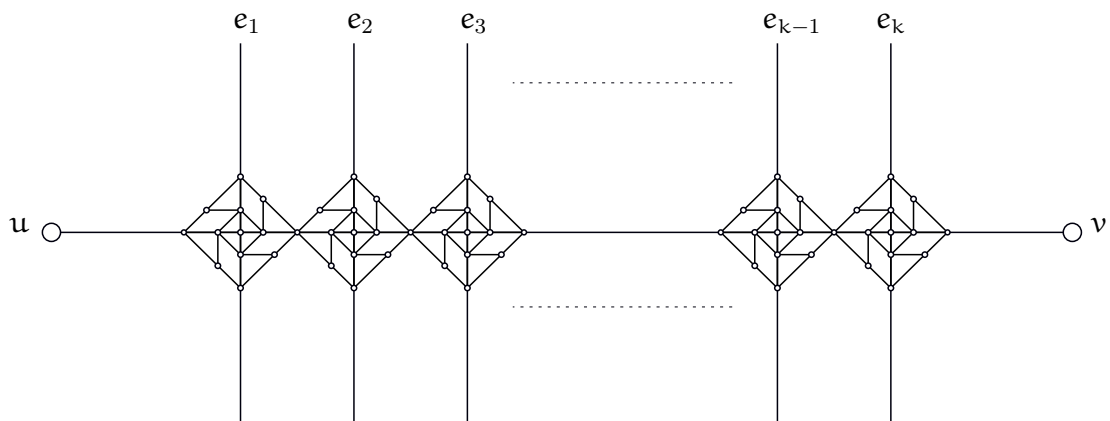
Това е добре дефинирано, понеже всяка p_j е обща крайна точка за точно четири планарни ребра на $\mathcal{G}_1$, а приспособленията имат по четири краища.

Но конструкцията е по-сложна. По протежение на всяко планарно ребро e на **оригиналния** планарен граф $\mathcal{G}$, приспособленията трябва да са слепени едно за друго и за единия от краищата на e по начин, който сега ще опишем подробно.

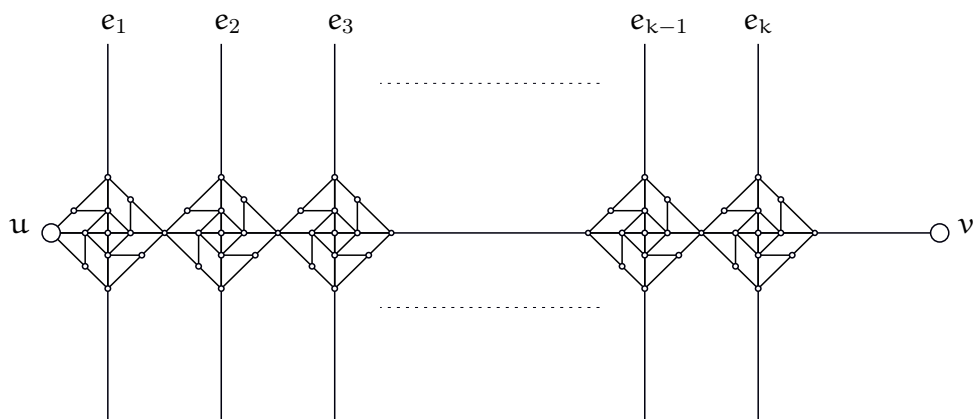
Нека $e = (u, v)$ е произволно планарно ребро на $\mathcal{G}$, което съдържа k точки от P за някое $k \in \{1, \dots, t\}$. БОО, нека точките са $p_1, \dots, p_k$, в този ред от u към v . Тези точки са общи за e и някакви други ребра $e_1, \dots, e_k$:



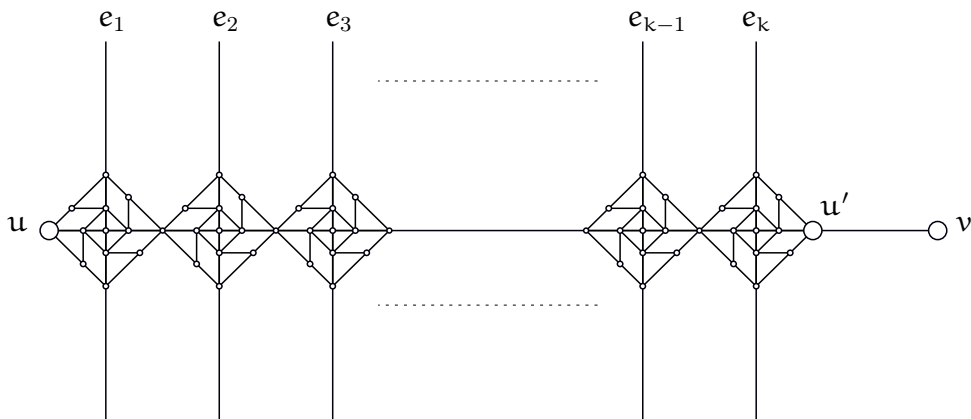
Слагаме по едно приспособление на мястото на всяко p_j , $1 \leq j \leq k$, като ги “слепваме” по следния начин:



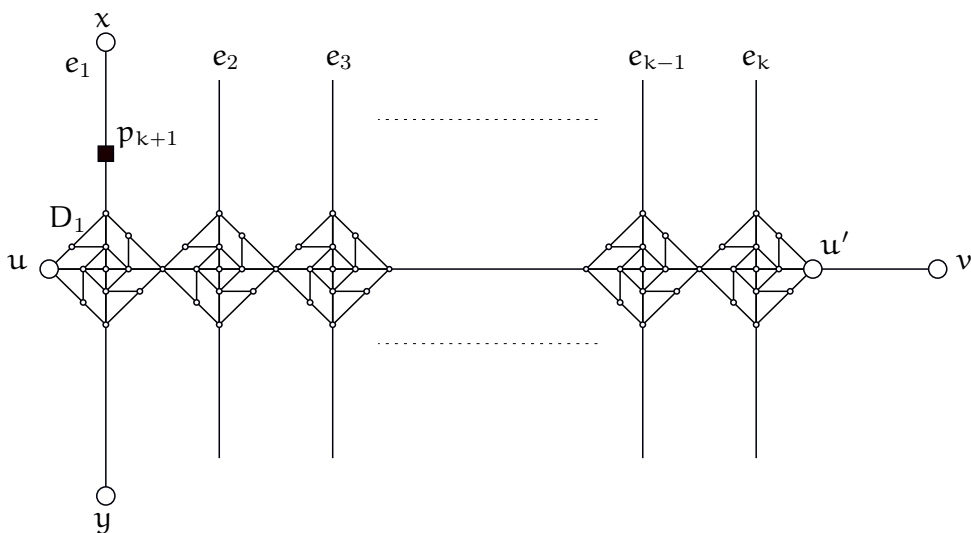
При това въпросните $p_1, \dots, p_k$ изчезват. След това избираме единия край на e , няма значение кой, за *особен*. Да кажем, че u е особеният край в примера. “Слепваме” u с най-близкия край на най-близкото до u приспособление, при което те стават един връх, а реброто между тях изчезва:



Разглеждаме и най-отдалеченият от u край на веригата от слепени приспособления. Това е върхът, който е съсед на v . Да го наречем u' :

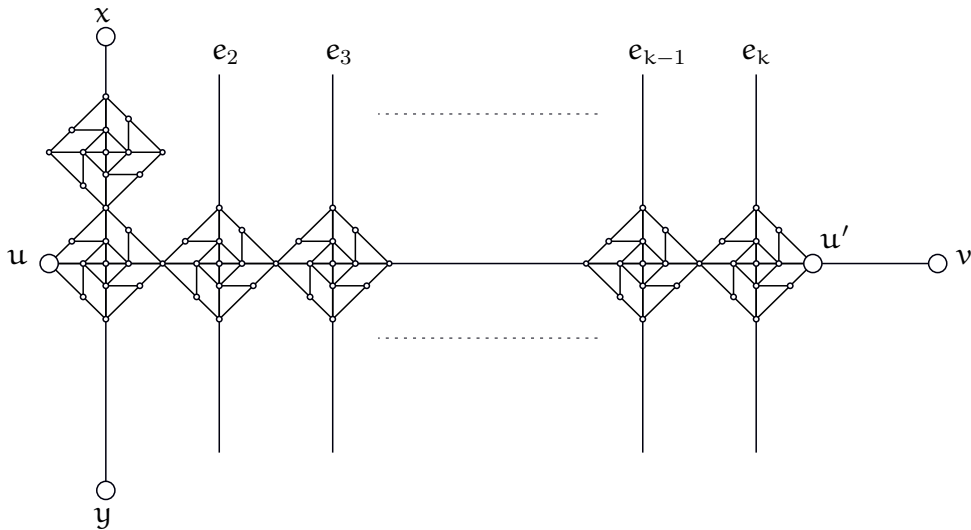


Според Garey и Johnson [56, стр. 88–89], с това описанието е конструкцията приключва. Според автора на записките, уместно е да се добави, че описаното слагане на верига от приспособления и идентифициране на два върха се извършва по точно този начин само за първото ребро $e = (u, v)$, върху което действаме. Ако след това действие все още има точки от P , които трябва да бъдат “обработени” (тоест, ако $k < t$), налага се да “обработим” поне още едно ребро, съдържащо точки от P . Ако това друго ребро е някое от $e_1, \dots, e_k$, няма да действаме по точно същия начин. БООО[†], нека е това друго ребро $e_1 = (x, y)$ и нека точката е само една – точка p_{k+1} . Когато “се захванем” с e_1 , то вече минава през приспособление D_1 , с което заменихме p_1 :

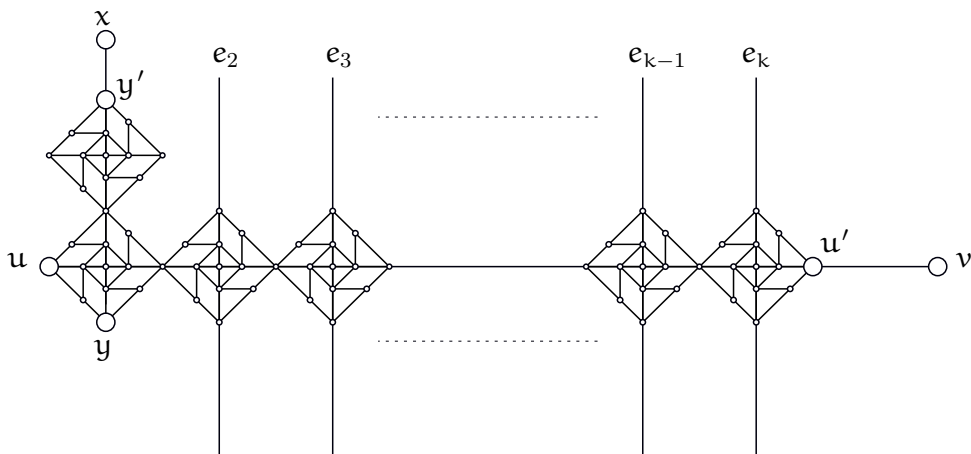


p_{k+1} бива заменена с приспособление, като се получава верига от две слепени в краищата си приспособления:

[†]Без Особено Ограничение на Общността.



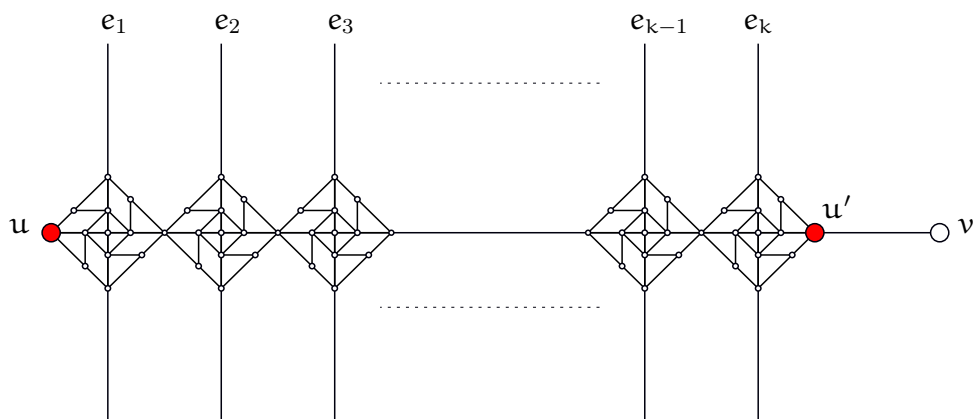
След което единият край на (бившето) e_1 , да кажем y , бива обявен за особен, което означава да го слепим с най-близкия край на веригата от приспособления, а другият край на веригата да стане y' :



Би трябвало да е ясно как да се действа, докато всички точки от P не изчезнат, бивайки заменени от приспособления. С това описанието на конструкцията приключва. Конструираното планарно вписване наричаме $\mathcal{G}_2$, а неговият съответен граф наричаме G_2 .

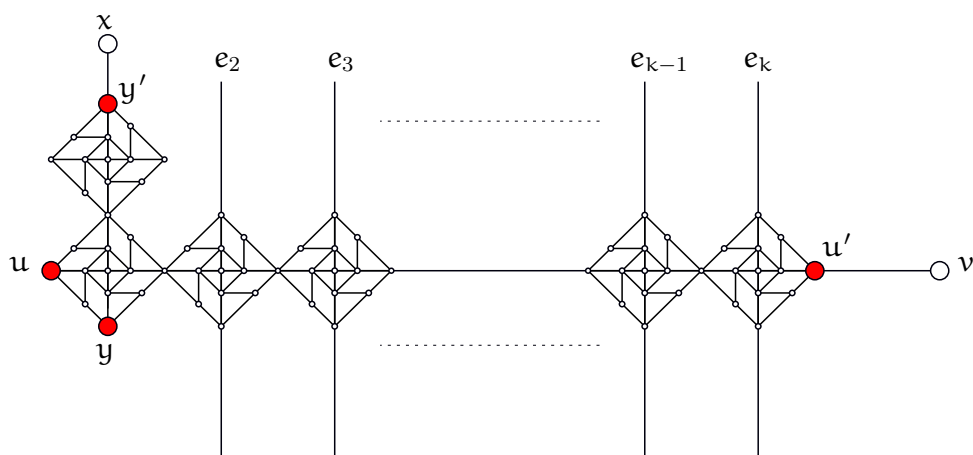
Ще докажем коректността. В едната посока, нека G е 3-оцветим. Ще докажем, че G_2 е също 3-оцветим. Нека f е оцветяване на върховете на G в не повече от 3 цвята. f се явява и 3-оцветяване на $\mathcal{G}$, ако дефинираме “оцветяване на планарно вписване” по най-естествения начин. Ще покажем, че f може да бъде разширено до 3-оцветяване f_2 на $\mathcal{G}_2$, откъдето то задава и 3-оцветяване на графа G_2 .

За всяко планарно ребро $e = (u, v)$ на $\mathcal{G}$, което има пресечни точки с други планарни ребра, редукцията заменя тези точки с приспособления по начин, който вече видяхме. Получаваме $\mathcal{G}_2$. Нека u е особеният връх на e . Ключовото наблюдение е, че веригата от приспособления “пренася” цвета на u към u' , в смисъл, че във всяко 3-оцветяване на $\mathcal{G}_2$, цветовете на u и u' съвпадат. Това следва тривиално от **Факт 2**, илюстриран на Фигура 16.15.

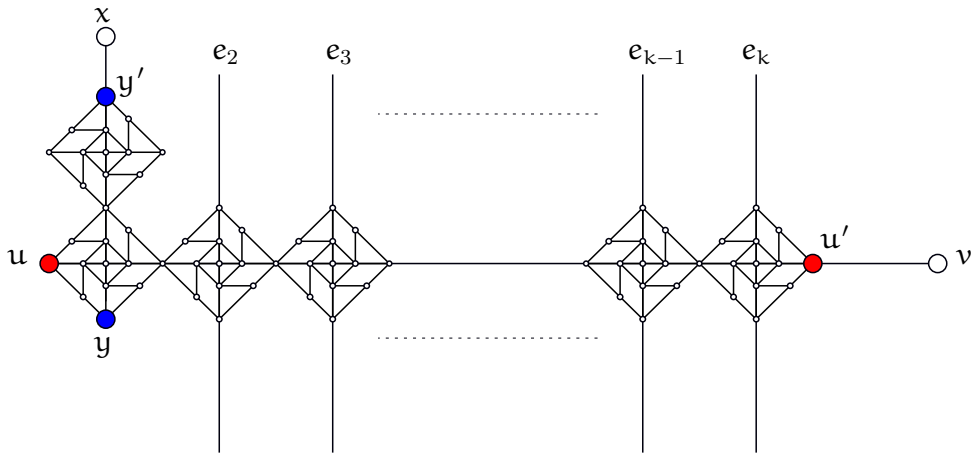


Иначе казано, за да получим цветове за новодобавените върхове, правим цвета на u' същия като цвета на u и знаем, че можем да раздадем цветове и на останалите върхове от приспособленията по протежение на e , така че 3-оцветимостта да е налице.

Нещо повече. Приспособленията, разположени по протежение на e , могат да пренасят цветове и в “ортогоналното направление”; тоест, по веригите, сложени върху планарните ребра $e_1, \dots, e_k$, пресичащи e . Така нареченото “пренасяне” отново означава, че особеният връх и неговото примовано съответствие в другия край на веригата от приспособления са в един и същи цвят: $f_2(u') \leftarrow f(u)$, $f_2(y') \leftarrow f(y)$, и така нататък. Пренасянето на цветове работи независимо от това дали цветовете на u и y съвпадат:



или не съвпадат:



f_2 е 3-оцветяването на $\mathcal{G}_2$, което се получава от f с добавяне на цветове на върховете на приспособленията по такъв начин, че за всяко планарно ребро (u, v) , върху което се слагат приспособления, ако u е особенният връх, то $f_2(u') \leftarrow f(u)$. Ако краищата на реброто (u, v) на G са в различни цветове под f , то краищата на планарното ребро (u', v) на $\mathcal{G}_2$ са в различни цветове под f_2 , а съгласно **Факт 3** има начин да оцветим и останалите върхове на приспособленията, така че да получим легитимно 3-оцветяване f_2 на $\mathcal{G}_2$.

В обратната посока, нека f_2 е 3-оцветяване на G_2 , оттам и на $\mathcal{G}_2$. Твърдим, че $f = f_2|_v$ е 3-оцветяване на $\mathcal{G}$, а оттам и на G . Да допуснем противното. Тогава има планарно ребро $(u, v) \in \mathcal{E}$, такова че $f(u) = f(v)$. Но (u, v) трябва да е планарно ребро, върху което редукцията е сложила приспособления. Съгласно **Факт 2**, хоризонталните или вертикалните краища на всяко приспособление са в един и същи цвят под каквото и да е 3-оцветяване, което влече, че по която и да е верига от приспособления, особенният връх и неговото примовано съответствие са в един и същи цвят. Ерго, $f_2(u) = f_2(u')$. Щом f_2 е легитимно 3 оцветяване, трябва $f_2(u') \neq f_2(v)$, понеже (u', v) е ребро на $\mathcal{G}_2$. Но $f(u) = f_2(u)$ и $f(v) = f_2(v)$. Заклучаваме, че $f(u) \neq f(v)$, в противоречие с прежде направеното допускане. ⚡

Показахме, че конструкцията е коректна. Това, че може да се извърши в полиномиално време, стига броят на точките на пресичане $p_1, \dots, p_t$ в $\mathcal{G}$ да е полиномиално ограничен от размера на G , е очевидно. $\square$

Заслужава си да се спомене и следното. В началото на Секция 16.2 казахме, че за всички задачи в нея, принадлежността към **NP** е очевидна. За PLANAR 3-COLORABILITY това не е точно така. Във формалното описание (Задача 84) се казва, че екземплярът е **планарен** граф. Недетерминираната машина на Turing M , която прави проверка за принадлежност към **NP**, първо трябва да провери дали стрингът, който получава като вход, наистина кодира **планарен** граф по отношение на избраното кодиране. И тази проверка трябва да стане в полиномиално в дължината на този стринг време.

Известни са полиномиални алгоритми, изчисляващи дали граф е планарен или не. Пример за такъв алгоритъм е алгоритъмът на Hopcroft и Tarjan с линейна сложност (вижте [78]). И така, ако M започне с викане на този алгоритъм върху входа, тя очевидно може да се реализира като недетерминирана машина с полиномиална сложност по време. Проверката за планарност е детерминирана; недетерминираното изчисление е отгатване на 3-оцветяване, последващо от проверка за валидност (че наистина е 3-оцветяване на този граф).

Още нещо, което си заслужава да се отбележи тук, е че всеки планарен граф е 4-оцветим, както знаем от курса Дискретни Структури. Съгласно теоремата за четирите цвята, за всеки

планарен граф G е вярно, че $\chi(G) \leq 4$. Нещо повече: има ефикасни алгоритми, изчисляващи 4-оцветяване на произволен планарен граф (вижте например [133]). Ерго, по отношение на PLANAR 3-COLORABILITY, неподатливостта се е само в това, че не можем да определим ефикасно дали $\chi(G) = 3$ или $\chi(G) = 4$. Дори да гледаме на PLANAR 3-COLORABILITY като на оптимизационна задача “Какъв е най-малкият брой цветове, с които можем да оцветим даден планарен граф G ?”, неподатливостта е в определянето на един бит информация: 2-оцветимостта е тривиална, 4-оцветимостта е налична винаги, и това, което не можем да пресметнем ефикасно, е 3-оцветимостта.

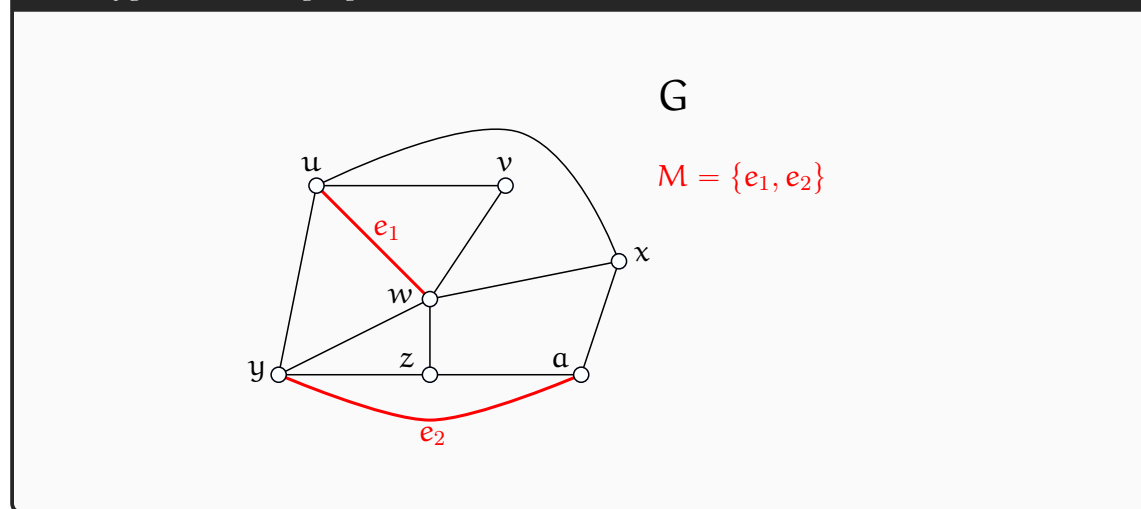
16.2.7 3-DIMENSIONAL MATCHING (3DM)

Пълното име на задачата е “3-DIMENSIONAL MATCHING”, а “3DM” е съкратеното име. Първо ще разгледаме двумерната ѝ версия СЪЧЕТАНИЕ.

16.2.7.1 СЪЧЕТАНИЕ – дефиниция, частни случаи и приложения

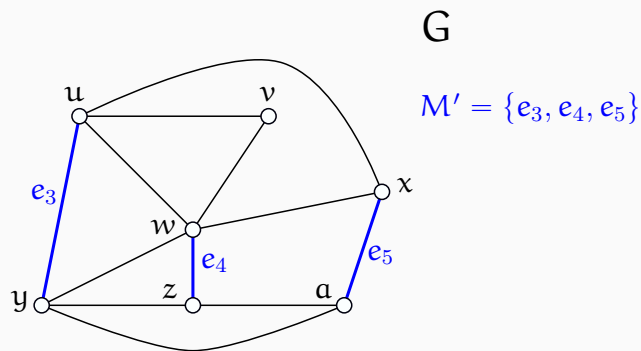
Става дума за добре известната задача СЪЧЕТАНИЕ, на английски MATCHING, в графи. Даден е граф $G = (V, E)$. Всяко $M \subseteq E$, такова че нито две ребра от M нямат общ край[†], се нарича *съчетание* в G , на английски *matching*. Ако $U \subseteq V$, казваме, че U е *съчетано от* M , ако всеки връх от U е инцидентен с ребро от M . Фигура 16.16 показва граф и съчетание M с две ребра в него (в червено). Върховете u, w, y и a са съчетани от M , а v, x и z не са съчетани от M .

Фигура 16.16 : Граф G и макс. по включване съчетание M в него.



Задачата СЪЧЕТАНИЕ като оптимизационна задача е максимизационна: да се намери съчетание с максимална мощност. Примерно, M на Фигура 16.16 е максимално по включване, понеже към него не може да добавим нито едно ребро и резултатът да остане съчетание. Фигура 16.17 показва максимално по мощност съчетание в същия граф. Наистина, в граф със седем върха не може да има съчетание с повече от три ребра.

[†]Такова множество от ребра на английски често се нарича *independent edge set*. В тези записки ще избягваме да казваме *независими ребра*, защото вече сме дефинирали *независими пътища* като пътища, които по двойки нямат общи вътрешни върхове; това допуска да имат общи краища. Очевидно двете дефиниции се несъвместими, понеже всяко ребро е и път.

Фигура 16.17 : Макс. по мощност съчетание M' в G от Фигура 16.16.

Числото на максималното съчетание на граф G е мощността на максимално по мощност съчетание в G . Често се бележи с " $\nu(G)$ ". На английски е *matching number*.

Да се намери съчетание, което е максимално по включване, е елементарно и става с очевидния алчен алгоритъм. Намирането на максимално по мощност съчетание е нетривиална задача. Дори се е смятало, че задачата е неподатлива, докато Edmonds не е открил кватричен алгоритъм за нея през 1965 г. [42]. Най-бързият известен алгоритъм за СЪЧЕТАНИЕ е на Micali и Vazirani, който има сложност $O(\sqrt{nm})$ [115].

Задачата има големи приложения на практика. За много задълбочено описание на нейни приложения, както и различни нейни варианти и обобщения, вижте книгата на Lovasz и Plummer [130].

Допълнение 83: Едно приложение на СЪЧЕТАНИЕ в общите графи

Обикновено учебниците и учебните помагала посочват приложения на СЪЧЕТАНИЕ върху двуделни графи. Lovasz и Plummer посочват интересно приложение на СЪЧЕТАНИЕ върху графи, които не са непременно двуделни [130, стр. vii-viii]. Да кажем, че са дадени n работи (computer jobs) $j_1, \dots, j_n$ и два еднакви компютъра и всяка работа j_i може да бъде изпълнена върху кой да е от компютрите. Всяка j_i се извършва за едно и също време; да кажем, за единица време. Дадени са и ограничения от вида " j_i трябва да се извърши преди j_k , понеже j_k зависи от резултата от j_i ". Да кажем, че тези ограничения са смислени; тоест, те представляват даг. За колко време най-малко може да бъдат изпълнени работите, така че ограниченията да бъдат спазени?

Ако компютърът беше само един, щяхме да топо-сортираме дага и да изпълним работите в този ред за n единици време – по-бързо не може, ако компютърът е един. При два компютъра, ако нямаше такива ограничения, щяхме да изпълним работите за $\lceil \frac{n}{2} \rceil$ единици време. Но наличието на ограничения усложнява нещата.

Ще моделираме задачата с **неориентиран** граф G , чиито върхове са $j_1, \dots, j_n$, а ребро (j_i, j_k) се слага тстк j_i и j_k са несравними в строгата частична наредба R , която съответства на дага (R е транзитивното затваряне на дага). Да са несравними означава, че могат да се изпълнят едновременно върху двата компютъра. Забележете, че несравнимостта е симетрична релация, така че наистина G е неориентиран, въпреки че графът на ограниченията е ориентиран.

За да извършим всички работи за минимално време, трябва да използваме и двата

компютъра колкото се може по-пълноценно. Това ще рече да максимизираме единиците време, в които двата компютъра работят едновременно. Сега ще се убедим, че това време е $\nu(G)$. Нека M е максимално по мощност съчетание в G .

- За всяко $(u, v) \in M$ е вярно, че работите u и v могат да бъдат изпълнени едновременно.
- Ако имаше начин да изпълним повече работи по двойки върху двата компютъра едновременно, щеше да има по-голямо съчетание в G .

Ерго, $|M|$ е максималният брой единици време, в които двата компютъра може да работят едновременно. В графа да има перфектно съчетание е същото като да можем да изпълним работите за оптималното време $\lceil \frac{n}{2} \rceil$.

Изработването на конкретното планиране—за всеки компютър кои работи и в какъв ред да се пуснат върху него—е повече от това, да се намери максимално по мощност съчетание. Мощността на съчетанието обаче дава долна граница: няма как двете машини да работят едновременно повече единици време.

В екстремния вариант, в който ограниченията задават линейна наредба, да кажем j_1 преди j_2 , j_2 преди j_3 , $\dots$, j_{n-1} преди j_n , графът няма ребра, понеже всеки две работи са сравними. Тогава максималното по мощност съчетание е празното множество и не можем да ползваме двете машини едновременно изобщо, така че оптималното време за изпълнение е n .

От особен интерес са съчетанията в двуделни графи, в които двата дяла отговарят на два различни вида обекти. Тогава ребрата може да се интерпретират като афинитет между обекти от различни видове. Класическата илюстрация на СЪЧЕТАНИЕ ВЪРХУ ДВУДЕЛНИ ГРАФИ е: единият дял са жени, другият дял са мъже, а ребрата са двойките, които се харесват взаимно и са склонни да се оженият. Иска се да оженим колкото може повече хора.

СЪЧЕТАНИЕ е значително по-лесна върху двуделни графи, отколкото върху графите по принцип. Двуделността помага за ефикасността на алгоритъма. Вижте Глава 26.3 **Maximum bipartite matching** учебника на курса [34, стр. 732–726]. Предложено е просто и елегантно решение чрез намиране на максимален поток в граф, като сложността е $O(nm)$. Най-бързият известен алгоритъм за СЪЧЕТАНИЕ ВЪРХУ ДВУДЕЛНИ ГРАФИ е на Hopcroft и Karp [77] със сложност $O(n^2\sqrt{n})$.

Перфектно съчетание в граф е съчетание, в което всеки връх е съчетан. Необходимо, но не достатъчно условие да има перфектно съчетание е броят на върховете да е четен. ПЕРФЕКТНО СЪЧЕТАНИЕ е задача за разпознаване или задача за търсене: по даден граф, да се намери дали има перфектно съчетание или да се намери едно перфектно съчетание.

Перфектно съчетание в двуделни графи, при интерпретацията с жени и мъже, които искат да се женят, е начин да бъдат оженени всички. Очевидно е, че трябва дяловете да са с еднаква мощност – жените да са колкото мъжете. Дяловете да са с еднаква мощност обаче не е достатъчно, за да има перфектно съчетание. Необходимо и достатъчно условие за съществуване на перфектно съчетание в двуделен граф следва веднага от Теорема 132, често наричана “the marriage theorem”.

Допълнение 84: Теорема на Kőnig и Hall

Да видим два класически резултата за съчетанията върху двуделни графи. Нека в това допълнение $G = (V, E)$ е двуделен граф с дялове X и Y .

Оригиналната статия, съдържаща Теорема 130, е на унгарски. Статията е преведена на английски и е свободно достъпна [148].

Теорема 130: Теорема на Kőnig, 1931 г.

Във всеки двуделен граф, размерът на максималното съчетание е равен на раз-
мера на минималното върхово покриване.

С други думи, $\nu(G) = \tau(G)$.

Доказателство: Известни са много доказателства. Примерно, в книгата на Diestel “Graph Theory” [38, стр. 36–38] се излага подробно доказателство чрез така наречените *augmenting paths*: много полезно понятие с огромно приложение в алгоритмите върху графи. Тук ще видим друго, по-кратко доказателство, което е от статия от 2000 г. [132].

Това, че $\nu(G) \leq \tau(G)$, е очевидно – за всяко върхово покриване U и за всяко съчетание M е вярно, че върховете от U покриват и ребрата от M . Но ребрата в M нямат общи краища, така че един връх от U може да покрие най-много едно ребро от M . Следователно, $|M| \leq |U|$.

Нека G е минимален контрапример за теоремата. Тогава $\nu(G) \neq \tau(G)$. Щом $\nu(G) \leq \tau(G)$, следва, че

$$\nu(G) < \tau(G) \quad (16.5)$$

G да е минимален контрапример означава, че при изтриването на връх от G се получава граф, който не е контрапример, и при изтриване на ребро от G също се получава граф, който не е контрапример.

G е свързан, иначе някоя от свързаните му компоненти би била по-малък контрапример. Нещо повече, G има връх u от степен ≥ 3 . В противен случай G би бил път или четен цикъл^a, а нито път, нито четен цикъл може да е контрапример.

Нека w е произволен съсед на u . Да допуснем, че $\nu(G - w) < \nu(G)$. Щом G е минимален контрапример за теоремата, $G - w$ не е контрапример, така че $\nu(G - w) = \tau(G - w)$. Нека U' е оптимално върхово покриване на $G - w$; това означава, че $|U'| = \tau(G - w)$. Ерго, $|U'| = \nu(G - w)$ и в текущите допускания, $|U'| < \nu(G)$. Оттук следва, че

$$|U' \cup \{w\}| \leq \nu(G) \quad (16.6)$$

От друга страна, $U' \cup \{w\}$ покрива върхово G . Тогава $\tau(G) \leq |U' \cup \{w\}|$. Предвид (16.5), заключаваме, че

$$\nu(G) < |U' \cup \{w\}| \quad (16.7)$$

Но (16.6) и (16.7) са в директно противоречие. ⚡

И така, $\nu(G - w) = \nu(G)$. Тогава има максимално съчетание M в G , което не съчетава w . Забележете, че M трябва да съчетава u , иначе можем да добавим реброто (u, w) към M , получавайки по-голямо съчетание. Ерго, има съсед x на u , такъв че $(u, x) \in M$ и $x \neq w$.

Разглеждаме $G - M$. Тъй като $d_G(u) \geq 3$, в G връх u има поне още един съсед y освен w и x . Но y е съсед на u и в $G - M$, понеже $(u, y) \notin M$. Тогава в $G - M$ съществува поне едно ребро e , инцидентно с u , но не инцидентно с w , а именно $e = (u, y)$.

Разглеждаме $G - e$. Този граф също не е контрапример, така че $\nu(G - e) = \tau(G - e)$. Нека U'' е върхово покриване на $G - e$, такова че $|U''| = |M|$. Защо съществува такова върхово покриване на $G - e$? Защото $G - e$ не е контрапример за теоремата, така че $\nu(G - e) = \tau(G - e)$. Очевидно $\nu(G - e) \leq \nu(G)$, така че $\tau(G - e) \leq \nu(G)$. Ерго, $G - e$ може да бъде покрит от най-много $\nu(G)$ върха, така че може да бъде покрит и от точно $\nu(G)$ върха; тоест, от $|M|$ върха, понеже M е максимално съчетание в G .

Твърдим, че $w \notin U''$. Но това следва веднага от факта, че $|U''| = |M|$: щом U'' покрива върхово ребрата на $G - e$, трябва U'' в частност да покрива ребрата от M ; припомняме си, че $M \subseteq E(G - e)$. Тъй като мощностите на U'' и M са равни, неизбежно върховете на U'' са краища на ребрата от M , като точно единият край на всяко ребро от M е елемент на U'' и в U'' няма други върхове. Щом w не е край на ребро от M , няма как w да е в U'' .

Това, че $w \notin U''$, е ключово. Припомняме си, че реброто (u, w) е ребро в $G - e$. Щом U'' покрива върхово $G - e$ и $w \notin U''$, трябва $u \in U''$, иначе реброто (u, w) би било непокрито. Щом $u \in U''$ и U'' покрива върхово $G - e$, вярно е, че U'' покрива върхово G . Но тогава G има върхово покриване с $\nu(G)$ върха, в противоречие с (16.5). $\square$

Щом в двуделните графи минималното върхово покриване е равно на максималното съчетание и можем да намираме ефикасно максимално съчетание в двуделен граф, можем да намираме ефикасно и размера на минималното върхово покриване на двуделен граф.

Теорема 131: VERTEX COVER е податлива върху двуделните графи

BIPARTITE VERTEX COVER $\in P$.

Преминаваме към разглеждане на Теоремата на Hall, като започваме с пример. В днешно време примерите с женитби са щекотливи и доста учебни помагала ги избягват, но оригиналните примери за приложенията на тази теория са най-често именно такива. Теорема 132 неслучайно е известна като “the marriage theorem”. И така, има n жени, n мъже и са дадени двойките на взаимно харесване. Искане се необходимо и достатъчно условие, така че да има начин всички тези хора да бъдат оженени (в рамките на това множество). За да се оженят двама души, трябва да се харесват взаимно. Да разгледаме една редица от възможности, всяка от които води до това, че **няма** решение. Тук разсъждаваме от гледна точка на жените, но лесно се вижда, че същите съображения са в сила и за мъжете.

- Има жена, която не харесва никого от тези мъже.
- Има две жени, които харесват един и същи мъж и никой друг.
- Има три жени, които харесват едни и същи двама мъже и никой друг.
- Има четири жени, които харесват едни и същи трима мъже и никой друг.
- И така нататък.

- Жените харесват $n - 1$ от мъжете и само тях. С други думи, има мъж, който не се харесва на нито една жена.

Тези възможности може да се изразят по-общо с неравенства и да се резюмират така. За всяко подмножество S от жени, нека $N(S)$ ⁶ е обединението от множествата от мъжете, които жените от S харесват. Задачата няма решение, ако $|N(S)| < |S|$ за поне едно подмножество S . Това е доста очевидно. Интересното е, че конверсното също е вярно: ако за **всяко** подмножество S е вярно, че $|N(S)| \geq |S|$, задачата има поне едно решение. Условието “ $\forall S : |N(S)| \geq |S|$ ” е известно като “the marriage condition”. Това, че the marriage condition влече наличие на решение, изобщо не е очевидно и е същността на Теорема 132, доказана от Philip Hall през 1935 г. [66].

Да бъдат оженени всички е същото като в двуделния граф на взаимните харесвания да се намери перфектно съчетание. За да има перфектно съчетание, двата дяла трябва да са равномошни. Теорема 132 е по-обща и не говори за перфектно съчетание, а за максимално възможно съчетание – такова, в което всички върхове на единия дял са съчетани. В частност, ако дяловете са равномошни, това е перфектно съчетание.

Теорема 132: Теорема на Hall, 1931 г.

В G има съчетание, което съчетава всички върхове на X тстк $\forall S \subseteq X : |N(S)| \geq |S|$.

Доказателство: Diestel [38, стр. 38–39] предлага три различни доказателства на Теоремата на Hall. Тук ще видим друго доказателство, ползващо Теоремата на König.

В едната посока: ако $\exists S \subseteq X : |N(S)| < |S|$, то няма съчетание, съчетаващо всички върхове на X . Това е очевидно.

В другата посока, нека няма съчетание, съчетаващо всички върхове на X . Ще покажем, че $\exists S \subseteq X : |N(S)| < |S|$. БОО, G е свързан. Ако G не е свързан, теоремата може да разглежда поотделно свързаните му компоненти, всяка от които е свързан двуделен граф.

Нека M е максимално по мощност съчетание в G . По допускане, $|M| < |X|$. Нека $U \subseteq V$ е минимално върхово покриване на G . Съгласно Теорема 130, $|M| = |U|$. Тогава $|U| < |X|$. Нека

$$U \cap X = X^+$$

$$U \cap U = Y^+$$

$$X \setminus X^+ = X^-$$

$$Y \setminus Y^+ = Y^-$$

Очевидно

$$|X| = |X^+| + |X^-|$$

$$|U| = |X^+| + |Y^+|$$

Тогава

$$|X| - |U| = |X^-| - |Y^+|$$

Щом $|U| < |X|$, то $|X| - |U| > 0$, така че $|X^-| - |Y^+| > 0$, тоест

$$|X^-| > |Y^+| \quad (16.8)$$

Твърдим, че $N(X^-) \subseteq Y^+$. Наистина, графът е свързан, така че изолирани върхове не може да има, ерго, всеки връх от X^- има поне едно инцидентно ребро. Другите краища на ребрата, инцидентни с върховете от X^- , са от Y , понеже графът е двуделен с дялове X и Y . Ключовото наблюдение е, че тези други краища трябва да са от Y^+ . Това е така, защото U е върхово покриване на графа и в частност всяко ребро, инцидентно с връх от X^- , трябва да бъде покрито. Тъй като $X^- \cap U = \emptyset$, това може да стане само от “другата страна”; по-точно, другият край на такова ребро трябва да е от $Y^+ = Y \cap U$. Но $N(X^-)$ е множеството от другите краища на ребрата, инцидентни с връх от X^- . Показахме, че $N(X^-) \subseteq Y^+$. Тогава

$$|Y^+| \geq |N(X^-)| \quad (16.9)$$

От (16.8) и (16.9) следва, че

$$|X^-| > |N(X^-)|$$

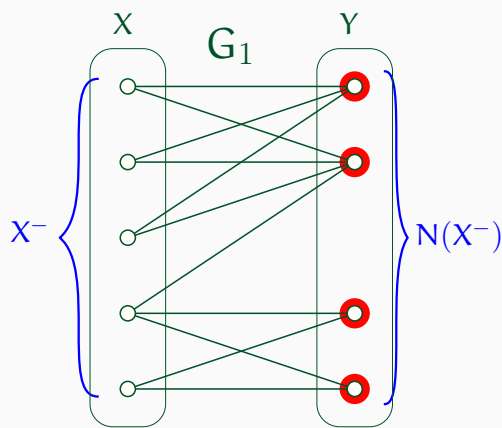
И това е краят на доказателството – търсеното множество S е X^- . □

Фигура 16.18 илюстрира доказателството на теоремата на Hall върху два двуделни графа G_1 и G_2 . И на двата дялове са наречени X и Y .

В G_1 , $|X| > |Y|$. Забележете, че няма съчетание, което съчетава всеки връх на X . Можем да сме сигурни в това и без да сме видели графа – фактът, че $|X| > |Y|$ и няма изолирани върхове влече невъзможността всеки връх на X да бъде съчетан. Но да видим как са нещата в светлината на доказателството на теоремата на Hall. Очевидно всяко максимално съчетание в G_1 е с мощност 4 и съгласно теоремата на Кőnig минималното върхово покриване е с мощност 4. Вляво на Фигура 16.18 е показано едно минимално върхово покриване U , като върховете му са в червено. То съвпада с Y . Тогава X^+ е празното множество и X^- съвпада с X . Тогава $N(X^-)$ е $N(X)$, което е Y . Виждаме, че в този пример несъществуването на съчетание, съчетаващо целия X , влече $|X^-| > |N(X^-)|$. Има и други върхови покривания на G_1 с мощност 4, но лесно може да се убедим, че за всяко от тях е изпълнено $|X^-| > |N(X^-)|$.

В G_2 , $|X| = |Y|$. Тогава всяко съчетание, съчетаващо всеки връх на X , би било перфектно съчетание. Такова не съществува в G_2 , но за да се убедим в това, трябва да разгледаме G_2 . Разглеждайки G_2 , виждаме, че перфектно съчетание няма, а максимално съчетание е с мощност 4. Съгласно теоремата на Кőnig, минималното върхово покриване на G_2 е с мощност 4. Вдясно на Фигура 16.18 е единственото минимално върхово покриване U , като върховете му са в червено. Както виждаме, отново несъществуването на съчетание, съчетаващо целия X , влече $|X^-| > |N(X^-)|$.

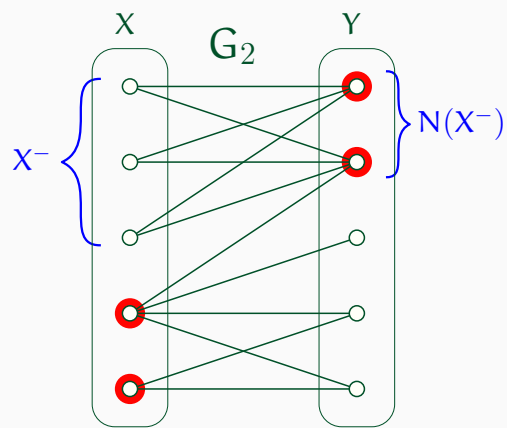
Фигура 16.18 : Теоремата на Hall следва от теоремата на Kőnig.



$\nu(G_1) = 4$ и оттам $\tau(G_1) = 4$ по теор. на Kőnig.

$\mathcal{U}$ е минимално върхово покриване на G_1 .

$X^+ = \emptyset, X^- = X, N(X^-) = Y^+ = Y = \mathcal{U}$.



$\nu(G_2) = 4$ и оттам $\tau(G_2) = 4$ по теор. на Kőnig.

$\mathcal{U}$ е минимално върхово покриване на G_2 .

$X^+ \neq \emptyset, Y^+ \neq \emptyset, X^- \subset X, N(X^-) = Y^+$.

^aЧетен, защото G е двуделен.

^bВижте Нотация 5.

16.2.7.2 Хиперграфите като обобщение на обикновените графи

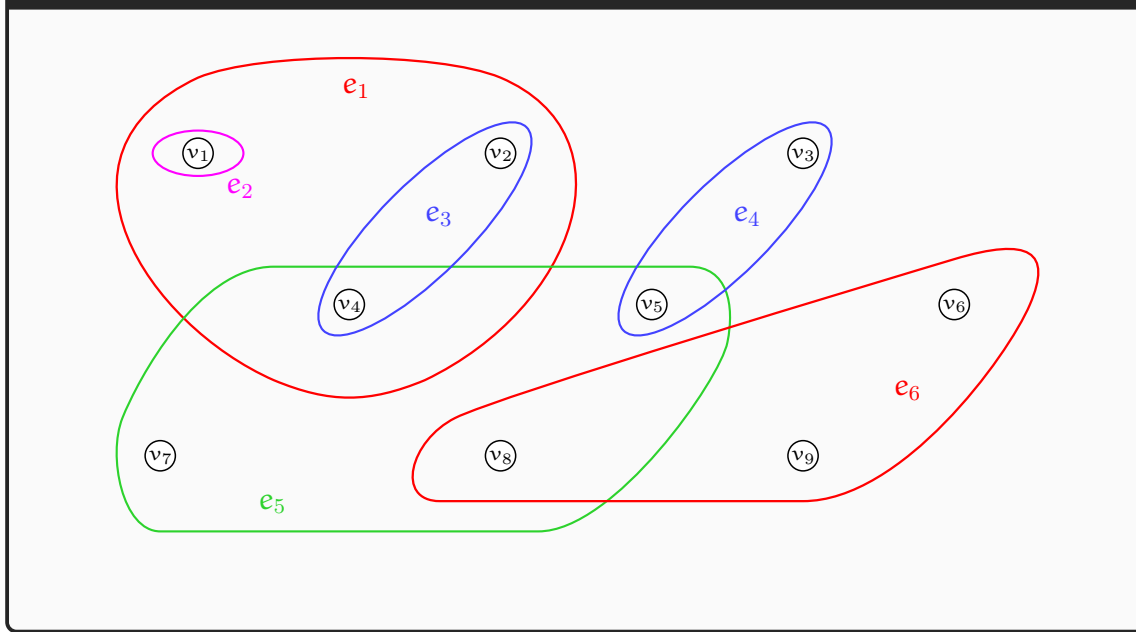
Хиперграф, на английски *hypergraph*, е естествено обобщение на “граф”. Нека $G = (V, E)$ е граф. Можем да мислим за V като за опорно множество, а за E като за фамилия над V . И то доста ограничен вид фамилия: всяко ребро е двуелементно множество от върхове. Мислейки за E като за фамилия над V , може ли да си представим ребра, които са с по-голяма мощност? Може, разбира се. Дали това е смислено и полезно на практика? В някои случаи, да.

За целите на тази лекция[†] дефинираме “хиперграф” като наредена двойка $H = (V, E)$ от множество от върхове V и фамилия E над V , като $\emptyset \notin E$. С други думи, $E \subseteq 2^V \setminus \{\emptyset\}$. Елементите на E се наричат *хиперребра*. *k-униформен хиперграф* е хиперграф, в който всички ребра имат мощност k . Очевидно обикновените графи без примки са 2-униформни хиперграфи.

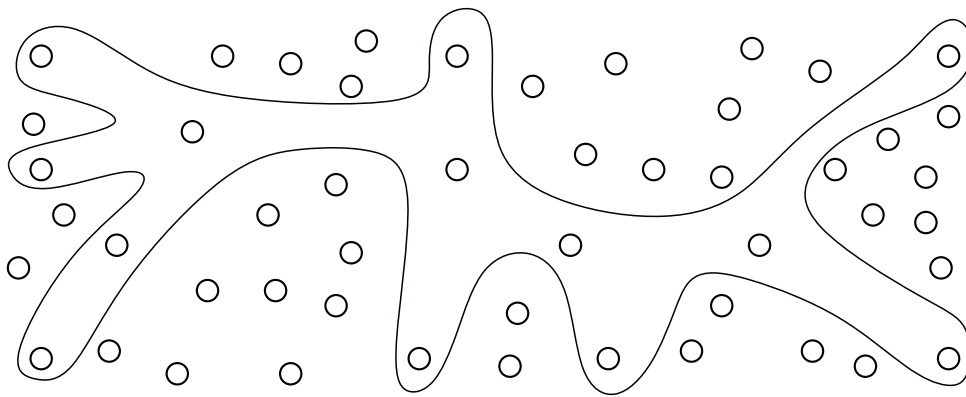
Фигура 16.19 показва пример за хиперграф. Типично, хиперребрата се рисуват с прости затворени криви—окръжности или овали, ако е възможно—като образа на хиперребро на рисунката огражда образите на неговите върхове в хиперграфа.

[†]За задълбочено изложение върху хиперграфите вижте класическите книги на Berge “Graphs and Hypergraphs” [19] и “Hypergraphs: Combinatorics of Finite Sets” [18], съвременната “Hypergraph Theory: An Introduction” на Bretto [28] и обзорната статия “Hypergraphs: an introduction and review” на Ouvrard [124]. За приложения на хиперграфите вижте статията “Learning with Hypergraphs: Clustering, Classification, and Embedding” [159].

Фигура 16.19 : Пример за хиперграф.



Хиперграфите се рисуват трудно. Докато при обикновените графи можем да изобразим граф с десетки върхове и много десетки ребра по такъв начин, че човек лесно да го осмисли, то хиперграф може да бъде възприет визуално само ако е много малък, както е примерът на Фигура 16.19. Ако хиперребрата на този хиперграф бяха много десетки, би било практически невъзможно да се нарисува ясно дори с умело ползване на цветовете. Читателят лесно може да си представи ситуация, в която хиперребра се рисуват с вдлъбнати “амеби”, за да не обхващат образите на върхове, които не влизат в тях:



Очевидно е, че рисунка с десетки такива “амеби” не би могла да бъде възприета визуално и поради това би била безполезна.

Ето пример, в който хиперграф възниква естествено. Да си представим множество от автори и множество от статии, написани от тези автори. Да кажем, че множеството от авторите е $\{v_1, \dots, v_9\}$, а множеството от статиите е $\{e_1, \dots, e_6\}$. Нека

- v_1, v_2 и v_4 са авторите на e_1 ,
- v_1 е авторът на e_2 ,
- v_2 и v_4 са авторите на e_3 ,

- v_3 и v_5 са авторите на e_4 ,
- v_4, v_5, v_7 и v_8 са авторите на e_5 , и
- v_6, v_8 и v_9 са авторите на e_6 .

Но Фигура 16.19 показва точно това!

Хиперграфите може да бъдат представени недвусмислено чрез обикновени графи, но начинът да стане това не е непременно очевидният. Да вземем за пример хиперграфа от Фигура 16.19. Човек може да се изкуши да го представи като граф със същото множество от върхове, в който клики отговарят на хиперребрата. Тази идея, разбира се, не работи. Както се вижда от Фигура 16.19, наличието на хиперребра, съдържащи други хиперребра (примерно, e_1 съдържа e_3), обезсмисля опитите за представяне на ребрата чрез клики, защото 3-кликата на e_1 би съдържала същински 2-кликата на e_3 , при което, при кликовото представяне, e_3 би изчезнало.

До същия извод можем да стигнем и с чисто комбинаторни съображения: върху дадено множество върхове, хиперграфите са много повече от обикновените графи.

Допълнение 85: За броя на графите и хиперграфите

Нека е фиксирано множество върхове V , като $|V| = n$. Както знаем от курса Дискретни структури, броят на всички възможни обикновени графи над V е

$$2^{\frac{n(n-1)}{2}}$$

Тук става дума за именувани графи.

Точната горна граница за броя на хиперграфите над V е:

$$2^{2^n - 1}$$

Изваждането -1 в степенния показател е заради това, че не може да има празно хиперребро, така че всички възможни хиперребра са $2^n - 1$ на брой, и всяко от тях или присъства, или не присъства.

Дори да се ограничим само до 3-униформните хиперграфи, броят е

$$2^{\binom{n}{3}} = 2^{\frac{n(n-1)(n-2)}{6}}$$

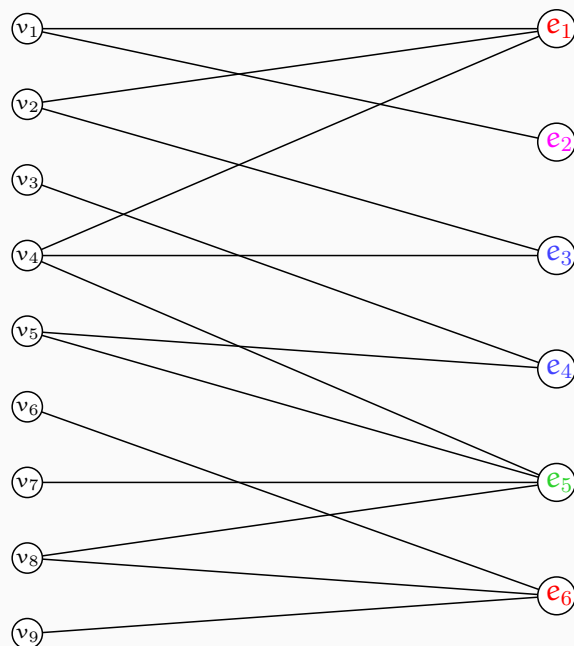
тъй като различните хиперребра са $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. Очевидно е, че

$$\lim_{n \rightarrow \infty} \frac{2^{\frac{n(n-1)(n-2)}{6}}}{2^{\frac{n(n-1)}{2}}} = \infty$$

И така, броят на обикновените графи е изчезващо малка функция пред броя на 3-униформните хиперграфи.

Може да представим хиперграф $H = (V, E)$ чрез обикновен двуделен граф с дялове V и E , като за всяко $x \in V$ и всяко $y \in E$, ребро (x, y) се слага тстк x е елемент на y (в H). Фигура 16.20 показва представяне на хиперграфа от Фигура 16.19 чрез двуделен граф.

Фигура 16.20 : Двуделният граф, описващ хиперграфа от Фигура 16.19.



Ясно е, че размерът на дяла E може да е експоненциален във $|V|$.

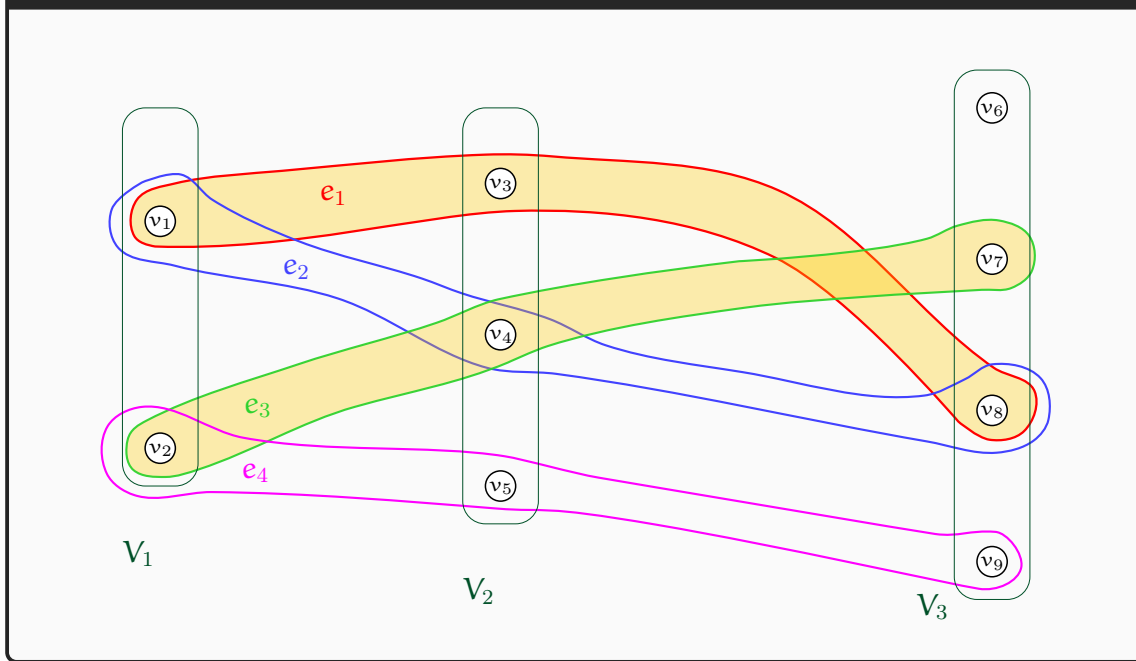
Ако видим само съответния двуделен граф, без да сме видели оригиналния хиперграф H , можем ли да разберем кой от дяловете (на двуделния) отговаря на върховете на H и кой, на хиперребрата на H ? Отговорът е категорично **не**. Примерно, G на Фигура 16.20 би могъл да представлява хиперграф с върхове $e_1, \dots, e_6$ и ребра $v_1, \dots, v_9$. Не е забранено върховете да се именуват с “ e ”, а ребрата, с “ v ”!

16.2.7.3 3DM – обобщение на ПЕРФЕКТНО СЪЧЕТАНИЕ за 3-регулярни триделни хиперграфи с равномощни дялове

Навсякъде в тази подподсекция, “хиперграф” означава “3-униформен хиперграф”. Нека $H = (V, E)$ е хиперграф. Казваме, че H е *триделен*, ако съществува разбиване $\{V_1, V_2, V_3\}$ на V , такова че за всяко хиперребро, единият му връх е от V_1 , другият е от V_2 , а третият е от V_3 . Понятието се явява очевидно обобщение на “двуделен граф” от света на обикновените графи. Казваме, че V_1 , V_2 и V_3 са *дяловете*.

Съчетание в H е всяко $M \subseteq E$, такова че нито две хиперребра от M нямат общ връх. Това е естественото обобщение на понятието при обикновените графи. Фигура 16.21 илюстрира съчетание в триделен хиперграф. Дяловете са V_1 , V_2 и V_3 , като $|V_1| = 2$, $|V_2| = 3$ и $|V_3| = 4$. Хиперребрата са e_1 , e_2 , e_3 и e_4 . Показаното съчетание $\{e_1, e_3\}$ е максимално.

Фигура 16.21 : Максимално съчетание в триделен хиперграф.



Нека $H = (V, E)$ е триделен хиперграф, чиито три дяла са с една и съща мощност k . Дали в H съществува съчетание с мощност k ? Такова съчетание би било обобщение на перфектно съчетание в двуделните графи. Но сложността на задачата върху двуделни графи и триделни хиперграфи е драстично различна. При обикновените графи е известно просто необходимо и достатъчно условие за съществуване на перфектно съчетание (Теорема 132) и задачата за съществуване на перфектно съчетание е решима ефикасно чрез алгоритъм за потоци в графи ([34, стр. 732–726]) или алгоритъма на Норсфот и Карп ([77]). От друга страна, задачата е NP-пълна върху триделните 3-униформни хиперграфи.

Както предстои да видим, формалната дефиниция на “3DM” не ползва “хиперграф”. Понятието “хиперграф” е полезно само за да видим 3DM като обобщение на добре известна, податлива графова задача.

16.2.7.4 3DM е неподатлива

Изч. Задача 85: 3-DIMENSIONAL MATCHING (3DM)

екземпляр: Равномощни множества B, C, D с празни сечения по двойки. Множество

$$A \subseteq \{\{a, b, c\} \mid a \in A, b \in B, c \in C\}$$

въпрос: Дали съществува $A' \subseteq A$, такова че $|A'| = n$ и

$$\forall X, Y \in A' : X \neq Y \rightarrow X \cap Y = \emptyset?$$

С други думи, A' представлява разбиване на $B \cup C \cup D$, такова че всеки негов елемент-множество съдържа по точно един елемент от B , от C и от D . И, разбира се, налице е ограничението A' да е подмножество на даденото A , иначе задачата би била тривиална. Такова A' ще наричаме “перфектно съчетание”, въпреки че не говорим за хиперграфи.

Редукция 22: $3SAT \leq_p 3DM$

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

Ще конструираме съответен екземпляр $\langle B, C, D, A \rangle$ на 3DM, където B, C и D са равномошни множества с празни сечения по двойки и $A \subseteq \{\{b, c, d\} \mid b \in B, c \in C, d \in D\}$, по такъв начин, че A има подмножество-перфектно съчетание тстк ϕ е удовлетворима. Нека $\text{Var}(\phi) = \{x_1, \dots, x_n\}$.

Първо конструираме множествата B, C и D :

$$B = \{u_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{v_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

$$C = \{y_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{s_{1,j} \mid 1 \leq j \leq m\} \cup \{g_{1,j} \mid 1 \leq j \leq m(n-1)\}$$

$$D = \{z_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{s_{2,j} \mid 1 \leq j \leq m\} \cup \{g_{2,j} \mid 1 \leq j \leq m(n-1)\}$$

Две по две не имат празни сечения. Очевидно, $|B| = |C| = |D| = 2nm$, така че $|B \cup C \cup D| = 6nm$. Множеството A е

$$A = \left(\bigcup_{i=1}^n A_{1,i} \right) \cup \left(\bigcup_{j=1}^m A_{2,j} \right) \cup \left(\bigcup_{k=1}^{m(n-1)} A_{3,k} \right)$$

където

$$A_{1,i} = \{\{u_{i,j}, y_{i,j}, z_{i,j}\} \mid 1 \leq j \leq m\} \cup \{\{v_{i,j}, y_{i,j+1}, z_{i,j}\} \mid 1 \leq j < m\} \cup \{\{v_{i,m}, y_{i,1}, z_{i,m}\}\}$$

$$A_{2,j} = \{\{u_{i,j}, s_{1,j}, s_{2,j}\} \mid \bar{x}_i \text{ е литерал във } \phi_j\} \cup \{\{v_{i,j}, s_{1,j}, s_{2,j}\} \mid x_i \text{ е литерал във } \phi_j\}$$

$$A_{3,k} = \{\{u_{i,j}, g_{1,k}, g_{2,k}\} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{\{v_{i,j}, g_{1,k}, g_{2,k}\} \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

Това е конструкцията, описана формално и прецизно. Лесно се вижда, че

$$|A_{1,i}| = 2m \quad \text{за } 1 \leq i \leq n$$

$$|A_{2,j}| = 3 \quad \text{за } 1 \leq j \leq m$$

$$|A_{3,k}| = 2mn \quad \text{за } 1 \leq k \leq m(n-1)$$

откъдето

$$\left| \bigcup_{i=1}^n A_{1,i} \right| = 2mn$$

$$\left| \bigcup_{j=1}^m A_{2,j} \right| = 3m$$

$$\left| \bigcup_{k=1}^{m(n-1)} A_{3,k} \right| = 2m^2(n-1)$$

Ясно, е, че примерът на 3DM може да се генерира в полиномиално време. Не е очевидно,

че конструкцията е коректна – че в изградения пример има перфектно съчетание тстк ϕ е удовлетворима. Поради сложността на конструкцията ще разгледаме пример и ще направим аргументация на неговата коректност, надявайки се това да е убедителен аргумент и за коректността на конструкцията по принцип. Екземплярът, който ще разгледаме, е

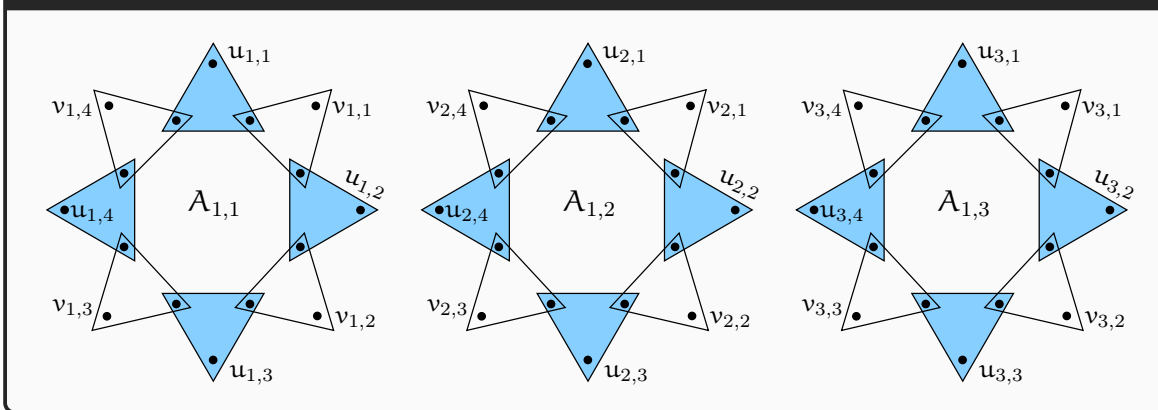
$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_2} \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{\phi_3} \underbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}_{\phi_4} \quad (16.10)$$

Този екземпляр на 3SAT вече ползвахме, например (16.3). ϕ е удовлетворима, например от валюацията $\langle 1, 1, 0 \rangle$. Ще видим, че в построеното A има подмножество-перфектно съчетание.

Приспособление звезда. Множествата $A_{1,i}$, за $1 \leq i \leq n$, са първият вид приспособления, които тази конструкция ползва. Ще ги наричаме *звездите*. Звездите отговарят на променливите. Същностното е, че всяка звезда се състои от тройки, на брой два пъти колкото са клаузите на ϕ , имащи общи краища в кръгова наредба. Всяка звезда има “върхове” и “основа”. А именно, звезда $A_{1,i}$ има върхове $u_{i,j}$ и $v_{i,j}$ и основа, състояща се от $y_{i,j}$ и $z_{i,j}$. Върховете участват и в други приспособления, докато елементите от основата – не; основата служи само реализиране на кръговата наредба на тройките.

В примера, който разглеждаме (16.10) има три променливи, така че конструкцията генерира три звезди, които са изобразени на Фигура 16.22. Елементите от основите не са именувани. Именуванни са върховете $u_{i,j}$ и $v_{i,j}$.

Фигура 16.22 : 3SAT $\leq_p$ 3DM върху екземпляра от (16.10): звездите.



Множеството от тройките на звезда $A_{1,i}$ се разбива на *u-тройките* и *v-тройките*: първите съдържат някой $u_{i,j}$, а вторите – някой $v_{i,j}$. На Фигура 16.22, *u-тройките* са нарисувани със син фон, а *v-тройките* са с бял фон.

Наблюдение 110

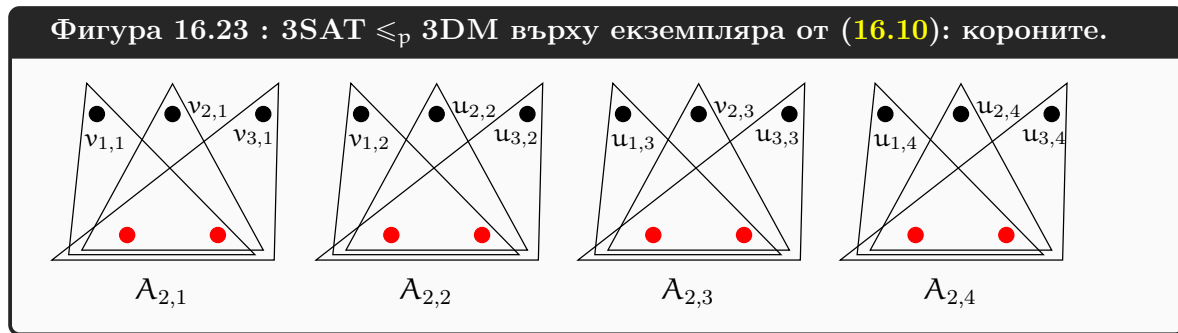
В текущия контекст, $A' \subseteq A$ е перфектно съчетание само ако за всяка звезда, A' съдържа или всички нейни *u-тройки* и нито една нейна *v-тройка*, или всички нейни *v-тройки* и нито една нейна *u-тройка*. Причината е, че елементите от основата на звездата не участват в други тройки, освен в тройките на звездата. Тогава тези елементи може да участват в (тройки на) перфектно съчетание само ако то съдържа точно половината от тройките на звездата, през една в кръговата наредба.

Читателят сигурно се досеща, че щом звездите отговарят на променливите, то дали за дадена звезда вземаме точно u -тройките или точно v -тройките в перфектно съчетание моделира дали тази променлива е истина или лъжа в съответната удовлетворяваща валуация на екземпляра на 3SAT. Дали u -тройките или v -тройките отговарят на 1, е друг въпрос. Важното е, че вземайки или точно u -тройките, или точно v -тройките, ние постигаме консистентност, гарантирайки, че стойността на променливата е една и съща във всички клаузи, в които се среща.

Приспособление корона. Множествата $A_{2,j}$, за $1 \leq j \leq m$, са вторият вид приспособления. Ще ги наричаме *короните*. Короните отговарят на клаузите. Всяка корона $A_{2,j}$ се състои от три тройки, които

- имат обща двойка $\{s_{1,j}, s_{2,j}\}$ от елементи, които принадлежат само на $A_{2,j}$,
- и три “върха” от вида $u_{i,j}$ или $v_{i,j}$, които са общи с върхове на някои звезди.

В примера, който разглеждаме (16.10) има четири клаузи, така че конструкцията генерира четири корони, които са изобразени на Фигура 16.23. Елементите от основите не са именувани. Именувани са върховете $u_{i,j}$ и $v_{i,j}$.



Докато звездите са конструирани без оглед на конкретиката на екземпляра ϕ , спрямо който работим, короните отразяват неговата конкретика. Примерно,

- корона $A_{2,1}$ има връх $v_{1,1}$, а не $u_{1,1}$, защото в клаузата ϕ_1 , променливата x_1 участва с положителния си литерал.
- корона $A_{2,3}$ има връх $u_{1,3}$, а не $v_{1,3}$, защото в клаузата ϕ_3 , променливата x_1 участва с отрицателния си литерал.

Наблюдение 111

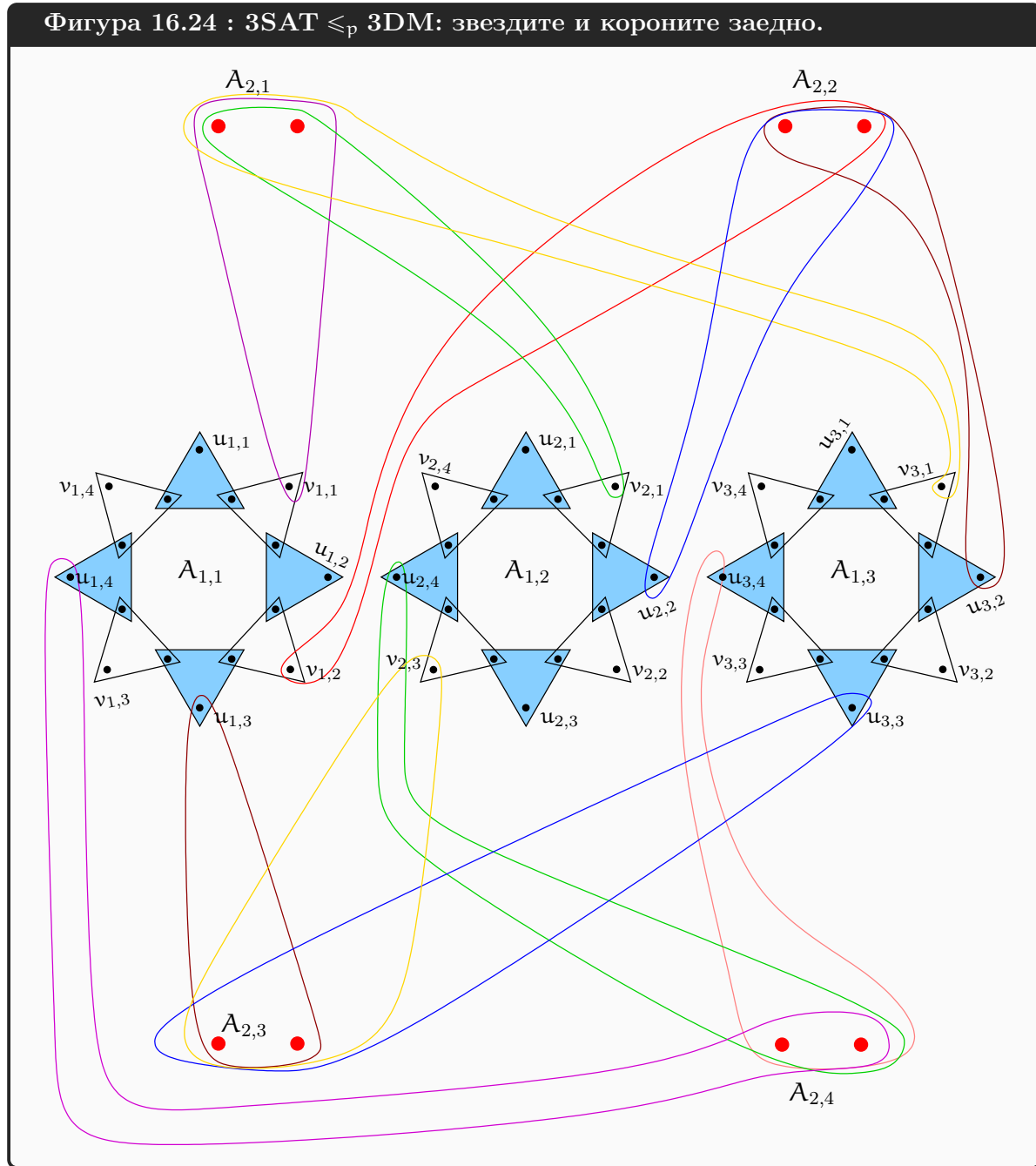
В текущия контекст, $A' \subseteq A$ е перфектно съчетание само ако за всяка корона, A' съдържа точно една нейна тройка. Причината е, че елементите от основата на короната не участват в други тройки, освен в тройките на короната.

Читателят сигурно се досеща, че щом короните отговарят на клаузите и всяка от трите тройки моделира участието на една от трите променливи в клаузата—било с положителен, било с отрицателен литерал—то коя от трите тройки на короната участва в перфектно съчетание моделира през коя променлива клаузата получава удовлетворение.

Забележете, че **точно** една от тройките на короната участва в съчетанието, докато е възможно клауза да бъде удовлетворена от литералите на **няколко** променливи. В това

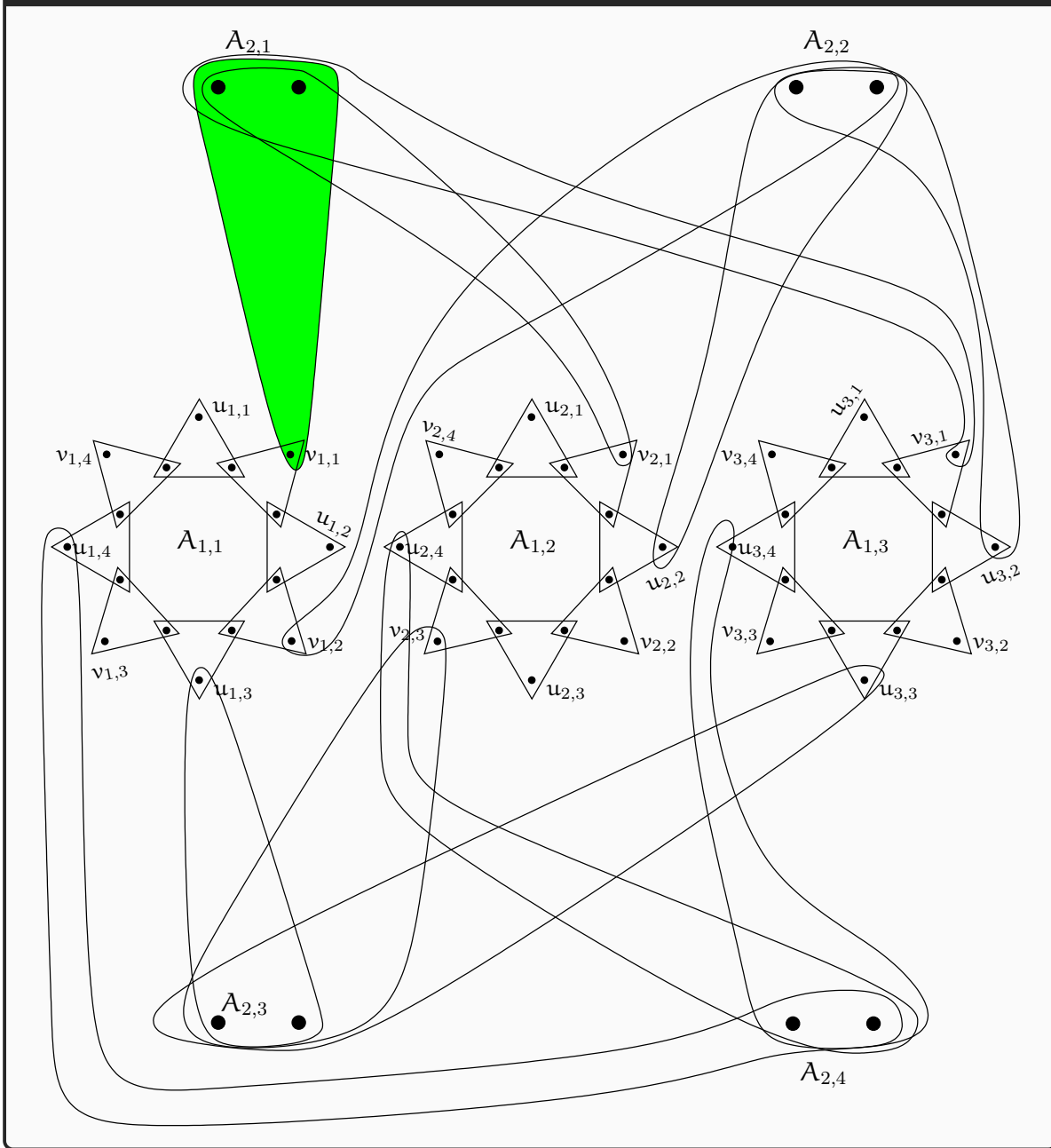
няма противоречие. Същественото е, че—както ще видим—наличието на удовлетворяваща валуация на формулата влече наличие на перфектно съчетание в хиперграфа, и обратно[†].

Да си представим звездите и короните заедно. Фигура 16.24 илюстрира това за примера, който разглеждаме (16.10). Звездите са нарисувани с половината си тройки в синьо, точно както бяха на Фигура 16.22. Тройките на короните са очертани с различни цветове за по-лесно проследяване.

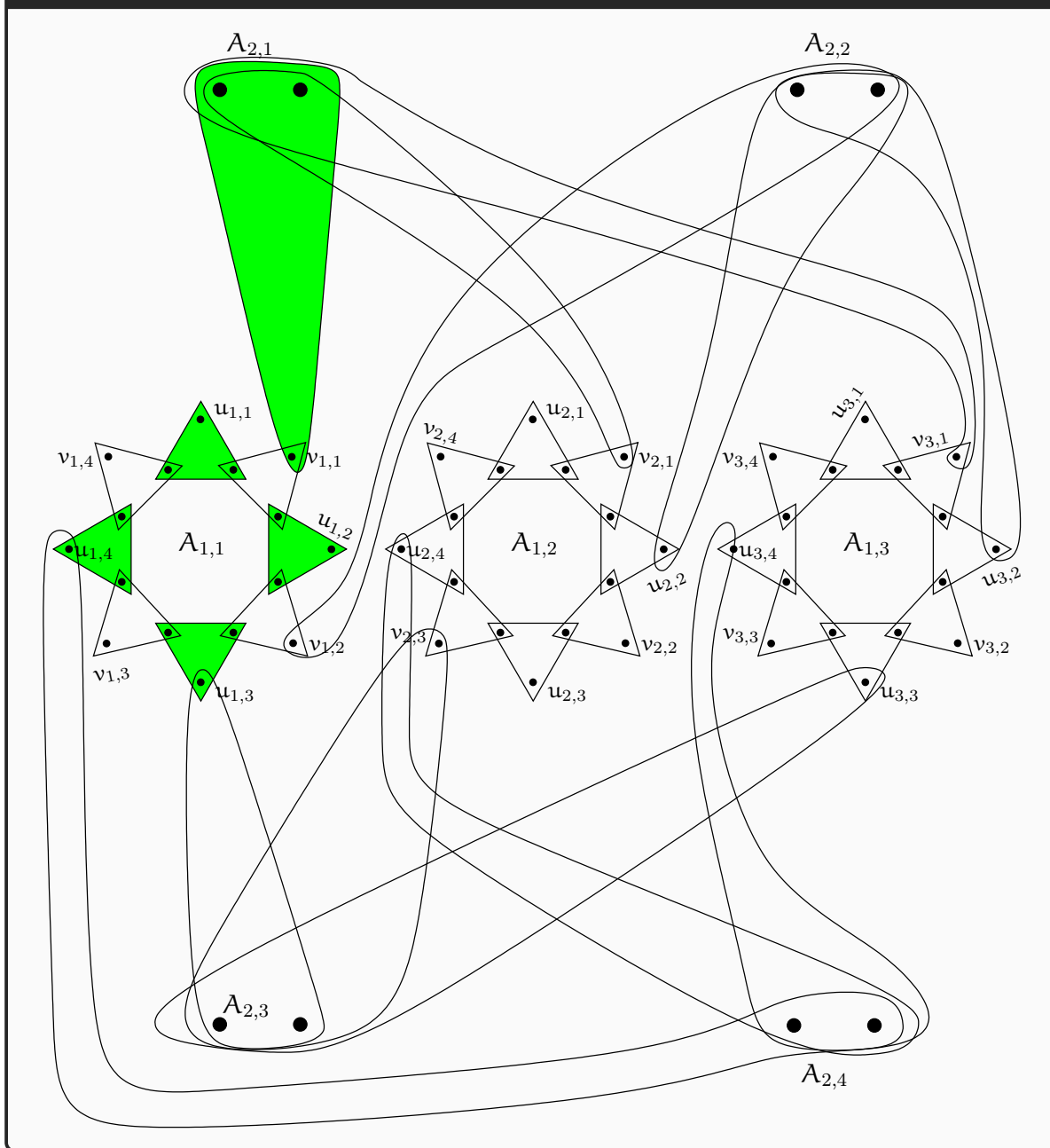


Сега да видим същината на конструкцията – защо това “работи”. За да има перфектно съчетание, точно една тройка на короната $A_{2,1}$ трябва да е в него. Да кажем, тройката с връх $v_{1,1}$. Това е показано на Фигура 16.25.

[†]Ситуацията напомня на привидното противоречие от $3SAT \leq_p \text{VERTEX COVER}$ (Наблюдение 108).

Фигура 16.25 : $3SAT \leq_p 3DM$: слагаме една тройка в съчетанието.

Изборът да сложим тази тройка в съчетанието има последици. Припомняме си Наблюдение 110: или u -тройките, или v -тройките на $A_{1,1}$ са в съчетанието. Тъй като тройката от $A_{2,1}$, която вече сложихме, има общ връх $v_{1,1}$ със звезда $A_{1,1}$, дължни сме да сложим в съчетанието u -тройките на $A_{1,1}$. Това е показано на Фигура 16.26.

Фигура 16.26 : $3SAT \leq_p 3DM$: u -тройките на $A_{1,1}$ са задължителни.

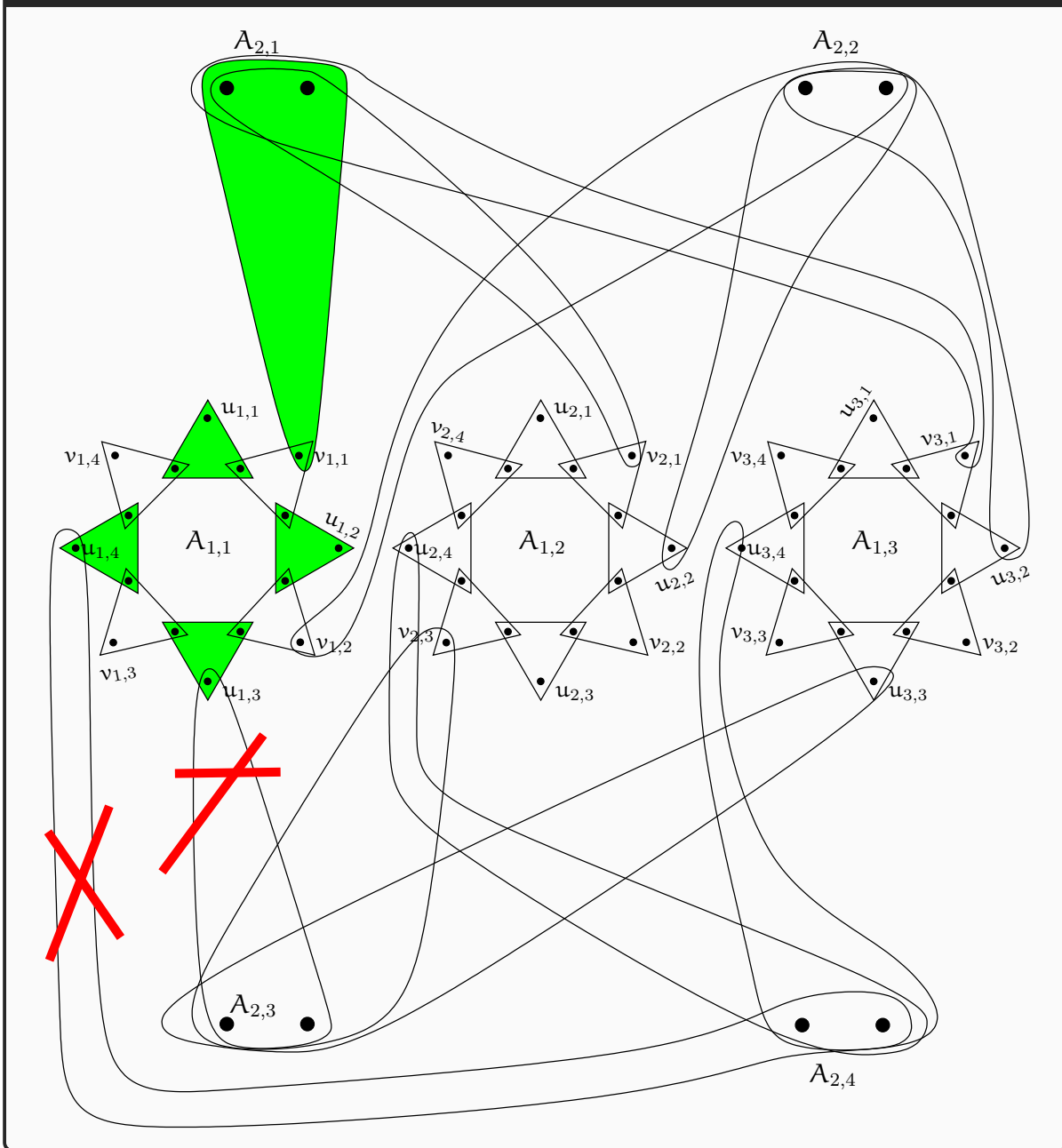
На свой ред, слагането на u -тройките на $A_{1,1}$ в перфектното съчетание води до това, че от всяка от трите засега необработени корони $A_{2,2}$, $A_{2,3}$ и $A_{2,4}$ трябва да изберем за съчетанието тройка, която няма връх $u_{1,j}$. Защо? – защото четирите върха $u_{1,j}$ вече се намират в тройка от дотук изграденото съчетание. Ерго, ако за коя да е от короните $A_{2,2}$, $A_{2,3}$ и $A_{2,4}$ изберем тройката с общ връх със звездата $A_{1,1}$, този общ връх трябва да е някой $v_{1,j}$.

Избирайки за $A_{2,1}$ тройка с връх $v_{1,1}$, ние налагаме избор на връх $v_{1,j}$ за всяка корона, чиято избрана тройка има общ връх със звездата $A_{1,1}$. Преведено на езика на 3SAT, ако изберем стойност за променливата x_1 , такава че ϕ_1 да е удовлетворена, ние можем да удовлетворим и други клаузи чрез променливата x_1 само ако x_1 участва в тези други клаузи със същия литерал, с който участва във ϕ_1 . И по-точно, във ϕ_1 , променливата x_1 е с положителен литерал; избирайки да удовлетворим ϕ_1 чрез x_1 , все едно избираме валуация, в която x_1 е 1, и тази стойност на x_1 бива “наложена” и по отношение на останалите клаузи.

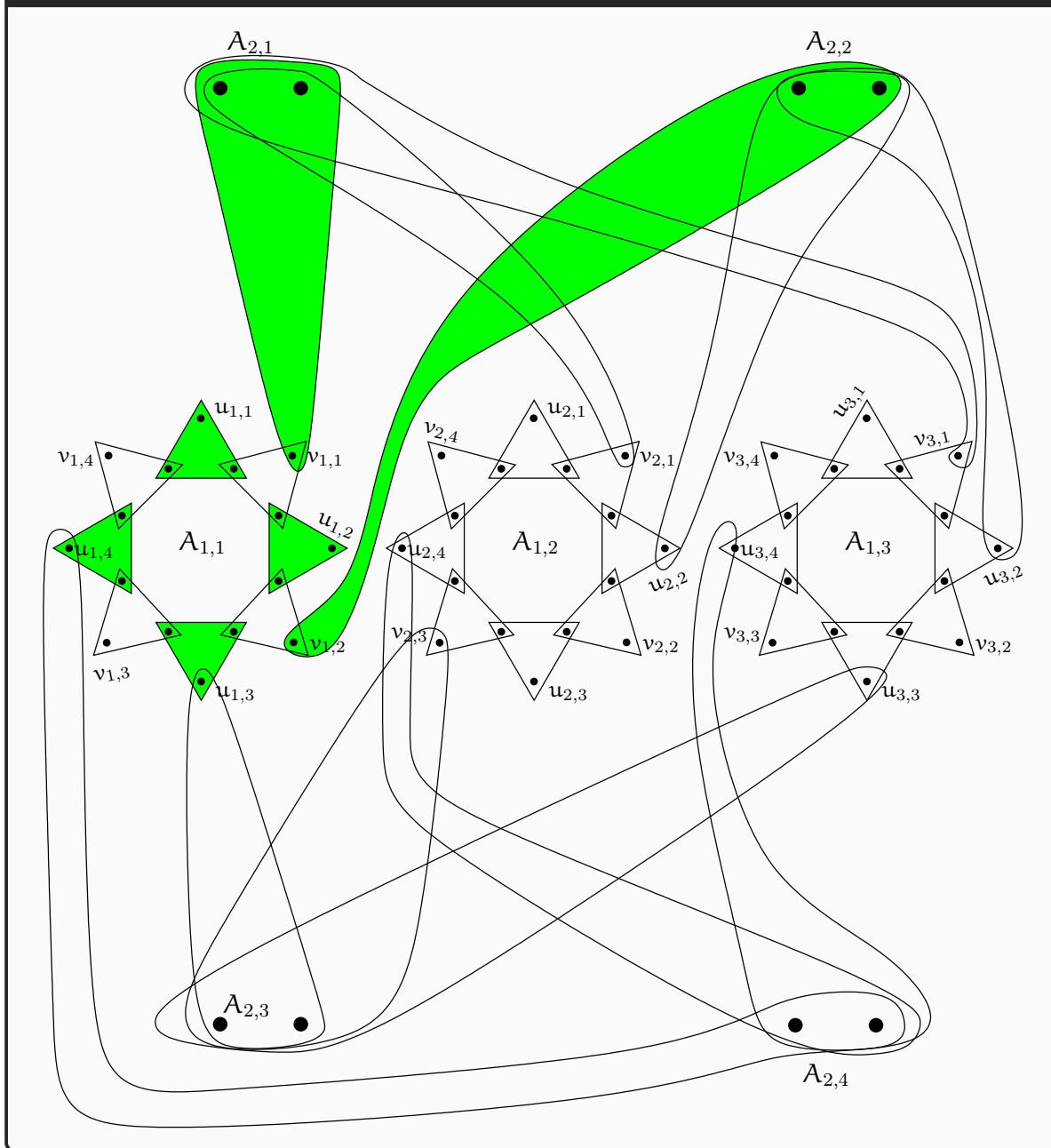
Фигура 16.27 показва как изборът на u -тройките от звездата $A_{1,1}$ прави невъзможно да се

изберат определени тройки от короните $A_{2,3}$ и $A_{2,4}$: а именно тези тройки, които съдържат съответно върховете $u_{1,3}$ или $u_{1,4}$. Ерго, клаузите ϕ_3 и ϕ_4 трябва да бъдат удовлетворени не през литерала на променливата x_1 . Кое е смислено, понеже променливата x_1 участва с отрицателния си литерал във ϕ_3 и ϕ_4 .

Фигура 16.27 : $3SAT \leq_p 3DM$: забранени тройки.

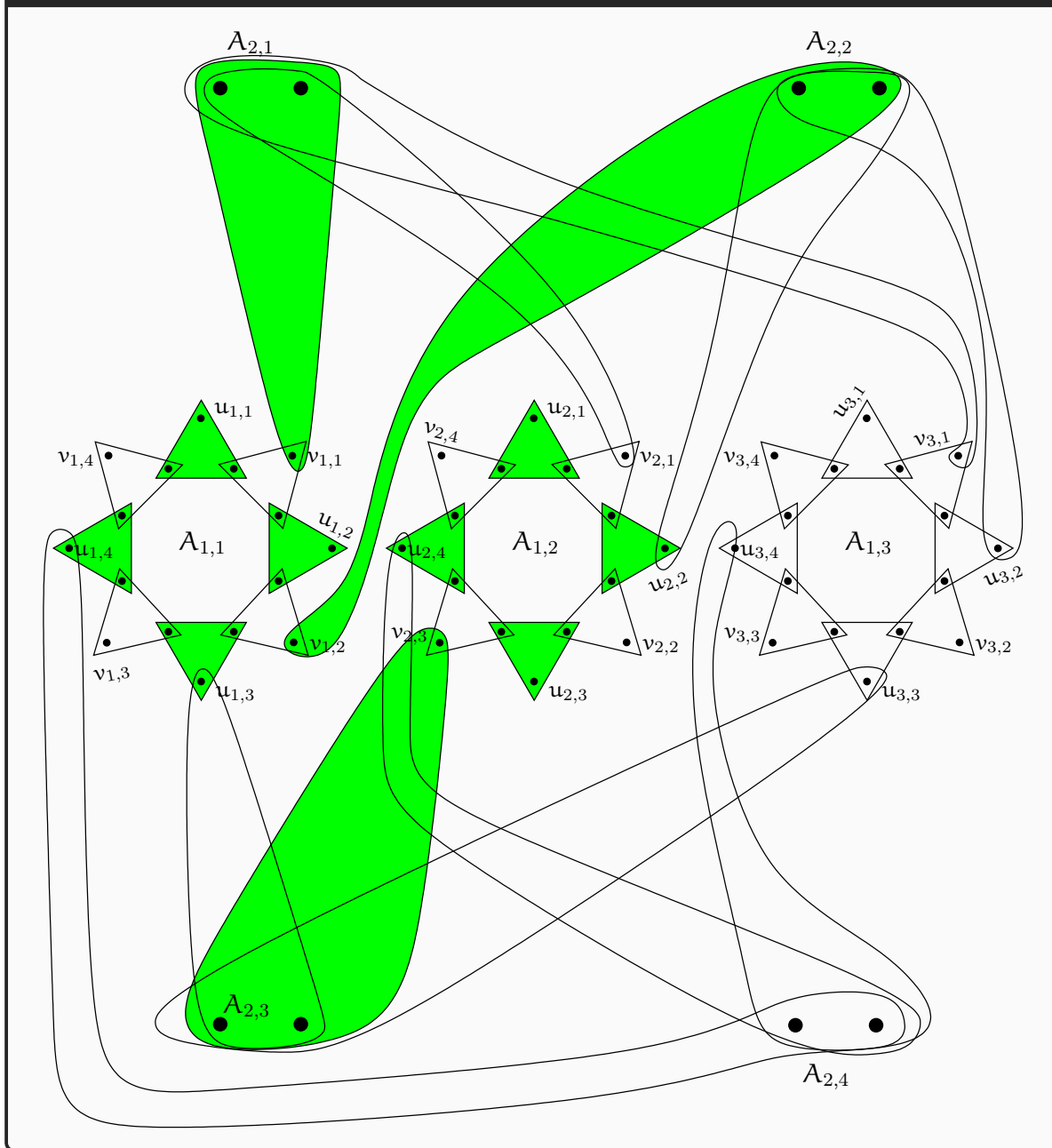


Във ϕ_2 обаче променливата x_1 обаче участва с положителния си литерал. Ерго, ϕ_2 бива удовлетворена от валуация, даваща 1 на x_1 . По отношение на екземпляра на 3DM, това се превежда така: можем да изберем за короната $A_{2,2}$ тази тройка, която съдържа $v_{1,2}$. Фигура 16.28 показва точно това.

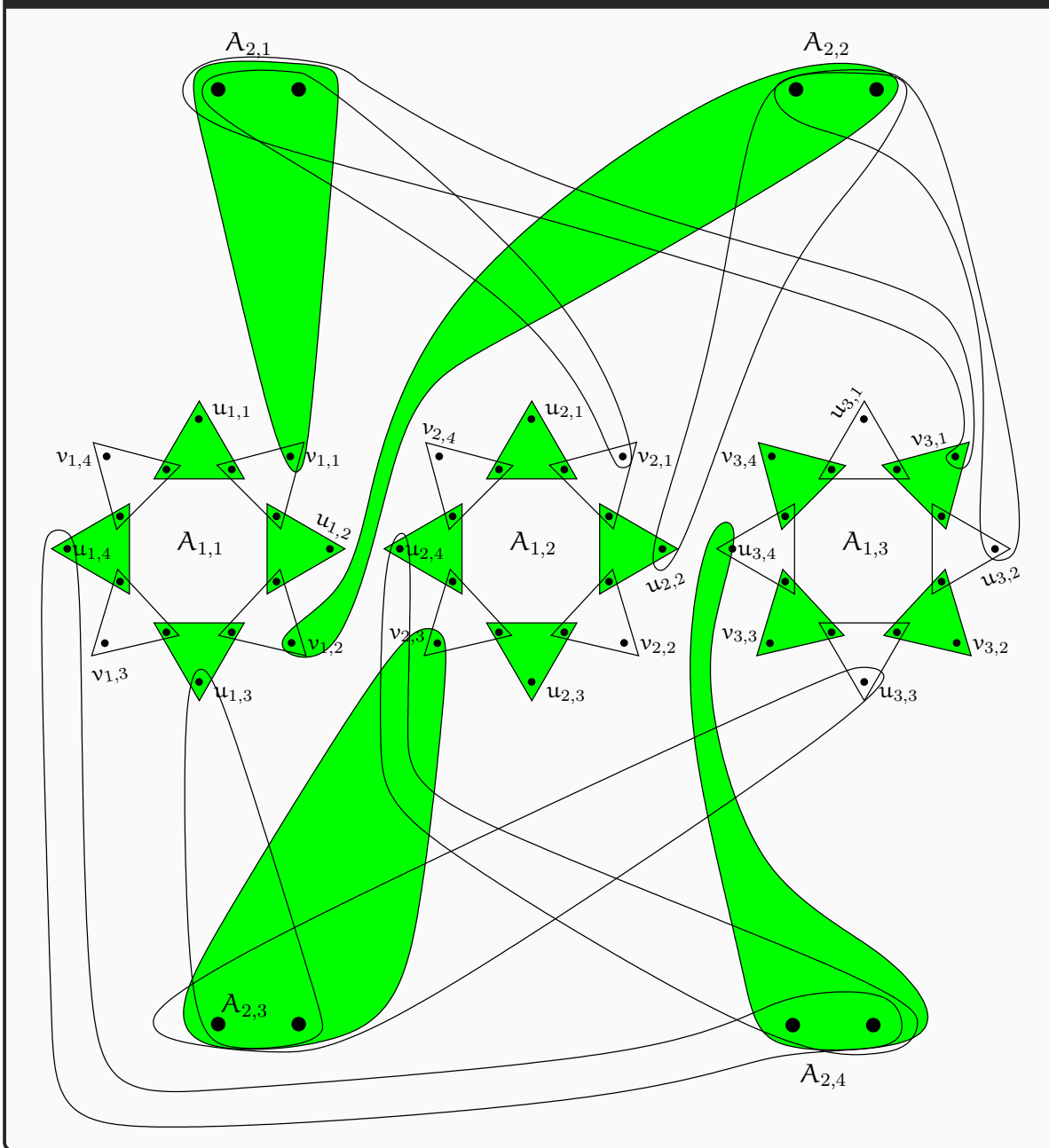
Фигура 16.28 : $3SAT \leq_p 3DM$: възможност за $A_{2,2}$.

Сега да разгледаме короната $A_{2,3}$. Както вече знаем, не можем да изберем нейната тройката, съдържаща $u_{1,3}$. Преведено на езика на 3SAT, клаузата ϕ_3 трябва да си “намери” удовлетворение не през литерала на x_1 , тъй като вече избрахме стойност 1 за x_1 , а във ϕ_3 , променливата x_1 участва с отрицателния си литерал.

Да изберем тройката от $A_{2,3}$, която съдържа $v_{2,3}$. Това е все едно да изберем стойност 1 за променливата x_2 ; досега тя беше свободна. Избирайки въпросната тройка за $A_{2,3}$, форсираме избора на u -тройките от звездата $A_{1,2}$, с което пък забраняваме избора на тройки, съдържащи $u_{2,j}$. Това е илюстрирано на Фигура 16.29.

Фигура 16.29 : $3SAT \leq_p 3DM$: възможност за $A_{2,3}$.

И накрая ще “обработим” короната $A_{2,4}$. Тук нямама избор: трябва да сложим в съчетанието тройката, съдържаща $u_{3,4}$, което форсира v -тройките на звездата $A_{1,3}$ да влязат в съчетанието. Преведено на езика на 3SAT, избираме стойност 0 за променливата x_3 и по този начин удовлетворяваме ϕ_4 , в която има литерал $\bar{x}_3$. Фигура 16.30 показва състоянието на нещата дотук.

Фигура 16.30 : $3SAT \leq_p 3DM$: от всяка корона влиза тройка.

Това обаче не може да е край на конструкцията. Осем елемента от екземпляра на 3DM не са в изграденото съчетание: това са $v_{1,3}$, $v_{1,4}$, $v_{2,1}$, $v_{2,2}$, $v_{2,4}$, $u_{3,1}$, $u_{3,2}$ и $u_{3,3}$. Смисълът от третият вид приспособления е да се “погрижи” за неучастващите до момента елементи.

Приспособление супер корона. Ще наричаме *супер корони* множествата $A_{3,k}$, за $1 \leq k \leq m(n-1)$. Това са третият вид приспособления и те са най-многобройните. Съвкупността от тях е известна като *garbage collector* [56, стр. 52]. Всяка супер корона $A_{3,k}$ прилича на корона с това, че се състои от основа $\{g_{1,k}, g_{2,k}\}$, чиито върхове не участват в нищо друго, и върхове, които са общи с върхове на звездите. Но короната има точно три върха, докато супер короната има $2mn$ върха: всяка нейна тройка съдържа точно един връх на всяка от звездите. Поради това няма и да се опитваме да рисуваме супер короните: рисунката би станала пренаситена и трудна за възприемане. Garbage collector-ът има общо $2m^2(n-1)$ тройки, докато звездите имат само $2mn$ тройки, а короните, само $3m$ тройки.

В примера от (16.10), $m = 4$, $n = 3$, така че garbage collector-ът има $4(3 - 1) = 8$ супер корони, всеки от които е двуелементна основа и $4 \times 3 = 12$ върха. Тези върхове са “слепени” с дванадесетте върха на звездите. За да има перфектно съчетание в целия изграден пример трябва осемте несъчетани върха на Фигура 16.30 да бъдат съчетани. Това става чрез супер короните. От всяка супер корона избираме тройка, която съдържа точно един от тези осем върха. Няма проблеми да го направим, защото всяка супер корона е “слепена” към всеки връх на всяка звезда. $\square$

16.2.8 HAMILTONIAN CYCLE, НАМ. PATH, LONGEST CYCLE, LONGEST PATH

Разглеждаме HAMILTONIAN CYCLE (Задача 2).

Редукция 23: VERTEX COVER $\leq_p$ HAMILTONIAN CYCLE

Конструкция: Нека е даден екземпляр $\langle G = (V, E), k \rangle$ на VERTEX COVER. Ще построим граф $J = (V_J, E_J)$, такъв че J има Хамилтонов цикъл тстк G има върхово покриване с размер $\leq k$. За разлика от редукцията на VERTEX COVER към DOMINATING SET (Редукция 17), в която входният и изходният граф си приличат много (примерно, вижте Фигура 16.9), тук графите G и J привидно нямат нищо общо, като J е много по-голям—но все пак, само полиномиално по-голям—от G .

Първо конструираме множества от върхове V' и V'' по следния начин. За всяко ребро $e \in E$, ако u и v са неговите краища, конструираме множество от дванадесет нови върхове

$$V'_e = \{\langle u, e, 1 \rangle, \dots, \langle u, e, 6 \rangle, \langle v, e, 1 \rangle, \dots, \langle v, e, 6 \rangle\}$$

V' е обединението на всички тези множества.

$$V' = \bigcup_{e \in E} V'_e$$

Конструираме и k нови върхове $\ell_1, \dots, \ell_k$, като

$$V'' = \{\ell_1, \ell_2, \dots, \ell_k\}$$

V_J е обединението на тези две множества

$$V_J = V' \cup V''$$

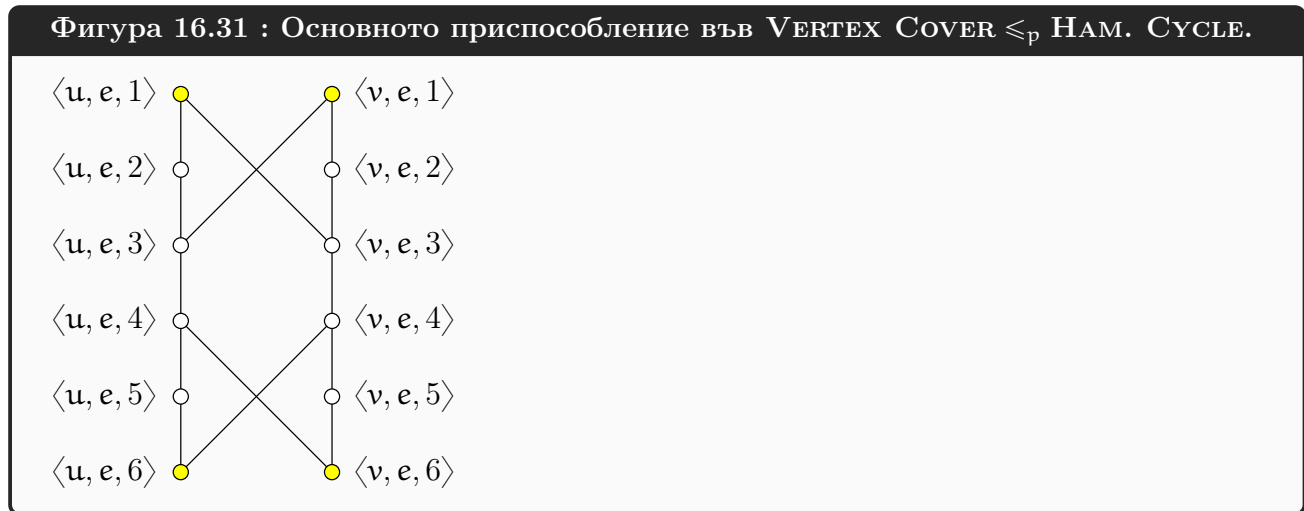
Сега ще конструираме множеството от ребрата на J като обединението на E' , E'' и E''' , които са следните множества. За всяко $e \in E$ изграждаме четиринадесет елементно множество от ребра E'_e , чиито краища са от V'_e :

$$\begin{aligned} E'_e = & \{(\langle u, e, i \rangle, \langle u, e, i+1 \rangle) \mid 1 \leq i \leq 5\} \cup \{(\langle v, e, i \rangle, \langle v, e, i+1 \rangle) \mid 1 \leq i \leq 5\} \cup \\ & \{(\langle u, e, 1 \rangle, \langle v, e, 3 \rangle), (\langle u, e, 3 \rangle, \langle v, e, 1 \rangle)\} \cup \\ & \{(\langle u, e, 4 \rangle, \langle v, e, 6 \rangle), (\langle u, e, 6 \rangle, \langle v, e, 4 \rangle)\} \end{aligned}$$

Множеството E' е обединението от тези множества:

$$E' = \bigcup_{e \in E} E'_e$$

Преди да разгледаме E'' и E''' , да осмислим E' . За всяко $e \in E$, графът (V'_e, E'_e) , чиито върхове и ребра вече дефинирахме, е едно приспособление. Конструкцията прави $|E|$ такива приспособления, по едно за всяко ребро на G . Фигура 16.31 ни дава представа за същината на такова приспособление.



Дефинираме, че четирите върха $\langle u, e, 1 \rangle$, $\langle v, e, 1 \rangle$, $\langle u, e, 6 \rangle$ и $\langle v, e, 6 \rangle$ (в жълто на Фигура 16.31) са *краищата* на това приспособление, като $\langle u, e, 1 \rangle$ и $\langle u, e, 6 \rangle$ са *u-краищата*, а $\langle v, e, 1 \rangle$ и $\langle v, e, 6 \rangle$ са *v-краищата*. В цялостната конструкция краищата са инцидентни с още ребра. Останалите осем върха са вътрешни за приспособлението в смисъл, че не са инцидентни с други ребра освен тези, които вече сложихме. Върховете $\langle u, e, 1 \rangle, \dots, \langle u, e, 6 \rangle$ индуцират подграф-път с дължина 5, който наричаме *u-страната* на приспособлението. Аналогично, върховете $\langle v, e, 1 \rangle, \dots, \langle v, e, 6 \rangle$ индуцират подграф-път с дължина 5, който наричаме *v-страната* на приспособлението.

Наблюдение 112: Точно три начина за Хамилтонов цикъл през джаджа

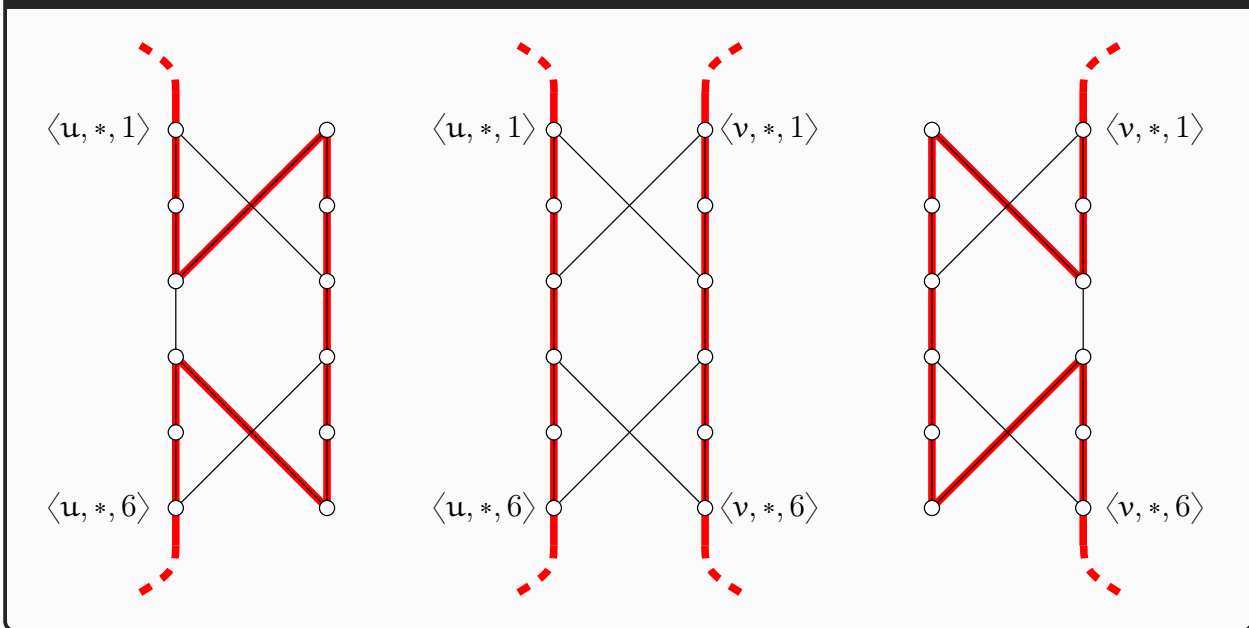
В контекста на цялостната конструкция има точно три начина, по които Хамилтонов цикъл C в J може да мине през дадено приспособление H . Ако краищата на H са $\langle u, *, 1 \rangle$, $\langle v, *, 1 \rangle$, $\langle u, *, 6 \rangle$ и $\langle v, *, 6 \rangle$, въпросните начини са следните.

- C “влиза” през единия u -край и излиза през другия u -край, както е показано вляво на Фигура 16.32. Сечението на C и H е Хамилтонов път в H .
- Огледално, C “влиза” през единия v -край и излиза през другия v -край, както е показано вдясно на Фигура 16.32. Сечението на C и H е Хамилтонов път в H .
- C съдържа пътищата

$$\langle u, *, 1 \rangle, \langle u, *, 2 \rangle, \dots, \langle u, *, 6 \rangle$$

$$\langle v, *, 1 \rangle, \langle v, *, 2 \rangle, \dots, \langle v, *, 6 \rangle$$

което е показано в средата на Фигура 16.32. Сечението на C и H се състои от u -страната и v -страната на H .

Фигура 16.32 : Трите начина за минаване на Хам. цикъл през джаджата.

Сега ще дефинираме E'' . Да разгледаме произволен $z \in V$. Нека ребрата, инцидентни със z в G , са $e_1, \dots, e_t$. Избираме една произволна наредба на тези ребра и я наричаме r_z , да кажем

$$r_z = \langle e_1, \dots, e_t \rangle$$

Подчертаваме, че r_z е произволна. Всяка от $t!$ наредби на тези ребра би свършила работа за конструкцията, но трябва да фиксираме една наредба и да работим спрямо нея. Конструираме множеството от ребра E_z'' в J спрямо r_z :

$$E_z'' = \{ (\langle z, e_i, 6 \rangle, \langle z, e_{i+1}, 1 \rangle) \mid 1 \leq i \leq t-1 \}$$

И правим това за всеки $z \in V$. Тогава

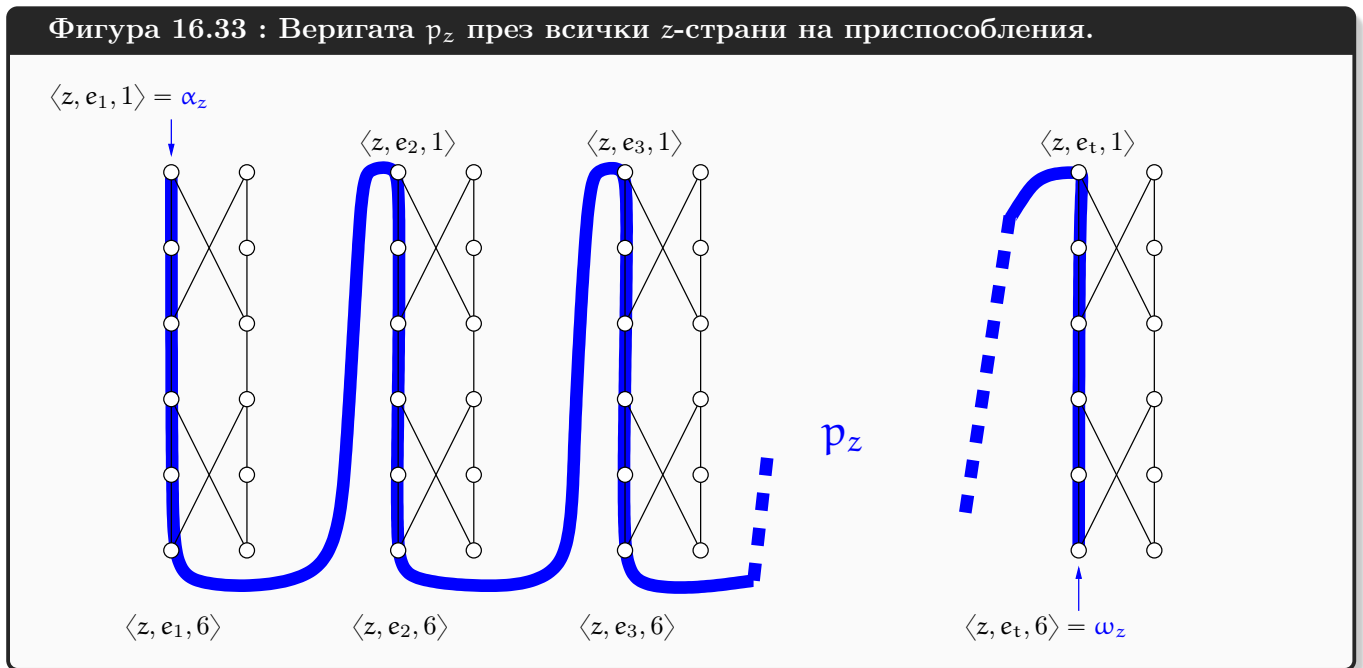
$$E'' = \bigcup_{z \in V} E_z''$$

За всеки $z \in V$, z -веригата в J ще наричаме пътя p_z , в който алтернират z -страна и ребро от E_z'' , и то в реда на r_z . Не особено формално казано, p_z започва със z -страна на първото приспособление, което има z -страна, минава през ребро от E_z'' , после през z -страна на следващото приспособление, което има z -страна, и така нататък, докато може. Дължината на p_z е $5t + (t - 1) = 6t - 1$, защото

- във всяко приспособление от тези, които имат z -страна, върховете от z -страната индуцират път с дължина 5 и има точно t такива приспособления,
- и освен това ребрата от E_z'' , на брой $t - 1$, реализират връзките между z -страните на приспособленията в p_z .

Крайщата на z -веригата са $\langle u, e_1, 1 \rangle$ и $\langle u, e_t, 6 \rangle$. За удобство в обосновката на коректността на конструкцията, даваме следните имена на крайщата на z -веригата: “ α_z ” е алтернативно име за $\langle u, e_1, 1 \rangle$ и “ ω_z ” е алтернативно име за $\langle u, e_t, 6 \rangle$.

Фигура 16.33 илюстрира една z -верига. Показани са само приспособленията, които имат z -страна. z -веригата е синият път между α_z и ω_z .



Ще наричаме *веригите* обединението на z -веригите, по всички $z \in V$.

Колко ребра има в E'' ? Общо в наредбите r_z , по всички $z \in V$, има $2m$ елемента, понеже техният брой е всъщност е сумата от степените на върховете, а знаем, че $\sum_{z \in V} d(z) = 2m$. Тогава

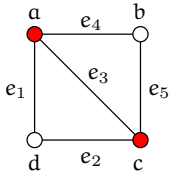
$$E'' = 2m - n$$

понеже за всяко $z \in V$, $|E_z''| = d(z) - 1$.

И накрая, дефинираме E''' така:

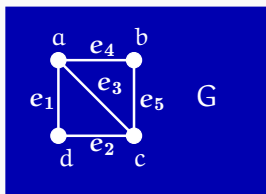
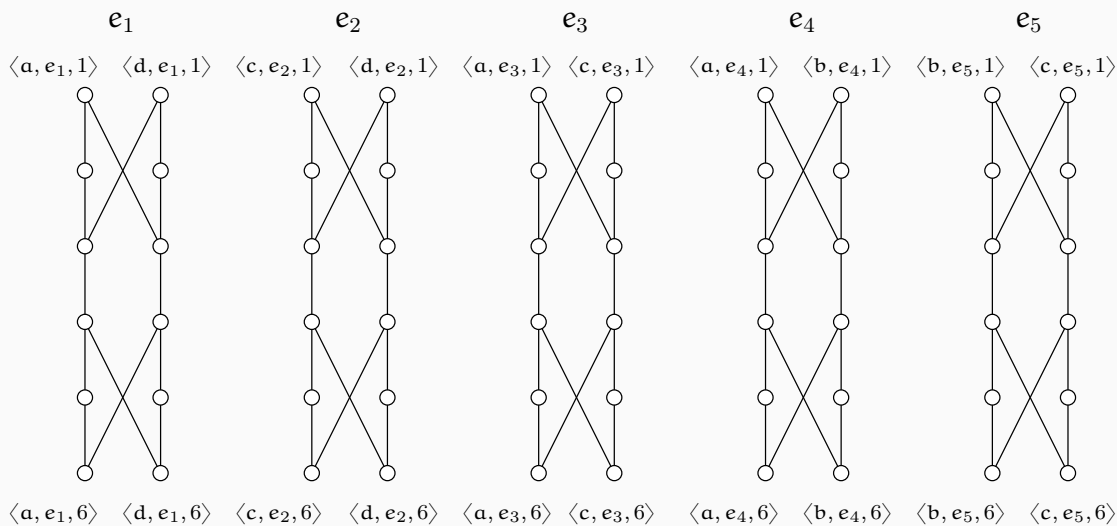
$$E''' = \{(\ell_i, \alpha_z) \mid i \in \{1, \dots, k\}, z \in V\} \cup \{(\ell_i, \omega_z) \mid i \in \{1, \dots, k\}, z \in V\}$$

С това приключи формалното описание на конструкцията на J .



Да видим малък пример за прилагането на конструкцията. Нека $G = (V, E)$ е графът, показан вляво. Очевидно $\langle G, 2 \rangle$ е ДА-екземпляр на VERTEX COVER, сертификат за което са двата върха в червено, които представляват върхово покриване. В крайна сметка, конструкцията генерира графа J , показан на Фигура 16.37. Има смисъл да разгледаме няколко междинни етапа от конструирането на J . Първо конструкцията генерира приспособленията. Те са пет на брой, колкото са ребрата на G , и са показани на Фигура 16.34. Всяко приспособление е именувано с реброто на G , на което съответства, и краищата му също са именувани. Отдолу е показан отново, за всеки случай, и G .

Фигура 16.34 : Приспособленията от примера за VERTEX COVER $\leq_p$ HAM. CYCLE.



Конструкцията добавя множество от ребра E'' , което се генерира спрямо някакви наредби на ребрата, инцидентни с върховете на G , като за всеки връх z наредбата r_z е произволна, но, след като я изберем веднъж, трябва да я спазваме. Нека наредбите са тези:

- $r_a = \langle e_1, e_3, e_4 \rangle$,
- $r_b = \langle e_5, e_4 \rangle$,
- $r_c = \langle e_5, e_3, e_2 \rangle$,
- $r_d = \langle e_1, e_2 \rangle$.

След като сме избрали наредбите, образуваме четирите множества от ребра

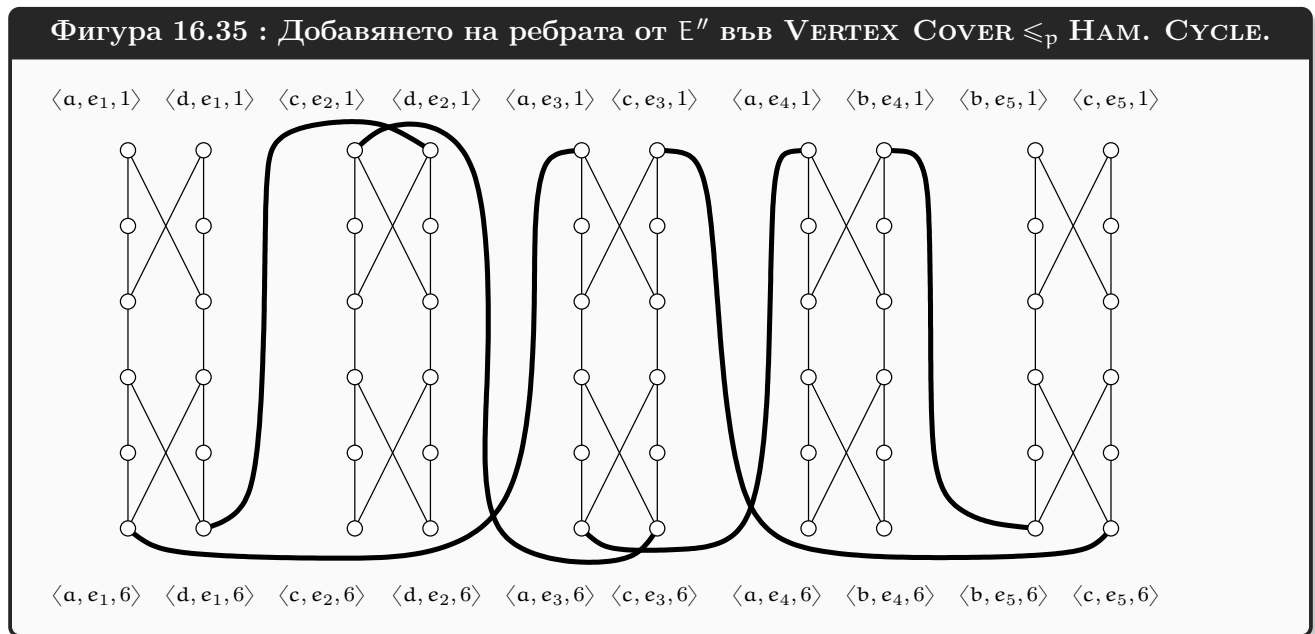
$$E''_a = \{(\langle a, e_1, 6 \rangle, \langle a, e_3, 1 \rangle), (\langle a, e_3, 6 \rangle, \langle a, e_4, 1 \rangle)\}$$

$$E''_b = \{(\langle b, e_4, 1 \rangle, \langle b, e_5, 6 \rangle)\}$$

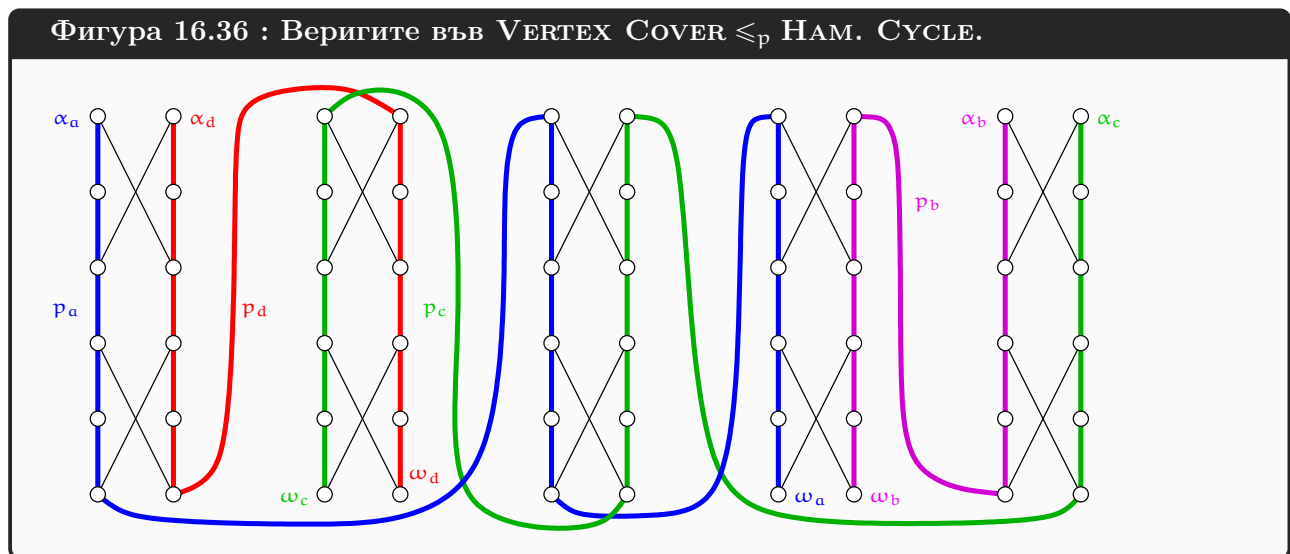
$$E''_c = \{(\langle c, e_2, 1 \rangle, \langle c, e_3, 6 \rangle), (\langle c, e_3, 1 \rangle, \langle c, e_5, 6 \rangle)\}$$

$$E''_d = \{(\langle d, e_1, 6 \rangle, \langle d, e_2, 1 \rangle)\}$$

Тяхното обединение е E'' . Фигура 16.35 показва E'' с дебели криви. Виждаме, че наистина E'' съдържа $2m - n = 10 - 4 = 6$ ребра.

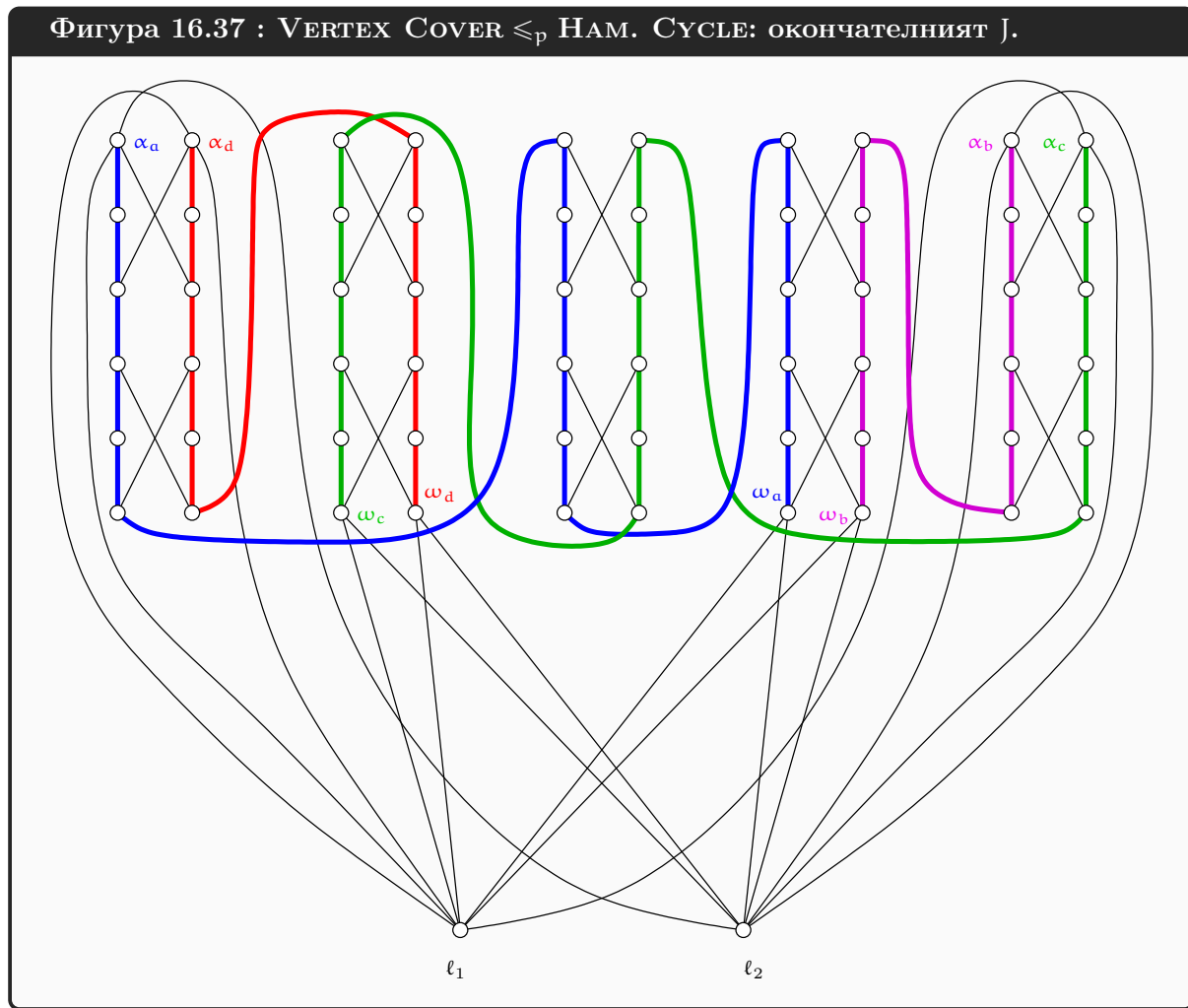


Да видим веригите в построеното досега. Те са четири, колкото са върховете на G . Именуваме ги p_a с краища α_a и ω_a , p_b с краища α_b и ω_b , и така нататък. Вижте Фигура 16.36.



Остава да построим E''' . Тъй като $k = 2$ в примера ни, слагаме два допълнителни върха ℓ_1 и ℓ_2 и свързваме с ребро всеки от тях с всеки край на верига. Веригите са четири и имат общо

осем края, така че E''' съдържа шестнадесет ребра. Вижте Фигура 16.37.



Да разгледаме коректността на редукцията. Започваме с наблюдението, че преди слагането на върховете $\ell_1, \dots, \ell_k$ и ребрата от E''' , Хамилтонов цикъл няма. Няма да правим формално доказателство на това – разгледайте Фигура 16.35 и ще се убедите, че в подграфа на J, състоящ се от приспособленията и ребрата от E'' , няма Хамилтонов цикъл. Имайте предвид Наблюдение 112: за да мине Хамилтонов цикъл през приспособление, той или прави това наведнъж, влизайки-излизайки едната страна, или прави това на два пъти, всеки път минавайки през една от страните.

Ключовият факт е, че върховете $\ell_1, \dots, \ell_k$ плюс ребрата между тях и краищата на веригите са това, което дава възможност да има Хамилтонов цикъл, стига k да е достатъчно голямо. Да видим следните крайни възможности.

- Ако $k = n$, Хамилтонов цикъл в J задължително съществува. При това положение е елементарно да построим Хамилтонов цикъл: той се състои точно от веригите, слепени чрез ребрата от E''' и върховете $\ell_1, \dots, \ell_n$. За примера, който разглеждаме, не е удачно да се рисува J при $k = 4$, защото това би означавало още два върха ℓ_3 и ℓ_4 и още шестнадесет ребра между тях и краищата на веригите и рисунката би станала претрупана. Така че да си представим нещата.
- ♦ Започвайки от ℓ_1 , отиваме в α_a по реброто (ℓ_1, α_a) , вървим по веригата p_a до ω_a и отиваме в ℓ_2 по реброто (ω_a, ℓ_2) ,

- ♦ от ℓ_2 отиваме в α_b по реброто (ℓ_2, α_b) , вървим по веригата p_b до ω_b и отиваме в ℓ_3 по реброто (ω_b, ℓ_3) ,
- ♦ от ℓ_3 отиваме в α_c по реброто (ℓ_3, α_c) , вървим по веригата p_c до ω_c и отиваме в ℓ_4 по реброто (ω_c, ℓ_4) ,
- ♦ от ℓ_4 отиваме в α_d по реброто (ℓ_4, α_d) , вървим по веригата p_d до ω_d и пак стигаме до ℓ_1 .

Построихме Хамилтонов цикъл, редувайки връх ℓ_i “долу” с верига “горе”. Щом всеки ℓ_i е съсед на всеки край на верига, нямаме проблеми да го направим. Хамилтоновият цикъл минава през всяко приспособление по начина, показан в средата на Фигура 16.32.

И така, ако $k = n$, построяваме Хамилтонов цикъл в J без да отчитаме особеностите на G . Това точно съответства на факта, че ако $k = n$, графът G задължително има върхово покриване с $k = n$ върха – такова, в което участва всеки връх. Какъвто и да е G , това е вярно.

- В другата крайност, нека $k = 1$. Сега е доста трудно да има Хамилтонов цикъл в J . Не е невъзможно, но понеже има само един ℓ_1 връх “долу”, Хамилтоновият цикъл би трябвало да тръгва от него (от ℓ_1), да се “качва” по едно от двете ребра на E'' до началото α_z на някоя верига p_z “горе”, да върви по нея, минавайки през всички приспособления, и от другия ѝ край ω_z да се връща в ℓ_1 . Ерго, трябва да има верига, минаваща през всички приспособления.

Но в J да има верига, минаваща през всички приспособления, е същото като в G да има връх, инцидентен с всички ребра. Нещо повече: ако има такава верига, не може да има друга верига, минаваща през повече от едно приспособление. Защо е така? Забележете, че верига, която минава през поне две приспособления, минава през u -страната и на двете, за някакъв връх u . Ако е вярно, че една верига минава през z -страните на всички приспособления и друга верига минава през u -страните на поне две приспособления, тогава тези две приспособления отговарят на ребра на G , всяко от които е инцидентно и с u , и със z . Но тогава G е мултиграф с паралелни ребра, което не е разрешено по условие. За илюстрация вижте пак Фигура 16.36: наистина има вериги, които минават през едно и също приспособление, но през само едно общо приспособление; повече от едно общо приспособление за две вериги би имало, ако G беше мултиграф.

Забележете, че верига, която е в рамките на само едно приспособление, отговаря на висящ връх в G . И така, ако една верига p_z минава през всички приспособления и по-точно през z страните им, G е **граф-звезда**, в който връх z е съсед на всички останали върхове, които са висящи. Такъв граф наистина има върхово покриване с размер единица: а именно връх z .

И така, ако $k = 1$, Хамилтонов цикъл в J има само в частния случай, в който G е граф-звезда. Това точно съответства на факта, че ако $k = 1$, графът G задължително има върхово покриване с 1 връх.

Фактът, че във всяка от тези крайности, наличието на Хамилтонов цикъл в J е същото като наличие на върхово покриване със съответния размер в G , би трябвало да ни дадат интуиция за коректността на редукцията. Сега да я докажем формално.

Теорема 133: Коректността на $\text{VERTEX COVER} \leq_p \text{HAMILTONIAN CYCLE}$

В текущия контекст, J съдържа Хамилтонов цикъл тстк G има върхово покриване с не повече от k върха.

Доказателство, 1: Да допуснем, че G има върхово покриване с размер точно k . Тук няма загуба на общност, защото, ако има върхово покриване с размер $\leq k$, то има и върхово покриване с размер точно k . Нека покриващото множество за G е $U = \{u_1, u_2, \dots, u_k\}$. Ще построим Хамилтонов цикъл C в J .

Разглеждаме множеството от вериги

$$P_U = \{p_{u_i} \mid 1 \leq i \leq k\}$$

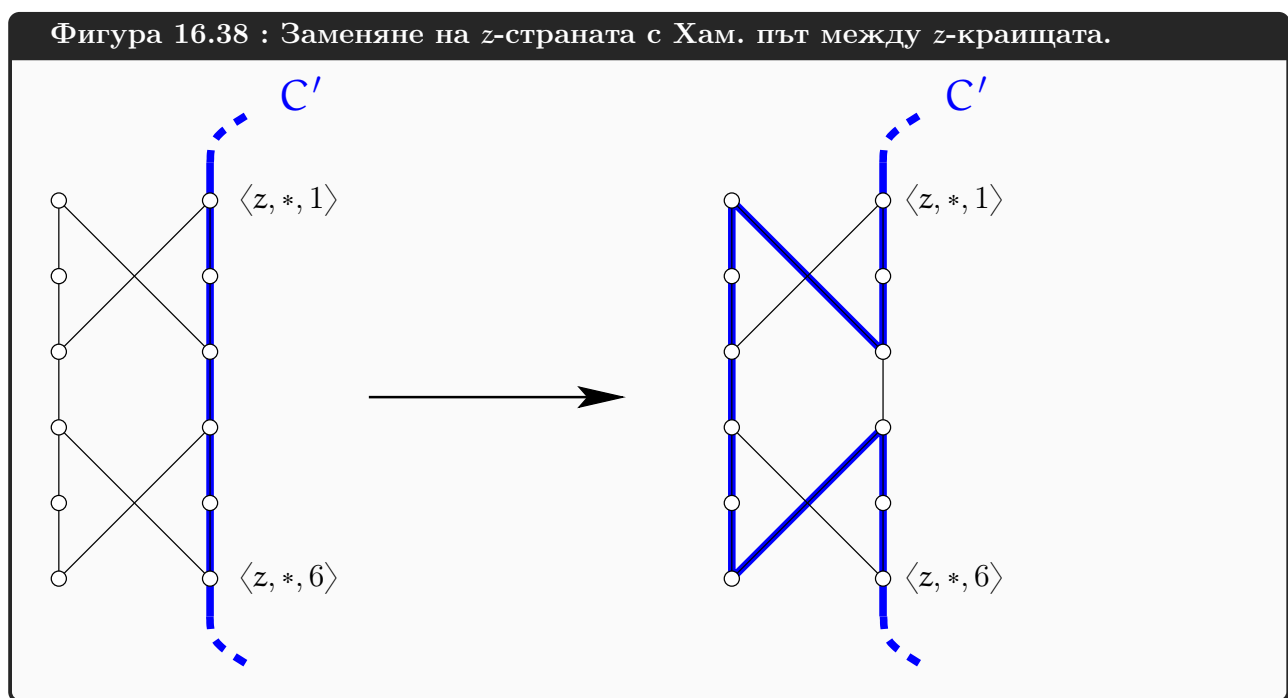
Разглеждаме и следното подмножество на E''' , което ще наречем E_U :

$$E_U = \{(\ell_1, \alpha_{u_1}), (\ell_1, \omega_{u_k})\} \cup \{(\ell_i, \omega_{u_{i-1}}) \mid 2 \leq i \leq k\} \cup \{(\ell_i, \alpha_{u_i}) \mid 2 \leq i \leq k\}$$

“Слепваме” веригите от P_U с ребрата от E_U . Получава се цикъл C' в J , който обаче не е Хамилтонов, освен ако k не е n . В общия случай, в който $k < n$, C' не е Хамилтонов, както предстои да видим.

Ключовото наблюдение е, че в J не може да има приспособление, през което не минава нито една верига от P_U . Ако има приспособление в J , през което не минава нито една верига от P_U , то съответното ребро в G не е инцидентно с нито един връх от U , което влече, че U не е върхово покриване.

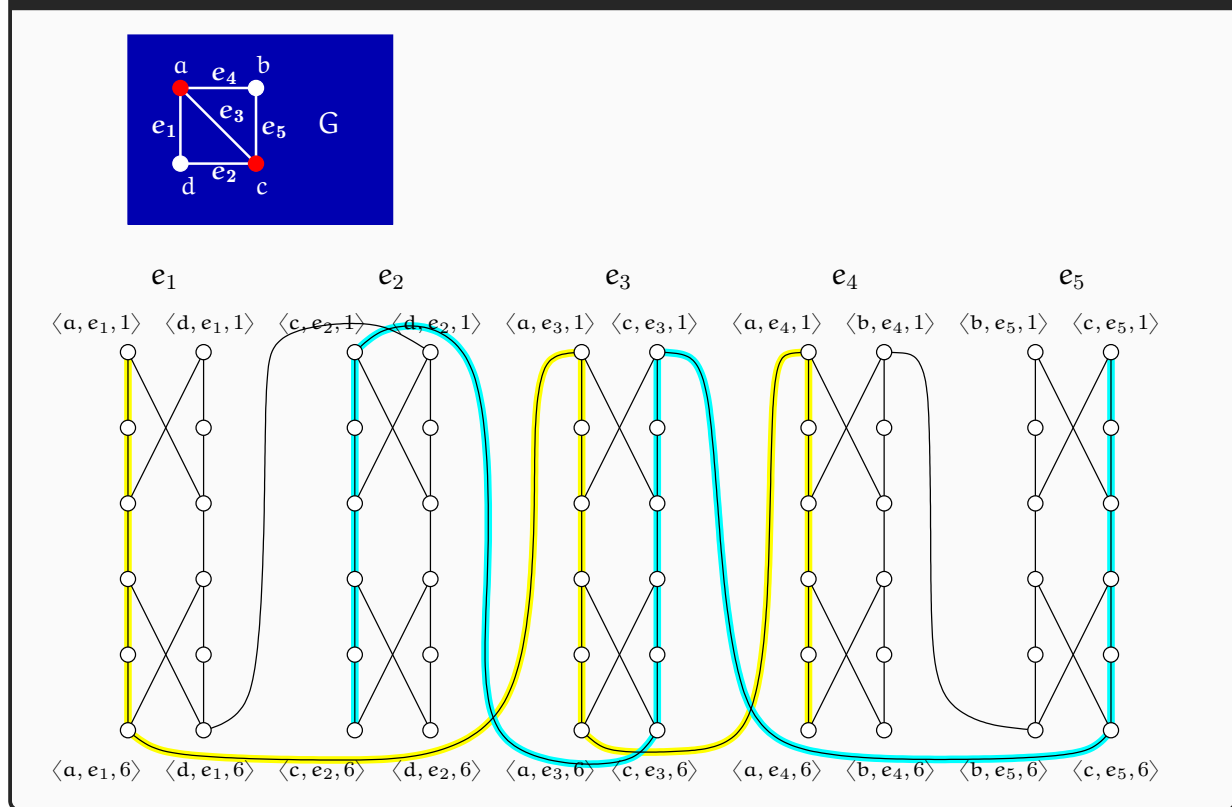
И така, през всяко приспособление минава поне една верига от P_U . Нещо повече. Приспособленията се категоризират в две категории: приспособления, през които минават две (различни) вериги от P_U , и приспособления, през които минава точно една верига от P_U . Вторият вид, и само те, са проблематични за Хамилтоновостта на C' : във всяко приспособление от втория вид, върховете от едната страна (а именно тази, през която не минава верига), не са в C' . Но това се решава елементарно! За всяко приспособление H , през което минава точно една верига p_z от P_U , сечението на p_z и H е z -страната на приспособлението и тя се явява подпът на C' . Тогава заменяме в C' този подпът с друг подпът с дължина 11, който е Хамилтонов път в H и чиито краища са z -краищата на H . По този начин цикълът C' вече съдържа всички върхове на H . Фигура 16.38 илюстрира това действие.



След като направим това за всички приспособления, през които минава точно една верига от P_U , наричаме получения C' с името “ C ” и забелязваме, че C е Хамилтонов цикъл в J .

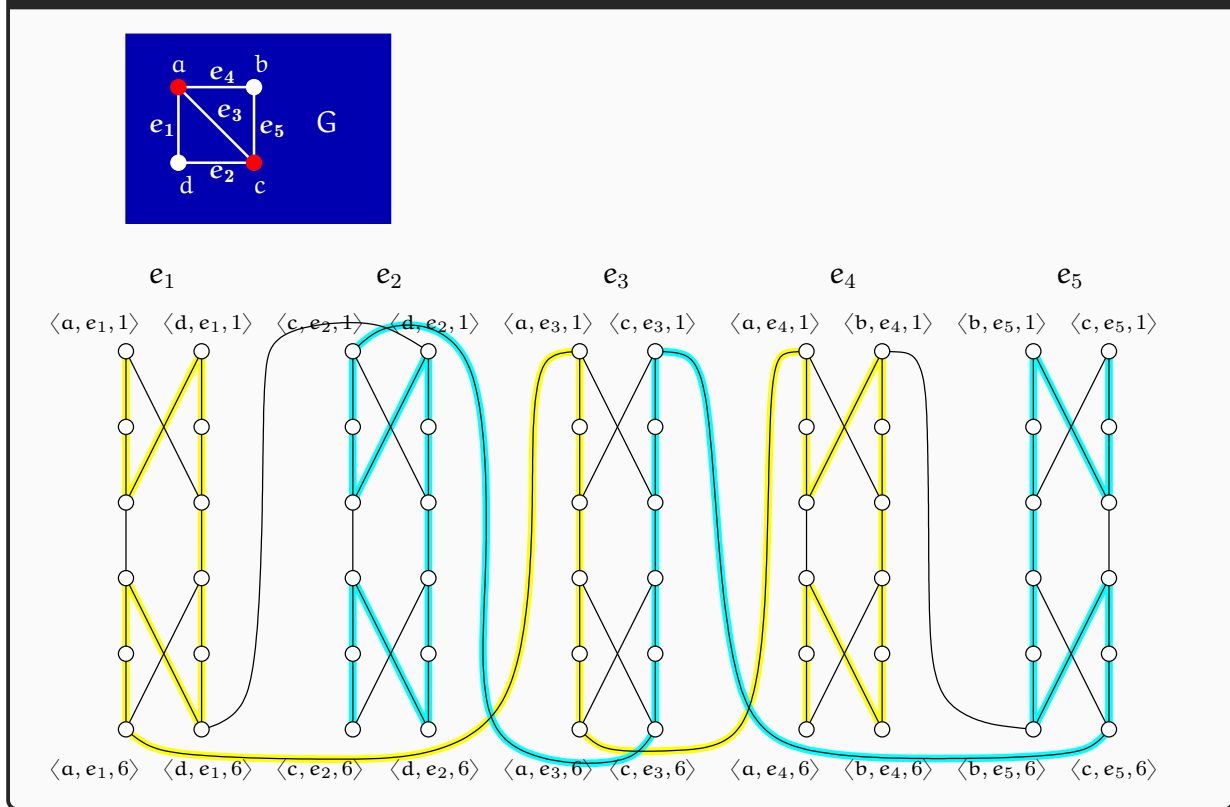
Преди да направим втората част от доказателството, да илюстрираме първата част върху примера, който разглеждахме горе. Фигура 16.39 показва графът G заедно с петте приспособления на съответния J и веригите от P_U . Множеството U е $\{a, c\}$, като $k = 2$. Множеството P_U е $\{p_a, p_c\}$. В жълто е веригата p_a , а в циан е веригата p_c .

Фигура 16.39 : Двете вериги от P_U в примера.



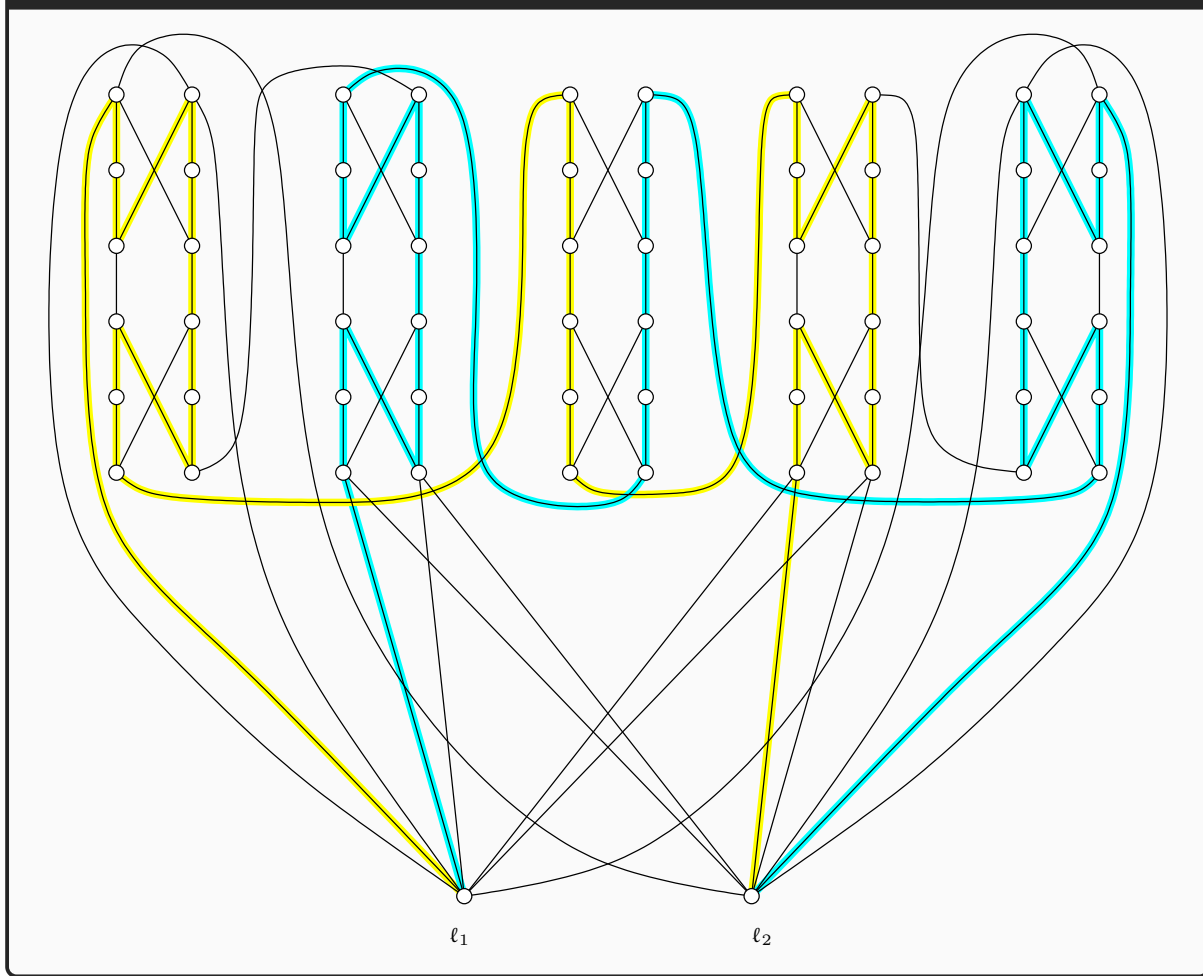
Ясно е, че от тези вериги и ребрата от E''' не можем да направим Хамилтонов цикъл, понеже четири от приспособленията има върхове, принадлежащи нито на p_a , нито на p_c . Модифицираме минаването на веригите през приспособленията по такъв начин, че всеки връх на приспособление да е в точно една верига от P_U , както е показано на Фигура 16.40.

Фигура 16.40 : Сега вече веригите минават през всички върхове на джаджи.



Сега вече има Хамилтонов цикъл, състоящ се от модифицираните вериги и ребра от E''' , както е показано на Фигура 16.41.

Фигура 16.41 : Има Хамилтонов цикъл!



Доказателство, 2: Нека съществува Хамилтонов цикъл C в J . Всеки от върховете $\ell_1, \dots, \ell_k$ се среща точно веднъж в C . Да изтрием $\ell_1, \dots, \ell_k$ от C . Получаваме колекция от k на брой пътя $q_1, \dots, q_k$. Смятаме за очевидно, че $\bigcup_{i=1}^k V(q_i) = V'$ и че краищата на всеки q_i съвпадат с краищата на някоя верига p_{u_i} , като обаче q_i не е непременно същият път като p_{u_i} . q_i минава през същите приспособления като p_{u_i} , и то в същия ред, но в рамките на едно приспособление H ,

- или q_i минава точно през u_i -страната на H , което значи, че някакъв друг q_j минава през другата страна на H ,
- или q_i минава през всички върхове на H като Хамилтонов път в него (в H).

Да има пътища $q_1, \dots, q_k$, минаващи по този начин през приспособленията, е същото като всяко ребро на G да е инцидентно с поне един връх от $u_1, \dots, u_k$. Показахме, че G има върхово покриване с размер k . $\square$

Разглеждаме HAMILTONIAN PATH (Задача 63). Вече показахме, че HAMILTONIAN CYCLE се редуцира до нея с Карг редукция (Редукция 3). Тогава HAMILTONIAN PATH $\in$ NP-с.

Дефинирахме задачата LONGEST PATH между два върха като оптимизационна задача в Допълнение 52 в Лекция 11 (Задача 31). За целите на тази лекция, разглеждаме задачите за най-дълъг цикъл и най-дълъг път като задачи за разпознаване, без специфициране на върхове.

Изч. Задача 86: LONGEST CYCLE, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ**екземпляр:** Неориентиран граф $G = (V, E)$, естествено число k .**въпрос:** Дали в G има цикъл с дължина $\geq k$?**Изч. Задача 87: LONGEST PATH, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ****екземпляр:** Неориентиран граф $G = (V, E)$, естествено число k .**въпрос:** Дали в G има път с дължина $\geq k$?Редукция 24: $\text{HAMILTONIAN CYCLE} \leq_p \text{LONGEST CYCLE}$

Конструкция: Това е най-лесният вид редукция: задачата, която редуцираме, се явява частен случай на задачата, към която редуцираме. Нещо подобно вече видяхме в Редукция 20. В случая, по даден екземпляр на HAMILTONIAN CYCLE , а именно граф G , генерираме съответен екземпляр $\langle G, n \rangle$ на LONGEST CYCLE . G да има Хамилтонов цикъл е същото като G да има цикъл с дължина n . Ерго, G е ДА-екземпляр на HAMILTONIAN CYCLE тстк $\langle G, n \rangle$ е ДА-екземпляр на LONGEST CYCLE . $\square$

Редукция 25: $\text{HAMILTONIAN PATH} \leq_p \text{LONGEST PATH}$

Конструкция: Напълно аналогично на Редукция 24, по даден екземпляр на HAMILTONIAN PATH , а именно граф G , генерираме съответен екземпляр $\langle G, n - 1 \rangle$ на LONGEST PATH . G да има Хамилтонов път е същото като G да има с дължина $n - 1$. Ерго, G е ДА-екземпляр на HAMILTONIAN PATH тстк $\langle G, n \rangle$ е ДА-екземпляр на LONGEST PATH . $\square$

16.2.9 2-PARTITION

Раззглеждаме Задача 36. За всеки случай, ето дефиницията отново. Ползваме името S за множеството, за да няма колизия с името “ A ”, което се появява в описанието на екземпляра на 2-PARTITION.

Изч. Задача: 2-PARTITION**екземпляр:** Крайно непразно множество S . Функция $w : S \rightarrow \mathbb{N}^+$.**въпрос:** Дали съществува $X \subset S$, такова че $\sum_{s \in X} w(s) = \sum_{s \in S \setminus X} w(s)$?Редукция 26: $3DM \leq_p 2\text{-PARTITION}$

Конструкция: Това е първата редукция, в която задачата, към която редуцираме, е задача с числа. Вече видяхме редукция, при която задачата, **от** която редуцираме, има числа – това беше $\text{HAMILTONIAN CYCLE} \leq_p \text{TSP}$ (Редукция 1). Но сега задачата, **към** която редуцираме, се дефинира чрез числа – това са числата $w(s)$ по всички $s \in S$.

Даден е екземпляр $x = \langle B, C, D, A \rangle$ на 3DM, където $B = \{b_1, \dots, b_n\}$, $C = \{c_1, \dots, c_n\}$ и $D = \{d_1, \dots, d_n\}$ са множества с празни сечения по двойки и $A \subseteq \{\{b, c, d\} \mid b \in B, c \in C, d \in D\}$.

Нека $A = \{a_1, \dots, a_m\}$, където $a_j = \{b_{p_j}, c_{q_j}, d_{r_j}\}$ за $j \in \{1, \dots, m\}$, където $p_j, q_j, r_j \in \{1, \dots, n\}$. Ще конструираме съответен екземпляр $y = \langle S, w \rangle$ на 2-PARTITION, който е

БОО, $m \geq n$, иначе x задължително е НЕ-екземпляр. Построяваме S , такова че $|S| = m + 2$. По начина, по който дефинирахме 2-PARTITION, елементите на S са просто два по два различни обекти. Единствено има значение стойностите на функцията w върху тях. За да опишем напълно редуцията, необходимо и достатъчно е да дефинираме $w(s)$ за всяко $s \in S$. Да кажем, че $S = \{s_1, \dots, s_m, s_{m+1}, s_{m+2}\}$. Елементите на S са два вида:

- $s_1, \dots, s_m$ са единият вид, като те съответстват биективно на елементите на A и получават своите тегла по едно правило,
- s_{m+1} и s_{m+2} не съответстват на елементи на A и техните тегла се образуват по съвсем различен начин.

Една полезна нотация: за всеки двоичен стринг x означаваме с " $v(x)$ " числото, което е записано с x в двоична позиционна бройна система.

Теглата на $s_1, \dots, s_m$. Конструираме двоични стрингове $\sigma_1, \sigma_2, \dots, \sigma_m$, като за всяко $j \in \{1, \dots, m\}$, $w(s_j) = v(\sigma_j)$. С други думи, дефинираме $w(s_j)$ чрез неговия запис в двоична позиционна бройна система σ_j .

Нека $g = \lfloor \log_2 m \rfloor + 1$ [†]. За всяко $j \in \{1, \dots, m\}$, σ_j е двоичен стринг с дължина $3ng$. σ_j е конкатенацията на три подстринга $\sigma_{j,B}$, $\sigma_{j,C}$ и $\sigma_{j,D}$, всеки с дължина ng :

$$\sigma_j = \sigma_{j,B} \sigma_{j,C} \sigma_{j,D}$$

Както предстои да видим,

- $\sigma_{j,B}$ описва участието на елемент от B в a_j ,
- $\sigma_{j,C}$ описва участието на елемент от C в a_j , а
- $\sigma_{j,D}$ описва участието на елемент от D в a_j .

За всяко $j \in \{1, \dots, m\}$, за всяко $\ell \in \{B, C, D\}$, $\sigma_{j,\ell}$ е конкатенация от n стринга $\sigma_{j,\ell,1}, \sigma_{j,\ell,2}, \dots, \sigma_{j,\ell,n}$, всеки с дължина g :

$$\sigma_{j,\ell} = \sigma_{j,\ell,1} \sigma_{j,\ell,2} \cdots \sigma_{j,\ell,n}$$

Ето какво е съдържанието на тези стрингове. За всяко $j \in \{1, \dots, m\}$:

- $\sigma_{j,B,p_j} = 00 \cdots 01$ и $\forall i \in \{1, \dots, n\} \setminus \{p_j\} : \sigma_{j,B,i} = 00 \cdots 0$;
- $\sigma_{j,C,q_j} = 00 \cdots 01$ и $\forall i \in \{1, \dots, n\} \setminus \{q_j\} : \sigma_{j,C,i} = 00 \cdots 0$;
- $\sigma_{j,D,r_j} = 00 \cdots 01$ и $\forall i \in \{1, \dots, n\} \setminus \{r_j\} : \sigma_{j,D,i} = 00 \cdots 0$.

И така, в σ_j има точно три единици. Лесно се вижда, че

$$v(\sigma_j) = 2^{g(3n-p_j)} + 2^{g(2n-q_j)} + 2^{g(n-r_j)} \quad (16.11)$$

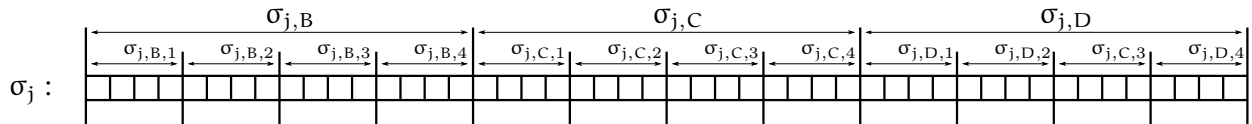
за всяко $j \in \{1, \dots, m\}$.

[†]Забележете, че g е броят на битовете в запис на m в двоична позиционна бройна система.

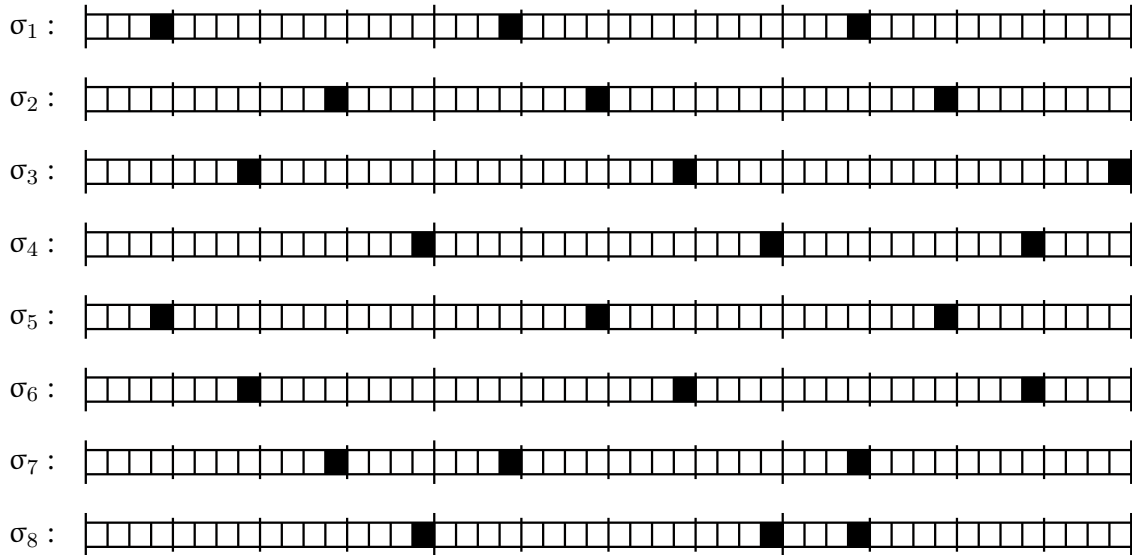
Ако това обяснение е прекалено формално, ето пример. Нека $B = \{b_1, b_2, b_3, b_4\}$, $C = \{c_1, c_2, c_3, c_4\}$, $D = \{d_1, d_2, d_3, d_4\}$ и

$$A = \{\{b_1, c_1, d_1\}, \{b_3, c_2, d_2\}, \{b_2, c_3, d_4\}, \{b_4, c_4, d_3\}, \\ \{b_1, c_2, d_2\}, \{b_2, c_3, d_3\}, \{b_3, c_1, d_1\}, \{b_4, c_4, d_1\}\}$$

В този пример $n = 4$, $m = 8$ и $g = 4$. Конструкцията генерира осем двоични стринга $\sigma_1, \dots, \sigma_8$, всеки от които има дължина $3 \times 4 \times 4 = 48$. Всеки σ_j е конкатенация от три подстринга, всеки с дължина 16, като на свой ред всеки от тези подстрингове е конкатенация от 4 подподстринга с дължина 4. Ето общата схема на кой да е σ_j .



А ето и самите стрингове. За по-голяма яснота, наместо да пишем нулите и единиците, рисуваме клетките и за всяко σ_j , точно три от тях—тези позиции, които съдържат единици—запълваме с черен цвят, а останалите (нулите) са празни.



Да видим какви числа са записани с тези двоични стрингове.

$$v(\sigma_1) = 2^{44} + 2^{28} + 2^{12} = 17\,592\,454\,483\,968$$

$$v(\sigma_2) = 2^{36} + 2^{24} + 2^8 = 68\,736\,254\,208$$

$$v(\sigma_3) = 2^{40} + 2^{20} + 2^0 = 1\,099\,512\,676\,353$$

$$v(\sigma_4) = 2^{32} + 2^{16} + 2^4 = 4\,295\,032\,848$$

$$v(\sigma_5) = 2^{44} + 2^{24} + 2^8 = 17\,592\,202\,821\,888$$

$$v(\sigma_6) = 2^{40} + 2^{20} + 2^4 = 1\,099\,512\,676\,368$$

$$v(\sigma_7) = 2^{36} + 2^{28} + 2^{12} = 68\,987\,916\,288$$

$$v(\sigma_8) = 2^{32} + 2^{16} + 2^4 = 4\,295\,032\,848$$

Да се убедим, че (16.11) е в сила за $v(\sigma_1)$. Наистина, $p_1 = q_1 = r_1 = 1$, така че (16.11) дава

$$v(\sigma_1) = 2^{g(3n-p_1)} + 2^{g(2n-q_1)} + 2^{g(n-r_1)} = 2^{4(12-1)} + 2^{4(8-1)} + 2^{4(4-1)} = 2^{4 \times 11} + 2^{4 \times 7} + 2^{4 \times 3}$$

Очевидно стринговете, които се конструират, са записи на грамадни числа дори при скромните размери на екземпляра, който разглеждаме.

Теглата на s_{m+1} и s_{m+2} . Нека

$$H = \sum_{t=0}^{3n-1} 2^{gt}$$

H е това число, което в двоична позиционна бройна система се записва със стринга с дължина $3ng$, който има $3n$ единици, а именно на позиции $0, g, \dots, (3n-1)g$, а на останалите позиции има нули. За примера, който видяхме току-що, $H = \sum_{t=0}^{11} 2^{4t} = 18\,764\,998\,447\,377$. Неговият запис в двоична позиционна бройна система, със запълнени и празни клетки, изглежда така:

H :

Използвайки H , теглата на s_{m+1} и s_{m+2} са следните.

$$w(s_{m+1}) = 2 \left(\sum_{j=1}^m v(\sigma_j) \right) - H \quad (16.12)$$

$$w(s_{m+2}) = \left(\sum_{j=1}^m v(\sigma_j) \right) + H \quad (16.13)$$

И това е краят на описанието на конструкцията.

В примера, който разглеждаме, $\sum_{i=1}^8 v(\sigma_i) = 37\,529\,996\,894\,769$. Тогава

$$w(s_{m+1}) = 2 \times 37\,529\,996\,894\,769 - 18\,764\,998\,447\,377 = 56\,294\,995\,342\,161$$

$$w(s_{m+2}) = 37\,529\,996\,894\,769 + 18\,764\,998\,447\,377 = 56\,294\,995\,342\,146$$

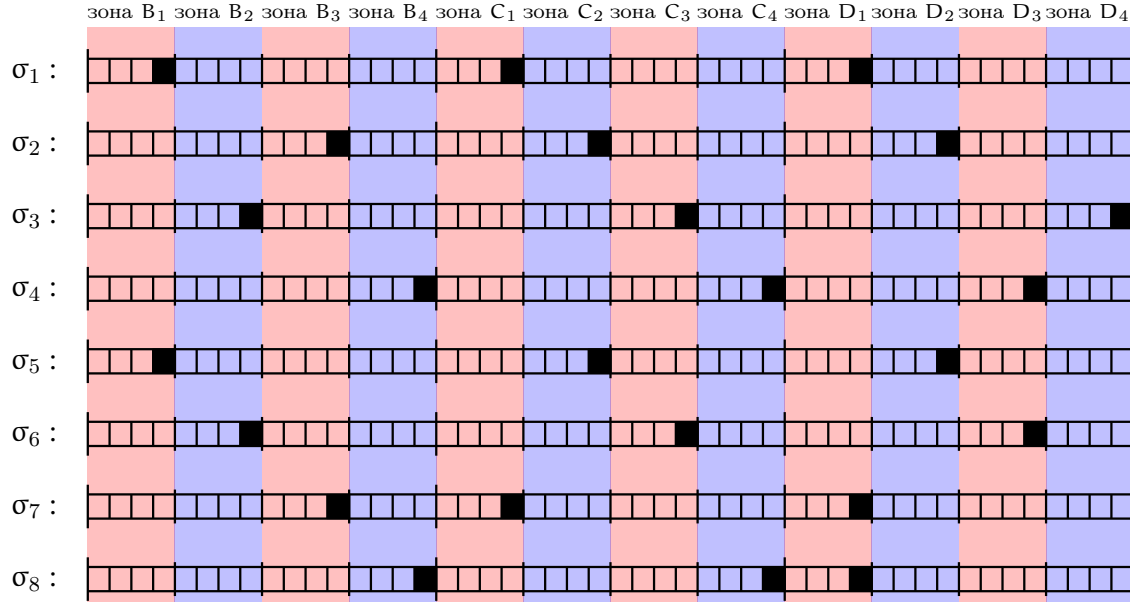
Коректност на конструкцията. Когато става дума за сумиране на числата $v(\sigma_j)$, както е в (16.12) и (16.13), удобно е да мислим за сумиране на директно на стринговете σ_j . Тъй като тези стрингове са записите на числата в двоична бройна система, това е нещо напълно естествено.

Удобно е да мислим за всички стрингове $\sigma_1, \dots, \sigma_m$, наредени един над друг и подравнени вляво (и в дясно, щом са с еднакви дължини). Да дефинираме, че

- подподстринговете $\sigma_{1,V,1}, \dots, \sigma_{m,V,1}$ образуват зона V_1 ,
- подподстринговете $\sigma_{1,V,2}, \dots, \sigma_{m,V,2}$ образуват зона V_2 ,
- и така нататък,
- подподстринговете $\sigma_{1,V,n}, \dots, \sigma_{m,V,n}$ образуват зона V_n ,
- подподстринговете $\sigma_{1,C,1}, \dots, \sigma_{m,C,1}$ образуват зона C_1 ,
- подподстринговете $\sigma_{1,C,2}, \dots, \sigma_{m,C,2}$ образуват зона C_2 ,
- и така нататък,

- подподстринговете $\sigma_{1,C,n}, \dots, \sigma_{m,C,n}$ образуват зона C_n ,
- подподстринговете $\sigma_{1,D,1}, \dots, \sigma_{m,D,1}$ образуват зона D_1 ,
- подподстринговете $\sigma_{1,D,2}, \dots, \sigma_{m,D,2}$ образуват зона D_2 ,
- и така нататък,
- подподстринговете $\sigma_{1,D,n}, \dots, \sigma_{m,D,n}$ образуват зона D_n .

По този начин дефинираме $3n$ зони. Ето зоните за примера, който видяхме преди.



Да се убедим, че $w(s_{m+1})$ е положително число, ако $\mathbf{x}$ е ДА-екземпляр на 3DM. Има смисъл да разсъждаваме върху това, понеже дефиницията (16.12) на $w(s_{m+1})$ използва разлика. Щом $\mathbf{x}$ е ДА-екземпляр, задължително има $a_j \in A$, такъв че $b_1 \in a_j$. Това е същото като да кажем, че в σ_j има единица в най-дясната клетка на зона В₁. След сумирането $\sigma_1 + \dots + \sigma_m$ тази единица може да се сумира с други и това да доведе до поява на единица още по-наляво. Умножаването по две на сумата, в двоичния запис, е преместване на единиците с една клетка наляво. Заклучаваме, че със сигурност в запис на удвоената сума има единица на позиция, която е вляво от позицията на най-лявата единица на Н. Тогава $2 \left(\sum_{j=1}^m v(\sigma_j) \right) - H > 0$.

Да видим каква е сумата от теглата на всички елементи. Очевидно $w(s_{m+1}) + w(s_{m+2}) = 3 \left(\sum_{j=1}^m v(\sigma_j) \right)$. Тогава

$$\sum_{j=1}^{m+2} w(s_j) = \left(\sum_{j=1}^m v(\sigma_j) \right) + w(s_{m+1}) + w(s_{m+2}) = 4 \left(\sum_{j=1}^m v(\sigma_j) \right) \quad (16.14)$$

Наблюдение 113: При сумиране на единиците от зона не се получава препълване

Дори да има зона, която съдържа m единици, при сумирането $\sigma_1 + \dots + \sigma_m$ не може да се получи препълване и прехвърляне на единици от тази зона в зоната вляво, ако има такава. Ширината $\lceil \log_2 m \rceil + 1$ на всяка зона е достатъчна, за да не се получи такава препълване. Следователно, след сумирането $\sigma_1 + \dots + \sigma_m$, за всяка зона е вярно, че единиците в нея “идват” само от нея, а не от препълване на зони вдясно.

Ключовото твърдение за доказателството за коректност е тази лема.

Лема 61

$A' \subseteq A$ е перфектно съчетание тстк

$$\sum_{\sigma \in Z'} v(\sigma) = H$$

където Z' е подмножеството на $\{\sigma_1, \dots, \sigma_m\}$, чиито стрингове отговарят на елементите на A' .

Доказателство: В едната посока, нека A' е перфектно съчетание. Тогава всяко b_i , всяко c_i и всяко d_i се появяват точно по един път в A' . Тогава, за съответните двоични стрингове е вярно, че във всяка зона се намира точно една единица, и то в най-дясната позиция. Тогава сумирането на (числата, съответни на) тези стрингове дава точно H .

В другата посока, нека подмножество Z' на стринговете има сума (на съответните числа) H . Тогава във всяка зона на сумата има точно една единица, и то в най-дясната позиция. Съгласно Наблюдение 113, единицата във всяка зона на сумата идва от точно една единица в някой подподстринг от Z' . Това означава, че елементите на A , съответни на стринговете от Z' , са такива, че всяко b_i , всяко c_i и всяко d_i се среща точно по един път. Но тогава тези елементи представляват перфектно съчетание. $\square$

Теорема 134: Коректността на $3DM \leq_p 2\text{-PARTITION}$

В текущия контекст, x е ДА-екземпляр на $3DM$ тстк y е ДА-екземпляр на 2-PARTITION .

Доказателство: В едната посока, нека x е ДА-екземпляр на $3DM$. Тогава A съдържа перфектно съчетание $A' \subseteq A$. Нека Z' е множеството от двоичните стрингове, съответни на елементите на A' . От Лема 61 знаем, че сумата от числата, съответни на стринговете от Z' , е точно H . Но тогава

$$\sum_{\sigma \in Z'} v(\sigma) + w(s_{m+1}) = H + 2 \left(\sum_{j=1}^m v(\sigma_j) \right) - H = 2 \left(\sum_{j=1}^m v(\sigma_j) \right)$$

Но от (16.14) знаем, че $\sum_{j=1}^{m+2} w(s_j) = 4 \left(\sum_{j=1}^m v(\sigma_j) \right)$. Тогава $\sum_{\sigma \in Z'} v(\sigma) + w(s_{m+1})$ е точно полусумата от всички тегла на елементи. Тогава y е ДА-екземпляр на 2-PARTITION .

В другата посока, нека y е ДА-екземпляр на 2-PARTITION . Тогава $\exists S' \subset \{s_1, s_2, \dots, s_m, s_{m+1}, s_{m+2}\}$, такова че $\sum_{s \in S'} w(s) = \sum_{s \in S \setminus S'} w(s)$. Както вече знаем, $\sum_{j=1}^{m+2} w(s_j) = 4 \left(\sum_{j=1}^m v(\sigma_j) \right)$. Тогава $\sum_{s \in S'} w(s) = 2 \left(\sum_{j=1}^m v(\sigma_j) \right)$.

Но тогава трябва единият от s_{m+1} и s_{m+2} да се намира в S' , а другият да се намира в $S \setminus S'$. Причината е, че теглата на s_{m+1} и s_{m+2} са прекалено големи: $w(s_{m+1}) + w(s_{m+2}) = 3 \left(\sum_{j=1}^m v(\sigma_j) \right)$. БОО, нека $s_{m+1} \in S'$. Тогава

$$\sum_{s \in S' \setminus \{s_{m+1}\}} w(s) = \underbrace{2 \left(\sum_{j=1}^m v(\sigma_j) \right)}_{\text{сумалното тегло на } S'} - \underbrace{\left(2 \left(\sum_{j=1}^m v(\sigma_j) \right) - H \right)}_{\text{теглото на } s_{m+1}} = H$$

И така, има подмножество от двоични стрингове, такова че числата, съответни на неговите елементи се сумират до N . Съгласно Лема 61, A съдържа перфектно съчетание. $\square$

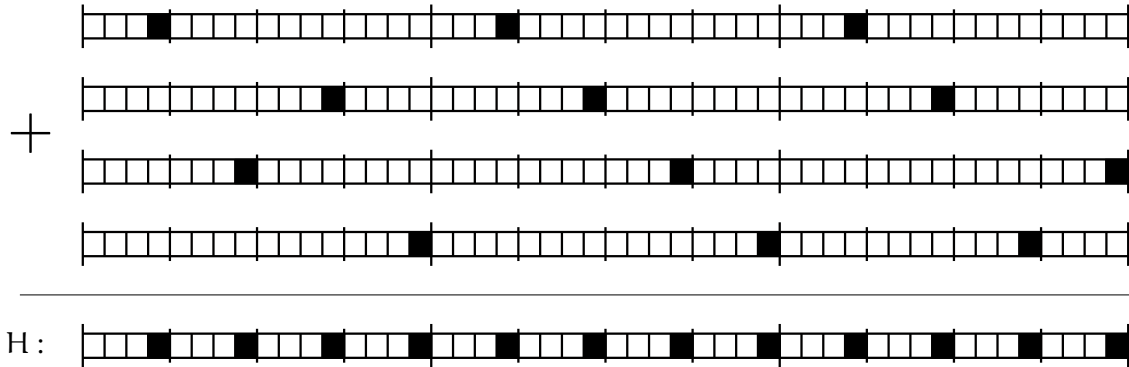
Да разгледаме отново примера, който въведохме горе:

$$A = \{\{b_1, c_1, d_1\}, \{b_3, c_2, d_2\}, \{b_2, c_3, d_4\}, \{b_4, c_4, d_3\}, \\ \{b_1, c_2, d_2\}, \{b_2, c_3, d_3\}, \{b_3, c_1, d_1\}, \{b_4, c_4, d_1\}\}$$

Той има перфектно съчетание, примерно

$$A' = \{\{b_1, c_1, d_1\}, \{b_3, c_2, d_2\}, \{b_2, c_3, d_4\}, \{b_4, c_4, d_3\}\}$$

Ето стринговете, съответни на тези тройки. Очевидно те се сумират точно до N .



16.2.10 SUBSET SUM

Изч. Задача 88: SUBSET SUM

екземпляр: Крайно непразно множество S . Функция $w : S \rightarrow \mathbb{N}^+$. Число $K \in \mathbb{N}^+$.

въпрос: Дали съществува подмножество $X \subseteq A$, такова че $\sum_{s \in X} w(s) = K$?

Числото K се нарича *цел*.

Задачата е очевидно обобщение на 2-PARTITION (Задача 36), поради което редукцията е елементарна.

Редукция 27: 2-PARTITION $\leq_p$ SUBSET SUM

Конструкция: По даден екземпляр $x = \langle S, w \rangle$ на 2-PARTITION, където където $S = \{s_1, s_2, \dots, s_n\}$ е множество и $w : S \rightarrow \mathbb{N}^+$ е тегловна функция върху него конструираме екземпляр $y = \langle S, w, K \rangle$ на SUBSET SUM. Трябва само да определим каква е целта K .

Ако сумата на S е нечетна, което влече, че x е НЕ-екземпляр, то конструираме такова K , че y да е също НЕ-екземпляр. Това е елементарно, примерно нека K е сумата на S .

Ако сумата на S е четна, то K е полусумата на S . Абсолютно очевидно е, че y е ДА-екземпляр тстк x е ДА-екземпляр. $\square$

16.2.11 UNBOUNDED KNAPSACK

Въведохме задачата в Подподсекция 12.5.3.1 като оптимизационна задача. За целите на редукцията ни трябва задачата във версия за разпознаване.

Изч. Задача 89: UNBOUNDED KNAPSACK, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Множество от видове предмети $A = \{a_1, \dots, a_n\}$, като има неограничено много предмети от всеки вид. За всеки вид a_i , $1 \leq i \leq n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$ и цел на крадеца $G \in \mathbb{N}^+$.

решение: Дали съществува $(x_1, \dots, x_n) \in \mathbb{N}^n$, такова че

$$\sum_{i=1}^n x_i w(a_i) \leq C$$

$$\sum_{i=1}^n x_i v(a_i) \geq G?$$

Редукция 28: SUBSET SUM $\leq_p$ UNBOUNDED KNAPSACK, РАЗПОЗНАВАНЕ

Конструкция: По даден екземпляр $x = \langle S, f, K \rangle$ на SUBSET SUM, където $S = \{s_1, s_2, \dots, s_n\}$ е множество и $f : S \rightarrow \mathbb{N}^+$ е тегловна функция върху него и $K \in \mathbb{N}^+$ е целта, конструираме екземпляр $y = \langle A, v, w, C, G \rangle$ на UNBOUNDED KNAPSACK, където A е множество, $v : A \rightarrow \mathbb{N}^+$ е функцията на стойностите, $w : A \rightarrow \mathbb{N}^+$ е функцията на теглата, $C \in \mathbb{N}^+$ е капацитетът и $G \in \mathbb{N}^+$ е целта.

Нека M е число, такова че

$$M > \max \{f(s_i) \mid i \in \{1, \dots, n\}\} \wedge M > K$$

Конструираме множеството A с $2n$ предмети, като за всяко $i \in \{1, \dots, n\}$, множеството A съдържа два предмета, “нулев” и “единичен”:

$$A = \{a_i^0 \mid i \in \{1, \dots, n\}\} \cup \{a_i^1 \mid i \in \{1, \dots, n\}\}$$

Цените и теглата на предметите се конструират така. За всяко $i \in \{1, \dots, n\}$,

$$v(a_i^0) = w(a_i^0) = (2^{n+1} + 2^i)nM \quad (16.15)$$

$$v(a_i^1) = w(a_i^1) = (2^{n+1} + 2^i)nM + f(s_i) \quad (16.16)$$

Капацитетът и целта са равни:

$$C = G = ((n+1)2^{n+1} - 2)nM + K$$

Очевидно конструкцията може да се извърши в полиномиално време. Да обосновем коректността ѝ. Нека x е ДА-екземпляр на SUBSET SUM. Тогава съществува $X \subseteq S$, такова че $\sum_{x \in X} f(x) = K$. Да кажем, че $X = \{s_{i_1}, s_{i_2}, \dots, s_{i_t}\}$. Да разгледаме цените на съответните единични елементи на A :

$$\begin{aligned} v(a_{i_1}^1) &= (2^{n+1} + 2^{i_1})nM + f(s_{i_1}) \\ v(a_{i_2}^1) &= (2^{n+1} + 2^{i_2})nM + f(s_{i_2}) \\ &\dots \\ v(a_{i_t}^1) &= (2^{n+1} + 2^{i_t})nM + f(s_{i_t}) \end{aligned} \quad (16.17)$$

За останалите индекси да разгледаме цените на съответните нулеви елементи на A . A именно, за всяко $j \in \{1, \dots, n\} \setminus \{i_1, \dots, i_t\}$,

$$v(a_j^0) = (2^{n+1} + 2^j)nM \quad (16.18)$$

Забележете, че (16.17) и (16.18) съдържат общо n равенства, защото от всяка двойка нулев-единичен елемент на A , точно единият се появява точно веднъж вляво. Предвид факта, че всяка дясна страна съдържа събираемо $(2^{n+1} + 2^p)nM$ за някое $p \in \{1, \dots, n\}$ и всяко $p \in \{1, \dots, n\}$ се среща като степенен показател на двойката в точно в едно равенство, ако сумираме десните страни на равенствата от (16.17) и (16.18), получаваме

$$(n2^{n+1} + 2^n + 2^{n-1} + \dots + 2^2 + 2^1)nM + f(s_{i_1}) + f(s_{i_2}) + \dots + f(s_{i_t})$$

Но това е същото като

$$(n2^{n+1} + 2^{n+1} - 1 - 1)nM + K$$

което може да оппростим до:

$$((n+1)2^{n+1} - 2)nM + K$$

Но това е точно G , целта на крадеца.

Напълно аналогично се показва, че сумата от теглата е точно C , което е капацитетът на раницата. Ерго, при тези тегла, цени, капацитет и цел има начин да бъдат подбрани n предмети—а именно, единичните, ако съответният елемент на S е в X , и нулевите, ако съответният елемент на S не е в X —по такъв начин, че ограниченията за крадеца и раницата са удовлетворени.

Сега да допуснем, че y е ДА-екземпляр на UNBOUNDED KNAPSACK. Но тогава съществува $A' \subseteq A$, такова че сумарната цена на предметите в A' е поне G и сумарното тегло на предметите в A' е най-много C . Първо ще покажем, че $|A'| = n$. По конструкция,

$$C = G = ((n+1)2^{n+1} - 2)nM + K \quad (16.19)$$

Но (16.19) е сума на събираеми от вида

$$(2^{n+1} + 2^i)nM + q \quad (16.20)$$

като $i \in \{1, \dots, n\}$, а q е или 0, или някое $f(s_j)$ (припомнете си (16.15) и (16.16)).

- Да допуснем, че $|A'| \leq n-1$. Ще покажем, че сумата на $n-1$ събираеми от вида (16.20) е задължително по-малка от (16.19).

Да намерим подходяща горна граница за сумата на $n-1$ събираеми от вида (16.20). Нека $r = \max \{f(s_i) \mid 1 \leq i \leq n\}$. Явно $(2^{n+1} + 2^n)nM + r$ е горна граница за събираемо от вида (16.20), откъдето сумата на $n-1$ събираеми от вида (16.20) не е по-голяма от $(2^{n+1} + 2^n)n(n-1)M + r(n-1)$. За съжаление, тази горна граница е прекалено груба и неточна и не ни дава желанния резултат. Ще намерим по-добра горна граница за въпросната сума на $n-1$ събираеми.

Разсъждаваме така: всяко събираемо е не по-голямо от $2^{n+1}nM + 2^i nM + r$, за някое $i \in \{1, \dots, n\}$. Когато сумираме всички $n-1$ такива събираеми, получаваме едно събираемо $X = 2^{n+1}(n-1)nM$, едно събираемо $Y = (n-1)r$ и още едно събираемо, да го наречем Z ,

което е сумата на всички $2^i nM$. Да намерим подходяща горна граница за Z . Ключовото наблюдение е, че

- ♦ от всички $n - 1$ събираеми в Z , най-много две могат да имат $i = n$, а именно тези, които отговарят на a_n^0 и a_n^1 ,
- ♦ от всички $n - 1$ събираеми в Z , най-много две могат да имат $i = n - 1$, а именно тези, които отговарят на a_{n-1}^0 и a_{n-1}^1 ,
- ♦ и така нататък.

Поради това може да ограничим Z отгоре с

$$2 \cdot 2^n + 2 \cdot 2^{n-1} + \dots + 2 \cdot 2^2 + 2 \cdot 2^1 = 2(2^n - 2)nM = (2^{n+1} - 4)nM$$

Тогава следната сума е горна граница за $X + Z + Y$:

$$\begin{aligned} & 2^{n+1}(n-1)nM + (2^{n+1} - 4)nM + (n-1)r = \\ & 2^{n+1}n^2M - 2^{n+1}nM + 2^{n+1}nM - 4nM + (n-1)r = \\ & 2^{n+1}n^2M - 4nM + (n-1)r \end{aligned} \quad (16.21)$$

Твърдим, че (16.19) е задължително по-голямо от (16.21). Наистина,

$$\begin{aligned} ((n+1)2^{n+1} - 2)nM + K &> 2^{n+1}n^2M - 4nM + (n-1)r &< \leftrightarrow \\ 2^{n+1}n^2M + 2^{n+1}nM - 2nM + K &> 2^{n+1}n^2M - 4nM + (n-1)r &< \leftrightarrow \\ 2^{n+1}nM + K &> -2nM + (n-1)r \end{aligned}$$

което със сигурност е вярно, понеже лявата страна е положително число, а дясната е отрицателно число. За да се убедим в последното, достатъчно е да си припомним, че по конструкция $M > \max \{f(s_i) \mid i \in \{1, \dots, n\}\}$, а $r = \max \{f(s_i) \mid i \in \{1, \dots, n\}\}$.

- Да допуснем, че $|A'| \geq n+1$. Ще покажем, че сумата на $n+1$ събираеми от вида (16.20) е задължително по-голяма от (16.19).

Да намерим подходяща долна граница за сумата на $n+1$ събираеми от вида (16.20). Разсъждаваме така: всяко събираемо е не по-малко от $2^{n+1}nM + 2^i nM$, за някое $i \in \{1, \dots, n\}$. Когато сумираме $n+1$ такива събираеми, получаваме едно събираемо $X = 2^{n+1}(n+1)nM$ и още едно събираемо, да го наречем Z , което е сумата на всички $2^i nM$. Сумата от всички степени на двойката е поне $2n$, поради което $Z > (2n)nM$. Тогава следната сума е долна граница за $X + Z$:

$$2^{n+1}(n+1)nM + (2n)nM \quad (16.22)$$

Твърдим, че (16.19) е задължително по-малко от (16.22). Наистина,

$$\begin{aligned} ((n+1)2^{n+1} - 2)nM + K &< 2^{n+1}(n+1)nM + (2n)nM &< \leftrightarrow \\ 2^{n+1}(n+1)nM - 2nM + K &< 2^{n+1}(n+1)nM + (2n)nM &< \leftrightarrow \\ -2nM + K &< (2n)nM \end{aligned}$$

което със сигурност е вярно, понеже $M > K$ по конструкция.

И така, отхвърлихме възможностите $|A'| < n$ и $|A'| > n$, което означава, че $|A'| = n$. На свой ред, това означава, че n събираеми, всяко от които е от един от тези два вида

$$2^{n+1}nM + 2^i nM$$

$$2^{n+1}nM + 2^i nM + f(s_i)$$

се сумират точно до

$$(n2^{n+1})nM + (2^n + 2^{n-1} + \dots + 2^2 + 2^1)nM + K$$

Тъй като M е по-голямо както от K , така и от всяко $f(s_i)$, единственият начин да се случи това е за всяко $i \in \{1, \dots, n\}$ да участва или $2^{n+1}nM + 2^i nM$, или $2^{n+1}nM + 2^i nM + f(s_i)$, като K е сумата на тези $f(s_i)$. Последното влече, че x е ДА-екземпляр на SUBSET SUM, понеже има подмножество S , чиято сума е K ; а именно тези елементи S , на които съответстват единични предмети от A , които крадецът е сложил в раницата.

В резюме, крадецът винаги слага в раницата точно n предмета, като за всяко $i \in \{1, \dots, n\}$ взема или единичния, или нулевия предмет от i -ия вид; ако при това бива достигнат капацитетът на раницата и целта на крадеца, гарантирано е вярно, че елементите на S , съответни на избраните единични предмети, имат сума точно K . $\square$

Да разгледаме примери. Първо разглеждаме ДА-екземпляр $x = \langle \{1, 9, 5, 3, 8\}_M, K = 13 \rangle$ на SUBSET SUM. Това наистина е ДА-екземпляр, защото $1 + 9 + 3 = 13$.

Да конструираме съответният екземпляр $y = \langle A, v, w, C, G \rangle$ на UNBOUNDED KNAPSACK съгласно конструкцията, която разгледахме. Щом $n = 5$, $|A| = 10$, като

$$A = \{a_1^0, a_1^1, a_2^0, a_2^1, a_3^0, a_3^1, a_4^0, a_4^1, a_5^0, a_5^1\}$$

Имаме

$$v(a_1^0) = w(a_1^0) = (2^6 + 2^1) \cdot 5 \cdot 20 = 6\,600$$

$$v(a_1^1) = w(a_1^1) = (2^6 + 2^1) \cdot 5 \cdot 20 + 1 = 6\,601$$

$$v(a_2^0) = w(a_2^0) = (2^6 + 2^2) \cdot 5 \cdot 20 = 6\,800$$

$$v(a_2^1) = w(a_2^1) = (2^6 + 2^2) \cdot 5 \cdot 20 + 9 = 6\,809$$

$$v(a_3^0) = w(a_3^0) = (2^6 + 2^3) \cdot 5 \cdot 20 = 7\,200$$

$$v(a_3^1) = w(a_3^1) = (2^6 + 2^3) \cdot 5 \cdot 20 + 5 = 7\,205$$

$$v(a_4^0) = w(a_4^0) = (2^6 + 2^4) \cdot 5 \cdot 20 = 8\,000$$

$$v(a_4^1) = w(a_4^1) = (2^6 + 2^4) \cdot 5 \cdot 20 + 3 = 8\,003$$

$$v(a_5^0) = w(a_5^0) = (2^6 + 2^5) \cdot 5 \cdot 20 = 9\,600$$

$$v(a_5^1) = w(a_5^1) = (2^6 + 2^5) \cdot 5 \cdot 20 + 8 = 9\,605$$

Капацитетът и целта са

$$C = G = (6 \cdot 2^6 - 2) \cdot 5 \cdot 20 + 13 = 38\,213 \quad (16.23)$$

И наистина, това е ДА-екземпляр на UNBOUNDED KNAPSACK. Крадецът взема точно 5 предмета, по един за всяко $i \in \{1, \dots, 5\}$; предметите, които слага в раницата, са акцентирани червен или син цвят. Веднага се вижда, че сумарното тегло и сумарната цена на избраните

предмети са точно равни на капацитета на раницата и целта на крадеца:

$$6\,601 + 6\,809 + 7\,200 + 8\,003 + 9\,600 = 38\,213$$

Ясно е в какъв смисъл единичните предмети, сложени в раницата (с червените тегла и цени) отговарят на подмножество на S със сума 13. Това 13 точно отговаря на събираемостта 13 в (16.23).

Сега разглеждаме НЕ-екземпляр $\mathbf{x} = \langle \{1, 9, 5, 3, 8\}_M, K = 7 \rangle$ на SUBSET SUM. Това наистина е НЕ-екземпляр, защото никое подмножество на $\{1, 9, 5, 3, 8\}_M$ няма сума 7, което се вижда и в таблицата на стр. 622.

Да конструираме съответният екземпляр $\mathbf{y} = \langle A, v, w, C, G \rangle$ на UNBOUNDED KNAPSACK съгласно конструкцията, която разгледахме. Отново, $n = 5$, $|A| = 10$ и

$$A = \{a_1^0, a_1^1, a_2^0, a_2^1, a_3^0, a_3^1, a_4^0, a_4^1, a_5^0, a_5^1\}$$

Теглата и цените пак са

$$\begin{aligned} v(a_1^0) &= w(a_1^0) = (2^6 + 2^1) \cdot 5 \cdot 20 = 6\,600 \\ v(a_1^1) &= w(a_1^1) = (2^6 + 2^1) \cdot 5 \cdot 20 + 1 = 6\,601 \\ v(a_2^0) &= w(a_2^0) = (2^6 + 2^2) \cdot 5 \cdot 20 = 6\,800 \\ v(a_2^1) &= w(a_2^1) = (2^6 + 2^2) \cdot 5 \cdot 20 + 9 = 6\,809 \\ v(a_3^0) &= w(a_3^0) = (2^6 + 2^3) \cdot 5 \cdot 20 = 7\,200 \\ v(a_3^1) &= w(a_3^1) = (2^6 + 2^3) \cdot 5 \cdot 20 + 5 = 7\,205 \\ v(a_4^0) &= w(a_4^0) = (2^6 + 2^4) \cdot 5 \cdot 20 = 8\,000 \\ v(a_4^1) &= w(a_4^1) = (2^6 + 2^4) \cdot 5 \cdot 20 + 3 = 8\,003 \\ v(a_5^0) &= w(a_5^0) = (2^6 + 2^5) \cdot 5 \cdot 20 = 9\,600 \\ v(a_5^1) &= w(a_5^1) = (2^6 + 2^5) \cdot 5 \cdot 20 + 8 = 9\,605 \end{aligned}$$

Капацитетът и целта сега са

$$C = G = (6 \cdot 2^6 - 2) \cdot 5 \cdot 20 + 7 = 38\,207$$

Нека читателят се убеди сам(a), че нито едно подмножество на

$$\{6\,600, 6\,601, 6\,800, 6\,809, 7\,200, 7\,205, 8\,000, 8\,003, 9\,600, 9\,605\}$$

не се сумира до 38 207.

16.2.12 0-1 KNAPSACK

Въведохме задачата в Подподсекция 12.5.3.2 като оптимизационна задача. За целите на редукцията ни трябва задачата във версия за разпознаване.

Изч. Задача 90: 0-1 KNAPSACK, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Множество от предмети $A = \{a_1, \dots, a_n\}$. За всяко a_i , $1 \leq i \leq n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет на раницата $C \in \mathbb{N}^+$ и цел на крадеца $G \in \mathbb{N}^+$.

въпрос: Дали съществува подмножество $X \subseteq A$, такова че:

$$\sum_{a \in X} w(a) \leq C$$

$$\sum_{a \in X} v(a) \geq G?$$

Редукция 29: 2-PARTITION $\leq_p$ 0-1 KNAPSACK, РАЗПОЗНАВАНЕ

Конструкция: По даден екземпляр $x = \langle S, f \rangle$ на 2-PARTITION, където $S = \{s_1, s_2, \dots, s_n\}$ е множество и $f : S \rightarrow \mathbb{N}^+$ е тегловна функция върху него, конструираме екземпляр $y = \langle A, v, w, C, G \rangle$ на 0-1 KNAPSACK, където A е множество, $v : A \rightarrow \mathbb{N}^+$ е функцията на стойностите, $w : A \rightarrow \mathbb{N}^+$ е функцията на теглата, $C \in \mathbb{N}^+$ е капацитетът и $G \in \mathbb{N}^+$ е целта. А именно,

$$A \leftarrow S$$

$$w \leftarrow f$$

$$v \leftarrow f$$

$$C \leftarrow \left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor$$

$$G \leftarrow \left\lceil \frac{1}{2} \sum_{s \in S} f(s) \right\rceil$$

Конструкцията е много проста: множеството от предметите е множеството от 2-PARTITION, тегловната функция и функцията на стойностите съвпадат с тегловната функция от 2-PARTITION, а капацитетът на раницата и целта на крадеца са полусумата от теглата.

Обосновката е елементарна. Ако x е ДА-екземпляр на 2-PARTITION, то съществува $X \subset S$, такова че $\sum_{s \in X} f(s) = \frac{1}{2} \sum_{s \in S} f(s)$. Веднага се вижда, че ограниченията $\sum_{s \in X} w(s) \leq C$ и $\sum_{s \in X} v(s) \geq G$ са спазени за така дефинираните C и G . В обратната посока, ако съществува подмножество от предмети X , за което $\sum_{a \in X} w(a) \leq C$ и $\sum_{a \in X} v(a) \geq G$ са спазени, то X е единият дял на разбиване, каквото се иска в 2-PARTITION. $\square$

16.2.13 BIN PACKING

Не особено формално, задачата е някакво множество от предмети, всеки с дадено тегло, да бъдат разпределени в минимален брой кофи по такъв начин, че сумарното тегло на предметите в коя да е кофа да е не повече от някакъв предварително даден лимит. Има очевидни приложения в транспортната дейност, където различни стоки се транспортират в стандартни контейнери и има смисъл всеки контейнер да е запълнен колкото може по-добре. В тази подсекция имаме предвид задачата във версия за разпознаване.

Изч. Задача 91: BIN PACKING

екземпляр: Крайно непразно множество S . Функция $w : S \rightarrow \mathbb{N}^+$. Числа $C, k \in \mathbb{N}^+$.

въпрос: Дали съществува разбиване $\{S_1, S_2, \dots, S_k\}$ на S , такова че

$$\forall j \in \{1, \dots, k\} : \sum_{s \in S_j} w(s) \leq C ?$$

За да няма формални проблеми, под “разбиване” имаме предвид обобщено разбиване – такова, в което може да има празни дялове.

Често срещаната в литературата редукция, показваща неподатливостта на BIN PACKING, е доста тежка и технически сложна ([125, стр. 204–205]). Причината е, че BIN PACKING е **NP**-пълна в силния смисъл (вижте Секция 17.1), но за да покаже това, трябва във веригата от редукции, стартираща от SAT, да не се ползват големи числа. Какво точно означава това, ще стане ясно в Секция 17.1. Тук само ще споменем, че редукцията $3DM \leq_p 2\text{-PARTITION}$ ползва големи числа. Поради което както 2-PARTITION , така и $\text{KNAPSACK}^\dagger$, са **NP**-пълни в слабия смисъл. Ако направим редукция на KNAPSACK към BIN PACKING, което и правим в тази подсекция, само показваме, че BIN PACKING е **NP**-пълна в слабия смисъл. За целите на тази лекция това е достатъчно. Ако обаче искаме да изследваме BIN PACKING задълбочено, трябва да покажем, че тя е **NP**-пълна в силния смисъл, което я прави качествено различна, тоест по-трудна, от KNAPSACK . Силната **NP**-пълнота на BIN PACKING обезсмисля опитите за конструиране на алгоритми по схемата **Динамично Програмиране**, каквито има за KNAPSACK , както и някои видове апроксимиращи алгоритми, за нея (за BIN PACKING) – ако имаше такива алгоритми, **P** щеше да съвпада с **NP**. Но това е материал от Лекция 17.

Редукция 30: $2\text{-PARTITION} \leq_p \text{BIN PACKING}$

Конструкция: Даден е екземпляр $\mathbf{x} = \langle S, w \rangle$ на 2-PARTITION . Ще конструираме екземпляр $\mathbf{y} = \langle A, f, C, k \rangle$ на BIN PACKING, където A е множеството, f е тегловната функция, C е капацитетът на контейнер, а k е броят на дяловете на разбиването, така че $\mathbf{x}$ е **Да**-екземпляр на 2-PARTITION тстк $\mathbf{y}$ е **Да**-екземпляр на BIN PACKING.

Ето конструкцията:

$$\begin{aligned} A &\leftarrow S \\ f &\leftarrow w \\ C &\leftarrow \left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor \\ k &\leftarrow 2 \end{aligned}$$

Доказателството за коректност е лесно. В едната посока, нека $\mathbf{x}$ е **Да**-екземпляр на 2-PARTITION . Тогава $\sum_{s \in S} w(s)$ е четно число, така че $\lfloor \frac{1}{2} \sum_{s \in S} w(s) \rfloor = \frac{1}{2} \sum_{s \in S} w(s)$, така че $C = \frac{1}{2} \sum_{s \in S} w(s)$. По определението на 2-PARTITION , съществува $S' \subset S$, такова че $\sum_{s \in S'} w(s) = \sum_{s \in S \setminus S'} w(s)$. Очевидно $\sum_{s \in S'} w(s) = \sum_{s \in S \setminus S'} w(s) = C$, така че наистина предметите могат да бъдат сложени в $k = 2$ кофи без препълване.

В другата посока, нека $\mathbf{y}$ е **Да**-екземпляр на BIN PACKING. Тогава съществува разбиване $\{S_1, S_2\}$ на S , такова че $\sum_{s \in S_1} f(s) \leq \lfloor \frac{1}{2} \sum_{s \in S} f(s) \rfloor$ и $\sum_{s \in S_2} f(s) \leq \lfloor \frac{1}{2} \sum_{s \in S} f(s) \rfloor$. Тогава

[†]Доказателството за неподатливостта на KNAPSACK , което направихме, е редукция от 2-PARTITION .

$\sum_{s \in S} f(s)$ е четно, така че $\lfloor \frac{1}{2} \sum_{s \in S} f(s) \rfloor = \frac{1}{2} \sum_{s \in S} f(s) = \sum_{s \in S_1} w(s) = \sum_{s \in S \setminus S_1} w(s)$. Тогава x е ДА-екземпляр на 2-PARTITION. $\square$

16.2.14 MULTIPROCESSOR SCHEDULING

В оптимизационната си версия, задачата е следната. Дадени са n задачи. Не става дума за изчислителни задачи, каквито разглеждаме в този курс, а за *computational jobs*: конкретни изчислителни задания, които трябва да бъдат изпълнени. Те са независими по двойки, в смисъл че нито една от тях не чака резултат от изпълнение на друга. Да кажем, че множеството от задачите е $T = \{t_1, \dots, t_n\}$. Задача t_i се изпълнява за $\ell(t_i)$ единици време, за $i \in \{1, \dots, n\}$, и тези времена са известни. Дадени са и p на брой еднакви процесори, всеки от които може да изпълни всяка от задачите. Търси се такова назначаване (scheduling) на задачи върху процесори, което минимизира времето за изпълнение. А времето за изпълнение е максималното време, за което процесор обработва назначените му задачи.

Ако формализираме назначаването на задачи върху процесори c

$$f : T \rightarrow \{1, \dots, p\}$$

то цената на назначаването f е

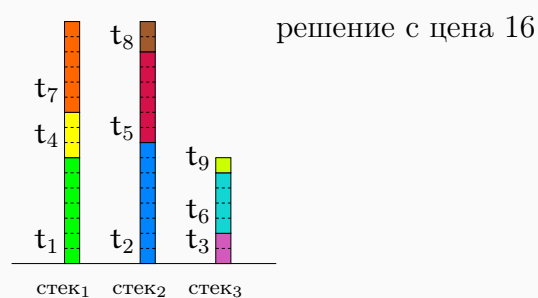
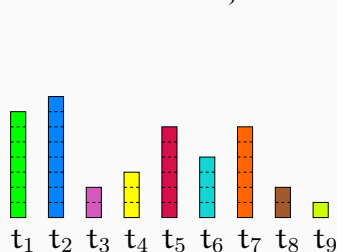
$$\max_{1 \leq i \leq p} \sum_{t_j \in T, f(t_j)=i} \ell(t_j)$$

Последното е известно под името *makespan*, което в оптимизацията означава изминалото време от началото до края на дадена дейност; в случая дейността е изпълнението на всички задачи. Поради това задачата е известна още като MINIMUM MAKESPAN SCHEDULING.

Задачата може да се онагледява визуално така. Времето за изпълнение на задача t_i се представя с бар с височина $\ell(t_i)$ и ширина единица. Процесорите са стекове, върху които се редят тези барове-задачи. Искане се такова разполагане на баровете в стековете, което минимизира максималната височина на стек. Фигура 16.42 онагледява това за $n = 9$ задачи и $p = 3$ стека. Показаното решение не е оптимално!

Фигура 16.42 : Multiprocessor scheduling като подреждане в стекове.

$$n = 9, p = 3, \sum_j \ell(t_j) = 39$$



В някакъв смисъл, MULTIPROCESSOR SCHEDULING е дуална на BIN PACKING. В BIN PACKING се искаше нещата да се разположат в контейнери с фиксиран капацитет, като се минимизира броя на контейнерите. В MULTIPROCESSOR SCHEDULING се иска нещата да се подредят във фиксиран брой контейнери, като се минимизира напълването на контейнер. Двете изчислителни задачи се получават една от друга с размяна на целевата функция и ограничението.

Трябва ни задачата MULTIPROCESSOR SCHEDULING във версия за разпознаване.

Изм. Задача 92: MULTIPROCESSOR SCHEDULING, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Крайно непразно множество S . Функция $w : S \rightarrow \mathbb{N}^+$. Числа $p, D \in \mathbb{N}^+$.

въпрос: Дали съществува разбиване $\{S_1, S_2, \dots, S_p\}$ такова че

$$\max_{1 \leq i \leq p} \sum_{s \in S_i} w(s) \leq D ?$$

“D” идва от deadline – краен срок за изпълнението на задачите.

Редукция 31: 2-PARTITION $\leq_p$ MULTIPROCESSOR SCHEDULING

Конструкция: Ще опишем конструкцията съвсем кратко. Фиксира се $p = 2$, тоест, само два процесора за обработка на задачи. Ако задачата е неподатлива за два процесора, тя е неподатлива за p процесора. На D се дава стойност $D = \lfloor \frac{1}{2} \sum_{s \in S} w(s) \rfloor$. Коректността е очевидна. $\square$

16.3 Когато принадлежността на задача към NP не е очевидна.

16.3.1 *-FREE REGEXP INEQ

Въпросът дали два регулярни изрази генерират един и същи език, тоест, дали са еквивалентни, е от голям интерес предвид широкото използване на регулярните езици. В Лекция 14 на стр. 840 видяхме, че задачата дали два регулярни изрази, ползващи и операцията “повдигане на квадрат”, са еквивалентни, е изключително трудна – тя иска експоненциална сложност по памет, което я поставя далеч-далеч отвъд NP, като това не е хипотеза, а сигурен факт. За нормалните регулярни изрази, които не ползват повдигане на квадрат, задачата е пълна за класа задачи PSPACE [145]. Това са задачите, имащи полиномиална сложност по памет. Не е известно дали PSPACE е различен от NP, въпреки че преобладаващото мнение е, че $NP \subset PSPACE$.

Да си припомним формалната дефиниция на основното понятие.

Определение 150: Регулярен израз над азбука

Дадена е азбука Σ .

1. Празното множество $\emptyset$ и празният стринг ϵ са *регулярни изрази над Σ* .
2. За всяка буква $a \in \Sigma$, a е регулярен израз над Σ .
3. Ако p и q са регулярни изрази над Σ , то pq е регулярен израз над Σ .
4. Ако p и q са регулярни изрази над Σ , то $p + q$ е регулярен израз над Σ .
5. Ако p е регулярен израз над Σ , то p^* е регулярен израз над Σ .

Нищо друго не е регулярен израз над Σ .

За да се избегне прекомерна употреба на скоби се разбираме, че звездата на Kleene има най-висок синтактичен приоритет, след това е конкатенацията, и с най-малък синтактичен приоритет е плюсьт. Използваме скоби като мета-символи само когато има нужда.

Примери за регулярни изрази над $\{a, b\}$ са

ϵ	a	b
$a + b$	$(a + b)(a + b)$	$\epsilon aab(a + a + (b + aa)bb)$
$(a^*)^*$	$(a + b)^*$	$\emptyset(a + b)^*$
$\emptyset^*$	$(a^*a^*ba^* + b^*a^*)^*$	$(a^*b^*)^*$

Определение 151: Език на регулярен израз

Нека Σ е азбука и r е регулярен израз над нея. *Езикът на r* означаваме с $L(r)$ и го дефинираме така.

- Ако $r = \emptyset$, то $L(r) = \emptyset$. Ако $r = \epsilon$, то $L(r) = \{\epsilon\}$.
- Ако $r = a$ за някое $a \in \Sigma$, то $L(r) = \{a\}$.
- Ако $r = pq$ за някои регулярни изрази над Σ p и q , то $L(r) = L(p)L(q)$. Ако $r = p + q$ за някои регулярни изрази над Σ p и q , то $L(r) = L(p) \cup L(q)$. Ако $r = p^*$ за някой регулярен израз над Σ r , то $L(r) = (L(p))^*$.

Примерно,

$$\begin{aligned}
 L(\epsilon) &= \{\epsilon\}, & L(a) &= \{a\}, & L(b) &= \{b\}, & L(a + b) &= \{a, b\} \\
 L((a + b)(a + b)) &= \{aa, ab, ba, bb\} \\
 L(\epsilon aab(a + a + (b + aa)bb)) &= \{aaba, aabbbb, aabaabb\} \\
 L((a^*)^*) &= \{\epsilon, a, aa, aaa, \dots\} \\
 L((a + b)^*) &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\} \\
 L(\emptyset(a + b)^*) &= \emptyset \\
 L(\emptyset^*) &= \{\epsilon\} \\
 L((a^*a^*ba^* + b^*a^*)^*) &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\} \\
 L((a^*b^*)^*) &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}
 \end{aligned}$$

От примера се вижда, че различни регулярни изрази може да генерират един и същи език. Освен това се вижда, че за да е безкраен езикът, трябва да участва звездата на Kleene.

В тази подсекция се ограничаваме до регулярни изрази, в които не участва звездата на Kleene; това са изразите съгласно Определение 150 без (5). Ще ги наричаме *регулярни изрази без звезди*. На английски терминът е *star-free regular expressions*. Както вече отбелязахме, всеки такъв израз генерира краен език, като никой стринг не може да е по-дълъг от израза. Имайте предвид обаче, че езикът може да е експоненциално по-голям от израза! Примерно, ако

$$r = \underbrace{(a + b)(a + b) \cdots (a + b)}_{n \text{ "множителы"}}$$

то $|L(r)| = 2^n$.

Пълното име на задачата, която е предмет на тази подсекция, е “STAR-FREE REGULAR EXPRESSION INEQUALITY”, но това е прекалено дълго, за да е практично. Някои автори я наричат “SF-REI” [118]. Именно “inequality”, защото NP-пълният вариант пита дали регулярните изрази генерират **различни** езици. Ако въпросът е дали генерират **един и същи** език, задачата е пълна за класа **coNP**, който не е предмет на тези лекции и се разглежда повърхностно в Допълнение 86.

Изн. Задача 93: $\star$ -FREE REGEXP INEQ

екземпляр: Азбука Σ и регулярни изрази без звезди r_1 и r_2 над Σ .

въпрос: Дали $L(r_1) \neq L(r_2)$?

На прост български, пита се дали има стринг в единия език, който не е в другия.

Това е първата от задачите, които разглеждаме, за която принадлежността към NP не е очевидна.

Теорема 135: $\star$ -FREE REGEXP INEQ $\in$ NP

Съществува НМТ M , такава че за всеки ДА-екземпляр x на $\star$ -FREE REGEXP INEQ, M приема x в полиномиален в $|x|$ брой стъпки.

Доказателство: За простота на доказателството ще разсъждаваме не за машина на Turing, а за алгоритъм, който обаче на първа стъпка генерира сертификат σ и след това, ползвайки сертификата, установява в полиномиално време, че x е ДА-екземпляр. Генерирането на сертификата става магически, в единица време, както подобава на недетерминирано изчисление.

По определение, $x = \langle \Sigma, r_1, r_2 \rangle$, където r_1 и r_2 са регулярни изрази без звезди над Σ . Нещо повече, можем да допуснем БОО, че r_i не съдържа нито $\emptyset$, нито ϵ , за $i = 1, 2$. Иначе казано, (1) в Определение 150 не се ползва. Да видим защо можем да допуснем това. Първо, БОО $L(r_i) \neq \emptyset$ и $\epsilon \notin L(r_i)$, за $i = 1, 2$. При това положение,

- всяка поява на $\emptyset$ като $\emptyset + p$ се редуцира до p ,
- всяка поява на $\emptyset$ като $\emptyset p$ се редуцира до $\emptyset$, но тъй като $\emptyset$ не е в езика, в крайна сметка това $\emptyset$ ще изчезне,
- всяка поява на ϵ като $\epsilon + p$ може да се редуцира до p , понеже ϵ не е в езика и в крайна сметка това ϵ ще изчезне
- всяка поява на ϵ като ϵp се редуцира до p .

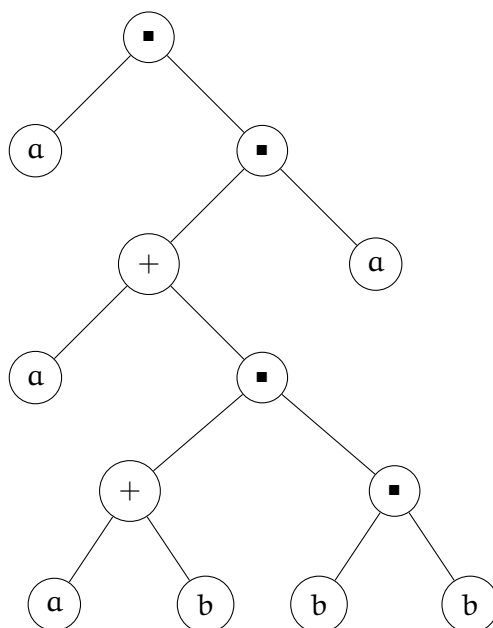
Сертификат за това, че $\langle \Sigma, r_1, r_2 \rangle$ е ДА-екземпляр, може да е стринг $\sigma \in \Sigma^*$, такъв че $\sigma \in L(r_1) \triangle L(r_2)$. Магически се отгатва такъв стринг σ и след това се проверява, че единият от r_1, r_2 може да генерира σ , а другият не може. Това обаче не е тривиално! Не е ефикасно да се генерират $L(r_1)$ и $L(r_2)$, понеже, както видяхме, всеки от тях може да е експоненциално по-голям от съответния регулярен израз.

Тези две проверки могат да се направят ефикасно по следния начин. На всеки регулярен израз без звезда (и без $\emptyset$ и ϵ) съответства биективно пълно двоично дърво, всяко листо на което е маркирано с точно една буква от азбуката, а всяко не-листо е маркирано с точно

едно от “■” и “+”, като “■” съответства на конкатенацията, а “+” е очевидно плюсът. Като пример, ето един регулярен израз без звезда над $\{a, b\}$:

$$a(a + (a + b)bb)a$$

Ето и съответното дърво:



Дървото е донякъде наредено в смисъл, че децата на връх ■ са наредени (има ляво дете и дясно дете), а децата на връх + не са наредени (няма ляво и дясно дете). На въпроса дали такова дърво T може да генерира стринг τ отговорът може да се даде така.

- Ако коренът е листо, то той е маркиран с някаква буква a и отговорът е ДА тстк $\tau = a$.
- В противен случай,
 - ◆ ако коренът е маркиран с ■, нека лявото му дете е u , а дясното е v ; отговорът е ДА тстк съществува нетривиална факторизация $\tau_1\tau_2$ на τ , такава че $T[u]$ може да генерира τ_1 , а $T[v]$ може да генерира τ_2 .
 - ◆ ако коренът е маркиран с +, отговорът е ДА тстк за поне едно от децата му, което ще наречем u , е вярно, че $T[u]$ може да генерира τ .

Тази идея може да се реализира чрез полиномиален в $|\sigma|$, $|r_1|$ и $|r_2|$ алгоритъм по схемата **Динамично Програмиране**. За всеки връх w изчисляваме множеството S_w от непразните подстрингове на σ , които може да бъдат генерирани от регулярния израз, отговарящ на $T[w]$:

- Ако w е листо, то е маркирано с някаква буква a и $S_w = \{a\}$.
- В противен случай,
 - ◆ ако w е маркиран с ■ и децата му са u и v , в този ред, то $S_w = S_u S_v$.
 - ◆ ако w е маркиран с + и децата му са u и v , то $S_w = S_u \cup S_v$.

Ако коренът е z и разполагаме със S_z , можем ефикасно да проверим дали $\sigma \in S_z$, което е вярно тстк σ е в езика на съответния регулярен израз. В крайна сметка, машината M проверява, че $\sigma \in L(r_1) \wedge \sigma \notin L(r_2)$ или $\sigma \in L(r_2) \wedge \sigma \notin L(r_1)$.

Предвид това, че непразните подстрингове на σ са $\Theta(|\sigma|^2)$ на брой и че комбинациите от връх на дърво и непразен подстринг на σ са само полиномиално много, алгоритъмът е с полиномиална сложност по време, ако се реализира грамотно. $\square$

Наблюдение 114

Идеята от доказателството на Теорема 135 **не работи** за комплементарната задача STAR-FREE REGULAR EXPRESSION EQUALITY. Ако трябва да се докаже равенство между $L(r_1)$ и $L(r_2)$, не е достатъчно да се разгледа един стринг и да се установи, че той принадлежи както на $L(r_1)$, така и на $L(r_2)$.

Редукция 32: 3SAT $\leq_p$ ★-FREE REGEXP INEQ

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

Да кажем, че $\text{Var}(\phi) = \{x_1, \dots, x_n\}$. Конструираме съответен екземпляр $\psi = \langle \Sigma, r_1, r_2 \rangle$ по следния начин.

- $\Sigma = \{0, 1\}$.
- $r_1 = \underbrace{(0+1)(0+1) \cdots (0+1)}_{n \text{ такива}}$.
- $r_2 = p_1 + p_2 + \cdots + p_m$. За $j \in \{1, \dots, m\}$,

$$p_j = q_{j,1} q_{j,2} \cdots q_{j,n}$$

За $i \in \{1, \dots, n\}$,

$$q_{j,i} = \begin{cases} 1, & \text{ако литералът } \bar{x}_i \text{ участва във } \phi_j \\ 0, & \text{ако литералът } x_i \text{ участва във } \phi_j \\ (0+1), & \text{ако променливата } x_i \text{ не участва във } \phi_j \text{ с литерал} \end{cases}$$

Първо да видим пример за прилагането на конструкцията. Добре е променливите да са повече от три, за да стане по-ясно. Нека $n = 6$ и

$$\phi = (x_1 \vee x_3 \vee x_6)(\bar{x}_2 \vee x_4 \vee x_6)(\bar{x}_3 \vee \bar{x}_5 \vee \bar{x}_6)(\bar{x}_2 \vee \bar{x}_4 \vee x_5)$$

Това е ДА-екземпляр на 3SAT. Удовлетворяваща валюация е, примерно, $t = \langle 101010 \rangle$. Конструираме $\psi = \langle \{0, 1\}, r_1, r_2 \rangle$, където

$$\begin{aligned} r_1 &= (0+1)(0+1)(0+1)(0+1)(0+1)(0+1) \\ r_2 &= 0(0+1)0(0+1)(0+1)0+ \\ &\quad (0+1)1(0+1)0(0+1)0+ \\ &\quad (0+1)(0+1)1(0+1)11+ \\ &\quad (0+1)1(0+1)10(0+1) \end{aligned}$$

$L(r_1)$ е езикът от бинарните стрингове с дължина шест. $L(r_2)$ е прекалено голям, за да го опишем подробно, но със сигурност $101010 \notin L(r_2)$. Ето защо, показано нагледно.

$$101010 \notin L(0(0+1)0(0+1)(0+1)0) \quad (16.24)$$

$$101010 \notin L((0+1)1(0+1)0(0+1)0) \quad (16.25)$$

$$101010 \notin L((0+1)(0+1)1(0+1)11) \quad (16.26)$$

$$101010 \notin L((0+1)1(0+1)10(0+1)) \quad (16.27)$$

Това, че примерът използва удовлетворяваща валюация като стринг, който не е в $L(r_2)$, не е случайно. Също така не е случайно, че

- (16.24) акцентира променливата x_1 , а клаузата $(x_1 \vee x_3 \vee x_6)$ може да бъде удовлетворена от $t = \langle 101010 \rangle$ през литерала на x_1 ,
- (16.25) акцентира променливата x_2 , а клаузата $(\overline{x_2} \vee x_4 \vee x_6)$ може да бъде удовлетворена от $t = \langle 101010 \rangle$ през литерала на x_2 ,
- (16.26) акцентира променливата x_6 , а клаузата $(\overline{x_3} \vee \overline{x_5} \vee \overline{x_6})$ може да бъде удовлетворена от $t = \langle 101010 \rangle$ през литерала на x_6 ,
- (16.27) акцентира променливата x_2 , а клаузата $(\overline{x_2} \vee \overline{x_4} \vee x_5)$ може да бъде удовлетворена от $t = \langle 101010 \rangle$ през литерала на x_2 .

Коректност на конструкцията. Първо да допуснем, че ϕ е удовлетворима. Нека $t \in \text{Val}(\text{Var}(\phi))$ удовлетворява ϕ . Ще покажем, че $L(r_1) \neq L(r_2)$. Тъй като и двата езика се състоят от стрингове с дължина n и освен това $L(r_1)$ е езикът от всички стрингове с дължина n , достатъчно е да покажем един стринг с дължина n , който не е в $L(r_2)$. Нека t означава t като стринг с дължина n (в примера горе имаме $t = 101010$). Ще покажем, че $t \notin L(r_2)$. Ще покажем, че $\forall j \in \{1, \dots, m\} : t \notin L(p_j)$. Щом t е удовлетворяваща валюация, клаузата ϕ_j съдържа поне един литерал, да го наречем α_j , такъв че $\text{TA}(\alpha_j, t) = 1$. Този литерал е или положителен, или отрицателен. Ако α_j е положителен литерал x_i , то от една страна, $t(x_i) = 1$, но от друга страна, $q_{j,i} = 0$. Ако α_j е отрицателен литерал $\overline{x_i}$, то от една страна, $t(x_i) = 0$, но от друга страна, $q_{j,i} = 1$. Следователно, за всеки стринг $y \in L(p_j)$ е вярно, че y и t се различават в i -тата позиция. И това е за всяко j . Следователно, t не може да бъде генериран от r_2 .

Сега да допуснем, че $\exists z \in L(r_1) \setminus L(r_2)$. Това означава, че $\forall j \in \{1, \dots, m\} : z \notin L(p_j)$. Подробно написано

$$\forall j \in \{1, \dots, m\} \exists i \in \{1, \dots, n\} : (z_i = 0 \wedge q_{j,i} = 1) \vee (z_i = 1 \wedge q_{j,i} = 0)$$

Но тогава $z \models \phi$. □

Допълнение 86: Задачи-допълнения и клас на сложност coNP

Определение 152: Задачата-допълнение на задача за разпознаване

Нека P е задача за разпознаване и J е множеството от нейните екземпляри. *Задачата-допълнение на P* , която бележим с $\bar{P}$, също има за множество от екземпляри J , но ДА-екземплярите и НЕ-екземплярите са разменени. Тоест, $\bar{P}_Y = P_N$ и $\bar{P}_N = P_Y$.

На прост български, $\bar{P}$ се получава от P с отрицание върху съждението във въпроса.

В Лекция 1 като примери за задачи за разпознаване видяхме СВЪРЗАНОСТ НА ГРАФИ (Задача 1) и HAMILTONIAN CYCLE (Задача 2). Тук ще въведем техните задачи-допълнения. Има една особеност обаче. В СВЪРЗАНОСТ НА ГРАФИ и HAMILTONIAN CYCLE въпросите бяха съответно “Дали G е свързан?” и “Дали има Хамилтонов цикъл в G ?”. Ако директно негираме тези въпроси, ще получим съответно “Дали G не е свързан?” и “Дали няма Хамилтонов цикъл в G ?”, а на разговорен български смисълът на тези въпроси не е този, който ни трябва. Поради това ще дефинираме задачите-допълнения по недвусмислен начин така.

Изч. Задача 94: СВЪРЗАНОСТ НА ГРАФИ

екземпляр: Неориентиран граф G .

въпрос: Дали G е несвързан?

ДА-екземплярите са графите, които не са свързани, а НЕ-екземплярите графите, които са свързани.

Изч. Задача 95: HAMILTONIAN CYCLE

екземпляр: Неориентиран граф G .

въпрос: Дали G е не-Хамилтонов?

ДА-екземплярите са графите, които нямат Хамилтонов цикъл, а НЕ-екземплярите графите, които имат Хамилтонов цикъл.

Читателят може да се изкуши да нарече тези задачи съответно НЕСВЪРЗАНОСТ НА ГРАФИ и НЕ-ХАМИЛТОНОВОСТ, но нотацията с черта отгоре е за предпочитане.

На този етап от изложението може би не е ясно защо изобщо въвеждаме задачи-допълнения. В крайна сметка, ние се интересуваме от алгоритмите, решаващи задачи, а ако имаме алгоритъм, решаващ задача за разпознаване P , то може да модифицираме този алгоритъм тривиално, разменяйки `return TRUE` и `return FALSE`, получавайки по този начин алгоритъм за $\bar{P}$. При това коректността е очевидна, а сложността по време остава същата.

Ако мислим не за алгоритми, а за машини на Turing, нещата са аналогични. Нека M е МТ и нека M решава задача за разпознаване P в смисъла на Определение 123. Тогава можем много лесно да конструираме МТ $\bar{M}$, която решава $\bar{P}$: просто разменяме q_Y и q_N в таблицата на преходите на M и получаваме (таблицата на преходите на) $\bar{M}$.

Задачите-допълнения са смислени в контекста на недетерминизма, и то ако отчитаме ефикасността. Нека M е полиномиална НМТ, която решава HAMILTONIAN CYCLE. Както знаем, можем да мислим за M по два начина.

- М е машина с размножаване. М приема или отхвърля (кодирания на) графи по начин, близък до описанието на стр. 861: за даден конкретен граф тя обхожда с нещо като BFS—включващо обаче размножаване—едно дърво и приема чрез q_Y тстк поне едно копие открие Хамилтонов цикъл.

Ключовото наблюдение, че от М **не можем да получим** машина за HAMILTONIAN CYCLE чрез проста размяна на q_Y и q_N по следните две причини.

1. Приемащото състояние и отхвърлящото състояние участват по различен начин в М: тя приема, ако поне едно копие достигне q_Y , и отхвърля, ако всяко копие достигне q_N (или катастрофира). Ако разменим q_Y и q_N , ще получим машина, която приема (кодирание на) граф, в който поне един цикъл не е Хамилтонов.
2. Задачата HAMILTONIAN CYCLE е дефинирана чрез екзистенциален квантор, макар това да не е експлицитно в Задача 2. В дефиницията на HAMILTONIAN CYCLE присъства имплицитно

$$\exists c \in \mathcal{C}(G) : H(c)$$

където $\mathcal{C}(G)$ е множеството от циклите в G , а $H(c)$ е предикатът “с е Хамилтонов”.

Ерго, задачата HAMILTONIAN CYCLE е дефинирана чрез универсален квантор, понеже нейният въпрос е негация на въпроса на HAMILTONIAN CYCLE. И така, в дефиницията на HAMILTONIAN CYCLE присъства имплицитно

$$\forall c \in \mathcal{C}(G) : \neg H(c)$$

Ясно се вижда, че задачата не е дали поне един цикъл е не-Хамилтонов, а дали всеки цикъл е не-Хамилтонов.

- М може да е със сертификат и работи, както е казано на стр. 863: сертификатът се генерира магически за единица време, а в детерминираната фаза М проверява, че сертификатът е пермутация (всеки връх се появява точно веднъж) и че необходимите (за да бъде тази пермутация Хамилтонов цикъл) ребра са налице. Проверката е ефикасна, защото сертификатът е къс: той се проверява в полиномиално време и това поставя HAMILTONIAN CYCLE в класа на сложност NP.

Ключовото наблюдение е, че от машината М **не изглежда да има начин да получим** машина за HAMILTONIAN CYCLE чрез генериране на къс сертификат. Не е доказано, че няма къс сертификат за не-Хамилтоновост, но никой досега не е намерил сертификат със само полиномиална дължина (за това, че графът няма Хамилтонов цикъл).

Най-естественият избор за сертификат за НМТ за задачата-допълнение е множеството от всички пермутации на върховете, може би с точност до ротация и рефлексия (за да са само $\frac{(n-1)!}{2}$). За всяка от тях, машината-верификатор във втората фаза от работата си проверява, че тази пермутацията не реализира Хамилтонов цикъл. По очевидни причини такъв сертификат не би бил къс и полиномиална НМТ няма да има време дори да го прочете целия, камо ли да провери нещо чрез него.

Разбира се, това не доказва, че към сертификат за $\overline{\text{HAMILTONIAN CYCLE}}$ няма, но опитите на компютърните учени в продължение на десетилетия да открият към сертификат за задача-допълнение на **NP**-пълна задача не са довели до успех. Прочее, както предстои да видим, откриването на такъв към сертификат би означавало, че $\mathbf{P} = \mathbf{NP}$.

Дано от този пример става ясно, че в общия случай недетерминирани машини на Turing за задачите-допълнения нямат нищо общо с недетерминирани машини на Turing за оригиналните задачи. Поради това има смисъл да се разглеждат задачи-допълнения и класове на сложност, базирани на тях.

Често срещана (примерно, [6, стр. 55, Definition 2.19]) дефиниция на “**coNP**” е следната.

Определение 153: Клас на сложност **coNP**

Класът на сложност **NP**, по отношение на дадена азбука Σ , е

$$\mathbf{coNP} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \bar{L} \in \mathbf{NP}\}$$

В това обаче има малък формален проблем: както знаем (Наблюдение 97), езикът-допълнение $\bar{L}$ включва не само кодиранятия на НЕ-екземпляри, а и стринговете, които са невалидни кодирания. Papadimitriou адресира този проблем [125, стр. 219, бележката под черта] така:

Recall that we say two languages are complements to each other if they are disjoint and their union is, not necessarily the set of all strings, but some other, trivial to recognize set; in this case, the set of all strings that are legitimate encodings of graphs.

Papadimitriou говори в контекста на задачата **HAMILTONIAN PATH**.

Papadimitriou казва [125, стр. 219], че задачите от **coNP** притежават към *дисквалификатори*. Дисквалификатор е обратното на сертификат: докато сертификатът помага, за да се убедим, че даденият екземпляр е ДА-екземпляр, то дисквалификаторът помага, за да се убедим, че даденият екземпляр е НЕ-екземпляр. Примерно, по отношение на **HAMILTONIAN PATH**, дисквалификатор е (кодиране на) пермутация на върховете, която представлява Хамилтонов цикъл. Ерго, това, което е сертификат за дадена задача, е дисквалификатор за нейната задача-допълнение.

Имайте предвид, че **NP** и **coNP** не са езици, а безкрайни множества от езици – ако разсъждаваме на ниво езици. Въпреки че Определение 153 говори за допълнение (на език), класовете **NP** и **coNP** не са взаимно комплементарни! Най-малкото, те имат непазено сечение: доста очевидно е, че $\mathbf{P} \subseteq \mathbf{coNP}$, понеже можем да мислим за езиците от **P** като за езици от **coNP** с празни дисквалификатори. Сравнете със Следствие 38.

Класът **coNP** съдържа пълни спрямо полиномиални редукции задачи, също като **NP**. Сравнете Определение 154 с Определение 139.

Определение 154: Клас на сложност coNP-с

Нека P е задача за разпознаване. Казваме, че P е *coNP-пълна*, ако

1. $P \in \text{coNP}$ и
2. за всяка задача $P' \in \text{coNP}$ е вярно, че $P' \leq_p P$.

Класът на сложност **coNP-с** се състои точно от **coNP-пълните** задачи.

На английски терминът е *coNP-complete*.

Да си припомним, че булева формула ϕ е тавтология тстк $\forall t \in \text{Val}(\text{Var}(\phi)) : t \models \phi$.

Изч. Задача 96: TAUTOLOGY

екземпляр: Булева формула ϕ .

въпрос: Дали ϕ е тавтология?

Каноничната **coNP-пълна** задача е TAUTOLOGY, аналогично на това, че каноничната **NP-пълна** задача е SATISFIABILITY. Доказателството ще направим с много по-малко усилия от доказателството на Теорема 126, понеже ще ползваме Теорема 126.

Теорема 136: TAUTOLOGY $\in$ coNP-с

TAUTOLOGY е **coNP-пълна**.

Доказателство: Първо ще покажем, че $\text{TAUTOLOGY} \in \text{coNP}$. Но това е почти очевидно, понеже $\overline{\text{TAUTOLOGY}} \in \text{NP}$: задачата $\overline{\text{TAUTOLOGY}}$ пита дали булева формула ϕ има поне една фалшифицираща валуация t , а НМТ отгатва t и после проверява в полиномиално време, че $t \not\models \phi$. Иначе казано, екземпляр на TAUTOLOGY може да бъде дисквалифициран бързо, ако някой даде подходящ дисквалификатор.

Сега ще покажем, че за $\forall P \in \text{coNP} : P \leq_p \text{TAUTOLOGY}$. Щом $P \in \text{coNP}$, вярно е, че $\overline{P} \in \text{NP}$. Щом $\overline{P} \in \text{NP}$, от Теорема 126 следва, че $\overline{P} \leq_p \text{SAT}$. Карп-редукцията на $\overline{P}$ към SAT е функция, която изобразява всеки екземпляр x на $\overline{P}$ в булева формула $\phi(x)$, която е удовлетворима тстк $x \in \overline{P}$.

Но $x \in \overline{P}$ тстк $x \in P_N$ съгласно Определение 152. Следва, че $\phi(x)$ е удовлетворима тстк $x \in P_N$. Но тогава $\phi(x)$ е противоречие тстк $x \in P_Y$. Прилагаме Наблюдение 102 и заключаваме, че $\neg\phi(x)$ е тавтология тстк $x \in P_Y$. Забележете, че $\neg\phi(x)$ е екземпляр на TAUTOLOGY, понеже е булева формула.

И това е краят на доказателството. Покажахме, функция, която изобразява произволен екземпляр x на P в екземпляр на TAUTOLOGY, и то така, че x е Да-екземпляр тстк изображението е тавтология. $\square$

TAUTOLOGY не е единствената **coNP-пълна** задача, чиято задача-допълнение е **NP-пълна**. За всяка **NP-пълна** задача P е вярно, че $\overline{P}$ е **coNP-пълна** [125, стр. 220, Proposition 10.1].

Както знаем от Теорема 126, удовлетворимостта на КНФ е неподатлива задача. Дуално, фалшифицируемостта на ДНФ е **NP-пълна** задача. От друга страна, удовлетворимостта на ДНФ, също както и фалшифицируемостта на КНФ, са тривиални: ДНФ е удовлетворима винаги и КНФ е фалшифицируема винаги. Очевидно е, че КНФ не

може да бъде тавтология и ДНФ не може да бъде противоречие. **coNP**-пълнотата ни дава възможност да кажем как стои въпросът с податливостта на задачата, дали КНФ е противоречие:

1. щом задачата дали КНФ ϕ е удовлетворима е в **NP-с**,
2. то задачата-допълнение дали КНФ ϕ е противоречие е в **NP-с**.

Но тогава заради дуалността е вярно, че задачата дали ДНФ ψ е тавтология също е в **NP-с**. Таблица 16.1 резюмира тези резултати.

Въпрос за ϕ Вид на ϕ	ϕ е удовлетворима?	ϕ е фалшифицируема?	ϕ е тавтология?	ϕ е противоречие?
ϕ е КНФ	NP-с	винаги	никога	coNP-с
ϕ е ДНФ	винаги	NP-с	coNP-с	никога

Таблица 16.1: Сложност на базови задачи върху КНФ/ДНФ.

Интересен въпрос е **NP** $\stackrel{?}{=}$ **coNP**. За съжаление, или за щастие, да се отговори на него е поне толкова трудно, колкото да се отговори на **P** $\stackrel{?}{=}$ **NP**.

Теорема 137: **P = NP** влече **NP = coNP**

Ако се окаже, че **P = NP**, ще следва, че **NP = coNP**.

Доказателство: **P** е затворен спрямо допълнение; тоест, **P = coP**, където

$$\text{coP} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \bar{L} \in \text{P}\}$$

Причината бе спомената в началото на това допълнение: всеки детерминиран алгоритъм за задача за разпознаване може тривиално да се преработи за задачата-допълнение, оставайки със същата сложност.

Ако **P = NP** и при положение, че **P = coP**, веднага следва **NP = coNP**. $\square$

Конверсното на Теорема 137 обаче не е вярно. Възможно е **P $\neq$ NP** и въпреки това **NP = coNP**.

Това е Proposition 10.2 в [125, стр. 220].

Теорема 138: **P $\in$ coNP-с $\wedge$ P $\in$ NP** влече **NP = coNP**

Ако се окаже, че **coNP**-пълна задача е в **NP**, ще следва, че **NP = coNP**.

Доказателство: Ще покажем, че ако задача **P** е в **NP** е **coNP**-пълна, то **coNP $\subseteq$ NP**. В обратната посока доказателството става симетрично.

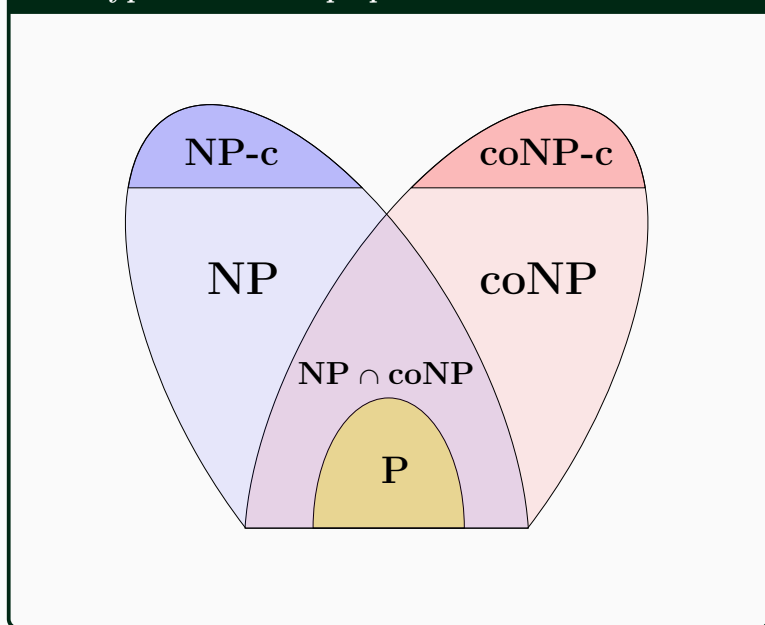
Щом **P $\in$ NP**, съществува НМТ **M**, която приема (недетерминирано) екземплярите на **P**.

Нека **P'** е произволна задача от **coNP**. Щом **P** е **coNP**-пълна, **P' $\leq_p$ P** с полиномиална редукция **f**. Нека **x** е екземпляр на **P'**. Тогава **f(x)** е екземпляр на **P**, построим от **x** в

полиномиално време. Тогава M приема $f(x)$ в полиномиално в $|x|$ време. Тогава P' е в **NP**. $\square$

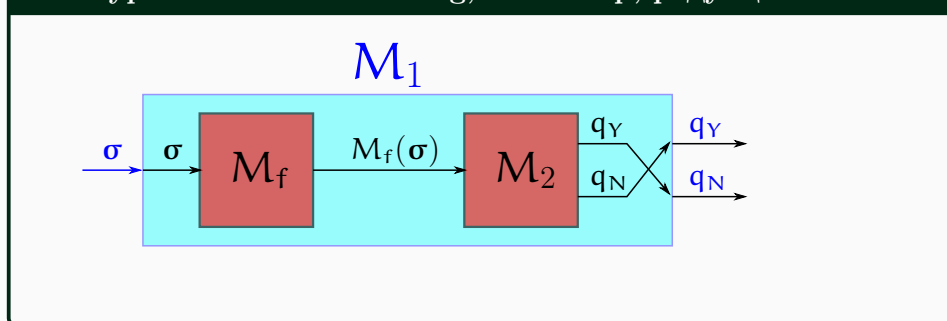
Широко разпространеното вярване сред специалистите е, че **NP** $\neq$ **coNP**, понеже десетилетни усилия да се намери към сертификат за задачи като TAUTOLOGY са се оказали безуспешни. Фигура 16.43 илюстрира сегашната широко приета представа за географията на **NP** и **coNP**, класовете от пълните за тях задачи, и **P**, който се намира в сечението им.

Фигура 15.5 : Географията на **NP** и **coNP**.



В края на допълнението си заслужава да се посочи нещо важно. Разлика между **NP** и **coNP** има само по отношение на Карп редукциите (Определение 134). Turing редукциите (Определение 138) са прекалено мощни и не различават тези два класа на сложност. За да се убедим в това, да си представим редукция, която прилича на Карп редукция (Фигура 14.9), но не е. В нея q_Y и q_N от M_2 са разменени в M_1 , в смисъл, че q_Y на M_2 дава q_N на M_1 и q_N на M_2 дава q_Y на M_1 . Фигура 16.44 онагледява ситуацията.

Фигура 14.9 : Това е Turing, а не Карп, редукция.



Това е Turing редукция, макар и съвсем проста. Истинската мощ на Turing редукциите е илюстрирана на Фигура 14.10: машината M' , реализираща редукцията, вика маши-

на/процедура M_2 многократно (макар и не повече от полиномиален брой пъти) и прави още изчисления (в които вероятно ползва резултати от тези викания), преди да даде окончателния отговор. Редукцията, показана на Фигура 16.44, е именно Turing редукция, въпреки че прави само едно обръщане към M_2 . Причината да е Turing редукция, а не Карп редукция е, че разменя q_Y с q_N в окончателния отговор. Макар и просто действие, то не е разрешено при Карп редукциите.

Извършвайки това действие, M_1 решава задачата-допълнение. Ако M_2 е полиномиална машина, то M_1 ще е полиномиална машина за задачата-допълнение. Ерго, при използване на Turing редукции, NP и coNP са неразличими.

16.3.2 ART GALLERY

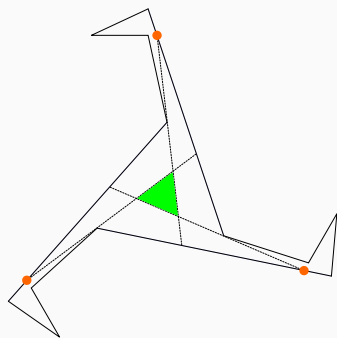
Това е задача от изчислителната геометрия с приложение в области като роботика, sensor networks и surveillance [147]. Има книга, посветена на нейните варианти [122], в предговора на която се разказва историята на задачата. През 1973 г. известният математик Václav Chvátal се обръща към Victor Klee с молба за интересна нерешена задача от геометрията. Klee дава следната задача: какъв е минималният брой охранители, необходим и достатъчен за охраната на изложбена зала с дадена геометрия. Приема се, че изложбената зала е прост многоъгълник—не се самопресича и няма дупки—но не непременно изпъкнал[†]. За да охраняват залата, трябва всяка нейна точка да е видима за поне един охранител. Приема се, че охранителите имат 360-градусово зрение, поради което, ако многоъгълникът е изпъкнал, един охранител е достатъчен — където и да застане, той или тя вижда цялата зала. Ако обаче многоъгълникът не е изпъкнал, в общия случай един охранител не стига; където и да застане, има части от залата, които не са видими за него или нея.

Задачата е в различни варианти. Може да се иска охранителите да виждат само стените на залата (многоъгълника), защото там са картините, които се охраняват. Може да се иска охранителите да виждат и вътрешността на залата (многоъгълника). От друга страна, може да е разрешено охранителите да са само във върховете на многоъгълника, може да е разрешено да са във върховете или околните ръбове[‡], а може да е разрешено да са и във вътрешността, тоест, навсякъде. Тези разлики в дефиницията на задачата имат значение за оптималното решение, както е показано на Фигура 16.45. Вляво виждаме пример за това, че минималният брой охранители за целия многоъгълник, което включва и вътрешността, може да е по-голям от минималния брой охранители за периферията: ако това беше зала, трима охранители стигат (и са необходими), за да виждат стените, но оцветеният триъгълник във вътрешността би бил невидим за тях. Вдясно виждаме пример за това, че минималният брой на охранителите може да нарастне, ако не им позволяваме да стъпват във вътрешността: за да може един охранител да вижда и двата “рога”, той или тя трябва да е в оцветения четириъгълник. Фигурата вляво е от [131]. “Кравешката глава” вдясно е фолклор.

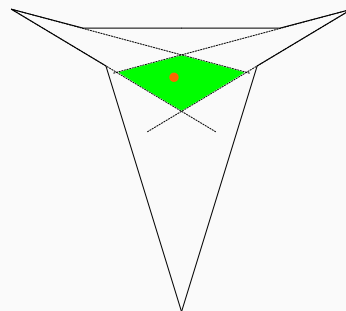
[†]По скромните наблюдения на автора, изложбените зали в реалния свят са *ортогонални многоъгълници*—такива, на които всяка стена е в едно от две ортогонални направления—но може да имат дупки, понеже може да има “хвърчащи стени” наред с залата, които не са свързани със стените, ограждащи залата.

[‡]В изложбените зали в реалния свят по правило охранителите се намират в ъглите или поне близо до стените.

Фигура 16.45 : Вариантите на ART GALLERY имат значение за оптимума.



Дали охранителите да гледат само стените или и вътрешността: трима охранители виждат всички стени, но не и цялата вътрешност.



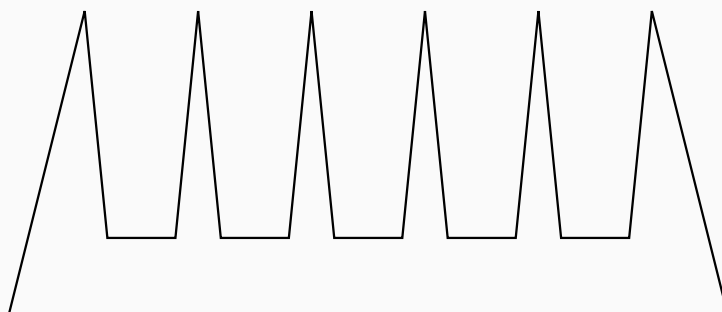
Дали охранителите да са до стените или може да са и във вътрешността: един охранител вижда всичко само ако е в оцветената област.

Chvátal разглежда задача във варианта, в който охранителите могат да са както във вътрешността, така и върху периферията (самият многоъгълник) и охраняват както вътрешността, така и периферията. По думите му [30]:

Let S be a subset of the Euclidean plane. We shall say that a subset A of S *dominates* S if for each $x \in S$ there is a $y \in A$ such that the entire segment xy lies within S . In a conversation, Professor Vicror Klee asked for the smallest number $f(n)$ such that every set bounded by a simple closed n -gon is dominated by a set of $f(n)$ points.

Chvátal показва [30], че точната долна граница за броя на охранителите на n -ъгълник е $\lfloor \frac{n}{3} \rfloor$. В едната посока, многоъгълник като “гребенът” на Фигура 16.46 иска поне толкова охранители, колкото са зъбите. Примерно, осемнадесетоъгълникът на фигурата не може да бъде охраняван от по-малко от шест охранители. Очевидно е как тази идея може да се обобщи за n -ъгълник-гребен, като за всеки зъб на гребена трябва да има поне един охранител, който да вижда вътрешността на зъба.

Фигура 16.46 : n -ъгълник, чието охраняване иска $\lfloor \frac{n}{3} \rfloor$ охранители.



В обратната посока, Chvátal показва, че $\lfloor \frac{n}{3} \rfloor$ охранители са достатъчни за произволен прост n -ъгълник, използвайки интересен аргумент по индукция по n . Заслужава да се отбележи,

че доказателството в тази посока може да бъде изключително кратко: през 1978 г. Steve Fisk публикува статия от една единствена страница [48], в която доказва на няколко реда, че $\lfloor \frac{n}{3} \rfloor$ охранители стигат. Неговото доказателство използва триангулация на n -ъгълника. Това, че триангулация винаги съществува, дори n -ъгълникът да не е изпъкнал, е азбучна истина сред хората, занимаващи се с изчислителна геометрия. Триангулация е най-фундаменталната декомпозиция на многоъгълник, чието значение е сходно с декомпозицията на прости множители в теорията на числата. Доказателство, че триангулация винаги съществува има, примерно, в споменатата книга на O'Rourke [122, стр. 12]. В [122, стр. 13] е доказано, че всяка триангулация може да бъде 3-оцветена; оцветяване на триангулация е даване на цветове на върховете по такъв начин, че краищата на никоя нейна отсечка (както диагонал, така и от периферията) да не са в един и същи цвят. Този факт също е азбучна истина в изчислителната геометрия. Ползвайки неявно тези два факта—че триангулация винаги съществува и че винаги е 3-оцветима—Fisk разглежда произволна триангулация на n -ъгълника и произволно нейно оцветяване в три цвята a , b и c . Той отбелязва два очевидни факта:

1. върховете от кой да е цвят виждат целия многоъгълник, защото всеки триъгълник има и трите цвята върху върховете си, а всеки връх на триъгълник вижда целия триъгълник, понеже триъгълникът е изпъкнала фигура;
2. съществува цвят, такъв че върховете, оцветени в него, са не повече от $\lfloor \frac{n}{3} \rfloor$.

Желаният извод следва веднага.

Допълнение 87: Point Visibility Graphs

Терминът “dominates” в цитата на Chvátal на предишната страница не е случаен. Сега ще видим, че нашата дефиниция на “доминиране” (Задача 55) е дискретният аналог на това, което Chvátal нарича “доминиране” по отношение на многоъгълниците.

Задачата ART GALLERY има дискретен вариант, в който охранителите са само върху върховете на многоъгълника, но и пазят само върховете – а не пазят вътрешността и дори не пазят точките от страните, които не са върхове. Тогава задачата се превръща именно в DOMINATING SET, но върху един граф, който се получава от многоъгълника. Върховете на графа са върховете на многоъгълника, а за всеки два върха x и y , в графа има ребро (x, y) тогава и само тогава, когато x и y се виждат в многоъгълника. Виждането е симетрична релация— x вижда y тук y вижда x —поради което графът е неориентиран. x и y да се виждат означава отсечката xy да лежи в многоъгълника. Дали трябва да е непременно във вътрешността или е разрешено да има общи точки с периферията е въпрос на дефиниция. Сякаш по-често използваната дефиниция разрешава общи точки с периферията. Важното е xy да не “излиза” извън многоъгълника. Така полученият граф се нарича Point Visibility Graph (PVG) на многоъгълника. PVG имат приложение в области като роботиката, разпознаване на обекти и дори в диагностицирането на болестта Алцхаймер [1].

Очевидно е, че ако многоъгълникът е изпъкнал и прост, въпросният граф е K_n . Не всеки граф е PVG на многоъгълник. Най-малкото, PVG винаги е Хамилтонов – периферията на многоъгълника отговаря на Хамилтонов цикъл в PVG. При даден многоъгълник, конструирането на неговия PVG е сравнително лесна задача, решима във време $O(n \lg n + m)$ и памет $O(n)$ [57]. Изненадващо, не е известен алгоритъм, който да решава дали даден граф е PVG; тоест, дали съществува многоъгълник, чийто PVG е дадения граф. Поне през 2009 г. това е било така [57]. Без съмнение, това се дължи на континуалната природа на многоъгълника.

Редица задачи като **INDEPENDENT SET**, **VERTEX COVER** и **DOMINATING SET** остават **NP-пълни** върху **PVG** [108].

Оттук насетне ще смятаме, че охранителите могат да стоят само във върховете на многоъгълника и че охраняват както страните, така и вътрешността на многоъгълника.

Изч. Задача 97: ART GALLERY

екземпляр: Прост, не непременно изпъкнал, многоъгълник P и число k .

въпрос: Дали съществуват k върха в P , такива че всяка точка от P се вижда от поне един от тях?

Принадлежността на задачата към **NP** не е очевидна и трябва да се аргументира. Естественият избор за сертификат е множество от k върха, които виждат целия многоъгълник, но не е очевидно как да се провери бързо, че наистина всяка точка на многоъгълника се вижда от поне един от тях? Точките на многоъгълника са неизброимо безкрайно много. Проблемът е в това, че многоъгълникът по природа не е дискретен обект, дори и върховете да са с целочислени или рационални координати.

Теорема 139: ART GALLERY $\in$ NP

При даден екземпляр x на **ART GALLERY** и сертификат-множество от k върха за x , в полиномиално време може да се установи, че x е **ДА-екземпляр**.

Скица на доказателство: Идеята е от [47, стр. 16]. След като някак сме отгатнали въпросните k върха, във време $O(n \lg n)$ изчисляваме така наречения *visibility polygon* за всеки от тях—името много точно казва за какво става дума—и пресмятаме обединението на тези *visibility polygons* във време $O(n^2 k^2)$. След това сравняваме получения многоъгълник-обединение с многоъгълника от x и отговорът е **ДА** тстк те съвпадат. Как се изчислява *visibility polygon* на даден връх е описано в [70]. $\square$

Оригиналната редукция от **NP-пълна** задача към **ART GALLERY** е от **3SAT** [104]. Тази редукция е разказана и в [122, стр. 239–242]. Известна е обаче и много по-лесно разбираема редукция, която е от **VERTEX COVER** Сега ще видим нея.

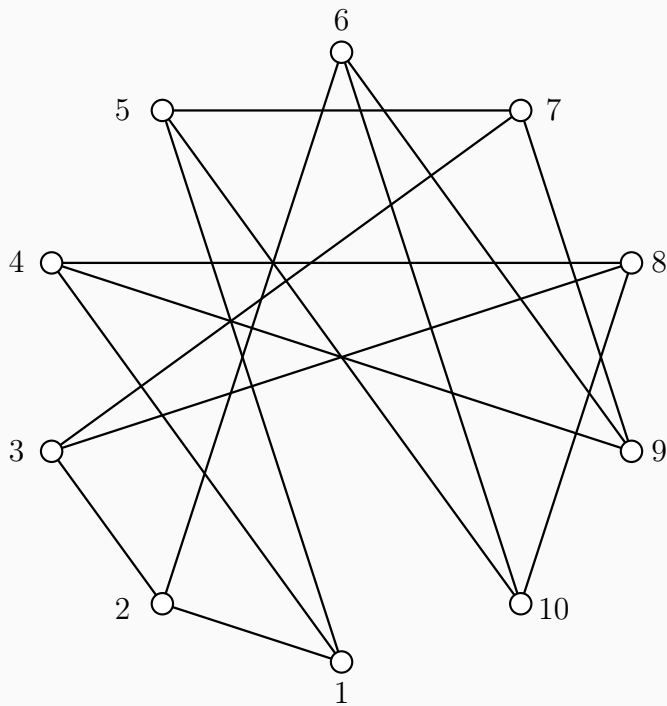
Редукция 33: VERTEX COVER $\leq_p$ ART GALLERY

Конструкция: Тази конструкция е описана, например, в [118, стр. 247–249]. Тук ще направим неин опростен, много по-нагледен и убедителен дори на пръв поглед вариант. Тъй като изложението далече не е формално прецизно, ще разгледаме пример за прилагане на конструкцията. От този пример трябва да стане кристално ясно каква е идеята на конструкцията.

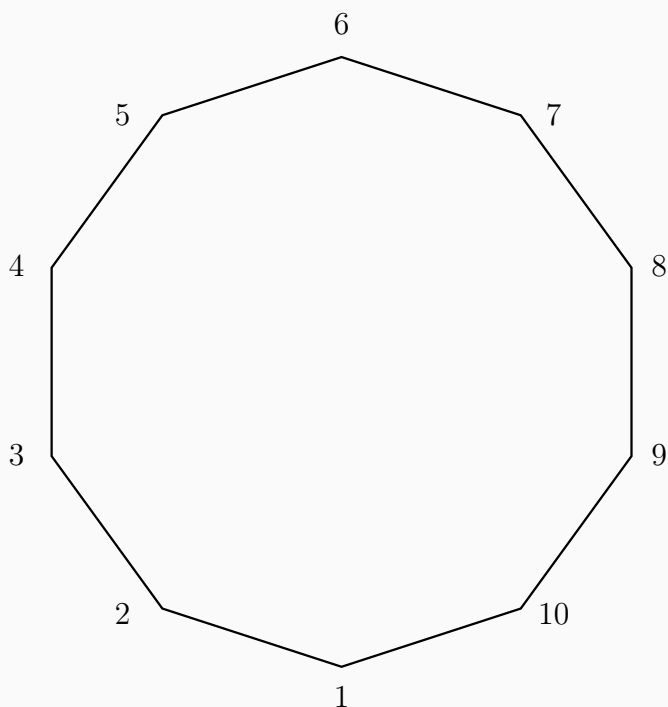
Даден е екземпляр $\langle G, k \rangle$ на **VERTEX COVER**. Ще построим екземпляр $\langle P, k \rangle$ на **ART GALLERY**, такъв че $\langle G, k \rangle$ е **ДА-екземпляр** на **VERTEX COVER** тстк $\langle P, k \rangle$ е **ДА-екземпляр** на **ART GALLERY**. Като пример в изложението ще ползваме графа на Petersen. И така, G е графът на Petersen, а $k = 6$. Както знаем от изложението на стр. 704, минималното върхово покриване на графа на Petersen има размер 6, така че това е **ДА-екземпляр** на **VERTEX COVER** и очакваме конструкцията да генерира **ДА-екземпляр** на **ART GALLERY**.

Каноничната рисунка на графа на Petersen в учебниците е тази, която е показана на Фигура 12.2. Тук ще нарисуваме графа на Petersen по друг начин: върховете са върхове на правилен десетоъгълник, именувани с $1, \dots, 10$, а ребрата са петнадесет отсечки между тях. Фигура 16.47 показва това.

Фигура 16.47 : VERTEX COVER $\leq_p$ ART GALLERY: графът на Petersen.



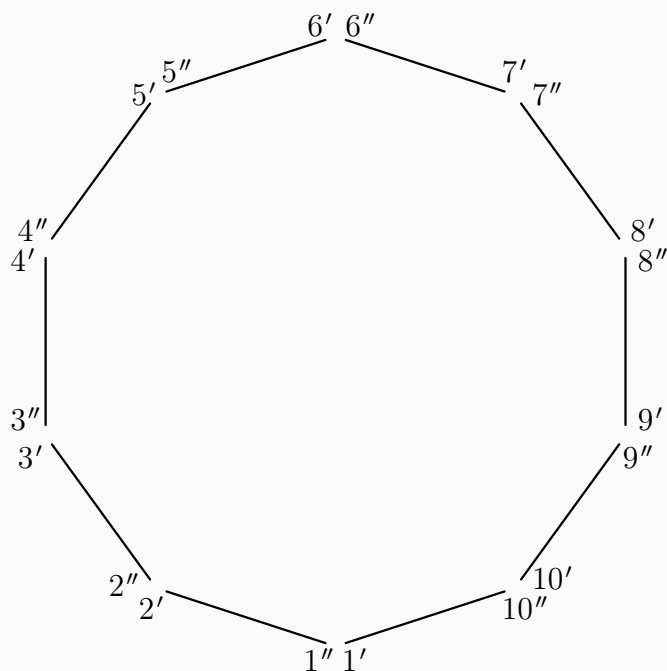
Конструкцията започва с правилен десетоъгълник, чиито върхове са именувани с $1, \dots, 10$. Вижте Фигура 16.48.

Фигура 16.48 : VERTEX COVER $\leq_p$ ART GALLERY: десетоъгълникът.

Махаме десет малки парчета от десетоъгълника: около всеки връх построяваме окръжност с радиус ϵ , за някакво достатъчно малко ϵ , и изтриваме сечението на десетоъгълника с вътрешността на окръжността. С други думи, изтриваме тези части от околните страни, които са на разстояние $< \epsilon$ от върховете.

По този начин получаваме десет отсечки с празни сечения по двойки, всяка от които е “остатъкът” от една от страните на бившия десетоъгълник. На всеки връх i от бившия десетоъгълник съответсват два от краищата на тези отсечки, които именуваме i' и i'' . Нека i' се появява преди i'' при обикаляне по часовниковата стрелка. Вижте Фигура 16.49.

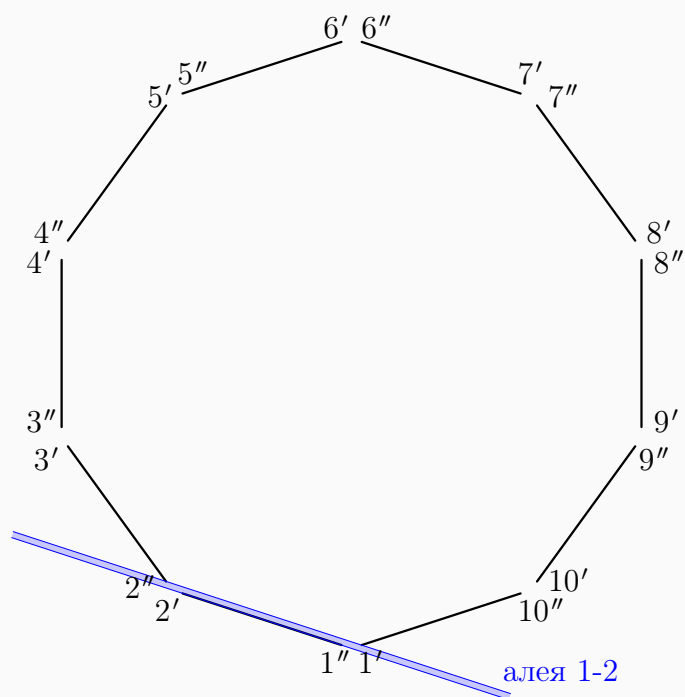
Точките $1', 1'', 2', 2'', \dots, 10', 10''$ наричаме *особените точки*. Между всяка $1'$ и $1''$ се намира *отворът* i .

Фигура 16.49 : Изрязваме ϵ около върховете и получаваме отвори.

Разглеждаме произволен връх от графа. Да кажем, връх 1. Ребрата, инцидентни с него, са $(1, 2)$, $(1, 4)$ и $(1, 5)$. За всяко от тях ще построим по две прави по следния начин.

Първо, заради реброто $(1, 2)$ построяваме права през особените точки $1''$ и $2'$ и права през особените точки $1'$ и $2''$. Иначе казано, през отвор 1 и отвор 2 прекарваме максимално отдалечени една от друга прави. Двете прави плюс неограничения район между тях наречаме *алея 1-2*. Фигура 16.50 показва алея 1-2 в синьо.

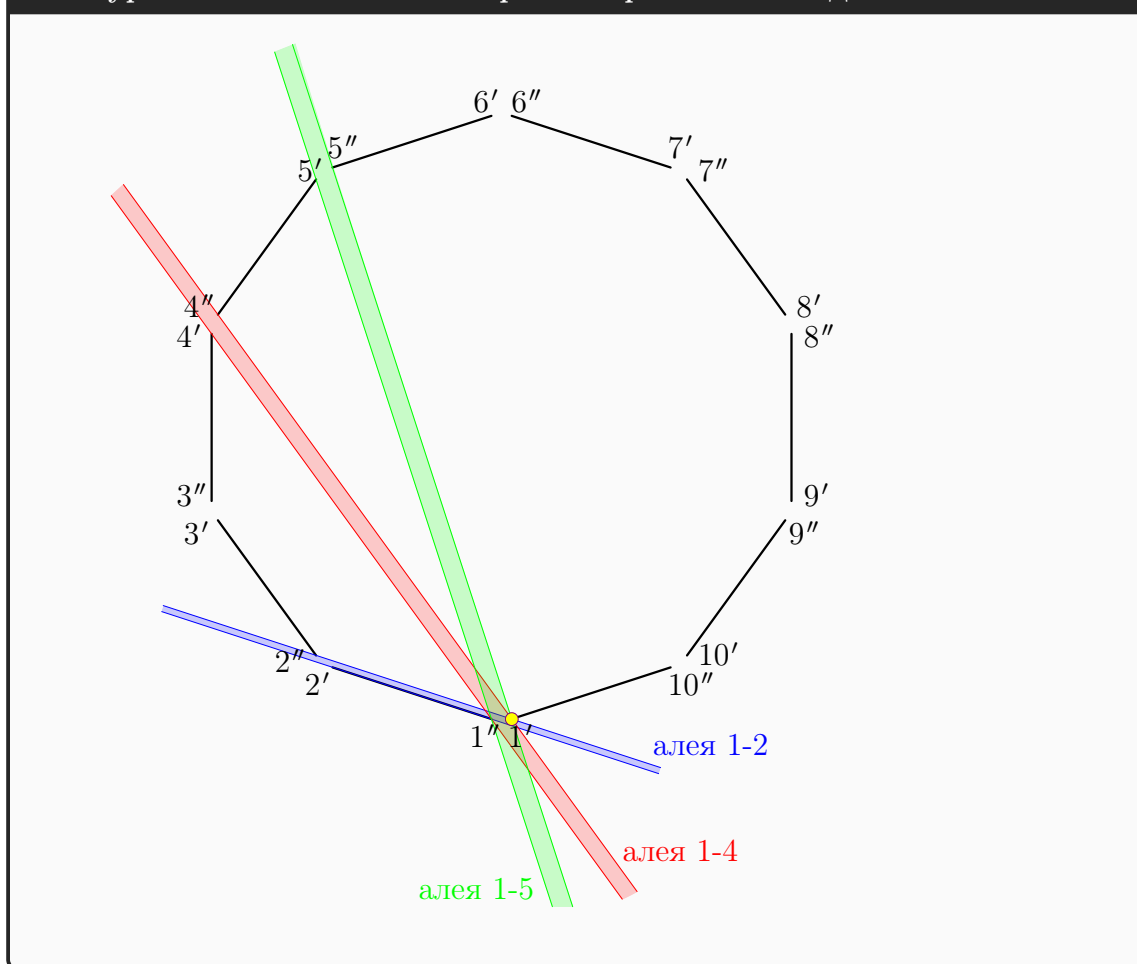
Фигура 16.50 : Построяваме алея 1-2.



После построяваме и алеи 1-4 и 1-5, които съответстват на ребрата $(1, 4)$ и $(1, 5)$ в графа, по напълно аналогичен начин. Вижте Фигура 16.51.

Особената точка $1'$ е акцентирана с жълт цвят. Забележете, че жълтата точка вижда както всяка точка както на всяка от трите алеи, така и всяка точка на целия бивш десетоъгълник. Разбира се, особената точка $1''$ също вижда тези области, но ние ще разглеждаме точките i' , за $1 \leq i \leq 10$, като точките, от които се вижда всичко.

Фигура 16.51 : Всички алеи през отвора 1. Те са видими от точка 1'.

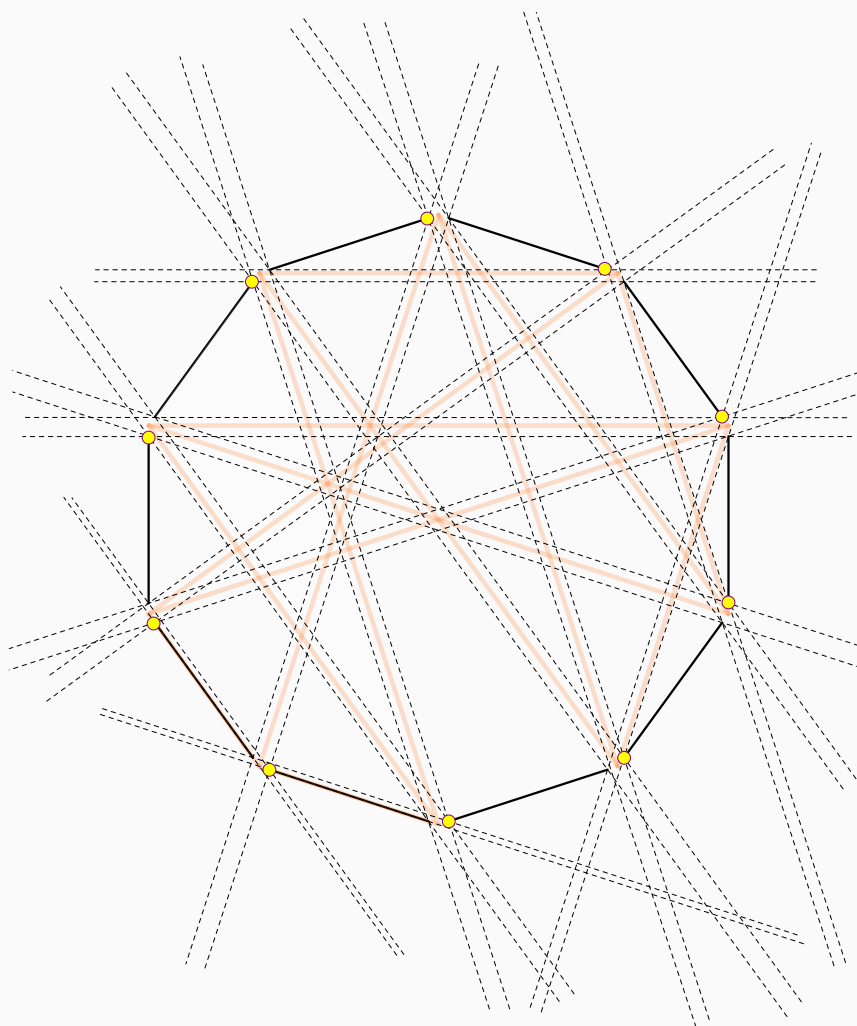


Конструираме по една такава алея и за останалите ребра на графа. Тъй като графът няма изолирани върхове, през всеки отвор минава поне една алея. За нашият пример, получаваме това, което е показано на Фигура 16.52. Страните на петнадесетте алеи са очертани с пунктирани линии.

С полупрозрачни оранжеви линии са показани ребрата на графа, за да е напълно ясно, че алеите точно съответстват на ребрата в многоъгълника, който строим.

С жълто са показани точките i' , за $1 \leq i \leq 10$. Очевидно е, че охранители, сложени върху тях, виждат абсолютно всичко: и всяка точка от всяка алея, и всяка точка от бившия десетоъгълник.

Фигура 16.52 : Всички алеи.

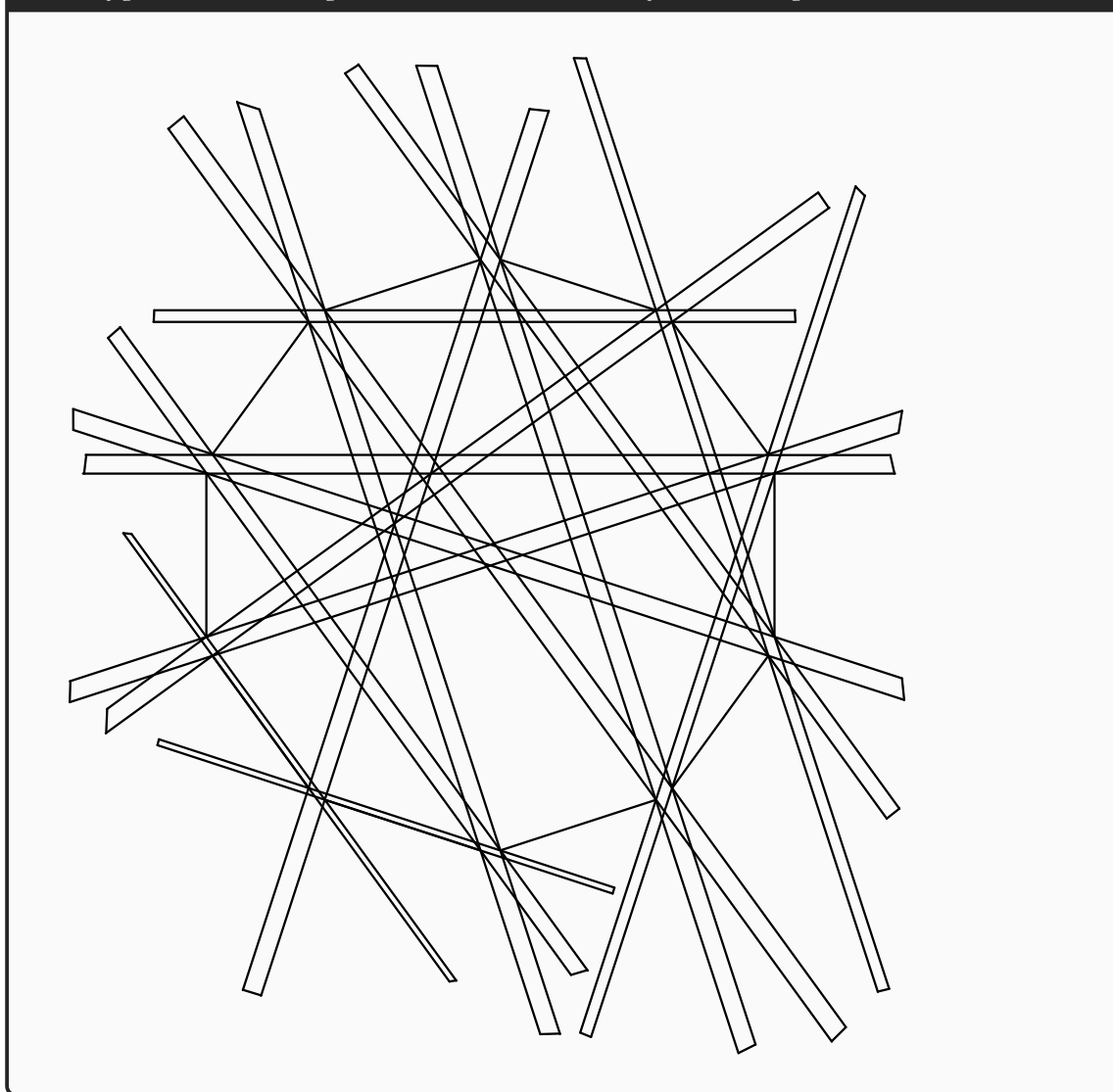


Този обект не е многоъгълник, понеже алеите са безкрайни. За да го превърнем в многоъгълник, във всяка алея слагаме две отсечки напреки и вземаме само частта от алеята между тях. Тази част е трапец, чиито бедра са въпросните две отсечки, а основите са подмножества на правите на алеята. За всяка алея, тези две отсечки-бедра се слагат

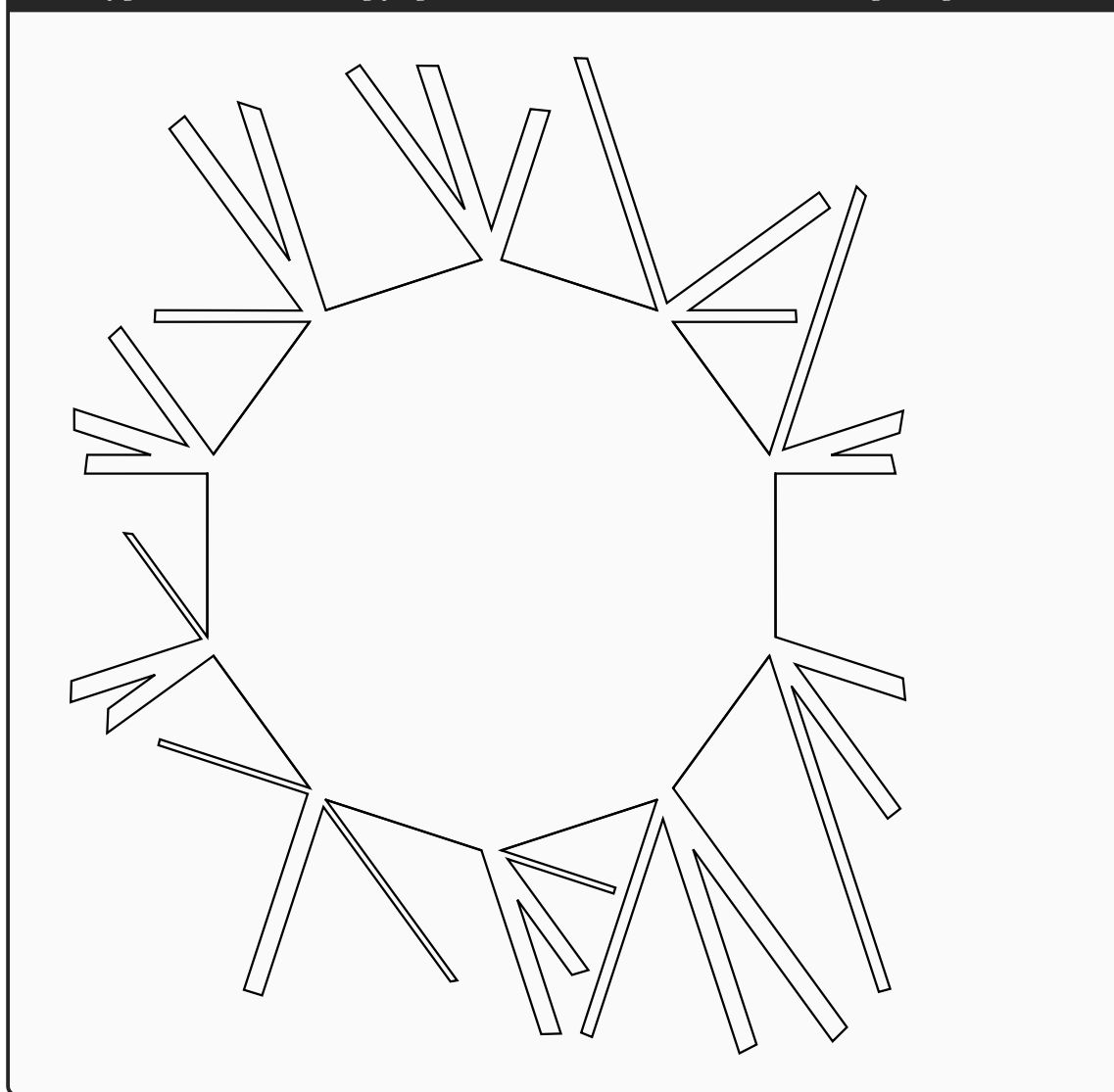
- извън бившия десетоъгълник, от двете му страни, така че полученият трапец да има непразно сечение с десетоъгълника,
- при това достатъчно далече от десетоъгълника, така щото всеки от четирите ъгъла на получения трапец да е извън всеки друг трапец, минаващ през същия отвор,
- но не прекалено далече от десетоъгълника; иска се никои два трапеца, минаващи през различни отвори на десетоъгълника, да не се пресичат, за да не получим многоъгълник с дупка.

Тази част от конструкцията е илюстрирана на Фигура 16.53.

Фигура 16.53 : Запречваме алеите и получаваме трапéци.

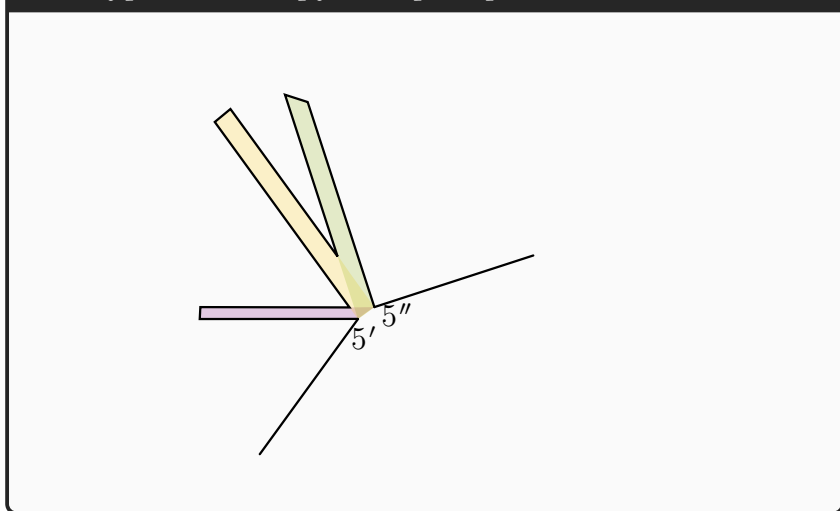


И накрая, вземаме обединението на десетоъгълника (от времето преди да махаме части от него) и трапéците и получаваме обекта, показан на Фигура 16.54. Той се състои от десетоъгълник с *коридори*, като коридорите са задънени, с единствен вход-изход към десетоъгълника, и са частите от трапéците, “стърчащи” извън десетоъгълника. Този обект-вдлъбнат многоъгълник е и крайният резултат от конструкцията. Вижте Фигура 16.54.

Фигура 16.54 : Конструираният P е десетоъгълник с коридори.

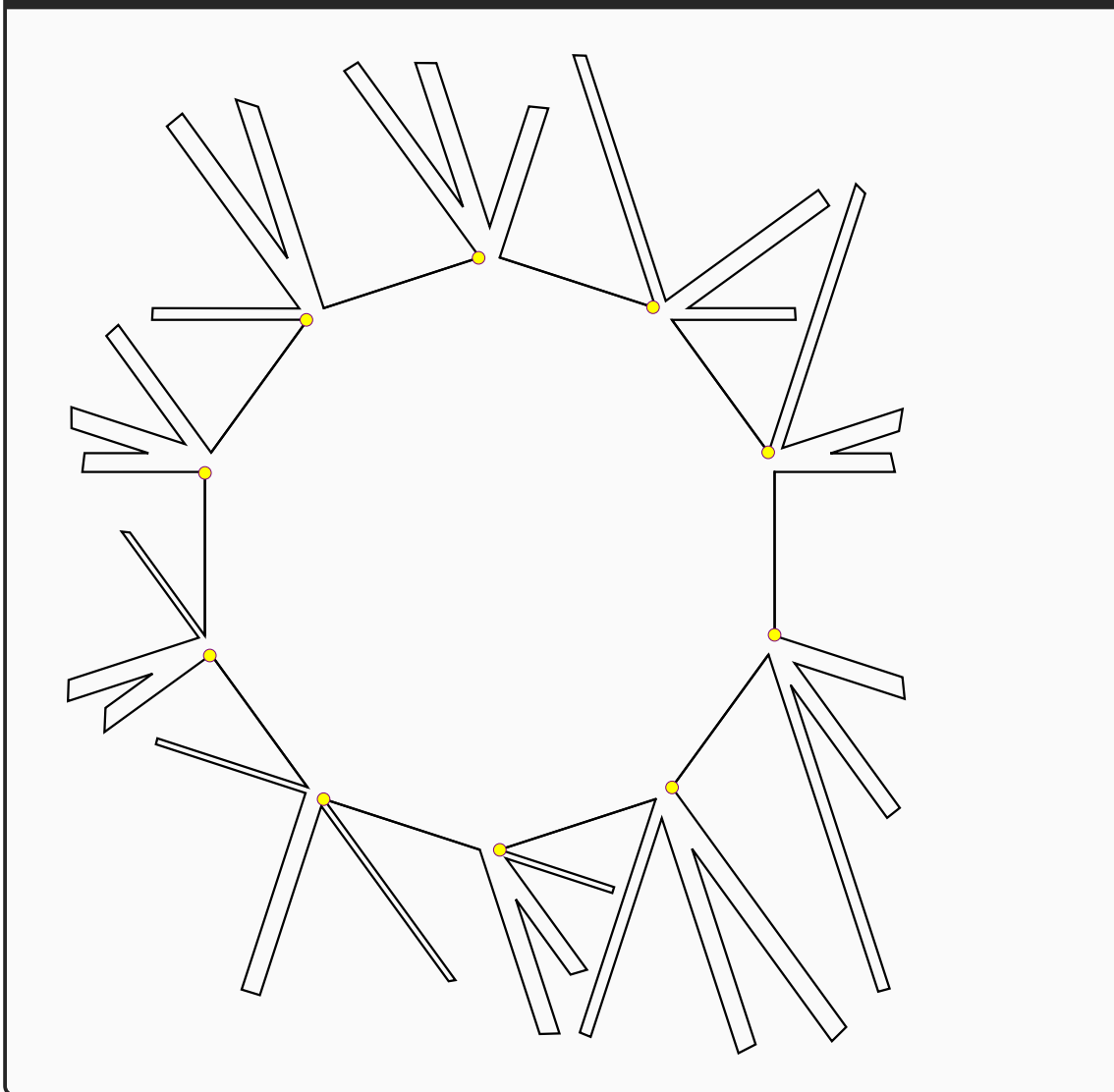
Нека дефинираме за $1 \leq j \leq 10$, *група коридори* j като обединението от коридорите, “излизаци” от отвор j . В нашия пример всяка група коридори има точно три коридора, но това се дължи на факта, че графът на Petersen е 3-регулярен. Фигура 16.55 показва група коридори 5, като трите нейни коридори са в различни цветове.

Фигура 16.55 : Група коридори 5.



Ако сложим по един охранител във всяка точка i' , за $1 \leq i \leq 10$, както е показано на Фигура 16.56 с десетте жълти точки, ще осигурим охрана на целия многоъгълник, защото точка i' вижда всяка точка на група коридори i , а освен това вижда и десетоъгълника.

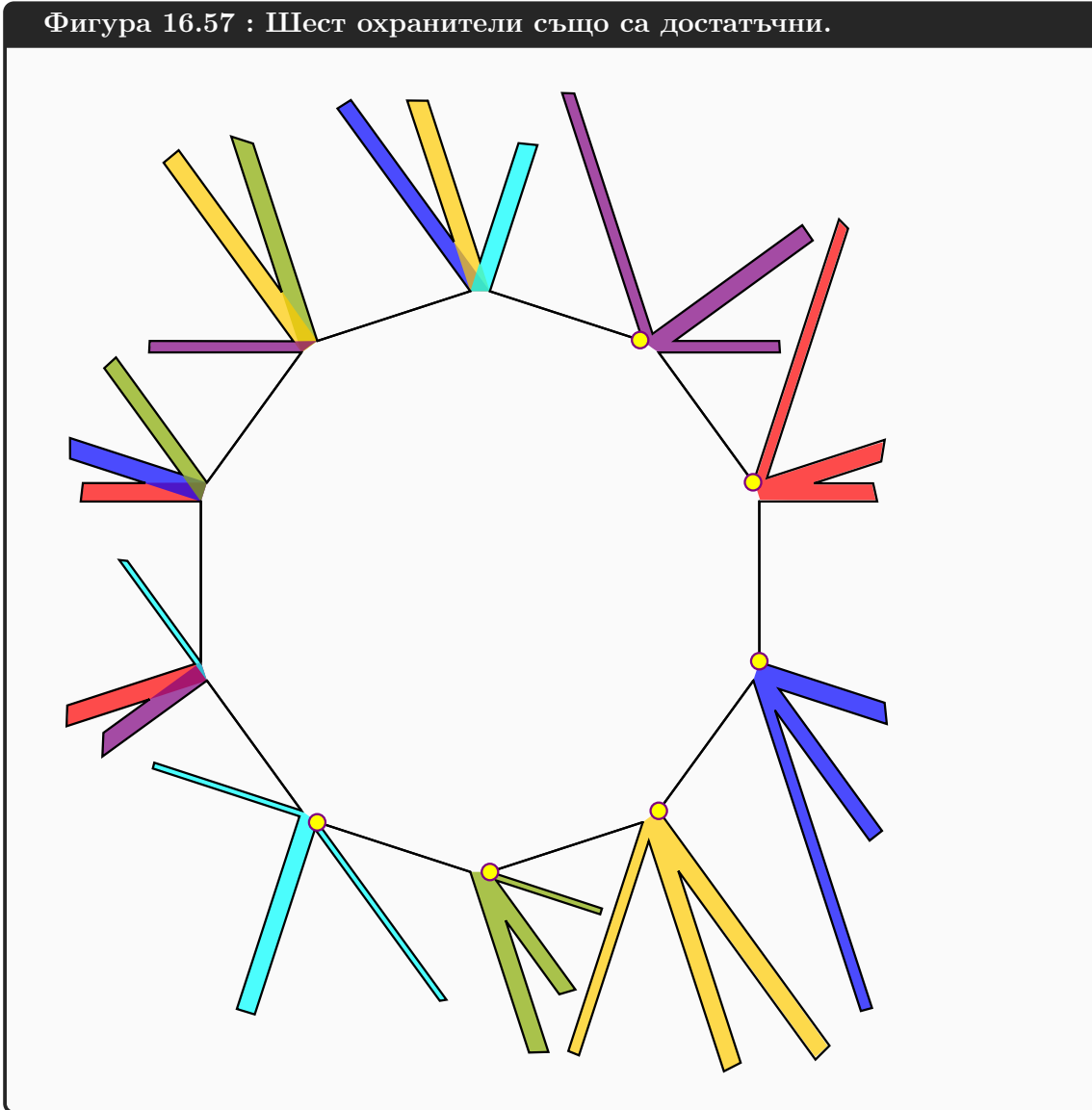
Фигура 16.56 : Десет охранители—в жълтите точки—са достатъчни.



И така, ако сложим десет охранители в десетте жълти точки, те ще виждат всичко. Работата е там, че тези десет точки са достатъчни (за разполагане на охранители), но не е необходимо да са десет. Можем да минем и с по-малко. Ключовото наблюдение е, че охранител в точка i' вижда група коридори i , но освен това, за всяко j , такова че $(i, j) \in E(G)$, вижда и (точно) единият от коридорите на група коридори j ; а именно, този от тях, който е част от алеята $i-j$. Ерго, за всяко подмножество $I \subseteq \{1', \dots, 10'\}$, такова че всяка алея има поне една обща точка с елемент на I , е вярно, че разполагането на охранители в точките на I осигурява охрана на всяка точка от P . Второто ключово наблюдение е, че точките от I отговарят на подмножество на $V(G)$, което е върхово покриване на G , понеже ребрата на G отговарят на алеите, които построихме. Трябва да помним, че коридорите са части от алеите в смисъл, че два коридора са подмножества на една и съща алея, така че, ако охранител вижда цялата алея, ще вижда и всяка точка от нейните два съответни коридора.

Вижте Фигура 16.57: шестте жълти точки виждат целия P . Тези шест точки отговарят на върхово покриване $\{1, 2, 7, 8, 9, 10\}$ на графа на Petersen от Фигура 16.47. Всяка жълта точка вижда една група коридори, от която е елемент, но също така вижда и други коридори от други групи. Това е акцентирано с цветове: монохромните групи коридори са точно тези, на изхода на които стои охранител, а в полихромните групи цветовете указват кои охранители виждат кои коридори.

Фигура 16.57 : Шест охранители също са достатъчни.



Забележете и нещо друго, което ясно се вижда на Фигура 16.57. Тъй като коридорите са дълги и тесни, за да може коридор от група, на отвора на която няма охранител, да бъде наблюдаван от охранител пред друга група коридори, този охранител трябва да е точка от алеята, от която въпросния коридор е част.

Ето скица на формално доказателство за коректността на конструкцията. В едната посока, нека G има върхово покриване с размер k . По дефиниция, има $U \subseteq V$, такова че всяко ребро има поне един връх в U и $|U| = k$. Тъй като

- върховете на G съответстват на отворите в конструкцията,
- а ребрата на G съответстват на алеите в конструкцията,

има множество от отвори, такива че ако от всеки отвор вземем точно една от особените му точки, тези точки виждат всички алеи. Освен това тези точки виждат и вътрешността на изпъкналия многоъгълник, бивайки негови периферни точки. Тогава това множество от точки вижда целия вдлъбнат многоъгълник P , който построихме. Ерго, k охранители може да виждат целия P .

В другата посока, нека k охранители виждат целия P . Във варианта на задачата, който разглеждаме (Задача 97), тези охранители се намират във върхове на P . За всеки охранител s :

- ако s е във връх i' , оставяме го на място,
- ако s е във връх i'' , преместваме го в i' ,
- ако s е друг връх v , то v е връх от някоя група коридори j ; тогава преместваме s във връх j' .

Ясно е, че при тези размествания охранителите продължават да виждат целия P . При това е възможно няколко охранителя да се окажат в една и съща точка i' . В това няма нищо нередно – може да им е дискомфортно да се намират в една и съща точка, но това не им пречи да охраняват. Алтернативно, можем да оставим само един от тях в i' , а останалите да пуснем в отпуск – при това броят на охранителите ще намалее, но тези, които ще останат, ще охраняват целия P .

Забележете, че групите коридори имат празни сечения по двойки, така че няма опасност един и същи охранител да трябва да отиде на различни места.

И така, има охранители върху точки от вида i' , тези охранители виждат всички коридори и не са повече от k на брой. Същественото наблюдение е, че тези охранители трябва да виждат и всички алеи. Наистина, алеите са безкрайно дълги, и то в две посоки спрямо правилния многоъгълник, а коридорите са с крайни дължини. Но, за да може да вижда един коридор напълно, без нито една скрита точка, даден охранител трябва да се намира върху алеята на този коридор. Коридорите са достатъчно тесни и дълги за целта.

Тогава има не повече от k охранители върху обединението на алеите. Предвид факта, че алеите съответстват биективно на ребрата на графа, заключаваме, че G има върхово покриване с размер $\leq k$.

За да довършим доказателството за коректност на конструкцията, трябва да направим още нещо. Трябва да покажем, че конструкцията може да се извърши в полиномиално време. Тук има потенциален проблем, какъвто в предните редукции не сме имали. Точките, които дефинират построения многоъгълник, трябва да са построими в полиномиално време.

- Първо, те трябва да са само полиномиално много в размера на (кодирането на) екземпляра $\langle G, k \rangle$ на VERTEX COVER. Това е изпълнено: започваме с правилен n -ъгълник, където $n = |V(G)|$, и после заменяме всеки връх u с група коридори, имаща общо $3d + 1$ точки, където d е степента на u в G .
- Всяка точка трябва да има координати, чието кодиране има най-много полиномиална големина в размера на (кодирането на) $\langle G, k \rangle$. И тук имаме проблем. Може лесно да се получат координати-ирационални числа, чието представяне в обикновено използваните кодирания е безкрайно.

Разбира се, ние можем да апроксимираме с координати-рационални числа, които имат крайни кодирания, но това може да ни докара други проблеми – примерно, дали правите, дефиниращи дадена алея, остават идеално успоредни след апроксимациите. Ако не, може да се появят части от коридори, макар и изключително малки, които не се виждат от точка i' , от която трябва да се виждат. А и рационалните числа трябва да са такива, че кодиранията им да са с размер, не по-голям от полиномиален.

По-горе, а именно на стр. 1048, бе споменато, че описанието на редукцията от VERTEX COVER в [118, стр. 247–249] е по-сложно от изложеното тук. Без съмнение, това идва

оттам, че Moret описва конструкция, в която точките, дефиниращи многоъгълника, имат целочислени координати. Читателят, който иска да види онази формално изпипана конструкция, може да прочете [118, стр. 247–249]. За целите на тези лекционни записки, изложената конструкция е напълно удовлетворителна, предавайки акуратно основната идея. $\square$

Лекция 17

Заобикаляне на неподатливостта: силна и слаба NP-пълнота, апроксимиране и параметризиране.

Резюме: Разглеждаме три възможни начина за съставяне на практически полезни алгоритми за NP-трудни задачи: възползване от малки числа във входа, генериране на приблизително решение, но с гаранции за приближението, и възползване от някакъв параметър на задачата, като за малки стойности на параметъра решението е поносимо бавен алгоритъм.

NP-трудността на дадена задача не може да е последната дума по въпроса. Тези задачи възникват в практиката на научни работници, инженери, дизайнери на системи и разработчици, чиято основна дейност не е доказване на NP-трудност. Практическите задачи искат някакви решения. Ако вярваме, че “NP-трудна” означава неизбежно “неподатлива по принцип”, няма смисъл да се опитваме да конструираме бърз—в най-лошия случай—алгоритъм за всеки вход. Но трябва да можем да предложим **някакво** решение.

Да заобиколим неподатливостта означава да не я конфронтираме директно; да я конфронтираме директно пък означава да искаме да докажем конструктивно, че $P = NP$. Избягвайки конфронтацията с това, което някои компютърни учени наричат *the daemon of intractability*, ние се примиряваме, че няма да конструираме алгоритъм с бързодействието и оптималността на алгоритмите на Prim, Kruskal, Dijkstra. Целта ни става по-скромна: алгоритъм, който все пак дава прилични резултати на практика. Ако обаче останем само с това, нашият алгоритъм е **евристика**, понятие, с което вече се запознахме на стр. 510. Искаме да отидем “на по-високото ниво” и да дадем някакви математически описания за входовете, върху които алгоритъмът работи прилично бързо, или математически гаранции за не-оптималността на получение резултат.

17.1 Силна и слаба NP-пълнота

17.2 Апроксимиращи алгоритми

17.3 Параметризирана сложност

Благодарности.

Благодарности на **Добрин Цветков** за откритите и коригирани грешки. Благодарности на **Емилиян Рогачев** за детайлното четене плюс откриване и коригиране на купища грешки. Благодарности на **Теодор Грозданов** за откритата и коригирана грешка в алгоритъма DUMMY-GOLDBACH на стр. 36. Благодарности на **Борис Дишов** за откритата и коригирана грешка в имплементацията на QUICK SORT. Благодарности на **Йордан Петров** за откритата и коригирана грешка в доказателството за коректност на алгоритъма НАРАСТВАНЕ С ЕДИНИЦА на стр. 38, както и на грешки във формулировките на Наблюдение 16 и Теорема 20. Благодарности на **Костадин Гаров** за откритата грешка в доказателството на Лема 59. Благодарности на **Ясен Алексиев** за откритата грешка в доказателството на Теорема 56. Благодарности на **доктор Добромир Кралчев** за посочването на факта, че отсъствието на малко-о отношение между две функции не влече голямо-Омега отношение между тях (вж. Допълнение 70). Благодарности на **Иван-Асен Чакъров** за откритите грешки в рекапитулацията на графите. Благодарности на **Ива Петрова Караджова** за откритите грешки във формулировката на Наблюдение 53. Благодарности на **Радослав Георгиев** за откритата грешка в дефиницията на “дървесно ребро”. Благодарности на **доктор Добромир Кралчев** за посочването на по-елегантно и илюминиращо доказателство на Лема 18. Благодарности на **Николай Пашов** за откритата и коригирана грешка в заключението на доказателството на Теорема 34. Благодарности на **Бойко Красимиров Борисов** за открит неверен резултат с невалидно “доказателство” в Допълнение 20. Благодарности на **Христо Трендафилов** за откритата груба грешка във формулировката и доказателството на Лема 29. Благодарности на **Цветан Алексиев** за откритата грешка в кода на PARTITION от Подсекция 6.2.2. Благодарности на **Ивайло Панайтов** за откритата и коригирана грешка в записа на Теорема 61. Благодарности на **доцент Стефан Герджиков**, който посочи на автора, че на български се казва “инвариант” в мъжки род, а не “инварианта” в женски род – който термин авторът поначало използваше. Благодарности на **Тодор Митрев** и **Андрей Дренски** за откритите и коригирани грешки във Фигура 5.2. Благодарности на **Теодоси Маринов Ташев** за откритите и коригирани многобройни грешки при писане и смислови грешки. Благодарности на **Емилиян Рогачев** за посочената и коригирана грешка в началните условия на рекурсивната декомпозиция в решението на SEQUENCE ALIGNMENT. Благодарности на **Теодор Христов** и **Лора Захаријева** за откритите многобройни грешки при писане. Благодарности на **Катерина Колева** за откритата и коригирана грешка в разяснението преди Редукция 12. Благодарности на **Борис Евгениев Велковски** за открита правописна грешка. Благодарности на **Андрей Дренски** за откритата и коригирана грешка в асимптотиката на f_3 в Задача 15. Благодарности на **Любомир Карагъзов** за откритата и коригирана грешка в индексването на ред 7 в ALG MATRIX-CHAIN MULTIPLICATION. Благодарности на **Тодор Дуков** за откритата неточност в дефиницията на нормална форма на Чомски на стр. 615. Благодарности на **Стоян Апостолов** за това, че посочи на автора критерия на Dirichlet за сходимост на несобствени интеграли, който критерий е ключов в доказателството на Теорема 32. Благодарности на **Тодор Дуков** за откритата и коригирана

на грешка във Фигура 16.23. Благодарности на **Михаела Гандева** за няколко открити и коригирани грешки. Благодарности на **Константин Философов** за откритата неточност в Определение 19 и откритата неконсистентност между кода на BFS и кода на DFS, както и за огромния брой правописни грешки. Благодарности на **Боян Щърбанов** за откритата правописна грешка. Благодарности на **Даниел Дамянов** за откритата тупо грешка в доказателството на Лема 38. Благодарности на **Кристиян Стоименов** за откритата грешка в доказателството на Лема 27. Благодарности на **Даниел Цветанов Дамянов** за откритата и коригирана тупо грешка в доказателството на Лема 43. Благодарности на **Димитър Стоилов** за откритата фундаментална грешка в аргументацията за коректност на алгоритъма на Kosaraju. Благодарности на **Боян Германов Щърбанов** за откритата тупо грешка в псевдокода на алгоритъма WEIGHTED INTERVAL SCHEDULING на стр. 651. Благодарности на **Мартин Георгиев Георгиев** за откритата и коригирана грешка в израза за сложността на алгоритъма на BELLMAN-FORD. Благодарности на **Тодор Дуков** за откритата и коригирана непрецизност в доказателството на Теорема 29. Благодарности на **Тодор Дуков** за откритата тупо грешка в доказателството на Лема 36. Благодарности на **Никола Драгнев** за откритата грешка в условията за входа на функцията MERGE на стр. 393. Благодарности на **Димитър Николаев Фенерски** за откритата грешка в номериране на ред при анализа на SELECTION SORT на стр. 105. Благодарности на **Емили Стоянова** за откритата грешка в доказателството на Теорема 18(2.36). Благодарности на **Антонио Бояджиев Бояджиев** за откритата грешка във фазата **Терминация** в доказателството на Лема 23 и за откритата грешка във формулировката на Инвариант 9. Благодарности на **Мартин Светлозаров Станчев** за откритата грешка във втория случай в индуктивната стъпка на доказателството на Теорема 59 и за откритата тупо грешка в доказателството на Теорема 56. Благодарности на **Александър Андреев Иванов** за откритите грешки в доказателството на Лема 34 и в решението на Задача 17. Благодарности на **Мартин Светлозаров Станчев** за откритата грешка в дефиницията на медианата на B_i в контекста на алгоритъма SELECT4 на стр. 386, за откритата груба грешка в обосновката на алгоритъма MERGE-MODIFIED на стр. 397, за откритата неточност в обяснението за работата на алгоритъма PARTITION-LOMUTO (Алгоритъм 14) и за откритата грешка в доказателството на Теорема 75. Благодарности на **Dimitar Dimitrov** за откритите неточности в доказателството на Теорема 24.

Специални благодарности на **Мартин Георгиев Георгиев** за посочването на значително по-просто и бързо решение на Задача 38 от това, което беше предложено поначало (рекурсивната декомпозиция 12.35 и 12.36).

Специални благодарности на **Йоан Руменов Василев** за посочения сериозен недостатък в доказателството на Теорема 64.

Библиография

- [1] M. Ahmadlou, H. Adeli и A. Adeli. “New diagnostic EEG markers of the Alzheimer’s disease using visibility graph”. В: *Journal of neural transmission* 117.9 (2010), с. 1099—1109. DOI: <https://doi.org/10.1007/s00702-010-0450-3>.
- [2] Alfred V. Aho, John E. Hopcroft и Jeffrey Ullman. *Data Structures and Algorithms*. 1st edition. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201000237.
- [3] Alfred V. Aho, John E. Hopcroft и Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. 1st edition. USA: Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN: 0201000296.
- [4] Mohamad Akra и Louay Bazzi. “On the Solution of Linear Recurrence Equations”. В: *Computational Optimization and Applications* 10.2 (1998), с. 195—210. ISSN: 0926-6003.
- [5] David Aldous и Persi Diaconis. “Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem”. В: *BULL. AMER. MATH. SOC. (N.S)* (1999). Достъпна онлайн на <https://www.ams.org/journals/bull/1999-36-04/S0273-0979-99-00796-X/S0273-0979-99-00796-X.pdf>, с. 413—432.
- [6] Sanjeev Arora и Boaz Barak. *Computational Complexity: A Modern Approach*. 1st edition. USA: Cambridge University Press, 2009. ISBN: 0521424267.
- [7] Bengt Aspvall, Michael F. Plass и Robert Endre Tarjan. “A linear-time algorithm for testing the truth of certain quantified boolean formulas”. В: *Information Processing Letters* 8.3 (1979). Достъпна онлайн на https://mathweb.ucsd.edu/~sbuss/CourseWeb/Math268_2007WS/2SAT.pdf, с. 121—123. ISSN: 0020-0190. DOI: [https://doi.org/10.1016/0020-0190\(79\)90002-4](https://doi.org/10.1016/0020-0190(79)90002-4).
- [8] Giorgio Ausiello и др. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. 1st edition. Berlin, Heidelberg: Springer-Verlag, 1999. ISBN: 3540654313.
- [9] Jørgen Bang-Jensen. *Lower bounds for selection of maximum, minimum, second smallest element and the median*. Достъпна онлайн на <https://imada.sdu.dk/~jbj/DM19/1b06.pdf>. 2006.
- [10] Francisco Barahona и др. “An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design”. В: *Operations Research* 36.3 (1988). Достъпна онлайн от СУ на <https://www.jstor.org/stable/170992>, с. 493—513. DOI: [10.1287/opre.36.3.493](https://doi.org/10.1287/opre.36.3.493).
- [11] Edward G. Belaga и Maurice Mignotte. “Walking Cautiously Into the Collatz Wilderness: Algorithmically, Number Theoretically, Randomly”. В: *Discrete Mathematics & Theoretical Computer Science DMTCS Proceedings vol. AG, Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities* (ян. 2006). Достъпна онлайн на <https://dmtdcs.episciences.org/3512>.

- [12] Richard Bellman. *Eye of the Hurricane: An Autobiography*. Series in modern applied mathematics. World Scientific, 1984. ISBN: 9789971966010.
- [13] Charles H. Bennett. “Notes on Landauer’s principle, reversible computation, and Maxwell’s Demon”. В: *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics* 34.3 (2003). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/S135521980300039X>, с. 501—510. ISSN: 1355-2198. DOI: [https://doi.org/10.1016/S1355-2198\(03\)00039-X](https://doi.org/10.1016/S1355-2198(03)00039-X).
- [14] S W Bent и J W John. “Finding the Median Requires $2n$ Comparisons”. В: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC ’85. Достъпна онлайн на <https://dl.acm.org/doi/10.1145/22145.22169>. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, с. 213—216. ISBN: 0897911512. DOI: [10.1145/22145.22169](https://doi.org/10.1145/22145.22169).
- [15] Jon Bentley. “Programming Pearls: Little Languages”. В: *Commun. ACM* 29.8 (авг. 1986), с. 711—721. ISSN: 0001-0782. DOI: [10.1145/6424.315691](https://doi.org/10.1145/6424.315691). URL: <http://doi.acm.org/10.1145/6424.315691>.
- [16] Jon Louis Bentley, Dorothea Haken и James B. Saxe. “A general method for solving divide-and-conquer recurrences”. В: 12 (1980). Достъпна онлайн на <https://apps.dtic.mil/dtic/tr/fulltext/u2/a064294.pdf>, с. 36—44.
- [17] Jon Louis Bentley и Michael Ian Shamos. “Divide-and-Conquer in Multidimensional Space”. В: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*. STOC ’76. Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.9611&rep=rep1&type=pdf>. Hershey, Pennsylvania, USA: Association for Computing Machinery, 1976, с. 220—230. ISBN: 9781450374149. DOI: [10.1145/800113.803652](https://doi.org/10.1145/800113.803652). URL: <https://doi.org/10.1145/800113.803652>.
- [18] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland Mathematical Library. Elsevier Science, 1984. ISBN: 9780080880235. URL: <https://books.google.bg/books?id=jEyfse-EKf8C>.
- [19] C. Berge и E. Minieka. *Graphs and Hypergraphs*. Graphs and Hypergraphs. North-Holland Publishing Company, 1973. ISBN: 9780444103994. URL: <https://books.google.bg/books?id=X32G1VfqXjsC>.
- [20] Garrett Birkhoff. *Lattice Theory*. 3rd edition. American Mathematical Society, 1967.
- [21] Andreas Blass и Yuri Gurevich. “Algorithms: A quest for absolute definitions”. В: *Bulletin of the European Association for Theoretical Computer Science* (2003). Достъпна онлайн на <http://research.microsoft.com/en-us/um/people/gurevich/Opera/164.pdf>.
- [22] Manuel Blum. “A Machine-Independent Theory of the Complexity of Recursive Functions”. В: *J. ACM* 14.2 (апр. 1967), с. 322—336. ISSN: 0004-5411. DOI: [10.1145/321386.321395](https://doi.org/10.1145/321386.321395). URL: <https://doi.org/10.1145/321386.321395>.
- [23] Manuel Blum и др. “Linear Time Bounds for Median Computations”. В: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC ’72. Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.309.9712&rep=rep1&type=pdf>. Denver, Colorado, USA: Association for Computing Machinery, 1972, с. 119—124. ISBN: 9781450374576. DOI: [10.1145/800152.804904](https://doi.org/10.1145/800152.804904).

- [24] Manuel Blum и др. “Time bounds for selection”. В: *Journal of Computer and System Sciences* 7.4 (1973). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/S0022000073800339>, с. 448—461. ISSN: 0022-0000. DOI: [https://doi.org/10.1016/S0022-0000\(73\)80033-9](https://doi.org/10.1016/S0022-0000(73)80033-9).
- [25] Kellogg S. Booth и George S. Lueker. “Linear Algorithms to Recognize Interval Graphs and Test for the Consecutive Ones Property”. В: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. STOC '75. Достъпна онлайн на <https://doi.org/10.1145/800116.803776>. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1975, с. 255—265. ISBN: 9781450374194. DOI: [10.1145/800116.803776](https://doi.org/10.1145/800116.803776).
- [26] Otakar Borůvka. *O jistém problému minimálním*. Czech. Práce moravské přírodovědecké společnosti 3 (1926), 37-58 (1926). Достъпна свободно онлайн на <https://dml.cz/handle/10338.dmlcz/500114>. 1926.
- [27] Robert S. Boyer и J Strother Moore. “MJRTY: A Fast Majority Vote Algorithm”. В: *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Под ред. на Robert S. Boyer. Automated Reasoning Series. Достъпна свободно онлайн на <http://www.cs.utexas.edu/users/boyer/mjrty.ps.Z> (gunzip mjrty.ps.Z) или <https://apps.dtic.mil/sti/tr/pdf/ADA131702.pdf> (недобро качество). Kluwer Academic Publishers, 1991, с. 105—118.
- [28] A. Bretto. *Hypergraph Theory: An Introduction*. Mathematical Engineering. Springer International Publishing, 2013. ISBN: 9783319000800. URL: <https://books.google.bg/books?id=1b5DAAAQBAJ>.
- [29] Neville Campbell. Непубликувана, достъпна онлайн на <https://nevillecampbell.com/Recurrences.pdf>.
- [30] V Chvátal. “A combinatorial theorem in plane geometry”. В: *Journal of Combinatorial Theory, Series B* 18.1 (1975). Свободно достъпна на <https://www.sciencedirect.com/science/article/pii/0095895675900611>, с. 39—41. ISSN: 0095-8956. DOI: [https://doi.org/10.1016/0095-8956\(75\)90061-1](https://doi.org/10.1016/0095-8956(75)90061-1).
- [31] Alan Cobham. “The Intrinsic Computational Difficulty of Functions”. В: *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)*. Под ред. на Yehoshua Bar-Hillel. Достъпна онлайн на https://www.cs.toronto.edu/~sacook/homepage/cobham_intrinsic.pdf. North-Holland Publishing, 1965, с. 24—30.
- [32] Louis Comtet. *Analyse Combinatoire Tome II*. Presses Universitaires de France, 1970. ISBN: 978-2130309819.
- [33] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. В: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Достъпна онлайн на <https://www.cs.toronto.edu/~sacook/homepage/1971.pdf>. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, с. 151—158. ISBN: 9781450374644. DOI: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047).
- [34] Thomas H. Cormen и др. *Introduction to Algorithms, Third Edition*. 3rd edition. The MIT Press, 2009. ISBN: 9780262033848.
- [35] Thomas T. Cormen, Charles E. Leiserson и Ronald L. Rivest. *Introduction to Algorithms*. 1st edition. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-03141-8.

- [36] George B. Dantzig. “Discrete-Variable Extremum Problems”. В: *Operations Research* 5.2 (1957). Достъпна свободно онлайн на <https://www.jstor.org/stable/167356>, с. 266—277. ISSN: 0030364X, 15265463. (Дата на посещ. 20.02.2024).
- [37] Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. 1st edition. USA: A. K. Peters, Ltd., 2011. ISBN: 1466505192.
- [38] Reinhard Diestel. *Graph Theory*. Fourth Edition. Т. 173. Graduate Texts in Mathematics. Springer, 2010. ISBN: 9783642142789 3642142788 9783642142796 3642142796.
- [39] Dorit Dor и Uri Zwick. “Selecting the Median”. В: *SIAM Journal on Computing* 28.5 (1999). Достъпна онлайн на <https://www.cs.unc.edu/~plaisted/comp550/fast.median.finding.pdf>, с. 1722—1758. DOI: [10.1137/S0097539795288611](https://doi.org/10.1137/S0097539795288611).
- [40] Dorit Dor и др. “On Lower Bounds for Selecting the Median”. В: *SIAM Journal on Discrete Mathematics* 14.3 (2001). Достъпна онлайн на <https://www.csc.kth.se/~johanh/mediansdma.pdf>, с. 299—311. DOI: [10.1137/S0895480196309481](https://doi.org/10.1137/S0895480196309481).
- [41] Jack Edmonds. “Covers and packings in a family of sets”. В: *Bulletin of the American Mathematical Society* 68.5 (1962). Достъпна онлайн на <https://www.ams.org/journals/bull/1962-68-05/S0002-9904-1962-10791-5/S0002-9904-1962-10791-5.pdf>, с. 494—499. DOI: [bams/1183524862](https://doi.org/10.1137/S0002-9904-1962-10791-5).
- [42] Jack Edmonds. “Paths, Trees and Flowers”. В: *CANADIAN JOURNAL OF MATHEMATICS* (1965). Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.494.9610&rep=rep1&type=pdf>, с. 449—467.
- [43] Saber N. Elaydi. *An Introduction to Difference Equations*. 3rd edition. Springer-Verlag New York, 2005. ISBN: 978-0-387-23059-7. DOI: [10.1007/0-387-27602-5](https://doi.org/10.1007/0-387-27602-5).
- [44] Salah E Elmaghraby. “The concept of “state” in discrete dynamic programming”. В: *Journal of Mathematical Analysis and Applications* 29.3 (1970). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/0022247X70900661>, с. 523—557. ISSN: 0022-247X. DOI: [https://doi.org/10.1016/0022-247X\(70\)90066-1](https://doi.org/10.1016/0022-247X(70)90066-1).
- [45] J. Erickson. *Algorithms*. Достъпна свободно онлайн на <https://jeffe.cs.illinois.edu/teaching/algorithms/>. Jeff Erickson, 2019. ISBN: 9781792644832.
- [46] Jeff Erickson. “Lower Bounds for Linear Satisfiability Problems”. В: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '95. Достъпна онлайн на <https://jeffe.cs.illinois.edu/pubs/pdf/linsat.pdf>. San Francisco, California, USA: Society for Industrial и Applied Mathematics, 1995, с. 388—395. ISBN: 0898713498.
- [47] Jeff Erickson, Ivor van der Hoog и Tillmann Miltzow. “Smoothing the gap between NP and ER”. В: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. Свободно достъпна на <https://jeffe.cs.illinois.edu/pubs/pdf/realram.pdf>. 2020, с. 1022—1033.
- [48] Steve Fisk. “A short proof of Chvátal’s Watchman Theorem”. В: *Journal of Combinatorial Theory, Series B* 24.3 (1978). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/009589567890059X>, с. 374. ISSN: 0095-8956. DOI: [https://doi.org/10.1016/0095-8956\(78\)90059-X](https://doi.org/10.1016/0095-8956(78)90059-X).
- [49] Robert W. Floyd. “Algorithm 245: Treesort”. В: *Communications of the ACM* 7.12 (дек. 1964). Достъпна онлайн на <http://doi.acm.org/10.1145/355588.365103>, с. 701—. ISSN: 0001-0782. DOI: [10.1145/355588.365103](https://doi.org/10.1145/355588.365103).

- [50] Robert W. Floyd. “Algorithm 97: Shortest Path”. В: *Commun. ACM* 5.6 (юни 1962). Достъпна онлайн на <https://dl.acm.org/doi/10.1145/367766.368168>, с. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168.
- [51] Lance Fortnow. “Diagonalization”. В: *Bulletin of the European Association for Theoretical Computer Science* 71 (юни 2000). Достъпна онлайн на <https://lance.fortnow.com/papers/files/diag.pdf>, с. 102—112.
- [52] Lance Fortnow. “The Status of the P Versus NP Problem”. В: *Commun. ACM* 52.9 (септ. 2009). Достъпна онлайн на <http://dl.acm.org/citation.cfm?id=1562186>, с. 78—86. ISSN: 0001-0782. DOI: 10.1145/1562164.1562186. URL: <http://doi.acm.org/10.1145/1562164.1562186>.
- [53] J.B. Fraleigh, R.A. Beauregard и V.J. Katz. *Linear Algebra*. 3rd edition. Addison-Wesley, 1995. ISBN: 0-201-52675-1. URL: <https://books.google.bg/books?id=ViqBvgAACAAJ>.
- [54] Michael L. Fredman. “On computing the length of longest increasing subsequences”. В: *Discrete Mathematics* 11.1 (1975). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/0012365X7590103X>, с. 29—35. ISSN: 0012-365X. DOI: [https://doi.org/10.1016/0012-365X\(75\)90103-X](https://doi.org/10.1016/0012-365X(75)90103-X).
- [55] Anka Gajentaan и Mark H. Overmars. “On a class of $O(n^2)$ problems in computational geometry”. В: *Computational Geometry* 45.4 (2012). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/S0925772111000927/pdf?md5=32a2f8abc82822ff176112364be58329&pid=1-s2.0-S0925772111000927-main.pdf>, с. 140—152. ISSN: 0925-7721. DOI: <https://doi.org/10.1016/j.comgeo.2011.11.006>.
- [56] M. R. Garey и D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. First Edition. W. H. Freeman, 1979. ISBN: 0716710455. URL: <http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455>.
- [57] Subir K. Ghosh и Partha P. Goswami. “Unsolved Problems in Visibility Graphs of Points, Segments, and Polygons”. В: *ACM Comput. Surv.* 46.2 (дек. 2013). Достъпна онлайн на https://static.aminer.org/pdf/PDF/000/842/846/visibility_graphs_and_oriented_matroids.pdf. ISSN: 0360-0300. DOI: 10.1145/2543581.2543589.
- [58] Kurt Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Превод В. Meltzer; има свободно достъпно копие на превода на https://homepages.uc.edu/~martinj/History_of_Logic/Godel/Godel%20%E2%80%93%20Formally%20Undecidable%20Propositions%20of%20Principia%20Mathematica%201931.pdf. Dover Publications, 1992. ISBN: 0486669807.
- [59] Omer Gold и Micha Sharir. “Improved Bounds for 3SUM, k-SUM, and Linear Degeneracy”. В: *25th Annual European Symposium on Algorithms (ESA 2017)*. Под ред. на Kirk Pruhs и Christian Sohler. Т. 87. Leibniz International Proceedings in Informatics (LIPIcs). Достъпна онлайн на <https://arxiv.org/abs/1512.05279>. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 42:1—42:13. ISBN: 978-3-95977-049-1. DOI: 10.4230/LIPIcs.ESA.2017.42.
- [60] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. ISSN. Elsevier Science, 2004. ISBN: 9780080526966. URL: https://books.google.bg/books?id=8xo-VrWo5%5C_QC.
- [61] Ronald L. Graham, Donald E. Knuth и Oren Patashnik. *Concrete Mathematics*. second edition. Addison-Wesley, 1994. ISBN: 0-201-55802-5.

- [62] B. Guenin, J. Könemann и L. Tunçel. *A Gentle Introduction to Optimization*. Cambridge University Press, 2014. DOI: [10.1017/CB09781107282094](https://doi.org/10.1017/CB09781107282094).
- [63] H.W. Guggenheimer. *Applicable Geometry: Global and Local Convexity*. Applied mathematics series. R. E. Krieger Publishing Company, 1977. ISBN: 9780882753683. URL: <https://books.google.bg/books?id=A2g6AAAAMAAJ>.
- [64] D. Gusfield, Cambridge University Press и John D. & Phyllis S. Harrah Library Fund. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press, 1997. ISBN: 9780521585194. URL: <https://books.google.bg/books?id=0fw5w1yuD8kC>.
- [65] F. Hadlock. “Finding a Maximum Cut of a Planar Graph in Polynomial Time”. В: *SIAM Journal on Computing* 4.3 (1975). Достъпна онлайн от СУ на <http://web.eecs.umich.edu/~pettie/matching/Hadlock-maxcut-planar-graph-reduction-to-T-join.pdf>, с. 221—225. DOI: [10.1137/0204019](https://doi.org/10.1137/0204019).
- [66] P. Hall. “On Representatives of Subsets”. В: *Journal of the London Mathematical Society* s1-10.1 (1935). Достъпна свободно онлайн на https://math.dartmouth.edu/archive/m38s12/public_html/sources/Hall1935.pdf, с. 26—30. DOI: <https://doi.org/10.1112/jlms/s1-10.37.26>.
- [67] Juris Hartmanis и Richard Edwin Stearns. “On the Computational Complexity of Algorithms”. В: *Transactions of the American Mathematical Society* 117 (1965). Достъпна онлайн на <https://www.ams.org/journals/tran/1965-117-00/S0002-9947-1965-0170805-7/S0002-9947-1965-0170805-7.pdf>, с. 285—306.
- [68] Michael Haythorpe. “Reducing the generalised Sudoku problem to the Hamiltonian cycle problem”. В: *CoRR* abs/1603.03019 (2016). Достъпна онлайн на <https://arxiv.org/abs/1603.03019>.
- [69] T.L. Heath и J.L. Heiberg. *Books 3-9. The Thirteen Books of Euclid’s Elements*. Достъпна онлайн на http://www.wilbourhall.org/pdfs/Heath_Euclid_II.pdf. The University Press, 1908.
- [70] Paul J. Heffernan и Joseph S. B. Mitchell. “An Optimal Algorithm for Computing Visibility in the Plane”. В: *SIAM Journal on Computing* 24.1 (1995). Разширен абстракт е достъпен свободно онлайн на <https://ecommons.cornell.edu/bitstream/1813/8838/1/TR000953.pdf>, с. 184—201. DOI: [10.1137/S0097539791221505](https://doi.org/10.1137/S0097539791221505).
- [71] S. Henikoff и J. Henikoff. “Amino acid substitution matrices from protein blocks.” В: *Proceedings of the National Academy of Sciences of the United States of America (PNAS)* 89 (1992). Достъпна онлайн на <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC50453/pdf/pnas01096-0363.pdf>, с. 10915—10919.
- [72] Daniel S. Hirschberg и C. K. Wong. “A Polynomial-Time Algorithm for the Knapsack Problem with Two Variables”. В: *J. ACM* 23.1 (1976). Достъпна свободно онлайн на <https://ics.uci.edu/~dan/pubs/p147-hirschberg.pdf>, с. 147—154. DOI: [10.1145/321921.321936](https://doi.org/10.1145/321921.321936).
- [73] C. A. R. Hoare. “Algorithm 63: Partition”. В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: [10.1145/366622.366642](https://doi.org/10.1145/366622.366642). URL: <http://doi.acm.org/10.1145/366622.366642>.
- [74] C. A. R. Hoare. “Algorithm 64: Quicksort”. В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: [10.1145/366622.366644](https://doi.org/10.1145/366622.366644). URL: <http://doi.acm.org/10.1145/366622.366644>.

- [75] C. A. R. Hoare. “Quicksort”. В: *The Computer Journal* 5.1 (1962), с. 10—16. DOI: 10.1093/comjnl/5.1.10. eprint: <http://comjnl.oxfordjournals.org/cgi/reprint/5/1/10.pdf>. URL: <http://comjnl.oxfordjournals.org/cgi/content/abstract/5/1/10>.
- [76] L. Hogben. *Handbook of Linear Algebra, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2013. ISBN: 9781466507289.
- [77] John E. Hopcroft и Richard M. Karp. “An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs”. В: *SIAM J. Comput.* 2 (1973). Достъпна свободно онлайн на <http://cse.unl.edu/~choueiry/Documents/MaxMatching-HopcroftKarp.pdf>, с. 225—231.
- [78] John Hopcroft и Robert Tarjan. “Efficient Planarity Testing”. В: *J. ACM* 21.4 (окт. 1974). Достъпна онлайн на <https://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/Efficient%20Planarity.pdf>, с. 549—568. ISSN: 0004-5411. DOI: 10.1145/321850.321852.
- [79] Hagen Höpfner и Christian Bunse. “Towards an Energy-Consumption Based Complexity Classification for Resource Substitution Strategies”. В: *Proceedings of the 22nd Workshop “Grundlagen von Datenbanken 2010”, Bad Helmstedt, Germany, May 25-28, 2010*. Под ред. на Wolf-Tilo Balke и Christoph Lofi. Т. 581. CEUR Workshop Proceedings. Достъпна онлайн на <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.228.9932&rep=rep1&type=pdf>. CEUR-WS.org, 2010.
- [80] Hsien-Kuei Hwang и Jean-Marc Steyaert. *On the Number of Heaps and the Cost of Heap Construction*. Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.9150&rep=rep1&type=pdf>. 2001.
- [81] R. Impagliazzo. “A Personal View of Average-case Complexity”. В: *Proceedings of the 10th Annual Structure in Complexity Theory Conference (SCT’95)*. SCT ’95. Достъпна онлайн на <http://www.cs.ucsd.edu/users/russell/average.ps>. Washington, DC, USA: IEEE Computer Society, 1995, с. 134—. ISBN: 0-8186-7052-5. URL: <http://dl.acm.org/citation.cfm?id=829497.829786>.
- [82] Ravi Jain, David Molnar и Zulfikar Ramzan. “Towards a Model of Energy Complexity for Algorithms”. В: *IEEE Wireless Communications & Networking Conference* (2005).
- [83] Alan S. Henry John N. Crossley. “Thus spake al-Khwārizmī: A translation of the text of Cambridge University Library Ms. Ii.vi.5”. В: *Historia Mathematica* 17 (2 1990). Достъпна онлайн на <http://www.sciencedirect.com/science/article/pii/031508609090048I>, с. 103—131.
- [84] David S Johnson. “The NP-completeness column: An ongoing guide”. В: *Journal of Algorithms* 8.2 (1987), с. 285—303. ISSN: 0196-6774. DOI: [https://doi.org/10.1016/0196-6774\(87\)90043-5](https://doi.org/10.1016/0196-6774(87)90043-5). URL: <http://www.sciencedirect.com/science/article/pii/0196677487900435>.
- [85] Yonggun Jun, Momčilo Gavrilov и John Bechhoefer. “High-Precision Test of Landauer’s Principle in a Feedback Trap”. В: *Physical Review Letters* 113.19 (ноем. 2014). ISSN: 1079-7114. DOI: 10.1103/physrevlett.113.190601. URL: <http://dx.doi.org/10.1103/PhysRevLett.113.190601>.
- [86] A. B. Kahn. “Topological Sorting of Large Networks”. В: *Commun. ACM* 5.11 (ноем. 1962). Статията е свободно достъпна на <https://dl.acm.org/doi/pdf/10.1145/368996.369025?download=true>, с. 558—562. ISSN: 0001-0782. DOI: 10.1145/368996.369025.

- [87] Dan Kalman. “The Generalized Vandermonde Matrix”. В: *Mathematics Magazine* 57.1 (1984). Достъпна онлайн на <https://www.jstor.org/stable/2690290>, с. 15–21. ISSN: 0025570X, 19300980.
- [88] Ravindran Kannan. “A Polynomial Algorithm for the Two-Variable Integer Programming Problem”. В: *J. ACM* 27.1 (1980). Достъпна свободно онлайн на https://www.researchgate.net/publication/220431950_A_Polynomial_Algorithm_for_the_Two-Variable_Integer_Programming_Problem, с. 118–122. DOI: 10.1145/322169.322179.
- [89] R. Karp. “Reducibility among combinatorial problems”. В: *Complexity of Computer Computations*. Под ред. на R. Miller и J. Thatcher. Достъпна онлайн на <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-113.pdf>. Plenum Press, 1972, с. 85–103.
- [90] Richard M. Karp. “On-Line Algorithms Versus Off-Line Algorithms: How Much is It Worth to Know the Future?” В: *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I*. Достъпна онлайн на <http://www.icsi.berkeley.edu/pubs/techreports/TR-92-044.pdf>. NLD: North-Holland Publishing Co., 1992, с. 416–429. ISBN: 044489747X.
- [91] Donald E. Knuth. “Big Omicron and big Omega and big Theta”. В: *SIGACT News* 8.2 (1976). Достъпна онлайн на http://www.phil.uu.nl/datastructuren/10-11/knuth_big_omicron.pdf, с. 18–24. URL: <http://doi.acm.org/10.1145/1008328.1008329>.
- [92] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.
- [93] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89684-2.
- [94] Donald E. Knuth. *The Art of Computer Programming, Volume 3 (2Nd Ed.): Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.
- [95] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. 1st edition. Addison-Wesley Professional, 2011. ISBN: 0201038048.
- [96] Mark Korenblit и Vadim E. Levit. “A One-Vertex Decomposition Algorithm for Generating Algebraic Expressions of Square Rhomboids”. В: *CoRR* abs/1211.1661 (2012). Достъпна онлайн на <http://arxiv.org/pdf/1211.1661v1>.
- [97] Daniel Krob и др. “Pseudo-Permutations I: First Combinatorial and Lattice Properties”. В: Достъпна онлайн на <https://igm.univ-mlv.fr/~novelli/ARTICLES/pp1.ps>. 2000.
- [98] Joseph B. Kruskal Jr. “On the shortest spanning subtree of a graph and the traveling salesman problem”. В: *Proc. Amer. Math. Soc.* 7 (1956). Достъпна свободно онлайн на <https://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/>, с. 48–50. ISSN: 0002-9939,1088-6826. DOI: 10.2307/2033241.
- [99] William Kuszmaul и Charles E. Leiserson. “Floors and Ceilings in Divide-and-Conquer Recurrences”. В: *Symposium on Simplicity in Algorithms (SOSA)*. Достъпна онлайн на <https://epubs.siam.org/doi/abs/10.1137/1.9781611976496.15>, с. 133–141. DOI: 10.1137/1.9781611976496.15.
- [100] Richard E. Ladner. “On the Structure of Polynomial Time Reducibility”. В: *J. ACM* 22.1 (ян. 1975), с. 155–171. ISSN: 0004-5411. DOI: 10.1145/321864.321877.

- [101] J C Lagarias. “The $3x+1$ problem: An annotated bibliography (1963–1999)”. Достъпна онлайн на <https://arxiv.org/abs/math/0608208>. 2012.
- [102] J C Lagarias. “The $3x+1$ Problem: An Annotated Bibliography, II (2000–2009)”. Достъпна онлайн на <https://arxiv.org/abs/math/0309224>. 2011.
- [103] Rolf Landauer. “Irreversibility and heat generation in the computing process”. В: *IBM Journal of Research and Development* 5.3 (1961). Достъпна, но не свободно, онлайн на <https://ieeexplore.ieee.org/document/5392446>, с. 183–191. DOI: 10.1147/rd.53.0183.
- [104] D. Lee и A. Lin. “Computational complexity of art gallery problems”. В: *IEEE Transactions on Information Theory* 32.2 (1986), с. 276–282. DOI: 10.1109/TIT.1986.1057165.
- [105] Leighton. *Note on Better Master Theorems for Divide-and-Conquer Recurrences*. Достъпна онлайн <http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>. 1996.
- [106] Vladimir Iosifovich Levenshtein. “Binary codes capable of correcting deletions, insertions and reversals.” В: *Soviet Physics Doklady* 10.8 (февр. 1966). Достъпна онлайн на <https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf>, с. 707–710.
- [107] Leonid A. Levin. “Universal Sequential Search Problems”. В: *Problems of Information Transmission* 9.3 (1973). Оригиналът на руски е достъпен онлайн на http://www.mathnet.ru/php/getFT.phtml?jrnid=ppi&paperid=914&what=fullt&option_lang=rus.
- [108] YAW-LING LIN и STEVEN S. SKIENA. “COMPLEXITY ASPECTS OF VISIBILITY GRAPHS”. В: *International Journal of Computational Geometry & Applications* 05.03 (1995), с. 289–312. DOI: 10.1142/S0218195995000179.
- [109] Seth Lloyd. “Ultimate physical limits to computation”. В: *Nature* 406 (6799 авг. 2000). Достъпна онлайн на <https://arxiv.org/abs/quant-ph/9908043>, с. 1047–1054. DOI: 10.1038/35023282.
- [110] Laszlo Lovasz и Neal E. Young. *Lecture Notes on Evasiveness of Graph Properties*. Достъпна свободно онлайн на <https://arxiv.org/abs/cs/0205031>. 2002. arXiv: cs/0205031 [cs.CC].
- [111] E. Mahmudov. *Single Variable Differential and Integral Calculus: Mathematical Analysis*. SpringerLink : Bücher. Atlantis Press, 2013. ISBN: 9789491216862. URL: <https://books.google.bg/books?id=oc9EAAAAQBAJ>.
- [112] Zohar Manna. *Mathematical Theory of Computation*. Dover Publications, Incorporated, 2003.
- [113] Silvano Martello и Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0471924202.
- [114] A. R. Meyer и L. J. Stockmeyer. “The equivalence problem for regular expressions with squaring requires exponential space”. В: *13th Annual Symposium on Switching and Automata Theory (swat 1972)*. 1972, с. 125–129. DOI: 10.1109/SWAT.1972.29.
- [115] Silvio Micali и Vijay V. Vazirani. “An $O(\sqrt{|v|} |E|)$ Algorithm for Finding Maximum Matching in General Graphs”. В: *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. Достъпна, но не свободно, онлайн на <https://ieeexplore.ieee.org/document/4567800>. IEEE Computer Society, 1980, с. 17–27. DOI: 10.1109/SFCS.1980.12.

- [116] Burkhard Monien. “The Bandwidth Minimization Problem for Caterpillars with Hair Length 3 is NP-Complete”. В: *SIAM Journal on Algebraic Discrete Methods* 7.4 (1986). Достъпна онлайн в недобро качество на <https://digital.ub.uni-paderborn.de/hsx/download/pdf/42089>, с. 505—512. DOI: [10.1137/0607057](https://doi.org/10.1137/0607057).
- [117] E.F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL: <https://books.google.com/books?id=IVZBHAACA AJ>.
- [118] Bernard M. Moret. *The Theory of Computation*. 1st edition. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201258285.
- [119] Gonzalo Navarro. “A Guided Tour to Approximate String Matching”. В: *ACM Comput. Surv.* 33.1 (март 2001). Достъпна онлайн на https://www.dcc.uchile.cl/TR/1999/TR_DCC-1999-005.pdf, с. 31—88. ISSN: 0360-0300. DOI: [10.1145/375360.375365](https://doi.org/10.1145/375360.375365).
- [120] Saul B. Needleman и Christian D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. В: *Journal of Molecular Biology* 48.3 (1970). Достъпна онлайн на www.cs.duke.edu/courses/compsci260/spring16/resources/AlignmentPapers/1970.needleman.wunsch.pdf, с. 443—453. ISSN: 0022-2836. DOI: [https://doi.org/10.1016/0022-2836\(70\)90057-4](https://doi.org/10.1016/0022-2836(70)90057-4).
- [121] Jaroslav Nešetřil, Eva Milková и Helena Nešetřilová. “Otakar Borůvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history”. В: *Discrete Mathematics* 233.1 (2001). Достъпна свободно онлайн на <https://www.sciencedirect.com/science/article/pii/S0012365X00002247>, с. 3—36. ISSN: 0012-365X. DOI: [https://doi.org/10.1016/S0012-365X\(00\)00224-7](https://doi.org/10.1016/S0012-365X(00)00224-7).
- [122] Joseph O’Rourke. *Art Gallery Theorems and Algorithms*. USA: Oxford University Press, Inc., 1987. ISBN: 0195039653.
- [123] Aurélien Ooms. “Algorithms and Data Structures for 3SUM and Friends”. Достъпна онлайн на <https://dipot.ulb.ac.be/dspace/bitstream/2013/293549/5/digital.pdf>. Докт. ... дис. Université libre de Bruxelles, 2019.
- [124] Xavier Ouvrard. *Hypergraphs: an introduction and review*. Свободно достъпна на <https://arxiv.org/pdf/2002.05014>. 2020.
- [125] Christos H. Papadimitriou. *Computational Complexity*. Chichester, UK: John Wiley и Sons Ltd., с. 260—265. ISBN: 0-470-86412-5.
- [126] Christos H. Papadimitriou и Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. USA: Prentice-Hall, Inc., 1982. ISBN: 0131524623.
- [127] Ian Parberry. *Problems on Algorithms*. Второто издание е достъпно със специален лиценз на <http://larc.unt.edu/ian/books/free/license.html>. Prentice Hall PTR, 1995. ISBN: 9780134335582.
- [128] Heinz-Otto Peitgen, Hartmut Jürgens и Dietmar Saupe. *Chaos and fractals - new frontiers of science (2. ed.)*. Springer, 2004, с. I—XIII, 1—864. ISBN: 978-0-387-20229-7.
- [129] M. Pirlot и P. Vincke. *Semiororders: Properties, Representations, Applications*. Theory and Decision Library B. Springer Netherlands, 1997. ISBN: 9780792346173. URL: <https://books.google.com/books?id=rXroQjbQHn0C>.
- [130] M.D. Plummer и L. Lovász. *Matching Theory*. ISSN. Връзка към нея: <https://books.google.bg/books?id=mycZP-J344wC>. Elsevier Science, 1986. ISBN: 9780080872322.

- [131] Pedro J. de Rezende и др. *Engineering Art Galleries*. Достъпна онлайн на <https://arxiv.org/abs/1410.8720>. 2014. DOI: 10.48550/ARXIV.1410.8720.
- [132] Romeo Rizzi. “A Short Proof of Königs’s Matching Theorem”. В: *J. Graph Theory* 33.3 (март 2000). Достъпна свободно онлайн на <https://www.lamsade.dauphine.fr/~cornaz/Enseignement/EcoleDoctorale/Rizzi2000.pdf>, с. 138—139. ISSN: 0364-9024.
- [133] Neil Robertson и др. “Efficiently Four-Coloring Planar Graphs”. В: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC ’96. Вариант на статията е достъпен на https://www.researchgate.net/publication/2430062_A_New_Proof_Of_The_Four-Colour_Theorem. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, с. 571—575. ISBN: 0897917855. DOI: 10.1145/237814.238005.
- [134] Dan Romik. “Stirling’s Approximation for $n!$: The Ultimate Short Proof?” В: *The American Mathematical Monthly* 107.6 (2000). Достъпна онлайн на <https://www.math.ucdavis.edu/~romik/data/uploads/papers/stirling.pdf>, с. 556—557.
- [135] Frank Ruskey. *Combinatorial Generation*. Достъпна онлайн на <http://page.math.tu-berlin.de/~felsner/SemWS17-18/Ruskey-Comb-Gen.pdf>. 2003.
- [136] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st edition. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201895390.
- [137] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0-471-11709-9.
- [138] Uwe Schöning. “A low and a high hierarchy within NP”. В: *Journal of Computer and System Sciences* 27.1 (1983). Достъпна онлайн на <https://www.sciencedirect.com/science/article/pii/0022000083900272>, с. 14—28. ISSN: 0022-0000. DOI: [https://doi.org/10.1016/0022-0000\(83\)90027-2](https://doi.org/10.1016/0022-0000(83)90027-2).
- [139] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.
- [140] Alexander Schrijver. *On the History of the Shortest Path Problem*. Достъпна онлайн на https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf. 2012.
- [141] Robert Sedgewick и Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley-Longman, 1996, с. I—XV, 1—492. ISBN: 978-0-201-40009-0.
- [142] Robert Sedgewick и Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011, с. I—XII, 1—955. ISBN: 978-0-321-57351-3.
- [143] Michael Sipser. *Introduction to the theory of computation: second edition*. 2nd edition. Boston: PWS Pub., 2006. ISBN: 978-0534950972.
- [144] Steven S. Skiena. *The Algorithm Design Manual*. 2nd edition. Springer Publishing Company, Incorporated, 2008. ISBN: 9781848000698.
- [145] L. J. Stockmeyer и A. R. Meyer. “Word Problems Requiring Exponential Time(Preliminary Report)”. В: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC ’73. Свободно достъпна на <https://dl.acm.org/doi/10.1145/800125.804029>. Austin, Texas, USA: Association for Computing Machinery, 1973, с. 1—9. ISBN: 9781450374309. DOI: 10.1145/800125.804029.
- [146] Williard Stone. “Abacists versus Algorists”. В: *Journal of Accounting Research Journal of Accounting Research* 10.2 (1972). Достъпна онлайн на <https://www.jstor.org/stable/2490013>, с. 345—350. DOI: 10.2307/2490013.

- [147] Evan Sultanik, Ali Shokoufandeh и William C. Regli. “Dominating sets of agents in visibility graphs: distributed algorithms for art gallery problems.” В: *AAMAS*. Под ред. на Wiebe van der Hoek и др. Достъпна онлайн на https://www.ifaamas.org/Proceedings/aamas2010/resources/_fullpapers.html. IFAAMAS, 2010, с. 797—804. ISBN: 978-0-9826571-1-9.
- [148] Gábor Szárnyas. *Graphs and matrices: A translation of “Graphok és matrixok” by Dénes Kőnig (1931)*. Достъпна свободно онлайн на <https://arxiv.org/abs/2009.03780>. 2020.
- [149] Robert Endre Tarjan. “Edge-Disjoint Spanning Trees and Depth-First Search”. В: *Acta Informatica* 6.2 (юни 1976). По-стара версия на статията е свободно достъпна като technical report на <http://i.stanford.edu/pub/cstr/reports/cs/tr/74/455/CS-TR-74-455.pdf>, с. 171—185. ISSN: 0001-5903. DOI: 10.1007/BF00268499.
- [150] B.A. Trakhtenbrot. “A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms”. В: *Annals of the History of Computing* 6.4 (1984). Достъпна онлайн на <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.408.7627&rep=rep1&type=pdf>, с. 384—400. DOI: 10.1109/MAHC.1984.10036.
- [151] Alan M. Turing. “On Computable Numbers, with an application to the Entscheidungsproblem”. В: *Proceedings of the London Mathematical Society*. 2-я пор. 42 (1936). Достъпна онлайн на http://www.dna.caltech.edu/courses/cs129/caltech_restricted/Turing_1936_IBID.pdf, с. 230—265.
- [152] Hui Wang и др. “Boson Sampling with 20 Input Photons and a 60-Mode Interferometer in a 10^{14} state spaces”. В: *Physical Review Letters* 123.25 (дек. 2019). Достъпна онлайн на <https://arxiv.org/abs/1910.09930>. ISSN: 1079-7114. DOI: 10.1103/physrevlett.123.250503.
- [153] Y. Wang. *The Goldbach Conjecture*. Series in pure mathematics. World Scientific, 2002. ISBN: 9789812776600. URL: <https://books.google.com/books?id=VAY9nTreXkcC>.
- [154] Stephen Warshall. “A Theorem on Boolean Matrices”. В: *J. ACM* 9.1 (ян. 1962). Достъпна онлайн на <https://dl.acm.org/doi/10.1145/321105.321107>, с. 11—12. ISSN: 0004-5411. DOI: 10.1145/321105.321107.
- [155] Lutz M. Wegner. *Sorting – The Turku Lectures*. Достъпна онлайн на http://tucs.fi/publications/attachment.php?fname=bWegner_LutzMx14a.full.pdf. 2014.
- [156] Andrew John Wiles. “Modular Elliptic Curves and Fermat’s Last Theorem”. В: *ANNALS OF MATH* 141 (1995). Достъпна онлайн на <http://math.stanford.edu/%7Elekheng/flt/wiles-small.pdf>, с. 141.
- [157] J. W. J. Williams. “Algorithm 232: Heapsort”. В: *Communications of the ACM* 7.6 (1964), с. 347—348.
- [158] Gerhard J. Woeginger и Zhongliang Yu. “On the Equal-Subset-Sum Problem”. В: *Inf. Process. Lett.* 42.6 (юли 1992), с. 299—302. ISSN: 0020-0190. DOI: 10.1016/0020-0190(92)90226-L.
- [159] Denny Zhou, Jiayuan Huang и Bernhard Schölkopf. “Learning with Hypergraphs: Clustering, Classification, and Embedding”. В: *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*. Свободно достъпна на <https://www.microsoft.com/en-us/research/publication/learning-hypergraphs-clustering-classification-embedding>. MIT Press, септ. 2007, с. 1601—1608.