Лекции по Алгоритми

работна версия, 2023 година

Минко Марков

Copyright © 2014 – 2023 Минко Марков (Minko Markov)
All rights reserved.
Maple is a trademark of Waterloo Maple Inc.

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 За разликата между алгоритми и изчислителни задачи 14
1.1.4 Елементарни 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 28
1.2.2 Съвременни варианти на Евклидовия алгоритъм. 30
1.3 Как описваме алгоритми 30
1.3.1 Псевдокод 30
1.3.2 Други видове описания на алгоритми 31

2 Анализ на алгоритми. 34
2.1 Анализ на коректността 35
2.1.1 Доказателство за финитност 36
2.1.2 Верификация на рекурсивни алгоритми: по индукция 39
2.1.2.1 Нараставане с единица 39
2.1.2.2 Максимум на несортиран масив, рекурсивен 40
2.1.2.3 Максимум на битоничен масив 41
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 50
2.1.3.3 Евклидовият алгоритъм в итеративен вариант 54
2.1.3.4 Alg2SUM 54
2.1.3.5 Alg3SUMcount 56
2.1.3.6 Една задача върху масиви 62
2.2 Анализ на сложността 64
2.2.1 Големина на входа 65
2.2.2 Въведение в сложността по време 66
2.2.3 Определение на сложността по време 70

2.2.4 Сложност по памет . . . 76
2.2.5 Проблеми със сложността по време, до които води нашият модел . . . 77
2.3 Асимптотични нотации . . . 78
2.3.1 Опит за намиране на точната сложност по време на прости алгоритми 78
2.3.2 Отказ от точна оценка и търсене на приблизителна оценка на сложността 81
2.3.3 Асимптотична нотация Θ . . . 83
2.3.4 Други асимптотични нотации: O, Ω, o и ω . . . 88
2.3.5 Релации $\asymp$, $\preceq$, $\prec$, $\succeq$ и $\succ$ и някои техни свойства . . . 92
2.3.6 Асимптотични еквивалентности на често срещани функции . . . 104
2.3.7 Задачи с решения върху асимптотични сравнения . . . 114
2.3.7.1 Свойства на релациите на асимптотични сравнения . . . 114
2.3.7.2 Асимптотични сравнения на двойки функции . . . 119
2.3.7.3 Подреждане на функции по асимптотично нарастване . . . 122
2.3.8 Релациите на асимптотични сравнения при целочислена променлива . 136
2.3.9 Асимптотичните нотации и релациите на асимптотични сравнения . . . 137
2.3.10 Релациите $\preceq$ и $\succeq$ като преднаредби . . . 138
2.3.11 Асимптотични сравнения на функции на повече от една променлива . 145
2.3.12 Релация $\sim$: друга релация на асимптотична еквивалентност . . . 145
2.4 Примери за изчисляване на сложност по време на итеративни алгоритми . . . 145
2.4.1 Алгоритми с единичен цикъл . . . 145
2.4.2 Алгоритми с вложени цикли . . . 146
2.4.3 Двоично търсене, итеративен вариант . . . 151
2.5 Винаги ли е лоша високата сложност . . . 154

3 Рекурентни уравнения. 158
3.1 Фундамент . . . 158
3.1.1 Определение . . . 158
3.1.2 Обобщения . . . 159
3.1.3 Рекурентни уравнения и рекурсивни алгоритми . . . 160
3.1.4 Класификации на рекурентните уравнения . . . 161
3.1.5 Винаги ли променливата в рекурентно уравнение е целочислена . . . 165
3.1.6 Решаване на рекурентни уравнения . . . 166
3.1.7 Рекурентни уравнения в изследването на алгоритми . . . 168
3.2 Методи за решаване на рекурентни уравнения . . . 169
3.2.1 Чрез развиване . . . 169
3.2.2 Чрез дървото на рекурсията . . . 181
3.2.3 По индукция . . . 185
3.2.4 Чрез Мастър теоремата (МТ) . . . 195
3.2.5 Чрез теоремата на Akra-Bazzi . . . 208
3.2.6 Чрез метода с характеристичното уравнение . . . 213

Част II Сортиране 236

4 Въведение. Елементарни сортиращи алгоритми. 237
4.1 За задачата Сортиране . . . 237
4.2 Защо сортирането е важно . . . 238
4.2.1 Двоично търсене . . . 239
4.2.2 Най-близки елементи . . . 242

4.2.3 Уникалност на елементите . . . 244
4.2.4 Откриване на еднакви елементи в два масива . . . 244
4.2.5 Мода . . . 245
4.2.6 Медиана . . . 250
4.2.7 2SUM, 3SUM, kSUM . . . 250
4.3 Анализ на сортиращи алгоритми. Стабилност. . . . 253
4.4 Елементарни Сортиращи Алгоритми . . . 255
4.4.1 Insertion Sort . . . 255
4.4.2 Selection Sort . . . 262

5 Двоична пирамида. Heapsort. Приоритетна опашка. 265
5.1 Двоични дървета и пирамиди . . . 265
5.1.1 Попълнено двоично дърво . . . 265
5.1.2 Адреси в попълнено двоично дърво . . . 272
5.1.3 Двоична пирамида . . . 277
5.1.4 Реализиране на пирамида чрез масив . . . 278
5.1.5 Подпирамида . . . 282
5.2 Построяване на пирамида . . . 284
5.2.1 Наивно построяване на пирамида . . . 284
5.2.2 Бързо построяване на пирамида: алгоритъм Build Heap . . . 288
5.2.2.1 Предварителни обяснения . . . 288
5.2.2.2 Неформално въвеждане на Heapify . . . 292
5.2.2.3 Heapify в итеративен вариант . . . 292
5.2.2.4 Heapify в рекурсивен вариант . . . 294
5.2.2.5 Алгоритъм Build Heap . . . 297
5.3 Сортиращ алгоритъм Heapsort . . . 298
5.4 Приоритетни опашки . . . 301
5.4.1 Абстрактен Тип Данни (АТД) . . . 301
5.4.2 Приоритетна опашка: вид АТД . . . 302
5.4.3 Реализация на приоритетни опашки с двоични пирамиди . . . 303
5.4.4 Функцията Increase-Key . . . 305

6 Алгоритмична схема Разделяй-и-Владей. Mergesort и Quicksort. 306
6.1 Разделяй-и-Владей . . . 306
6.2 Примери за ефикасни Разделяй-и-Владей алгоритми . . . 307
6.2.1 Най-близки елементи 2D . . . 307
6.2.2 Избор на елемент по големина . . . 321
6.3 Mergesort . . . 334
6.4 Quicksort . . . 340
6.4.1 Имплементация на фазата **Разделяй** чрез Partition–Hoare . . . 341
6.4.2 Имплементация на фазата **Разделяй** чрез Partition–Lomuto . . . 345
6.4.3 Самият Quicksort . . . 348
6.4.4 Сложност по време на Quicksort . . . 349

7 Сортиране в линейно време. Counting Sort. Radix Sort. 353
7.1 Въведение . . . 353
7.2 Counting Sort . . . 353
7.3 Radix Sort . . . 358

Част III Алгоритми върху графи 361

8 Въведение в алгоритмите върху графи. Обхождания на графи. 362
8.1 Рекапитулация на графите . . . 362
8.2 Обхождания на графи . . . 367
8.3 Обща схема за обхождане . . . 368
8.4 BFS . . . 369
8.4.1 Псевдокод на BFS от един стартов връх . . . 371
8.4.2 Псевдокод на BFS, обхождащ целия граф . . . 376
8.4.3 Приложения на BFS . . . 377
8.5 DFS . . . 377
8.5.1 Неформално въведение и сравнение с BFS . . . 377
8.5.2 Псевдокод на DFS . . . 379
8.5.3 Свойства на DFS . . . 380
8.5.4 Приложения на DFS . . . 384

9 Топосортиране, силно св. комп., срязващи върхове, мостове. 387
9.1 Фундамент . . . 387
9.2 Алгоритъм на Tarjan за топологическо сортиране . . . 390
9.3 Алгоритъм на Kahn за топологическо сортиране . . . 393
9.4 Алгоритъм на Kosaraju за намиране на силно свързаните компоненти . . . 396
9.5 Алгоритми за намиране на срязващите върхове и на мостовете . . . 405
9.5.1 Фундамент . . . 405
9.5.2 Алгоритъм за ефикасно намиране на срязващите върхове . . . 408
9.5.3 Алгоритъм за за ефикасно намиране на мостовете . . . 413

10 Минимални покриващи дървета. Алгоритми на Prim и Kruskal. 416
10.1 Фундамент . . . 416
10.2 Алчни алгоритми . . . 419
10.3 Алгоритъм на Prim . . . 424
10.3.1 Базов вариант на алгоритъма на Prim . . . 425
10.3.2 Изтънчен вариант на алгоритъма на Prim . . . 427
10.4 Алгоритъм на Kruskal . . . 431
10.5 Union-Find структури данни . . . 434
10.5.1 Два крайни подхода, които не вършат работа . . . 434
10.5.2 Реализация на разбиване чрез ориентирана гора . . . 436

11 Най-къси пътища. Dijkstra, дагове, Bellman-Ford. 445
11.1 Фундамент . . . 445
11.1.1 Базови дефиниции . . . 445
11.1.2 Варианти на задачата . . . 447
11.1.3 Най-къс път се състои от най-къси подпътища . . . 449
11.1.4 Дърво на най-късите пътища . . . 449
11.1.5 Пак за отрицателните тегла . . . 452
11.1.6 Релаксация . . . 455
11.2 Алгоритъмът на Dijkstra в два варианта . . . 458
11.3 Най-къси пътища в дагове . . . 461
11.4 Алгоритъмът на Bellman-Ford . . . 464

Част IV Динамично програмиране 466

12 Схемата Динамично Програмиране. Примери. 467
12.1 Въведение с примери . . . 467
12.1.1 Задачата за опашката пред касата . . . 467
12.1.2 Числата на Fibonacci . . . 473
12.2 Фундамент . . . 474
12.3 Най-къс път за всяка двойка върхове . . . 478
12.3.1 Алгоритъм, реализиращ матрично умножение . . . 479
12.3.2 Алгоритъмът на Floyd-Warshall . . . 486
12.4 Задачи за броене на комбинаторни структури . . . 489
12.4.1 Биномни коефициенти . . . 490
12.4.2 Числа на Stirling от втори род . . . 493
12.4.3 Числа на Stirling от първи род . . . 495
12.4.4 Брой на целочислени разбивания . . . 497
12.5 Задачи за оптимални скобувания на редици (триъгълна таблица) . . . 501
12.5.1 Matrix-Chain Multiplication . . . 501
12.5.2 Minimum Weight Triangulation . . . 515
12.5.3 Факторизация при неасоциативно умножение . . . 522
12.5.4 Деривация в контекстно-свободна граматика . . . 525
12.6 Задачи за оптимални подмножества . . . 527
12.6.1 2-Partition . . . 527
12.6.2 2 Equal Sum Subsets (2-ESS) . . . 535
12.6.3 Задачата за раницата (Knapsack) . . . 538
12.6.3.1 Unbounded Knapsack . . . 539
12.6.3.2 0-1 Knapsack . . . 541
12.6.3.3 Bounded Knapsack . . . 542
12.7 Задачи за планиране (Scheduling) . . . 544
12.7.1 Задачата за безработния актьор (Interval Scheduling) . . . 544
12.7.1.1 Алчно решение . . . 545
12.7.1.2 Решение по схемата **Динамично Програмиране** . . . 550
12.8 Задачи върху редици и стрингове . . . 554
12.8.1 Longest Increasing Subsequence . . . 554
12.8.1.1 Дин. Прогр. със сложност $\Theta(n^2)$ . . . 555
12.8.1.2 Дин. Прогр. със сложност $\Theta(n \lg n)$ . . . 556
12.8.1.3 PATIENCESORT . . . 562
12.8.1.4 Building Bridges . . . 562
12.8.2 Longest Common Subsequence . . . 562
12.8.3 Sequence Alignment . . . 568
12.8.4 Edit Distance . . . 579
12.9 **NP**-трудни задачи върху дървета . . . 587
12.9.1 Минимално върхово покриване на дърво . . . 588
12.9.2 Минимално доминиращо множество на дърво . . . 594
12.10 Игри . . . 604
12.10.1 The Daisy Petals Game . . . 606
12.10.2 The Coins in a Line Game . . . 610
12.11 Мемоизация . . . 614
12.12 За границите на възможностите на динамичното програмиране . . . 617

Част V Долни граници 619

13 Долни граници върху сложност на задачи. 620
13.1 Въведение . . . 620
13.2 Дървета за вземане на решение . . . 622
13.2.1 Задачите THE BALANCE PUZZLE и THE TWELVE-COIN PUZZLE . . . 622
13.2.2 Долна граница $\Omega(n \lg n)$ за СОРТИРАНЕ . . . 626
13.2.2.1 Сортиране, базирано на сравнения . . . 626
13.2.2.2 Дървото на сравненията на INSERTION SORT за $n = 3$ . . . 627
13.2.2.3 Дървото на сравненията на SELECTION SORT за $n = 3$ . . . 632
13.2.2.4 Дървото на вземане на решение на произволен сортиращ алг. . . 633
13.2.2.5 Дървото за вземане на решение е изчислителен модел . . . 634
13.2.2.6 За структурата на сортиращите дървета за вземане на решение 635
13.2.2.7 Долна граница $\Omega(n \lg n)$ чрез дървета за вземане на решение . 637
13.2.2.8 Минималният брой на сравненията в най-добрия случай е $n - 1$ 637
13.2.3 Долна граница $\Omega(n \lg n)$ за УНИКАЛНОСТ НА ЕЛЕМЕНТИТЕ . . . 641
13.2.4 Долна граница $\Omega(\lg n)$ за ТЪРСЕНЕ . . . 645
13.2.5 Лампи и ключове . . . 646
13.3 Редукции . . . 652
13.3.1 Долна граница $\Omega(n \lg n)$ за НАЙ-БЛИЗКИ ЕЛЕМЕНТИ . . . 652
13.3.2 Долна граница $\Omega(n \lg n)$ за МОДА . . . 654
13.3.3 Долна граница $\Omega(n \lg n)$ за 2SUM във версия за броене . . . 654
13.3.4 Долна граница $\Omega(n \lg n)$ за INTERVAL SCHEDULING. . . . 655
13.3.5 Относителна долна граница: DEGENERACY TESTING е 3SUM-hard . . . 656
13.4 Аргументиране чрез противник (Adversary argument) . . . 658
13.4.1 Неформално обяснение . . . 658
13.4.2 Формално определение . . . 664
13.4.3 Долна граница $\lceil \log_2 n \rceil$ за намиране на число от $\{1, \ldots, n\}$ . . . 666
13.4.4 Долна граница $n - 1$ за намиране на максималния елемент . . . 667
13.4.5 Долна гр. $\lceil \frac{3n}{2} \rceil - 2$ за намиране на максималния и минималния елемент 669
13.4.6 Долна граница $\lfloor \frac{3n-2}{2} \rfloor$ за намиране на медиана . . . 675
13.4.7 Долна граница n^2 за установяване на свързаност на граф с $2n$ върха . 685
13.4.8 Долна гр. $2n - 1$ за търсене в сортиран по редове и колони, $n \times n$ масив 686
13.4.9 Задачата за яйцето . . . 692

Част VI Алгоритмична неподатливост 693

14 NP-пълнота и NP-трудност. 694
14.1 Въведение с пример и илюстрация . . . 694
14.2 Алгоритмична неподатливост . . . 697
14.3 Задачите като формални езици . . . 701
14.3.1 Всяка задача за разпознаване в някакъв смисъл е формален език . . . 701
14.3.2 Подходящи кодирания на задачи за разпознаване . . . 703
14.4 Детерминирани машини на Turing . . . 704
14.4.1 Фундамент . . . 704
14.4.2 Съответствие между разпознаване на езици и решаване на задачи за разпознаване от машини на Turing . . . 707
14.4.3 Пример за машина на Turing . . . 707

14.4.4 Машини на Turing-изчислители на стрингови функции . . . 711
14.4.5 Сложност по време на машина на Turing. Клас на сложност **P**. . . 712
14.4.6 Сложност по памет на машина на Turing. . . 714
14.5 Недетерминизъм и недетерминирани машини на Turing . . . 714
14.5.1 НМТ с размножаване . . . 714
14.5.2 НМТ-верификатор . . . 721
14.5.3 Еквивалентност на двете дефиниции на НМТ . . . 723
14.5.4 Какви нови възможности ни дава недетерминизмът . . . 726
14.5.5 Сложност по време на НМТ. Клас на сложност **NP**. . . . 726
14.6 Полиномиални редукции . . . 730
14.6.1 Фундамент . . . 731
14.6.2 Основни свойства . . . 732
14.6.3 Примери за полиномиални редукции . . . 736
14.6.4 Karp редукции срещу Turing редукции . . . 739
14.7 **NP**-пълнота . . . 745
14.8 **NP**-трудност . . . 746

15 SAT и теоремата на Cook. 748
15.1 Синтаксис на булевите формули . . . 748
15.2 Булеви функции . . . 749
15.3 Семантика на булевите формули. Удовлетворимост. . . . 749
15.4 Конюнктивни и Дизюнктивни нормални форми . . . 752
15.4.1 Дизюнктивна нормална форма . . . 752
15.4.2 Конюнктивна нормална форма . . . 754
15.4.3 Превръщане на формули в КНФ или ДНФ . . . 755
15.5 Satisfiablity (SAT) и моделиращата сила на КНФ . . . 757
15.5.1 Моделиране на судоку позиция чрез КНФ. . . . 757
15.5.2 Моделиране на ограничено поставяне на етикети чрез КНФ. . . . 763
15.5.3 Моделиране на наличие на Хамилтонов път чрез КНФ. . . . 766
15.6 SAT е **NP**-пълна . . . 767
15.7 SAT е началото на редици от доказателства на **NP**-пълнота . . . 777
15.8 Географията на **NP** . . . 778

16 Основни NP-пълни задачи. 782
16.1 Дървото на доказателствата на **NP**-пълнота. . . . 782
16.2 Редукциите. . . . 784
16.2.1 3SAT, Max2SAT и NAESAT . . . 784
16.2.2 MaxCut . . . 804
16.2.3 Vertex Cover . . . 808
16.2.4 Independent Set, Clique и Dominating Set . . . 813
16.2.5 0-1 Integer Programming . . . 820
16.2.6 3-Colorability, k-Colorability и Planar 3-Colorability . . . 825
16.2.7 3-Dimensional Matching (3DM) . . . 842
16.2.7.1 Съчетание – дефиниция, частни случаи и приложения . . . 842
16.2.7.2 Хиперграфите като обобщение на обикновените графи . . . 849
16.2.7.3 3DM – обобщение на Перфектно Съчетание . . . 852
16.2.7.4 3DM е неподатлива . . . 853
16.2.8 Hamiltonian Cycle, Ham. Path, Longest Cycle, Longest Path . 864
16.2.9 2-Partition . . . 876

16.2.10 Knapsack . . . 882
16.2.11 Bin Packing . . . 883
16.2.12 Multiprocessor Scheduling . . . 884
16.3 Когато принадлежността на задача към **NP** не е очевидна. . . . 886
16.3.1 ⋆-Free Regexp Ineq . . . 886
16.3.2 Art Gallery . . . 898

17 Заобикаляне на неподатливостта. 915
17.1 Силна и слаба **NP**-пълнота . . . 915
17.2 Апроксимиращи алгоритми . . . 915
17.3 Параметризирана сложност . . . 915

Благодарности 916

Библиография 917

Списък на фигурите

1.1 Dixit algorismi. 4
1.2 Вход, алгоритъм, изход. 15
1.3 Блок-схема на Евклидовия алгоритъм. 32

2.1 Видове сложност по време. 75
2.2 Графиките на n^2, $3n^2$ и $(2 + \sin n)n^2$. 85
2.3 Графиките на n, $n^{1+\sin n}$, 1 и n^2. 95
2.4 Графиките на $2^{2^{\lfloor n \rfloor}}$ и $2^{2^{\lceil n \rceil}}$. 96
2.5 $\boldsymbol{\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx}$. 105
2.6 $\boldsymbol{\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx}$. 106
2.7 $\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx = \int_1^4 \frac{4-\lfloor x \rfloor}{x} dx$. 106
2.8 $\int_1^n \frac{1}{x} dx$ като площ на район. 111
2.9 $\sum_{i=1}^{n-1} \frac{1}{i}$ като площ на район. 112
2.10 $\sum_{i=2}^{n} \frac{1}{i}$ като площ на район. 112
2.11 Графиките на n и $n^{1+\sin 2\pi n}$. 136
2.12 Диаграма на преднаредба. 139
2.13 Фактор-релацията на пълна преднаредба. 141
2.14 Преднаредба, която не е пълна. 142
2.15 Фактор-релация на преднаредба, която не е пълна. 143

3.1 Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. 182
3.2 Схема на дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. . . 182
3.3 Друга схема на дървото при $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. 183
3.4 Дърво на рекурсията при $T(n) = 2T(n-1) + 1$. 184
3.5 Дърво на рекурсията при $T(n) = T(n-1) + 2T(n-2) + 2$. 185
3.6 МТ: дърво на рекурсията при $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. 197

4.1 Блок-схема на Compute Mode. 246

5.1 Попълнено двоично дърво съгласно Определение 49. 268
5.2 Височините на върховете в попълнено двоично дърво T. 271
5.3 Изтриване на листата на попълнено двоично дърво. 272
5.4 Попълнено дърво с 14 върха и техните адреси. 280
5.5 Пирамида с 26 върха. 283
5.6 Линеаризацията на пирамида от Фигура 5.5. 283
5.7 Листата на пирамидата от Фигура 5.5. 283
5.8 Подпирамидата $A[\![3]\!]$. 283
5.9 Листата на $A[\![3]\!]$ в червено. 283
5.10 Почти-пирамидата, получена от пирамидата от Фиг. 5.5 с един добавен връх. 285

5.11 Илюстрация на работата на NAIVE BUILD HEAP. 285
5.12 Понякога по-изгодно коренът да бъде разменян с детето с по-малкия ключ. . 289
5.13 Ако разменяме връх с детето с по-голям ключ 290
5.14 След 4 размени на връх с детето с по-голям ключ получаваме пирамида. . . 290
5.15 След 2 размени на връх с детето с по-малък ключ получаваме пирамида. . . 291
5.16 Може разменянето на връх с по-малкото дете да доведе до $\Omega(n)$ разме́ни. . . 291
5.17 Пирамидата, получена от дървото на Фигура 5.16 с $\Omega(n)$ размени. 291
5.18 АТД приоритетна опашка като черна кутия. 303

6.1 Точките, които ще ползваме за пример в НАЙ-БЛИЗКИ ЕЛЕМЕНТИ 2D. . . . 308
6.2 Бисекторът ℓ. 312
6.3 Дегенеративни случаи. ℓ си остава бисектор. 313
6.4 $2d$-лентата. 315
6.5 $2d$-лентата в примера от Фигура 6.1. 316
6.6 Възможните други краища на къса прекосяваща отсечка са малко. 317
6.7 Прозорецът върху примера от Фигура 6.1. 318
6.8 Прозорецът може да обхваща най-много осем точки. 319
6.9 Диаграма на Hasse на данните след ред 9 на SELECT. 324
6.10 Диаграма на Hasse на данните след ред 11 на SELECT. 326
6.11 Диаграмата на Hasse след като m стане медианата на медианите. 328

8.1 Обхождане на дърво от BFS и от DFS. 379
8.2 Ориентиран граф със сдвоени ребра и неориентиран граф. 383
8.3 Различен брой ребра назад. 385

9.1 Даг и неговото топо-сортиране. 388
9.2 Ориентиран граф и неговият фактор-граф. 397
9.3 SCC-DUMMY при различни наредби π_1 и π_2 на върховете. 398
9.4 Транспонираният граф на графа от Фигура 9.2. 401
9.5 Различни наредби на V индуцират различни наредби на $\mathsf{SCC}(G)$. 402
9.6 Срязващ връх и срязващо ребро. 405
9.7 Разцепване на връх. 406
9.8 Блокове в графи. 407
9.9 Кога коренът на дървото на DFS е срязващ връх. 409
9.10 Кога вътрешен връх на дървото на DFS е срязващ връх. 409
9.11 В алгоритъма FCV-REC: когато променливата x съдържа вътрешен връх u. 412
9.12 В алгоритъма FCV-REC: черните върхове се прескачат. 413

10.1 Прехвърляне на връх от границата в дървото при Prim. 426
10.2 След Union$(1, 2)$ и Union$(3, 4)$. 437
10.3 Два варианта за Union$(1, 3)$ върху гората от Фигура 10.2. 437
10.4 Два варианта за Union$(1, 5)$ върху Вар. 1 от Фиг. 10.2. 438
10.5 Union by rank: правилен и неправилен. 439

12.1 Минимално върхово покриване в графа на Petersen. 588
12.2 Редуване на нива́ не дава минимално върхово покриване на дърво. 591
12.3 Намиране на максимална антиклика в дърво с 22 върха. 593
12.4 Минимално доминиращо множество в графа на Petersen. 595
12.5 $D(T)$ е м-во с мин. мощност измежду тези $k + 1$ м-ва. 601
12.6 $\widetilde{D}(T)$ е м-во с мин. мощност измежду тези две м-ва. 601

13.1 Схема за измервания за The Balance Puzzle. 623
13.2 Схема от измервания за The Twelve-coin Puzzle. 625
13.3 Дървото на сравненията на Insertion Sort върху $\langle a_1, a_2, a_3 \rangle$. 629
13.4 Дървото на сравненията на Insertion Sort не различава псевдо-пермутациите. 630
13.5 Дървото на сравненията на Selection Sort върху $\langle a_1, a_2, a_3 \rangle$. 632
13.6 Не е достатъчно всяка двойка да е сравнена някъде в дървото, за да бъдат различавани пермутациите. 635
13.7 Пример за даг на сравненията при гарантирано уникални елементи. 640
13.8 Даг на сравненията с Хамилтонов път. 640
13.9 В най-добрия случай, с две сравнения се намира повтаряне на елементи. . . . 642
13.10 Пример за даг на сравненията при възможни повтаряния на елементи. 643
13.11 Да-екземпляр на 3SUM поражда Да-екземпляр на Degeneracy Testing. 657
13.12 Аргументация с противник: два алгоритъма. 664
13.13 Диаграма на състоянията и преходите при сравняване на числа. 671
13.14 Няма Хамилтонов път, но a_2 е в средата на всяко топо-сортиране. 676
13.15 Единствена медиана е сравнима с всички елементи. 678
13.16 Полезни и безполезни ребра за намиране на медианата. 679

14.1 Машина на Turing приема палиндром. 708
14.2 Машина на Turing събира числа. 711
14.3 Композиция на машини-изчислители и на съответните функции. 712
14.4 Дървото на конфигурациите на недетерминирана машина на Turing. 716
14.5 Граф, в който търсим с връщане Хамилтонов цикъл. 719
14.6 Сравнение на търсене с връщане с недетерминирано търсене. 720
14.7 Машина-верификатор симулира машина с размножаване. 724
14.8 Композицията на МТ, илюстрираща композиция на редукции. 733
14.9 Схема на полиномиална редукция $L_1 \leqslant_p L_2$. 735
14.10 Схема на Turing редукция $L_1 \leqslant_T L_2$. 741
14.11 Turing редукция Hamiltonian Cycle към Hamiltonian Path. 742
14.12 Karp редукция Hamiltonian Cycle към Hamiltonian Path. 744

15.1 Да-екземпляр на Restricted Label Placement. 763
15.2 Не-екземпляр на Restricted Label Placement. 764
15.3 Началното съдържание на лентата на $M(x)$ в теоремата на Cook. 771
15.4 Пространство-времето в теоремата на Cook. 772
15.5 Географията на **NP** в светлината на теоремата на Ladner. 779

16.1 Дървовидната структура на доказателствата за **NP**-трудност. 782
16.2 2SAT формулата ϕ_1 и нейният импликационен граф G_1. 791
16.3 Импликационният граф G_1 от Фигура 16.2, нарисуван по-прегледно. 791
16.4 Импликационният граф от Фигура 16.2 след махане на $(x_2 \vee \overline{x_4})$ от ϕ_1. . . . 792
16.5 Неудачен имликационен граф. 792
16.6 Фактор-графът на графа от Фигура 16.4. 797
16.7 Топо-сортиране на фактор-графа от Фигура 16.6. 797
16.8 Пример за NAESAT $\leqslant_p$ MaxCut. 807
16.9 И двата графа от Vertex Cover $\leqslant_p$ Dominating Set. 816
16.10 Оптимумът на ILP не опт. на LP след приближение. 822
16.11 Първият вид джаджи в редукцията 3SAT $\leqslant_p$ 3-Colorability. 825
16.12 Частично 3-оцветяване на екземпляра от Фигура 16.11. 826
16.13 Да-екземпляр на 3SAT и съответният Да-екземпляр на 3-Colorability. . 829

16.14 Превръщане на непланарен в планарен граф с добавяне на връх. 831
16.15 3-оцветяванията на джаджата от 3-Color $\leqslant_p$ Planar 3-Color. 835
16.16 Граф G и максимално по включване съчетание M в него. 842
16.17 Максимално по мощност съчетание M' в G от Фигура 16.16. 842
16.18 Теоремата на Hall следва от теоремата на Kőnig. 848
16.19 Пример за хиперграф. 849
16.20 Двуделният граф, описващ хиперграфа H от Фигура 16.19. 851
16.21 Максимално съчетание в триделен хиперграф. 852
16.22 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): звездите. 855
16.23 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): короните. 856
16.24 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): звездите и короните заедно. . . 857
16.25 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): слагаме една тройка в съчетанието.857
16.26 3SAT $\leqslant_p$ 3DM върху екз. от (16.10): u-тройките на $A_{1,1}$ са задължителни. . 858
16.27 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): забранени тройки. 860
16.28 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): възможност за $A_{2,2}$. 860
16.29 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): възможност за $A_{2,3}$. 861
16.30 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): от всяка корона влиза тройка. . 862
16.31 Основното приспособление във Vertex Cover $\leqslant_p$ Hamiltonian Path. . . . 865
16.32 Трите начина за минаване на Хамилтонов цикъл през джаджата. 866
16.33 Веригата p_z през всички z-страни на приспособления. 867
16.34 Петте приспособления от примера за Vertex Cover $\leqslant_p$ Hamiltonian Cycle.868
16.35 Добавянето на ребрата от E'' във Vertex Cover $\leqslant_p$ Hamiltonian Cycle. . 869
16.36 Веригите във Vertex Cover $\leqslant_p$ Hamiltonian Cycle. 869
16.37 Vertex Cover $\leqslant_p$ Hamiltonian Cycle: окончателният J. 870
16.38 Заменяне на z-страната с Хамилтонов път между z-краищата. 872
16.39 Двете вериги от P_u в примера. 873
16.40 Сега вече веригите минават през всички върхове на джаджи. 873
16.41 Има Хамилтонов цикъл! 874
16.42 Multiprocessor scheduling като подреждане в стекове. 885
16.43 Географията на **NP** и **coNP**. 896
16.44 Това е Turing, а не Karp, редукция. 897
16.45 Вариантите на Art Gallery имат значение за оптимума. 898
16.46 n-ъгълник, чието охраняване иска $\left\lfloor \frac{n}{3} \right\rfloor$ охранители. 899
16.47 Пример за Vertex Cover $\leqslant_p$ Art Gallery: започваме с графа на Petersen. 902
16.48 Vertex Cover $\leqslant_p$ Art Gallery: десетоъгълникът. 902
16.49 Изрязваме ϵ около върховете и получаваме отвори. 903
16.50 Построяваме алея 1-2. 904
16.51 Всички алеи през отвора 1. Те са видими от точка 1′. 905
16.52 Всички алеи. 906
16.53 Запречваме алеите и получаваме трапéци. 907
16.54 Конструираният P е десетоъгълник с коридори. 908
16.55 Група коридори 5. 909
16.56 Десет охранители—в жълтите точки—са достатъчни. 910
16.57 Шест охранители също са достатъчни. 911

Списък на таблиците

8.1 Сложностите на действията върху графи при двете представяния. 365

12.1 Таблица за изчисляване на биномни коефициенти. 491
12.2 Таблица за изчисляване на числа на Stirling от втори род. 495
12.3 Таблица за изчисляване на числа на Stirling от първи род. 497
12.4 Таблица за изчисляване на броеве на целочислени разбивания. 500
12.5 Печеливши и губещи целочислени разбивания. 608

13.1 Стратегията на противника при намиране максимум и минимум. 672

14.1 Машина на Turing M_{pal}, която решава задачата Палиндроми. 708

15.1 Променливите в редукцията в Теоремата на Cook. 773

16.1 Сложност на базови задачи върху КНФ/ДНФ. 896

Списък на допълненията

1 За произхода на думата "алгоритъм" . . . 4
2 Не може ли изч. задача да е функция $\Pi : \mathcal{I} \to 2^{\mathcal{S}}$? . . . 7
3 Функциите са повече от алгоритмите! . . . 14
4 Изчислителни модели . . . 16
5 On-line и off-line алгоритми . . . 20
6 Алгоритмичната нерешимост на Halting Problem . . . 21
7 Рандомизирани алгоритми . . . 23
8 За разликата между *ефективност* и *ефикасност* . . . 34
9 Когато коректността не е стопроцентова. . . . 35
10 Не-финитност и частични функции . . . 36
11 Други ресурси освен времето и паметта . . . 64
12 Принцип на Landauer . . . 68
13 P_n: числата на Bell с наредба . . . 72
14 Сл. по време ограничава сл. по памет отгоре и отдолу . . . 76
15 Друга интерпретация на появата на Θ вляво . . . 86
16 Нотациите O и o в анализа . . . 91
17 За преднаредбите . . . 138
18 Няколко рекурентни уравнения чрез развиване . . . 174
19 Няколко рекурентни уравнения по индукция . . . 191
20 Решение на $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$ . . . 198
21 Условието за регулярност в MT влече **Случай 3** . . . 202
22 Ограничена MT следва от метода с хар-чното у-ние . . . 203
23 Едно разширение на Теорема 27 . . . 206
24 Доказателство на теоремата на Akra-Bazzi . . . 209
25 Извеждане на решението на линейните рек. у-ния . . . 214
26 Когато има комплексни характеристични корени . . . 233
27 Коректността на двоичното търсене . . . 240
28 Коректността на Compute Mode . . . 245
29 Неформално за доказателствата за коректност . . . 256
30 За броя на пирамидите на n върха . . . 280
31 Heapsort има сложност $\Theta(n \lg n)$ в най-добрия случай . . . 300
32 Как да изчисляваме индексите на елементите на B . . . 310
33 Когато константата в алгоритъма Select не е $\mathbf{5}$ . . . 327
34 Най-лош случай за първото викане в Select . . . 331
35 Бързо намиране на броя на инверсиите в масив. . . . 338
36 За избора на pivot в Quicksort. . . . 341
37 Разликата между BFS и DFS, илюстрирана с шах . . . 378
38 За произхода на "топологическо сортиране". . . . 388
39 Команда `tsort`. . . . 389

40 Една транспортна задача с алчно решение . . . 419
41 Частична наредба $\sqsubseteq$ върху разбиванията . . . 433
42 Генерирането на всички най-къси пътища е неефикасно. . . . 448
43 За структурата на най-дългите пътища . . . 453
44 Намиране на арбитражи чрез отрицателни цикли . . . 454
45 За подобието между МПД Prim2 и SSShP Dijkstra2. . . . 459
46 Колко са булевите вектори без съседни единици . . . 468
47 Не всеки Разделяй-и-Владей води до Дин. Прогр. . . . 475
48 За произхода на "Динамично Програмиране" . . . 476
49 Имплементацията на `binomial(n,k)` в Maple(TM) . . . 491
50 За числата на Catalan . . . 504
51 Броят на триангулациите на изпъкнал n-ъгълник . . . 516
52 Още за псевдополиномиалната сложност . . . 532
53 Алчността на алчните алгоритми: глупава или умна . . . 547
54 Максимална антиклика върху интервални графи . . . 548
55 За алгоритъма на Needleman и Wunsch . . . 578
56 Edit Distance срещу Sequence Alignment . . . 580
57 Vertex Cover и Edge Cover са различни задачи . . . 590
58 Долни граници, задачи и изчислителни модели . . . 621
59 Защо допуснахме, че a_1, a_2 и a_3 са уникални? . . . 629
60 Thinking outside the box – добро или лошо? . . . 647
61 $f(n) \neq \Omega(g(n))$ не влече $f(n) = o(g(n))$ . . . 653
62 Морски шах с дявола . . . 660
63 Безсмислено за макс&мин сравнение е смислено за сорт. . . . 674
64 Накратко за историята на неподатливостта . . . 698
65 Backtracking и дървото на конфигурациите на НМТ . . . 717
66 Защо $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ е толкова важен . . . 728
67 За името "Traveling Salesman Problem" . . . 736
68 Удовлетворимост на всякакви формули с булев смисъл . . . 750
69 Bounded Halting Problem е **NP**-пълна задача . . . 767
70 2SAT $\in \mathbf{P}$ . . . 788
71 Намиране на хроматичното число чрез Maple (TM) . . . 832
72 Едно приложение на Съчетание в общите графи . . . 843
73 Теореми на Kőning и Hall . . . 845
74 За броя на графите и хиперграфите . . . 851
75 Задачи-допълнения и клас на сложност **coNP** . . . 891
76 Point Visibility Graphs . . . 900

Списък на теоремите

1 Halting Problem е алгоритмично нерешима . . . 21
2 Голяма Теорема на Fermat . . . 37
3 Коректността на алгоритъма Нарастване с единица . . . 39
4 Коректността на алгоритъма Max Unsorted, rec . . . 40
5 Коректността на алгоритъма Max Bitonic . . . 42
6 Коректността на алгоритъма Euclid, rec . . . 44
7 Коректността на алгоритъм за сумата от елементите на масив . . . 46
8 Коректността на алгоритъма Max Unsorted, iter . . . 49
9 Коректността на Alg2SUM . . . 55
10 Коректността на Alg3SUMcount . . . 56
11 Коректността на алгоритъма AlgDist . . . 62
12 Различните възможности при асимптотично сравняване на функции . . . 95
13 Повдигането на константна степен запазва $\asymp$ . . . 98
14 Водещото събираемо определя асимптотиката . . . 98
15 Експонирането запазва $\prec$ . . . 99
16 Логаритмуването запазва $\asymp$ . . . 100
17 $(\log_a f(n))^k \prec (f(n))^c$ . . . 101
18 $(f(n))^k \prec a^{(f(n))^c}$ . . . 102
19 Итерирането на логаритъма дава по-малко асимпт. нарастване . . . 104
20 Апроксимация на Stirling, слаба версия . . . 104
21 Асимптотиката на $\lg n!$ . . . 109
22 Асимптотиката на $\sum_{k=1}^{n} k \lg k$ . . . 110
23 Асимптотиката на $\binom{n}{n/2}$ . . . 110
24 Асимптотиката на H_n . . . 111
25 Сложността на двоичното търсене е логаритмична . . . 154
26 Решенията на клас рекурентни уравнения (Theorem 2.1 в [3]) . . . 195
27 Мастър теорема (Theorem 4.1 от [31, стр. 94]) . . . 196
28 Разширение на МТ . . . 206
29 Теорема на Akra-Bazzi (Theorem 1 от [96]) . . . 208
30 Решение на линейно хомогенно рекурентно уравнение . . . 213
31 Решение на линейно нехомогенно рекурентно уравнение . . . 214
32 Множество решения е фундаментално тстк дет. на Casorati е ненулева . . . 221
33 Theorem 2.18 от [39, стр. 72]: съществуване на фунд. множество от решения . . . 222
34 Принцип на суперпозицията . . . 223
35 Фундаментално множество от решения и конкретно решение . . . 223
36 Решенията на хомогенно лин. рек. у-ние образуват линейно пространство . . . 224
37 Theorem 2.23 от [39, стр. 77]: фунд. м-во от решения при кратни корени . . . 227
38 Разликата м-у две реш. на нехомогенното е реш. на хомогенното у-ние . . . 230
39 Theorem 2.40 от [39, стр. 84]: общото решение на нехомогенното уравнение . . . 230

40 Коректността на BinSearchIter . . . 240
41 Коректността на BinSearchRec . . . 241
42 Коректността на Insertion Sort . . . 261
43 Коректността на Selection Sort . . . 263
44 Броят на върховете с височина k в попълнено дърво. . . . 272
45 Коректността на наивното построяване на пирамида . . . 287
46 Коректността на бързото построяване на пирамида . . . 297
47 Бързото построяване на пирамида става в линейно време . . . 298
48 Counting Sort е стабилен сортиращ алгоритъм. . . . 357
49 Коректността на BFS (Theorem 22.5 в [31, стр. 599]) . . . 374
50 Скобова структура на интервалите при DFS . . . 380
51 Theorem 22.10 в [31, стр. 610] . . . 384
52 Граф е цикличен тстк DFS открива ребро назад . . . 385
53 НДУ за съществуване на топологическо сортиране . . . 387
54 Поне един източник и поне един сифон в даг . . . 388
55 Коректността на топологическото сортиране на Tarjan . . . 391
56 Коректността на топологическото сортиране на Kahn . . . 394
57 Коректността на алгоритъма на Kosaraju . . . 403
58 Theorem 23.1 от [31, стр. 627] (МПД теоремата) . . . 418
59 Коректността на Transport . . . 421
60 Коректността на МПД Prim2 . . . 428
61 Максимална височина на дърво след Union by rank . . . 442
62 Най-къс път се състои от най-къси подпътища. . . . 449
63 Коректността на алгоритъма на Dijkstra . . . 460
64 Коректността на алгоритъма DAG SSShP . . . 462
65 Най-рано приключване е оптимално за Interval Scheduling . . . 546
66 Обосновка на бързия алг. за Longest Increasing Sequence . . . 560
67 Vertex Cover и Independent Set са дуални . . . 589
68 В граф без изол. в-ве, всяко върхово покриване е доминиращо множество . . . 595
69 Височината на сортиращо дърво за вземане на решение е $\Omega(n \lg n)$ . . . 637
70 Сортиращите дървета сравняват всяка двойка съседни по големина елементи . 638
71 За дърветата на EU върху Да-екземплярите . . . 644
72 Дърветата за вземане на решение на EU различават пермутациите . . . 645
73 Задачата Degeneracy Testing е 3SUM-hard . . . 657
74 Намирането на максималния елемент иска поне $n-1$ сравнения. . . . 668
75 Експоненциална долна граница за $\mathsf{RSQ}(\Sigma)$ (Theorem 2.1, [104]) . . . 700
76 Машината на Turing M_{pal} решава задачата Палиндроми върху $\{0,1\}$. . . . 710
77 Верификатор може да симулира ефикасно машина с размножаване . . . 723
78 Машина с размножаване може да симулира ефикасно верификатор . . . 725
79 Полиномиалните редукции са транзитивни. . . . 733
80 Класът **P** и полиномиалните редукции . . . 735
81 **P** е минималният клас на еквивалентност на $\leqslant_p$ . . . 736
82 Halting Problem $\in$ **NP-h** . . . 747
83 Всяка булева формула може да се препише в ДНФ или КНФ . . . 756
84 Bounded Halting Problem $\in$ **NP-c** . . . 768
85 Теорема на Cook . . . 769
86 $\Pi \in \mathbf{NP} \wedge \Pi' \in \mathbf{NP\text{-}c} \wedge \Pi' \leqslant_p \Pi \rightarrow \Pi \in \mathbf{NP\text{-}c}$ . . . 777
87 Теорема на Ladner . . . 778
88 Екземпляр на 2SAT е удовлетворим тстк импл. граф е удачен . . . 794

89 Теорема на Kőning, 1931 г. 845
90 VERTEX COVER е податлива върху двуделните графи 846
91 Теорема на Hall, 1931 г. 847
92 Коректността на VERTEX COVER $\leqslant_p$ HAMILTONIAN CYCLE 871
93 Коректността на 3DM $\leqslant_p$ 2-PARTITION 881
94 $\star$-FREE REGEXP INEQ $\in$ **NP** 888
95 TAUTOLOGY $\in$ **coNP-c** 895
96 **P = NP** влече **NP = coNP** 896
97 **Π ∈ coNP-c ∧ Π ∈ NP** влече **NP = coNP** 896
98 ART GALLERY $\in$ **NP** 901

Списък на нотациите

1 $\forall n \nearrow$ . . . 83
2 “$\{x\}$” наместо “$x - \lfloor x \rfloor$” . . . 104
3 “H_n” означава n-тата парциална сума на хармоничния ред . . . 111
4 Фактор-релацията $R/_{\simeq}$ . . . 141
5 $N(u)$ и $N(U)$: съседите на u и U . . . 363
6 $\mathsf{SC}(G)$ и $\mathsf{SCC}(G)$: релацията на силна свързаност и нейните класове на екв. . . . 363
7 “adj[x]” означава списъка на съседство на връх x . . . 364
8 $G/_{\mathsf{SC}(G)}$ е фактор-графът на G . . . 396
9 $u \leadsto v$, $u \overset{p}{\leadsto} v$, $\mathsf{Paths}(u, v)$ . . . 445
10 “$\left\{ {n \atop k} \right\}$” означава числото на Stirling от втори род n-подмножество-k . . . 493
11 “$\left[{n \atop k} \right]$” означава числото на Stirling от първи род n-цикъл-k . . . 495
12 “$\left| {n \atop k} \right|$” означава броя на целочислените разбивания на n на k части . . . 497
13 $\uplus$ . . . 596
14 Бележим полиномиалните редукции с “$\leqslant_p$” . . . 732
15 $\mathsf{Var}(\phi)$ . . . 748
16 $\mathsf{Val}(\mathcal{Y})$ . . . 750
17 $t \models \phi$ . . . 751

Списък на изчислителните задачи

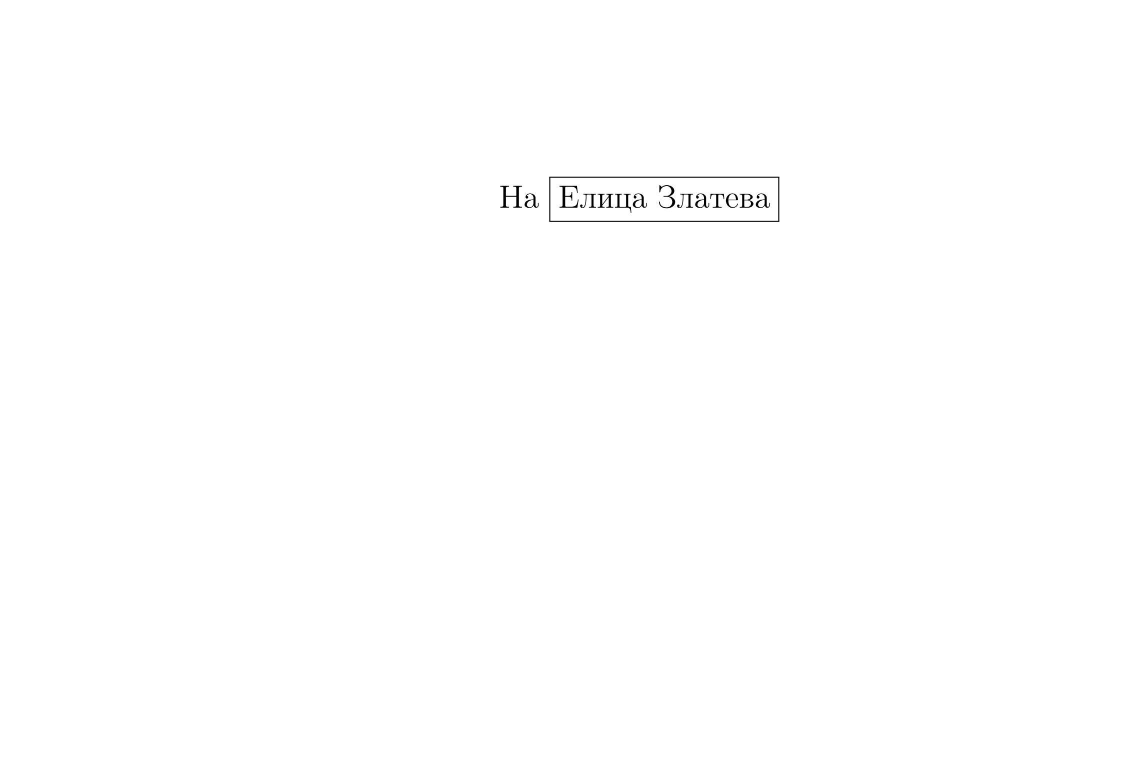
1 Свързаност на графи . . . 11
2 Hamiltonian Cycle . . . 11
3 Най-къс път в граф, педантично описание . . . 12
4 Най-къс път в граф, човешко описание . . . 12
5 Най-къс път в граф, версия за разпознаване . . . 12
6 Сортиране на числа . . . 13
7 Път в граф . . . 13
8 Halting Problem . . . 20
9 Сортиране . . . 237
10 Сортиране, абстрактна дефиниция . . . 238
11 Търсене, версия за разпознаване . . . 239
12 Търсене, версия за търсене . . . 239
13 Най-близки елементи, олекотният вариант . . . 242
14 Най-близки елементи, тежкият вариант . . . 242
15 Уникалност на елементите (Element Uniqueness / EU) . . . 244
16 Медиана . . . 250
17 kSUM, версия за разпознаване . . . 250
18 kSUM, версия за търсене . . . 251
19 Най-близки елементи 2D, олекотеният вариант . . . 308
20 Най-близки елементи 2D, тежкият вариант . . . 308
21 Избор на елемент по големина . . . 321
22 Минимално покриващо дърво (Minimum Spanning Tree) . . . 417
23 Транспортиране със загуби . . . 420
24 Single Pair Shortest Path . . . 447
25 Single Source Shortest Paths . . . 447
26 Single Destination Shortest Paths . . . 447
27 All Pairs Shortest Paths . . . 448
28 Longest Path между два върха, оптимизационна версия . . . 453
29 Matrix-Chain Multiplication . . . 507
30 Minimum Weight Triangulation . . . 516
31 Non-Associative Operation . . . 523

32 Context-Free Grammar Derivation, версия за разпознаване 525
33 2-Partition, версия за разпознаване 527
34 2-Partition, оптимизационна версия 527
35 2-ESS, версия за разпознаване 535
36 2-ESS, оптимизационна версия 535
37 Unbounded Knapsack 539
38 0-1 Knapsack 541
39 Bounded Knapsack 542
40 Interval Scheduling 545
41 Longest Increasing Sequence 554
42 Longest Common Sequence 563
43 Sequence Alignment 570
44 Edit Distance 581
45 Vertex Cover 588
46 Independent Set 589
47 Edge Cover 590
48 Independent Set, weigthed 593
49 Dominating Set 594
50 Dominating Set, weighted 603
51 Longest Directed Path 617
52 Мода 654
53 Degeneracy Testing 656
54 Палиндроми 702
55 TSP, оптимизационна версия 736
56 TSP, версия за разпознаване 736
57 Hamiltonian Path 742
58 Satisfiability (SAT) 757
59 Судоку 758
60 Restricted Label Placement 765
61 Bounded Halting Problem 768
62 Graph Isomorphism 780
63 Factoring, версия за търсене 780
64 Factoring, версия за разпознаване 780
65 3SAT 784
66 2SAT 788
67 Max2SAT, оптимизационна версия 799
68 Max2SAT, версия за разпознаване 799
69 NAESAT 802
70 MaxCut, версия за разпознаване 804

71 Vertex Cover, версия за разпознаване . . . 808
72 Independent Set, версия за разпознаване . . . 814
73 Clique, версия за разпознаване . . . 814
74 Dominating Set, версия за разпознаване . . . 814
75 0-1 Integer Programming . . . 823
76 3-Colorability . . . 825
77 k-Colorability . . . 830
78 Planar 3-Colorability . . . 830
79 3-Dimensional Matching (3DM) . . . 853
80 Longest Cycle, версия за разпознаване . . . 876
81 Longest Path, версия за разпознаване . . . 876
82 0-1 Knapsack, версия за разпознаване . . . 882
83 Bin Packing . . . 883
84 Multiprocessor Scheduling, версия за разпознаване . . . 885
85 ⋆-Free Regexp Ineq . . . 888
86 Свързаност на графи . . . 892
87 Hamiltonian Cycle . . . 892
88 Tautology . . . 895
89 Art Gallery . . . 901

Списък на редукциите

1 Hamiltonian Cycle $\leqslant_p$ TSP . . . 737
2 Четност $\leqslant_p$ Halting Problem . . . 738
3 Hamiltonian Cycle $\leqslant_p$ Hamiltonian Path . . . 743
4 Hamiltonian Path $\leqslant_p$ Hamiltonian Cycle . . . 745
5 Судоку $\leqslant_p$ SAT . . . 759
6 Restricted Label Placement $\leqslant_p$ SAT . . . 765
7 Hamiltonian Path $\leqslant_p$ SAT . . . 766
8 $\forall \Pi \in \mathbf{NP} : \Pi \leqslant_p$ Bounded Halting Problem . . . 769
9 $\forall \Pi \in \mathbf{NP} : \Pi \leqslant_p$ SAT . . . 770
10 SAT $\leqslant_p$ 3SAT . . . 784
11 3SAT $\leqslant_p$ Max2SAT . . . 799
12 3SAT $\leqslant_p$ NAESAT . . . 802
13 NAESAT $\leqslant_p$ MaxCut . . . 804
14 3SAT $\leqslant_p$ Vertex Cover . . . 808
15 Vertex Cover $\leqslant_p$ Independent Set . . . 814
16 Independent Set $\leqslant_p$ Clique . . . 814
17 Vertex Cover $\leqslant_p$ Dominating Set . . . 814
18 3SAT $\leqslant_p$ 0-1 Integer Programming . . . 823
19 3SAT $\leqslant_p$ 3-Colorability . . . 825
20 3-Colorability $\leqslant_p$ k-Colorability . . . 830
21 3-Colorability $\leqslant_p$ Planar 3-Colorability . . . 830
22 3SAT $\leqslant_p$ 3DM . . . 853
23 Vertex Cover $\leqslant_p$ Hamiltonian Cycle . . . 864
24 Hamiltonian Cycle $\leqslant_p$ Longest Cycle . . . 876
25 Hamiltonian Path $\leqslant_p$ Longest Path . . . 876
26 3DM $\leqslant_p$ 2-Partition . . . 876
27 2-Partition $\leqslant_p$ 0-1 Knapsack, разпознаване . . . 882
28 2-Partition $\leqslant_p$ Bin Packing . . . 884
29 2-Partition $\leqslant_p$ Multiprocessor Scheduling . . . 885
30 3SAT $\leqslant_p$ $\star$-Free Regexp Ineq . . . 890
31 Vertex Cover $\leqslant_p$ Art Gallery . . . 901



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 елементарните инструкции, 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 [20]. Steve Skiena [133, 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 [38] 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 [107]:

> *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)

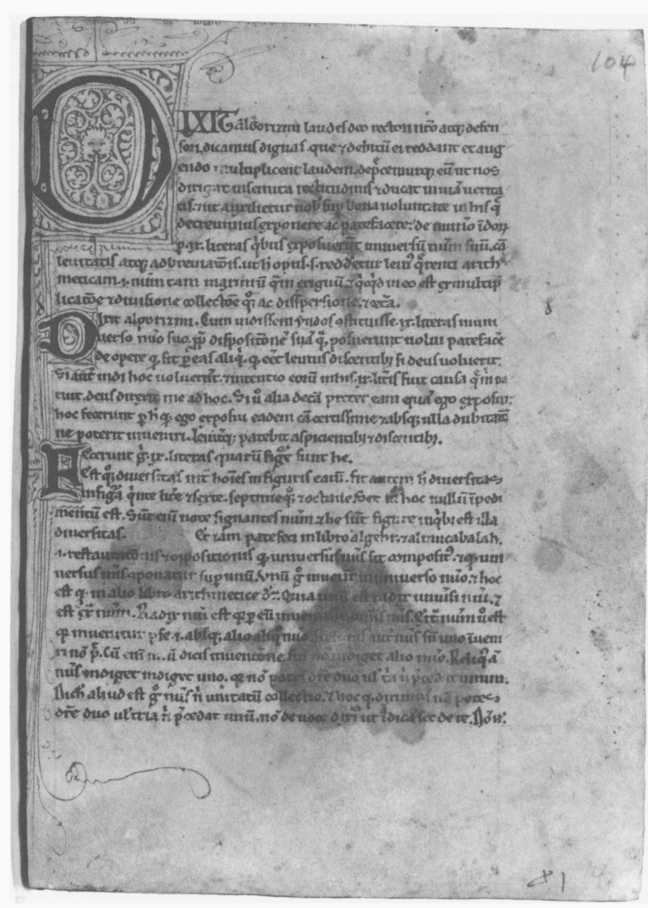
часовникови механизми, които ги извършват. Днес вместо "механична процедура" бихме казали "алгоритъм"[†], защото механичната имплементация не е съществена – процедурата може да се имплементира върху електрическа, електронна или квантово-фотонна машина [141]. Същественото е, че процедурата в никой момент 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 [84, 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 англоезичната 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 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 основовоположен 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 [76], свободно достъпна онлайн на *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". 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 не-позиционна 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 паралелно, 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 [135], 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".

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 съответства нула, едно или повече *решения*, на английски *solutions*.

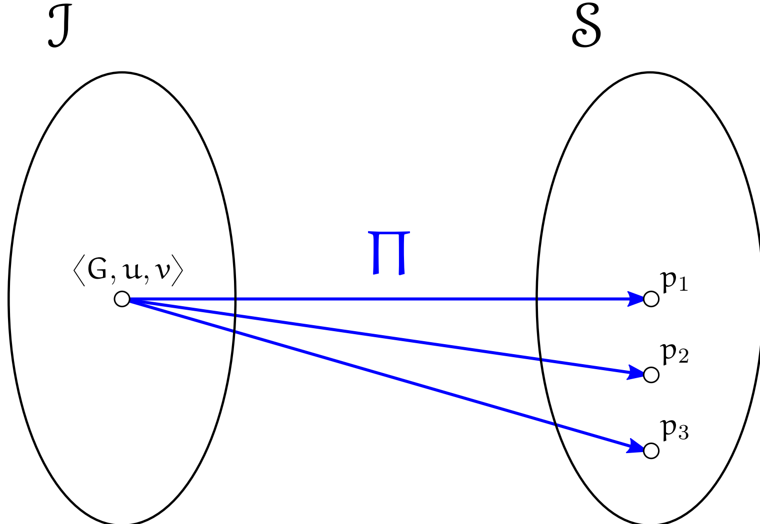
По принцип всяка интересна изчислителна задача има безброй екземпляри. Освен това, както ще видим в Лекция 2, общоприетата теория за анализ на алгоритми не е приложима, ако задачата има краен брой екземпляри. И така, разглеждаме само задачи с безкрайно много екземпляри. Обаче **изброимо** безкрайно. 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", за което понятие няма формална дефиниция, "изчислителна задача" има формална дефиниция.

Определение 1: Изчислителна задача

Изчислителна задача Π е двуместна релация $\Pi \subseteq \mathcal{I} \times \mathcal{S}$, 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 $(x, y) \in \Pi$, казваме, че y *е решение за* x.

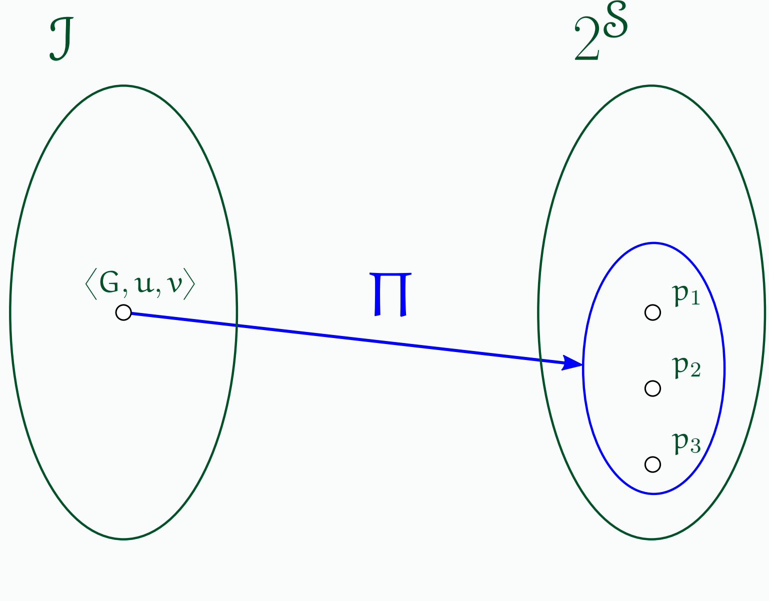
Следната фигура илюстрира нашата представа за изчислителната задача Път в граф като релация: първият домейн (екземплярите) е множеството от всички наредени тройки от неориентиран граф, един връх в него и друг връх в него, а вторият домейн (решенията) е множеството от пътищата (във всички графи). За всеки 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 $(\langle G, u, v \rangle, p_1) \in \Pi$, $(\langle G, u, v \rangle, p_2) \in \Pi$, $(\langle G, u, v \rangle, p_3) \in \Pi$ и в Π няма други наредени двойки с първи елемент $\langle G, u, v \rangle$. Ако си представим Π 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 Π е функция $\Pi : \mathcal{I} \to \mathcal{S}$.

Допълнение 2: Не може ли изч. задача да е функция $\Pi : \mathcal{I} \to 2^{\mathcal{S}}$?

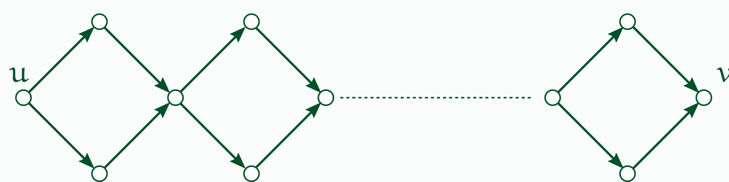
Човек може да се изкуши да дефинира "изчислителна задача" не като релация с първи домейн екземплярите и втори домейн, решенията, а като функция с домейн екземплярите и кодомейн степенното множество на множеството от решенията. По този начин може успешно да изрази това, че даден екземпляр има множество 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 Π 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 съвкупността** от всички тях.

2. Дори в простия пример с пътищата, те може да са прекалено много. Лесно може да измислим граф и два върха, такива че всички пътища от единия до другия са експоненциално много в размера на графа[a].



Очевидно е, че ако настояваме решението да е съвкупността от пътищата, а индивидуалните пътища не са решения, то никой алгоритъм няма да може да генерира решение 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.

[a]Същата конструкция се ползва и Допълнение 42.

Определение 2: Функция, съответна на релация

Нека е дадена изчислителна задача $\Pi \subseteq \mathcal{I} \times \mathcal{S}$. Нека $\star$ е елемент, непринадлежащ на $\mathcal{S}$. Нека $\mathcal{S}' = \mathcal{S} \cup \{\star\}$. *Функция, съответна на* Π, е всяка функция $f : \mathcal{I} \to \mathcal{S}'$, 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 $\{\textsc{True}, \textsc{False}\}$. Още може да кажем, че $\mathcal{S} = \{1, 0\}$.
Ако Π е задача за разпознаване, множеството от нейни екземпляри се разбива на две подмножества Π_Y и Π_N. Π_Y се състои точно от екземплярите, за които отговорът е TRUE; това са ДА-*екземплярите*. Π_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. Екземпляриите 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 и празно. 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 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. Накратко, може да се иска множеството от оптималните решения, а то е празното множество тстк оптимални решения няма.

Забележете, че множеството от оптималните решения е празно тстк множеството от допустимите решения е празно, понеже множеството от допустимите решения е крайно и всяко допустимо решение има цена. С други думи, ако съществува поне едно допустимо решение, то съществува и поне едно оптимално решение.

[a]Като 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 42).

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 y, а върху наредените двойки (x, y), където x е екземпляр, а y е решение за него. Ако приемем това, то приемаме, че едно и също решение y може да има различна цена в контекста на различни екземпляри, чието решение се явява. В тези лекции никога няма да е така. Примерно, 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 $\forall y \in Y : (x, y) \in \Pi$, където Π 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" 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.

Изч. 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.

Изч. 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, човешко 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 Π може да бъде превърната изкуствено в задача за разпознаване Π', 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 Π' *е задачата за разпознаване, съответна на* Π. Или че Π' *е задачата* Π *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 $w(p) \leqslant k$?

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 $A = (a_1, a_2, \ldots, a_n)$ от числа.
решение: Пермутация $(a'_1, a'_2, \ldots, a'_n)$ на числата от A, такава че $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

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 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 Глава 4 ще видим по-обща дефиниция на задачата.
[‡] Ако разполагаме с решение, да намерим цената на това решение е тривиално.

Има смисъл може 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 За разликата между алгоритми и изчислителни задачи

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

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 Π", а не за "реализация на Π".

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 $\mathcal{I}$ и кодомейн $\mathcal{S}$, като елементите от $\mathcal{I}$, 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 $f : \mathbb{N} \to \{0, 1\}$ е неизброимо безкрайно. Да допуснем, че функциите от този вид може да бъдат изброени като $f_0, f_1, f_2, \ldots$.

Да разгледаме редицата $f_0(0), f_1(1), f_2(2), \ldots$. 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 $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \ldots$, която също е редица от нули и единици, което означава, функция с домейн $\mathbb{N}$ и кодомейн $\{0, 1\}$. Но функцията $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \ldots$ се различава, **за всяко** $j \in \mathbb{N}$, от f_j върху поне една стойност, а именно j. Следователно, $\overline{f_0(0)}, \overline{f_1(1)}, \overline{f_2(2)}, \ldots$ не е нито една от функциите в наредбата. Щом функциите от вида $f : \mathbb{N} \to \{0, 1\}$ са неизброимо безкрайно много, то и функциите от вида $f : \mathbb{N} \to \mathbb{N}$ също са неизброимо безкрайно много. Следва, че възможните функции са повече от алгоритмите в същия смисъл, в който реалните числа са повече от

естествените. Това означава, че само безкрайно малко подмножество от тези функции имат алгоритми!

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

Фигура 1.2 : Вход, алгоритъм, изход.

Както бе казано горе, алгоритъмът **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. Тук само ще повторим следното; то не може да се нарече "определение", а е просто разяснение.

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.

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 въвеждаме 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 [125]. RAM моделът там се въвежда на страница 19.

1.1.4 Елементарни инструкции

И така, алгоритъм е реализация на някаква функция. Тази реализацията се състои от крайна редица от елементарни инструкции. Това е съдържанието на кутията на Фигура 1.2 – редица от елементарни инструкции.

Всеки алгоритъм съществува в *дискретно време*, което се състои от *моменти* (още казваме *стъпки*). Във всеки момент от дискретното време се изпълнява точно една от елементарните инструкции.

Какви инструкции са допустими? Няма общоприета схема за това, какви са елементарните инструкции, но е очевидно, че ако допуснем съвсем произволни инструкции, ще обезсмислим конструирането на алгоритми. Например, ако допуснем елементарна 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 елементарни инструкции са тези, които могат да бъдат реализирани **физически** така, че да се изпълняват за време, близко до единица (един машинен такт), върху реален компютърен процесор. Например,

> *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* 99

са смислени елементарни инструкции. С 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.

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 съставни променливи, например масиви, и атомарни (елементарни) променливи, например целите числа. Това, разбира се, е абстракция по отношение на работата на истински компютър.

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 формална 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. Елементарните типове данни са `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 0 и не-нула и т. н.

Приемаме, че всяка елементарна инструкция се изпълняват в една стъпка независимо от големината на операндите. Например, събирането и умножаването на числа стават за една стъпка *независимо от големината на числата*. От практическа гледна точка, разбира се, това е нереалистично допускане по отношение на много големите числа. За всеки реален компютър има числа, които са достатъчно големи, за да не се побират (по-точно, тяхното представяне да не се побира) в една машинна дума. Нещо повече, всеки реален компютър може да работи само с крайно подмножество на безкрайното множество на целите числа; независимо от избраното кодиране на числата, само безкрайно малка част от всички тях са достъпни за него. Но за целите на курса ще приемаме, че всяко число може да се представи върху нашата абстрактна машина и че основните операции върху кои да е две числа стават за единица време. Това допускане 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.

Както казахме, инструкциите на алгоритъма са подредени и са краен брой. В изложението нататък ще смятаме, че са номерирани. Започвайки от първата инструкция на алгоритъма, във всеки следващ момент (стъпка) се изпълнява точно една инструкция[†]. Коя ще е следващата инструкция след инструкция $\mathtt{i}$ се определя така:

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

- Ако инструкция $\mathtt{i}$ е условна, например

  `if` ⟨някакъв израз с булева интерпретация⟩ `then`

[†]С други думи, няма паралелизъм.

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 кажем на алгоритъм AlgX, има две възможности. Ще ги наречем условно "прагматичната" и "теоретичната".

  - Прагматичната възможност е да мислим, че AlgX присъства като последователност 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 викащия 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$(a_1, \ldots, a_n)$ 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 викащия алгоритъм).

    При това можем да отчитаме времето за изпълнение на AlgX[†], 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 това също е полезна абстракция.

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

[†]Въпреки че в тази възможност не гледаме на AlgX като на последователност 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

- 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`.

1.1.5 Knuth за алгоритмите

Knuth [84, 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 *изчиситeлен метод*. Има два начина алгоритъм да не завърши работата си.

- Може да "зацикли", което означава да влезе повторно в конфигурация, в която вече е бил. *Конфигурация* е съвкупността от стойностите на всички използвани променливи плюс номера на текущата инструкция. Тоест, състоянието на паметта, което включва в себе си и 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 състояние, в което вече е бил. В посочения пример, паметта, която реализира променливата `j`, рано или късно ще се препълни и `j` пак ще стане единица.

  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 – различни читатели може да тълкуват по различен начин дадени инструкции.

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[‡].

[†]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 с абсолютна сигурност, че е *детерминиран* краен автомат. Input от околния свят може да е нещо, което идва по мрежата, или нещо, въведено от потребителя.

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

Допълнение 5: On-line и off-line алгоритми

Някои автори, например Карп [82], 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 стартирана 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, в някакъв смисъл, затворени или капсуловани, понеже за тях не съществува външен свят. Те връщат резултата си във външния свят, но по време на работата си са капсуловани. В контраст на това, on-line алгоритмите (може да) получават информация от външния свят, докато работят.
Алгоритмите в този курс са off-line без изключение. В учебника [31] 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.

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 елементарни инструкции, наречени от Knuth “operations”, 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 ...

където P е произволна програма, а x е произволен неин вход. Тази операция, или елементарна инструкция в нашата терминология, 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 $\mathfrak{P}$ и неин вход $\mathfrak{I}$.
въпрос: Дали $\mathfrak{P}$ с вход $\mathfrak{I}$ терминира?

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

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 $\mathfrak{P}$ и неин вход $\mathfrak{I}$.
въпрос: Дали $\mathfrak{P}$ с вход $\mathfrak{I}$ терминира?

Alan Turing е доказал [140], че няма алгоритъм, който решава Halting Problem. Следва опростена версия на доказателството. За по-задълбочено разглеждане, добър източник е, например, книгата на Sipser [132].

Теорема 1: Halting Problem е алгоритмично нерешима

Halting Problem е алгоритмично нерешима.

Доказателство: Да допуснем противното. Тогава съществува програма $\mathfrak{Q}$, чийто вход е наредена двойка от компютърна програма $\mathfrak{P}$ и неин вход $\mathfrak{I}$[a], като $\mathfrak{Q}(\mathfrak{P}, \mathfrak{I})$ връща True, ако $\mathfrak{P}$ с вход $\mathfrak{I}$ терминира, и False в противен случай. Да дефинираме програма $\mathfrak{S}(\mathfrak{P})$ по следния начин: $\mathfrak{S}(\mathfrak{P}) = \mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$. Забележете, че $\mathfrak{S}(\mathfrak{P})$ се състои от един единствен ред:

```
return Q(P, P)
```

Дефинираме още една програма $\mathfrak{T}(\mathfrak{P})$ по следния начин:

```
if S(P) then loop forever
else return True
```

[†]Повече информация за хипотезата на Goldbach има в [142]. Оригиналното писмо на Goldbach до Euler, изказващо хипотезата, може да бъде видяно на *този сайт*.

Да анализираме $\mathfrak{T}(\mathfrak{T})$. Ако $\mathfrak{T}(\mathfrak{T})$ не терминира, трябва да е вярно, че $\mathfrak{S}(\mathfrak{T})$ връща TRUE. Но тогава трябва да е вярно, че $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ връща TRUE. Тогава трябва да е вярно, че $\mathfrak{T}$ терминира с вход $\mathfrak{T}$. Това е противоречие.
Тогава да допуснем, че $\mathfrak{T}(\mathfrak{T})$ терминира. Тогава тя връща TRUE. Тогава трябва да е вярно, че $\mathfrak{S}(\mathfrak{T})$ връща FALSE. Тогава $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ връща FALSE. Тогава $\mathfrak{T}$ не терминира с вход $\mathfrak{T}$. Но това също е противоречие. □

[a] Всъщност, входът на $\mathfrak{Q}$ е **кодирането** на $\mathfrak{P}$ и **кодирането** на нейния вход $\mathfrak{I}$.

Базирайки се на казаното от Knuth, има два принципно различни начини дадена елементарна инструкция да е неподходяща за инструкция на алгоритъм: 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.

Има и трети начин една елементарна инструкция да е неподходяща: да е не-ефикасна, което ще рече, допускането, че тя може да бъде изпълнена за единица време, да е прекалено нереалистично. За разликата между ефективно и ефикасно, вижте Допълнение 8. Пример за такава инструкция е

```
if isprime(n) then ...
```

където `n` е естествено число, а `isprime(n)` е TRUE тогава и само тогава, когато `n` е просто число. Доколкото е известно в момента, задачата дали дадено число е просто или не, е *достатъчно алгоритмично трудна*, за да не може да твърдим, че се извършва в единица време. Следователно, `isprime(n)` не може да е елементарна инструкция. В контраст на това,

```
if iseven(n) then ...
```

може да бъде елементарна инструкция, защото тестването за четност е алгоритмично тривиална задача и можем да мислим, че се извършва за единица време.

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 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 $i \in \{1, \ldots, k\}$, 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 [84, 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 заe термина "итератор" от книгата "Chaos and Fractals: New Frontiers of Science" [117]. Обикновено в компютърните науки под "итератор" се разбира *средство за обхождане на някакви агрегирани данни*, примерно списъци. В [117, 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

$$n \mapsto f(n)$$

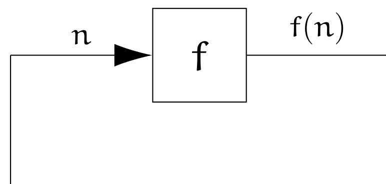
където n е реално число, а f е реална функция. Това описва безкраен изчислителен метод, който започва с n, от него получава $f(n)$, от него получава $f(f(n))$, и така нататък, без край.

$\boxed{n \mapsto f(n)}$ 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 ѝ се подава като вход, отново и отново.

Въпросният итератор може да се опише и така:

$$n \mapsto f(n) \mapsto f(f(n)) \mapsto f(f(f(n))) \mapsto \cdots$$

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

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

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

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

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, после 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 "всемогъщ" е дали може да симулира произволна машина на Turing[‡]. Подробно изложение за машини на Turing има в

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

[‡]Или, алтернативно, универсалната машина на Turing

Секция 14.4. "Всемогъщ" 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 решима) задача, стига да знаем как да го направим.

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 на стр. 239, който позволява излизане от средата на цикъла (ред 8) 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 ∈ ℕ)
1   if n = 0
2       return 1
3   else
4       return n × FACTORIAL(n − 1)
```

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

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).
- Factorial(3) проверява условието на ред 1 и понеже то е Лъжа, на ред 4 изпълнява Factorial(2).
- Factorial(2) проверява условието на ред 1 и понеже то е Лъжа, на ред 4 изпълнява Factorial(1).
- Factorial(1) проверява условието на ред 1 и понеже то е Лъжа, на ред 4 изпълнява Factorial(0).
- Factorial(0) проверява условието на ред 1 и понеже то е Истина, на ред 2 връща стойност 1 на Factorial(1).
- Factorial(1) получава стойност 1 от викането на Factorial(0), след което умножава 1×1 и връща резултата, който е 1, на Factorial(2).
- Factorial(2) получава стойност 1 от викането на Factorial(1), след което умножава 2×1 и връща резултата, който е 2, на Factorial(3).
- Factorial(3) получава стойност 2 от викането на Factorial(2), след което умножава 3×2 и връща резултата, който е 6, на Factorial(4).
- Factorial(4) получава стойност 6 от викането на Factorial(3), след което умножава 4×6 и връща резултата, който е 24, на Factorial(5).
- Factorial(5) получава стойност 24 от викането на Factorial(4), след което умножава 5×4 и връща 120.

Работата на 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 *спирачката 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 бащa на геометрията. Неговата поредица от книги, наречена *Елементи*, е в основата на курсовете по класическа геометрия и до днес. В седмата книга от *Елементи* [63] 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 Задача 1 [63, стр. 296] и Задача 2 [63, стр. 298].

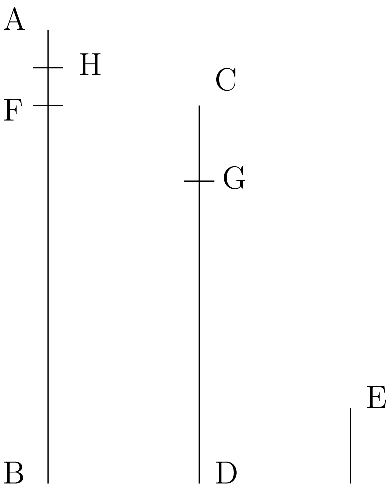
1.2.1 Оригиналният Евклидов алгоритъм

Задача 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 по-голямото[a], и най-накрая остане единица, оригиналните числа са взаимно прости.

[a]Има 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. □

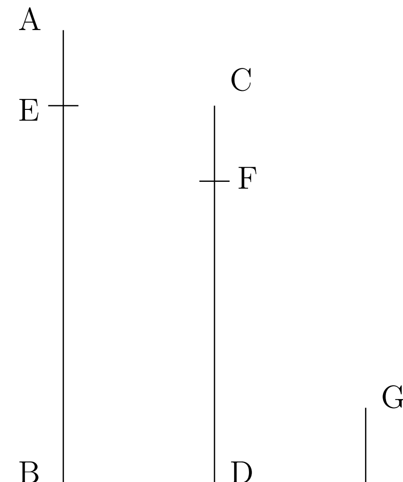
[†]Древните гърци са мислели за числата като за дължини на отсечки.
[‡]Древните гърци не са смятали единицата за число, а за нулата дори не са имали понятие.
[§]Става дума отново за E.

Задача 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 Задача 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 някаква 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 Задача 1, трикратното прилагане е:

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

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 [85].

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, ITERATIVE(a, b ∈ ℕ⁺, a ⩾ b)
1  while a mod b > 0 do
2      r ← a mod b
3      a ← b
4      b ← r
5  return b
```

```
EUCLID, REC(a, b ∈ ℕ, a ⩾ b, a > 0)
1  if b = 0
2      return a
3  else
4      return EUCLID, REC(b, a mod b)
```

В Глава 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 [31]. 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 "←" 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. Така е в Pascal. В C и Java, постусловията са изградени по обратната конвенция: следващо изпълнение има ако и само ако постусловието е TRUE.
- 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.

Писането на псевдокод има потенциален недостатък. Свободата при писането на псевдокода създава опасност да започнем да пишем елементарни 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 Задача 1 и Задача 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: "An algorithm is the idea behind any reasonable computer program" (на стр. 3). 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

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

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, ITERATIVE 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.

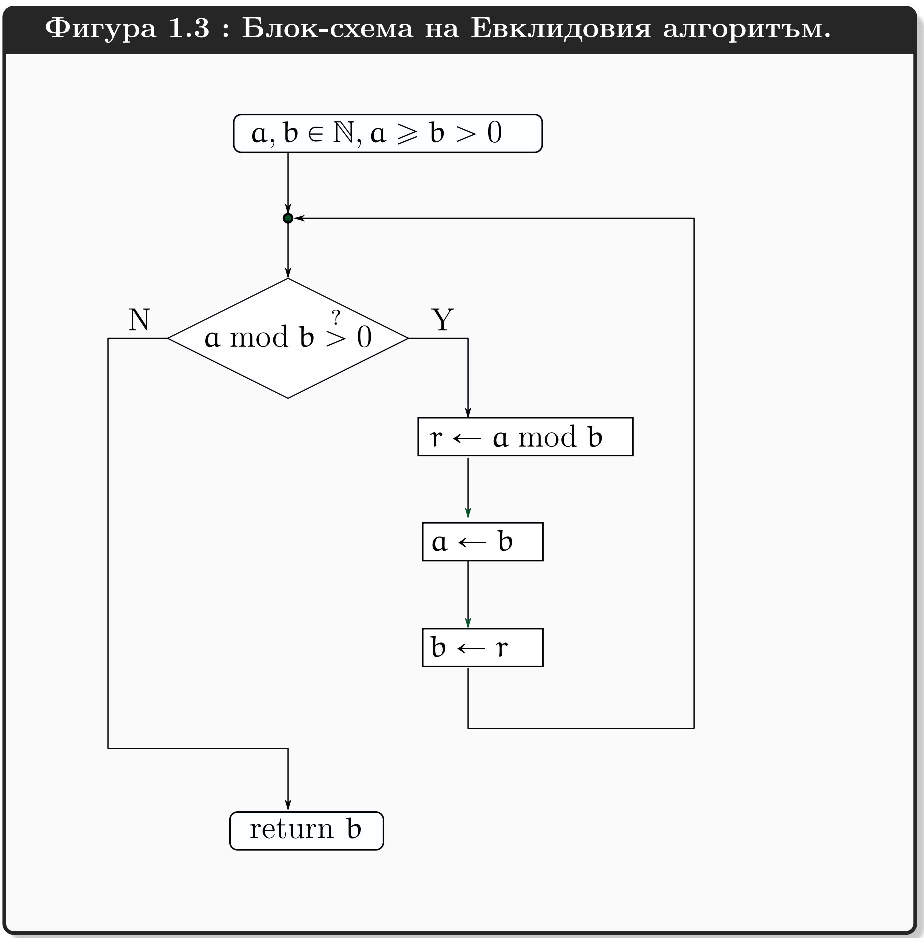
Стрелките показват как върви изпълнението, което по правило се изобразява отгоре надолу. Върхът-начало има степен на входа нула, защото от него започва изпълнието, и степен на изхода единица. Върховете-условия и върховете-императивни инструкции, както и върхът-край, имат степен на входа единица. Върхът-край 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 начина[‡], преди него слагаме връх от тип ветрило, който има повече от един входове и само един изход. Връх-ветрило рисуваме с дебела точка.

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.

[†]На английски "блок-схема" е *flowchart*.

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

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 **ефект**ивен, понеже неговата работа дава желания **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 достигаме 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.

[a] "В 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 греши. 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 краят 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 Π. Както казахме вече, можем да мислим за Π 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 39), 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 [102].

[a]Да 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(∅)
1  n ← 2
2  do
3     n ← n + 2
4     more ← FALSE
5     for i ← 2 to 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 $n \geqslant 3$.

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 [145], 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 $C : \mathbb{N}^+ \to \mathbb{N}^+$ 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) $\boxed{n \mapsto C(n)}$ 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 [10] и огромната библиография от статии за хипотезата на Collatz [92], [93].

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 ∈ ℕ⁺)
1  while n ≠ 1 do
2      if iseven(n)
3          n ← n/2
4      else
5          n ← 3n + 1
```

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

Наблюдение 6

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 са 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 големини. 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(⌊n/2⌋)
```

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 $n \geqslant 1$. 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 $2 \times$ НАРАСТВАНЕ С ЕДИНИЦА $\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Тъй като $\left\lfloor \frac{n}{2} \right\rfloor < n$, може да приложем 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

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

Това е и краят на доказателството. □

В това доказателство 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 −∞
3  else
4      max ← MAX UNSORTED, REC(A[1..n − 1])
5      if A[n] > max
6          max ← 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 предната 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 $n-1 \geqslant 0$, MAX UNSORTED, REC($A[1..n-1]$) връща максималния елемент на $A[1..n-1]$. Разглеждаме викането MAX UNSORTED, REC($A[1..n]$). 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 $A[1..n-1]$). Следните две възможности са изчерпателни:

- $A[n]$ е по-голям от всеки елемент на $A[1..n-1]$. Прилагаме индуктивното предположение и заключаваме, че $A[n] > max$. 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 $A[1..n]$. После изпълнението отива на ред 7.

- Не е вярно, че $A[n]$ е по-голям от всеки елемент на $A[1..n-1]$. Тогава максимумът на $A[1..n]$ е максимумът на $A[1..n-1]$. Прилагаме индуктивното предположение и заключаваме, че на ред 4, на *max* се присвоява максимума на $A[1..n]$. 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 $A[1..n]$. Заключаваме, че алгоритъмът връща максимума на $A[1..n]$. □

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 $A[1..n]$ е числен масив, то A е битоничен, ако съществува индекс k, такъв че $1 \leqslant k \leqslant n$ и

$$A[1] < A[2] < \cdots < A[k]$$
$$A[k+1] > A[k+2] > \cdots > A[n]$$

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 големини на $A[k]$ и $A[k+1]$.

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

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

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

Следователно, има или точно един максимален елемент, или точно два максимални елемента, които са съседи 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.

```
Max Bitonic(A[1..n], ℓ, h; изпълнено е 1 ⩽ ℓ ⩽ h ⩽ n)
1  if ℓ = h
2      return A[ℓ]
3  mid ← ⌊(ℓ+h)/2⌋
4  if A[mid] < A[mid + 1]
5      return Max Bitonic(A, mid + 1, h)
6  else if A[mid] > A[mid + 1]
7      return Max Bitonic(A, ℓ, mid)
8  else
9      return A[mid]
```

Теорема 5: Коректността на алгоритъма Max Bitonic

Ако $A[1\,..\,n]$ е битоничен и $1 \leqslant \ell \leqslant h \leqslant n$, то Max Bitonic$(A, \ell, h)$ връща максималния елемент на подмасива $A[\ell\,..\,h]$.

Доказателство:
Управляваш 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 съответна разлика и "спирачката" на рекурсията се "задейства" от тази разлика.

Базовият случай е $h - \ell = 0$. Той "задейства спирачката" на рекурсията, като условието на ред 1 е истина и алгоритъмът връща $A[\ell]$ на ред 2. Това е коректно, понеже единственият елемент е максимален.

Да разгледаме стойности на ℓ и h, които не "задействат спирачката" на рекурсията. С други думи, $\ell < h$. Индуктивното предположение е, че за всички h' и ℓ', такива че $1 \leqslant \ell' \leqslant h' \leqslant n$ и $h' - \ell' < h - \ell$, викането на Max Bitonic(A, ℓ', h') връща максимума на $A[\ell'\,..\,h']$.

Изпълнението отива на ред 3, където mid става $\left\lfloor \frac{\ell+h}{2} \right\rfloor$. Преди да продължим с доказателството, да се убедим, че елементите $A[\text{mid}]$ и $A[\text{mid}+1]$ се намират в масива $A[\ell\,..\,h]$; с други думи, че няма опит за достъп извън подмасива от входа.

- Първо, $\ell \leqslant \text{mid}$, защото $\ell \leqslant \left\lfloor \frac{\ell+h}{2} \right\rfloor$ при $\ell < h$, като равенство е възможно.
- Второ, $\text{mid} < h$, защото

$$
\begin{aligned}
\left\lfloor \frac{\ell+h}{2} \right\rfloor &< h \\
\left\lfloor \frac{\ell+h}{2} - h \right\rfloor &< 0 \\
\left\lfloor \frac{\ell-h}{2} \right\rfloor &< 0 \quad // \text{ нека } h = l + k, \text{ където } k > 0 \\
\left\lfloor \frac{-k}{2} \right\rfloor &< 0
\end{aligned}
$$

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

като последното е вярно (почти) директно от дефиницията на $\lfloor\ \rfloor$.

И така, $A[\mathrm{mid}]$ и $A[\mathrm{mid}+1]$ се намират в масива $A[\ell \mathinner{..} h]$.

Следните възможности са изчерпателни:

- $A[\mathrm{mid}] < A[\mathrm{mid}+1]$. В такъв случай нито един от $A[\ell], \ldots, A[\mathrm{mid}]$ не е максимален. Тогава максималният елемент се намира в подмасива $A[\mathrm{mid}+1 \mathinner{..} h]$. Но $h-(\mathrm{mid}+1) < h-\ell$ и можем да приложим индуктивното предположение, от което следва, че рекурсивното викане на ред 5 ще върне максимума на $A[\mathrm{mid}+1 \mathinner{..} h]$. Както вече установихме, той е и максимумът на $A[\ell \mathinner{..} h]$.

- $A[\mathrm{mid}] > A[\mathrm{mid}+1]$. В такъв случай нито един от $A[\mathrm{mid}+1], \ldots, A[h]$ не е максимален. Тогава максималният елемент се намира в подмасива $A[\ell \mathinner{..} \mathrm{mid}]$. Но $\mathrm{mid} - \ell < h - \ell$ и можем да приложим индуктивното предположение, от което следва, че рекурсивното викане на ред 7 ще върне максимума на $A[\ell \mathinner{..} \mathrm{mid}]$. Както вече установихме, той е и максимумът на $A[\ell \mathinner{..} h]$.

- $A[\mathrm{mid}] = A[\mathrm{mid}+1]$. В такъв случай имаме двоен максимум $A[\mathrm{mid}] = A[\mathrm{mid}+1]$, и алгоритъмът връща точно тази стойност на ред 9. □

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

Следствие 1: Коректността на началното викане на Max Bitonic

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: $\gcd(a, b)$

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

Следствие 2

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

Съгласно [56, (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 1

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, за някакви цели положителни p и q, $a = cp$ и $b = cq$. Но $a \bmod b = a - b\left\lfloor \frac{a}{b} \right\rfloor$ от Определение 9, така че $a \bmod b = cp - cq\left\lfloor \frac{a}{b} \right\rfloor = ck$ за някое естествено k. Ерго, c дели $a \bmod b$. □

Следната лема е ключова за верификацията на Евклидовия алгоритъм.

Лема 2

Нека $a, b \in \mathbb{N}$ и $a \geqslant b$, $a > 0$. 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 2. Нека $a \geqslant b > 0$.

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 2. От друга страна, $\gcd(a, b) = b$ в този случай, така че твърдението пак е вярно.

Нека b не дели a. Нека $a \bmod b = c$, като $c > 0$. Тогава $c = a - bk$, за някое цяло k. Ако някое цяло d дели както a, така и b, то d дели и $a - bk$; тоест, d дели c и b. От друга страна, ако d дели c и b, то d дели и $c + bk$, което е a; тоест, d дели a. Ерго, d е общ делител на c и b тстк d е общ делител на a и b. Ерго, множеството от делителите на c 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. □

Ето отново рекурсивната версия на Евклидовия алгоритъм от Подсекция 1.2.2.

```
EUCLID, REC(a, b ∈ ℕ, a ⩾ b, a > 0)
1  if b = 0
2      return a
3  else
4      return EUCLID, REC(b, a mod b)
```

Теорема 6: Коректността на алгоритъма Euclid, rec

Алгоритъмът Euclid, rec(a, b) връща $\gcd(a, b)$ за всички естествени a и b, такива че $a \geqslant b$.

Доказателство: Рекурсията е двумерна, в смисъл, че има два аргумента и началното условие е едномерното $b = 0$. Приемаме за очевидно, че за всяко $a, b \in \mathbb{N}$, такива че $a \geqslant b$:

- итераторът $\langle a, b \rangle \mapsto \langle b, a \bmod b \rangle$ винаги достига до $\langle x, 0 \rangle$ за някое $x > 0$,
- итераторът $\langle a, b \rangle \mapsto \langle b, a \bmod b \rangle$ изобразява наредената двойка $\langle a, b \rangle$ в наредена двойка $\langle p, q \rangle$, която е по-малка в смисъл, че $p \leqslant a$ и $q < b$.

Второто наблюдение ни дава право да правим доказателството със силна индукция по наредените двойки $\langle a, b \rangle$.

База. В базовия случай разглеждаме наредените двойки $\langle a, 0\rangle$ за $a > 0$. Тогава условието на ред 1 е истина и изпълнението отива на ред 2 и връща a. Но по дефиниция, $\gcd(a, 0) = a$, ако $a > 0$. ✓

Индуктивно Предположение. Нека за някакви естествени a и b, такива че $a \geqslant b > 0$, за всяка наредена двойка $\langle p, q\rangle$, такава че $p \leqslant a$ и $q < b$, EUCLID, REC(p, q) връща $\gcd(p, q)$.

Индуктивна стъпка. Разглеждаме EUCLID, REC(a, b). Щом $b > 0$, условието на ред 1 е лъжа и изпълнението отива на ред 4, където се прави викане на EUCLID, REC$(b, a \bmod b)$. Но $b \leqslant a$ и $a \bmod b < b$, ерго, съгласно индуктивното предположение, викането на EUCLID, REC$(b, a \bmod b)$ връща $\gcd(b, a \bmod b)$. От Лема 2 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. □

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 пътите, в които изпълнението е на ред ℓ, е **краен**. Ако ред ℓ се изпълнява точно m пъти, то индукцията е върху крайното множество $\{1, 2, , \ldots, m\}$

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):

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

По отношение на някои алгоритми предикатът може да се дефинира по-просто: директно върху управляващата променлива на цикъла. Това не винаги е възможно, но, ако е възможно, доказателството става по-елегантно. Нека цикълът е **for**-цикъл с инкрементиране на

управляващата променлива с единица:

for $i \leftarrow n_1$ **to** n_2
    «тяло-на-цикъла»

за някакви фиксирани n_1 и n_2 и освен това i не бива променяна вътре в тялото на цикъла. Тогава може да вземем за домейн на предиката множеството $\{n_1, n_1 + 1, \ldots, n_2 + 1\}$[†] и да дефинираме предиката директно като $P(i)$, където i е управляващата променлива на цикъла. Но при по-сложни цикли, примерно **while**-цикли, в които управляващата променлива се мени в тялото на цикъла по някакъв сложен начин, по-удачно е предикатът да е върху броя пъти на достигането на реда с условието.

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..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 7: Коректността на алгоритъм за сумата от елементите на масив

Find Sum връща сумата от елементите на входния масив $A[1..n]$.

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 $A[1..i-1]$.

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 $A[1..i-1]$ в този момент е празен, защото i съдържа стойност 1 заради присвояването на ред 2, следователно $A[1..i-1]$ всъщност е $A[1..0]$, 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 $A[1..i-1]$ 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. И така, *sum* съдържа сумата от елементите на подмасива $A[1..i-1]$.

[†]Това е множеството от стойностите, които управляващата променлива взема. Забележете, че най-голямата от тях е $n_2 + 1$, а не n_2, понеже при последното достигане на реда с условието—когато условието вече не е изпълнено и тялото на цикъла няма да се изпълнява повече—управляващата променлива е именно $n_2 + 1$.

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 $A[1..i-1, i]$. Но след това на ред 5, променливата i се инкрементира. Изразено чрез новото i, заключението става " *sum* съдържа сумата от елементите на подмасива $A[1..i-1]$". 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 $A[1..(n+1)-1]$.". По-просто казано, *sum* съдържа сумата от елементите на масива $A[1..n]$.

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 [31]. 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, който вече дискутирахме горе, е следният. Да кажем, че цикълът е **for**-цикъл от вида (ℓ е номерът на съответния ред 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 $\mathsf{P}(a)$ е вярно при първото достигане на ред ℓ.
- Допускайки, че $\mathsf{P}(i)$ е вярно при някое достигане на ред ℓ, което не е последното, (което значи, че $i \leqslant b$), доказваме $\mathsf{P}(i+1)$ 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($A[1..n]$: масив от цели числа)
  1 **for** $i \leftarrow 1$ **to** n
  2 $\quad A[i] \leftarrow i$

  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 $A[1..i-1]$ 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 $[1, 2, \ldots, n-1, n]$, 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 реализира сортиран масив, а трябва да реализира 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 7

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..n]: array of integers)
1   max ← −∞
2   i ← 1
3   while i ⩽ n do
4       if A[i] > max
5           max ← A[i]
6       i ← i + 1
7   return max
```

Теорема 8: Коректността на алгоритъма MAX UNSORTED, ITER

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

Доказателство: Заслужава си да се отбележи, че това 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 $\boxed{a'}$ и $\boxed{b'}$, за да означим съдържанието съответно на a и b преди началото на алгоритъма,
- в доказателството за коректност на INSERTION SORT, в Лема 25 използваме означението $\boxed{A'}$ за съдържанието на масива A в самото начало на едно изпълнение на външния цикъл, а в Теорема 42 използваме означението $\boxed{A''}$ за съдържанието на A самото начало на алгоритъма.

Доказателството на Теорема 8 се основава на следния инвариант.

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* $< A[i]$. Следователно, 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* $\not< A[i]$. Следователно, 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 $A[1..(n+1)-1]$". По-просто казано, "променливата *max* съдържа стойността на максималния елемент в подмасива $A[1..n]$". Но на ред 7, алгоритъмът връща именно *max*. Заключаваме, че MAX UNSORTED, ITER(A) връща максимума на A. □

2.1.3.2 Странен алгоритъм за сумиране на елементите на масив

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

```
int A[n];
int strangesum(int n) {
  int i, s = 0;
  for(i = 0; i < n; i ++) {
    if (i%2 == 0)
      s += A[i/2];
    else
      s += A[n - 1 - i/2]; }
  return s; }
```

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

Инвариант

При всяко достигане на ред 4:

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

База При първото достигане на ред 4, s съдържа 0 заради присвояването на ред 3. От друга страна, $s = \sum_{j=0}^{\left\lfloor \frac{i+1}{2} \right\rfloor - 1} A[j] + \sum_{j=\left\lfloor \frac{i}{2} \right\rfloor + 1}^{n-1} A[j] = 0 + 0 = 0$, защото i е 0, така че $\left\lfloor \frac{0+1}{2} \right\rfloor = 0$ и $\left\lfloor \frac{0}{2} \right\rfloor = 0$, което на свой ред означава, че множествата $\{0, \ldots, \left\lfloor \frac{i+1}{2} \right\rfloor - 1\} = \{0, \ldots, -1\}$ и $\{n - \left\lfloor \frac{i}{2} \right\rfloor, \ldots, n-1\} = \{n-0, \ldots, n-1\}$ са празни. ✓

Поддръжка Нека твърдението е вярно при някое достигане, 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 $s = \sum_{j=0}^{\left\lfloor \frac{i+1}{2} \right\rfloor - 1} A[j] + \sum_{j=n-\left\lfloor \frac{i}{2} \right\rfloor}^{n-1} A[j]$ от предположението. След присвояването, имаме

$$\begin{aligned} s &= \left(\sum_{j=0}^{\left\lfloor \frac{i+1}{2} \right\rfloor - 1} A[j] + \sum_{j=n-\left\lfloor \frac{i}{2} \right\rfloor}^{n-1} A[j] \right) + A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \\ &= \left(\sum_{j=0}^{\left\lfloor \frac{i+1}{2} \right\rfloor - 1} A[j] \right) + A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] + \sum_{j=n-\left\lfloor \frac{i}{2} \right\rfloor}^{n-1} A[j] \qquad (2.1) \end{aligned}$$

Тъй като i е четно, $i+1$ е нечетно. Лесно се вижда, че $\left\lfloor \frac{i+1}{2} \right\rfloor = \left\lfloor \frac{i}{2} \right\rfloor$, следователно (2.1) е равно

на:

$$
\begin{aligned}
&\left(\sum_{j=0}^{\left\lfloor\frac{i}{2}\right\rfloor-1} A[j]\right)+A\left[\left\lfloor\frac{i}{2}\right\rfloor\right]+\sum_{j=n-\left\lfloor\frac{i}{2}\right\rfloor}^{n-1} A[j]= \\
&\left(\sum_{j=0}^{\left\lfloor\frac{i}{2}\right\rfloor} A[j]\right)-A\left[\left\lfloor\frac{i}{2}\right\rfloor\right]+A\left[\left\lfloor\frac{i}{2}\right\rfloor\right]+\sum_{j=n-\left\lfloor\frac{i}{2}\right\rfloor}^{n-1} A[j]= \\
&\sum_{j=0}^{\left\lfloor\frac{(i+1)+1}{2}\right\rfloor-1} A[j]+\sum_{j=n-\left\lfloor\frac{i+1}{2}\right\rfloor}^{n-1} A[j]
\end{aligned}
\tag{2.2}
$$

При следващото достигане на ред 4, i се инкрементира с единица. Очевидно, спрямо новата стойност на i, израз (2.2) е:

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

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

$$
\begin{aligned}
s &= \left(\sum_{j=0}^{\left\lfloor\frac{i+1}{2}\right\rfloor-1} A[j]+\sum_{j=n-\left\lfloor\frac{i}{2}\right\rfloor}^{n-1} A[j]\right)+A\left[n-1-\left\lfloor\frac{i}{2}\right\rfloor\right] \\
&= \sum_{j=0}^{\left\lfloor\frac{i+1}{2}\right\rfloor-1} A[j]+\left(\left(\sum_{j=n-\left\lfloor\frac{i}{2}\right\rfloor}^{n-1} A[j]\right)+A\left[n-1-\left\lfloor\frac{i}{2}\right\rfloor\right]\right) \\
&= \sum_{j=0}^{\left\lfloor\frac{i+1}{2}\right\rfloor-1} A[j]+\sum_{j=n-1-\left\lfloor\frac{i}{2}\right\rfloor}^{n-1} A[j]
\end{aligned}
\tag{2.3}
$$

При нечетно i, в сила е равенството $\left\lfloor\frac{i+1}{2}\right\rfloor=\left\lfloor\frac{i+2}{2}\right\rfloor$. Освен това, за всяко i е вярно, че $n-1-\left\lfloor\frac{i}{2}\right\rfloor=n-1+\left\lceil-\frac{i}{2}\right\rceil=n+\left\lceil-1-\frac{i}{2}\right\rceil=n-\left\lfloor 1+\frac{i}{2}\right\rfloor=n-\left\lfloor\frac{i+2}{2}\right\rfloor$. При нечетно i, последният израз е равен на $n-\left\lfloor\frac{i+1}{2}\right\rfloor$. Следователно, (2.3) е равно на:

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

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

на i, (2.4) е:

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

Виждаме, че инвариантът се запазва.

Терминация При последното достигане на ред 4, в сила е

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

Следователно,

$$s = \sum_{j=0}^{\left\lfloor \frac{n+1}{2} \right\rfloor - 1} A[j] + \sum_{j=n-\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} A[j] \qquad (2.5)$$

Ще покажем, че $\left\lfloor \frac{n+1}{2} \right\rfloor - 1$ и $n - \left\lfloor \frac{n}{2} \right\rfloor$ са съседни стойности, нарастващи в този ред, за всяко естествено n. Първо да допуснем, че е n е четно, тоест $n = 2k$ за някое естествено k. Тогава

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

а

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

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

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

а

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

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

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

са разбиване на множеството $\{0, 1, \ldots, n-1\}$. Следователно, (2.5) е равно на

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

2.1.3.3 Евклидовият алгоритъм в итеративен вариант

Ето отново итеративната версия на Евклидовия алгоритъм от Подсекция 1.2.2.

```
EUCLID, ITERATIVE(a, b ∈ ℕ⁺, a ⩾ b)
1  while a mod b > 0 do
2      r ← a mod b
3      a ← b
4      b ← r
5  return b
```

И неговото доказателство се основава на Лема 2.

Инвариант

Нека a' и b' означават съответно входните a и b в алгоритъм EUCLID, ITERATIVE. 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 Лема 2 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 $b_2 = a_1 \bmod b_1$. Показахме, 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 $\gcd(a', b') = \gcd(a_2, b_2)$. ✓

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

2.1.3.4 ALG2SUM

Изчислителната задача 2SUM се споменава в Подсекция 4.2.7 и Подсекция 13.3.3. Тук само ще разгледаме алгоритъм за нея и ще докажем коректността му.

```
ALG2SUM(A[1..n]: масив от цели числа, такъв че A[1] ⩽ A[2] ⩽ ··· ⩽ A[n]; m ∈ ℤ)
1  i ← 1, j ← 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          i++
7      else
```

```
8        j--
9    return 0
```

За удобство да дефинираме *диада* като всяко двуелементно мултимножество $\{A[p], A[q]\}_M$, такова че $1 \leqslant p, q \leqslant n$, $p \neq q$ и $A[p] + A[q] = m$.

Теорема 9: Коректността на Alg2SUM

Нека входният масив на Alg2SUM е сортиран ненамаляващо. Алгоритъмът връща 1, ако във входния масив има поне една диада, а в противен случай връща 0.

Доказателство: Следното твърдение е инвариант за **while**-цикълът (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 $A[i] + A[j] \neq m$, инак 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.

- $A[i] + A[j] < m$. 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 $\leqslant$ 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 $A[i..j]$ се намират в $A[i+1..j]$. Тъй като по допускане всички диади в целия $A[1..n]$ са в $A[i..j]$, следва, че всички диади в $A[1..n]$ се намират в $A[i+1..j]$.

  Изпълнението отива на ред 6, където i бива инкрементирано. Спрямо новото i, твърдението става "всички диади в $A[1..n]$ се намират в $A[i..j]$". Виждаме, че инвариантът се запазва.

- $A[i] + A[j] > m$. 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 $\geqslant$ 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 $A[i..j]$ се намират в $A[i..j-1]$. Тъй като по допускане всички диади в целия $A[1..n]$ са в $A[i..j]$, следва, че всички диади в $A[1..n]$ се намират в $A[i..j-1]$.

  Изпълнението отива на ред 8, където j бива декрементирано. Спрямо новото j, твърдението става "всички диади в $A[1..n]$ се намират в $A[i..j]$". 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, $i \geqslant j$. Но инвариантът казва, че всички диади се намират в $A[i..j]$. Ако $i \geqslant j$, подмасивът $A[i..j]$ 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 $A[i..j]$ "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 Alg3SUMCount

За формалната дефиниция и практическото значение на задачата 3SUM вижте Подсекция 4.2.7. Тук само ще разгледаме алгоритъм за нея, наречен Alg3SUMCount, и ще докажем коректността му. За разлика от Alg2SUM, който просто индикира дали има или няма диади, Alg3SUMCount връща броя на триадите (дефиницията на "триада" е очевидната). Както ще видим, това прави доказателството за коректност значително по-трудно.

В алгоритъма Alg2SUM, числото m, което трябва да е сумата на всяка диада, е част от входа, докато в алгоритъма Alg3SUMCount сумата на триада е фиксирана на 0. Тази разлика между Alg2SUM и Alg3SUMCount е незначителна.

```
Alg3SUMCount(A[1..n]: сортиран масив от две по две различни цели числа)
 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 редица[†] $\langle A[x], A[y], A[z] \rangle$, такава че $1 \leqslant x < y < z \leqslant n$ и $A[x] + A[y] + A[z] = 0$.

Теорема 10: Коректността на Alg3SUMCount

За всеки входен масив $A[1..n]$, който е сортиран и няма повтарящи се елементи, Alg3SUMCount$(A[1..n])$ връща броя на триадите в $A[1..n]$.

Доказателство: Alg3SUMCount е първият итеративен алгоритъм с вложеност на цикли измежду алгоритмите, които верифицираме. Наблюдателният читател сигурно е забелязал,

[†]По-удобно за доказателството е триадите да са подредени обекти, тоест редици, а не множества.

че може да се каже, че ALG3SUMCOUNT за всяко i пуска вариант на ALG2SUM, който присвоява на променливата d броя на диадите в $A[i+1..n]$, сумиращи се до $-A[i]$, и променливата c акумулира тези бройки. Но верификацията на ALG3SUMCOUNT ще бъде направена формално и от първи принципи – все едно, че не знаем за ALG2SUM.

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 10, 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, заествайки 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 ALG3SUMCOUNT

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, а i съдържа 1. Инвариантът става "0 е броят на триадите, чийто минимален елемент се намира в подмасива $A[1..1-1]$". Но подмасивът $A[1..1-1]$ е празният масив $A[1..0]$, така че всъщност инвариантът става "Броят на триадите, чийто минимален елемент се намира в празния масив, е нула". Това вярно ли е? Да, разбира се, понеже множеството от триадите, чиито минимален елемент е в празния масив, е празното множество, а мощността на празното множество е нула. ✓

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, имаха единствен цикъл. При някои, например FIND SUM на стр. 46, преминаването през тялото на цикъла е "линейно" в смисъл, че се първо изпълнява една точно определена инструкция, след това друга точно определена инструкция, и така нататък, докато тялото бъде изпълнено. При други като MAX UNSORTED, ITER на стр. 49 преминаването е по-сложно, защото тялото съдържа **if**, така че може да се изпълни една последователност или друга последователност от инструкции в зависимост от верността на булевото условие на **if**-а. Това прави доказателството по-дълго, но то остава с **фиксирана дължина**. Докато при сегашния алгоритъм ALG3SUMCOUNT ние не можем да направим доказателство (за поддръжката на инварианта на **for**-а) с фиксирана дължина, ако се опитаме да "разгънем" изпълнението на **while**-а, защото **не знаем точно колко пъти ще се изпълни while-а**. Така че се налага да осмислим работата на **while**-а, 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—после да формулираме нов инвариант, който касае **while**-а, и да докажем този нов инвариант. Този нов инвариант е напълно различен от инварианта за **for**-цикъла! Когато го докажем, във фазата **Терминация** от неговото доказателство 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 3: За ефекта от изпълнението на вътр. цикъл на ALG3SUMCOUNT

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

Доказателство: Ще докажем Лема 3 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 $\langle A[i], A[j], A[k] \rangle$ е вярно, че $j \in \{i+1, \dots, p-1\}$ тстк $k \in \{q+1, \dots, n\}$.

② d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, 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 $A[1..n]$ е сортиран, без повтарящи се елементи, и нека $1 \leqslant i < j < k \leqslant n$.

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

Факт2 Ако $A[i] + A[j] + A[k] \geqslant 0$, 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 $j \in \varnothing$ и $k \in \varnothing$." 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 $j \in \{i+1, \ldots, p\}$ тстк $k \in \{q, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ тстк $k \in \{q-1, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p\}$ и $k \in \{q, \ldots, n\}$, е с единица по-голям от броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i+1, \ldots, p-1\}$ и $k \in \{q+1, \ldots, n\}$. От ② знаем, че d съдържа броя на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i+1, \ldots, p-1\}$ и $k \in \{q+1, \ldots, n\}$. Тогава броят на триадите $\langle A[i], A[j], A[k] \rangle$, където $j \in \{i+1, \ldots, p\}$ и $k \in \{q, \ldots, n\}$, 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 $j \in \{i+1, \ldots, p\}$ и $k \in \{q, \ldots, n\}$, 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 $j \in \{i+1, \ldots, p-1\}$ и $k \in \{q+1, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ тстк $k \in \{q, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ тстк $k \in \{q+1, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ и $k \in \{q, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ и $k \in \{q+1, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p\}$ тстк $k \in \{q+1, \ldots, n\}$. 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 $j \in \{i+1, \ldots, p-1\}$ тстк $k \in \{q+1, \ldots, n\}$. 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. Щом е на ред 5 за последен път, вярно е, че $\neg(\tilde{p} < \tilde{q})$, което е същото като $\tilde{p} \geqslant \tilde{q}$. 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 и 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 доказателството 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 3). 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 $y \in \{i+1, \ldots, r-1\}$. Но това директно противоречи на ①, което казва "за всяка триада $\langle A[i], A[j], A[k]\rangle$ е вярно, че $j \in \{i+1, \ldots, r-1\}$ тстк $k \in \{r+1, \ldots, n\}$": щом средният елемент $A[y]$ е от $A[i+1..r-1]$, няма как максималният елемент да не е от $A[r+1..n]$.
- $A[r]$ е средният елемент на тази триада. Тогава тя е от вида $\langle A[i], A[r], A[z]\rangle$, като $z \in \{r+1, \ldots, n\}$. Но това директно противоречи на ①, което казва "за всяка триада $\langle A[i], A[j], A[k]\rangle$ е вярно, че $j \in \{i+1, \ldots, r-1\}$ тстк $k \in \{r+1, \ldots, n\}$": щом максималният елемент $A[z]$ е от $A[r+1..n]$, няма как средният елемент да не е от $A[i+1..r]$.

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 $\tilde{q}$ е $r-1$. 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 $j \in \{i+1, \ldots, r-1\}$ тстк $k \in \{r, \ldots, n\}$.

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

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

С това доказателството на Лема 3 приключва. □

Връщаме се към фазата **Поддръжка** от доказателството на Инвариант 1. Съгласно Лема 3, на ред 13, d съдържа съдържа броя на триадите, чиито минимален елемент е $A[i]$. Тогава след присвояването на ред 13, c вече съдържа броя на триадите, чийто минимален елемент се намира в подмасива $A[1..i]$. При следващото достигане на ред 2, променливата i се инкрементира. По отношение на новото i е вярно, че c съдържа броя на триадите, чийто минимален елемент се намира в подмасива $A[1..i-1]$. Виждаме, че инвариантът се запазва. ✓

Терминация: Да разгледаме последното достигане на ред 2. Тогава $i = n-1$. Получаваме, че c съдържа броя на триадите, чийто минимален елемент се намира в $A[1..n-2]$. 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 ALG3SUMCOUNT връща броя на всички триади в $A[1..n]$. □

2.1.3.6 Една задача върху масиви

Задачата има графова интерпретация. Да си представим свързан граф G, в който има път $p = u_1, u_2, \ldots, u_k$, където $u_1, \ldots, u_k \in V(G)$, и всяко ребро от p е мост (Определение 73). Това

означава, че ако изтрием от G ребрата на p, то G ще се разпадне на k свързани компоненти $G_1, \ldots, G_k$, като $u_i \in G_i$, за $1 \leqslant i \leqslant k$. Да допуснем, че $1 \leqslant i \leqslant k$ знаем ℓ_i: дължината на най-дълъг път в G_i с един край u_i. Искаме да намерим дължината на най-дълъг път q в G, единият край на който е в някое G_i, а другият край е в някое G_j, където $1 \leqslant i < j \leqslant n$. Неформално, q е път, който започва в някой G_i, после "минава по p" в смисъл, че има общ подпът с p, и после завършва в някой друг G_j. Очевидно $|q| = \ell_i + j - i + \ell_j$, защото q се състои от "слепването" на изброените три подпътя; ℓ_i и ℓ_j са дадени, а $j - i$ е броят на ребрата от p, които се намират и в q.

Ако игнорираме графовата интерпретация, задачата е, при даден масив $A[1..n]$ да се намери максималната сума на два различни елемента и разстоянието между тях в масива, за всички двойки различни елементи. Има тривиален алгоритъм, който опитва всички двойки елементи. Тук ще направим нещо по-умно.

```
AlgDist(A[1..n]: масив от цели положителни числа, като n ⩾ 2)
 1  създай масив B[1..n] от цели положителни числа
 2  B[1] ← A[1]
 3  t ← −∞
 4  i ← 2
 5  do
 6      B[i] ← max (B[i − 1], A[i] − (i − 1))
 7      t ← max (t, B[i − 1] + A[i] + (i − 1))
 8      i++
 9  while i ⩽ n
10  return t
```

Теорема 11: Коректността на алгоритъма AlgDist

AlgDist($A[1..n]$) връща максималната сума на два различни елемента и разстоянието между тях във входния масив, за всички двойки различни елементи.

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 $\max\{A[k] - (k-1) \mid 1 \leqslant k \leqslant i-1\}$,

② t съдържа $\max\{A[p] + A[q] + (q-p) \mid 1 \leqslant p < q \leqslant i-1\}$

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 $\{A[k]-(k-1) \mid 1 \leqslant k \leqslant i-1\}$, за което говори инварианта, е $\{A[1] - 0, A[2] - 1\}$. Заключаваме, че ① е истина.

От една страна, ако четем псевдокода, t 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 $\{A[p] + A[q] + (q - p) \,|\, 1 \leqslant p < q \leqslant i - 1\}$, 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,

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

Тогава

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

което може да запишем накратко като

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

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

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

Виждаме, че ① отново е истина.

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

$$t_{\text{old}} = \max\{A[p] + A[q] + (q-p) \,|\, 1 \leqslant p < q \leqslant i-1\}$$

а $B[i-1]$ е

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

Тогава $B[i-1] + A[i] + (i-1)$ е

$$\begin{aligned}
&\max\{A[1], A[2] - 1, A[3] - 2, \ldots, A[i-1] - (i-2)\} + A[i] + (i-1) = \\
&\max\{A[1] + A[i] + (i-1), A[2] - 1 + A[i] + (i-1), \ldots, A[i-1] - (i-2) + A[i] + (i-1)\} = \\
&\max\{A[k] + A[i] + (i-k) \,|\, 1 \leqslant k \leqslant i-1\}
\end{aligned}$$

И така:

- t_{old} съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1..i-1]$,
- 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 $A[1..i]$, единият от които обаче е $A[i]$.

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

$$t = \max\{\{A[p] + A[q] + (q-p) \,|\, 1 \leqslant p < q \leqslant i-1\}$$

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 t съдържа стойността на максимална сума на два различни елемента и разстоянието между тях, по всички елементи на $A[1..n]$. И на ред 10 алгоритъмът връща точно това. □

2.2 Анализ на сложността

"Сложност на алгоритъм" е мярка за това, колко ресурси полза този алгоритъм. "Сложен" в този смисъл означава "ползва много ресурси". За какви ресурси става дума?

- Ресурсът, който най-често имаме предвид, е времето. Естествено е, че 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 е, че 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 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 [114, стр. 156], има много други възможни ресурси, наречени там *complexity measures*, които можем да разглеждаме. 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 [75] и [72].
Има елегантна, много обща дефиниция на Blum на "complexity measure", която покрива времето, паметта и "мастилото". Интересуващият се читател може да я намери в [114, стр. 156], като оригиналната статия е [21].
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 10: Големина на входа – определение, което няма да използваме

За произволен алгоритъм, *големина на входа* при конкретен вход е дължината на стринга, който кодира този вход в избраната система за кодиране.

Това определение е полезно за теория на сложността, която отчита и кодирането на данните. Тъй като вече приехме, че няма да разглеждаме кодирания, Определение 10 не върши работа за целите на този курс. Терминът "големина на входа" ще дефинираме непрецизно, с голяма доза условност.

Конвенция 1: Всяко число има големина на представянето единица

За целите на тези лекции приемаме, че всяко число, цяло или рационално, е записано в единица памет, независимо от стойността си.

Това не означава, че ползваме само един бит за число—това очевидно би било невъзможно—а че има константа големина на компютърна памет, предварително фиксирана, такава, че всяко число се "побира" в нея, колкото и да е голямо. 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 тривиално се опровергава с принципа на Dirichlet, но е удобно опростяване.

Определение 11: 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 8 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. Всъщност броят на елементите на матрицата е n^2. Следователно, изразявайки сложността като функция на n, ние я изразяваме не чрез големината на входа, а чрез корен-квадратен от големината на входа[†]. Вижте Конвенция 14.

[†]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 $\Theta\left(N^{\frac{3}{2}}\right)$, където N е големината на входа. За да разберете тази забележка под черта, трябва да познавате съдържанието на Подсекция 2.3.3.

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) [31].

```
Insertion Sort(A[1 .. 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 ← j − 1
7      A[j + 1] ← key
```

Колко бързо работи той? От практическа гледна точка, този въпрос е за физическото време, което отнема работата на някаква негова програмна реализация върху истински компютър. От тази гледна точка трябва да отчитаме следните фактори:

- За какъв вход става дума. На свой ред, този фактор има две компоненти:
  - Колко числа има във входа. Ясно е, че колкото по-голямо е n, толкова по-бавно работи алгоритъмът.
  - При една и съща големина, какви са конкретните стойности на числата. Лесно се забелязва, че Insertion Sort ще работи по-бързо, ако входът вече е сортиран, например $[1, 2, 3, 4]$. Нататък ще видим, че този алгоритъм работи най-бавно, когато входът е сортиран обратно, например $[4, 3, 2, 1]$.
- Каква е конкретната програмна реализация. Според Skiena (вж. определението на стр. 3), алгоритъмът е идеята зад някаква програма. Но истинските програми са повече от идеите зад тях, те имат конкретна реализация на конкретен език за програмиране. Очевидно някои реализации ще са по-бързи от други, защото са по-грамотно написани. Компилаторът не може да компенсира неграмотно писане на софтуер.
- Какъв е компилаторът.
- Каква е виртуалната машина, ако има такава.
- Каква е операционната система.
- Каква е компютърната архитектура.

Да се даде точен отговор колко бързо би работила конкретна програма—например, с точност до наносекунда—реализираща Insertion Sort, на конкретен език, върху конкретен компютър, е практически невъзможно дори за фиксиран вход[†]. Толкова прецизен отговор не е и необходим.

За да може да правим смислени изводи за бързодействието на алгоритми, правим серия от опростявания.

[†]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.

- Първото е, че се фокусираме именно 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. **По отношение на сложните, нетривиални задачи, печелившата стратегия за бързодействието е подобрение на алгоритъма.** Печалбата от по-бързия алгоритъм е толкова по-видима, колкото по-голям е входът.

- Второто е допускането, че всички елементарни инструкции (стъпки) се изпълняват за единица време[†], така че времето за изпълнение на алгоритъма върху някакъв вход е точно броят на елементарните инструкции, които се изпълняват по време на работата му.

Ако в този опростен модел алгоритъм A е по-бърз от алгоритъм B за една и съща задача, най-вероятно в реалния свят програмната реализация на A ще е по-бърза от тази на B, 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 (2.10) на стр. 80, в най-лошия случай работи във време, пропорционално на на n^2. Сортиращият алгоритъм HEAPSORT, както ще видим на стр. 299, в най-лошия случай работи във време, пропорционално на на $n \cdot \lg n$. Да разгледаме стойностите на n^2 и $n \cdot \log n$ 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
$n \cdot \log n$	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 тази, която реализира 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 с много над предимството на по-бързата компютърна технология, било хардуер, било софтуер.

Съществуват много по-екстремни примери за предимствата, които дават бързите 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 неголямо n, примерно $n = 200$. От

[†] Инструкциите на истински процесор се изпълняват за различен брой тактове.

[‡] При нормално грамотна реализация на двата алгоритъма.

[§] Наивен алгоритъм е алгоритъм, който може да бъде предложен от най-общи съображения. По правило терминът се използва в контекста на задачи, за които съществуват значително по-добри, но и по-неочевидни алгоритми.

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, която трябва да изпълни 2^{200} стъпки, няма да завърши работата си **НИКОГА**, независимо от изпозваната 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 [94] 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. Само че 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 [12]:

> *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 1.38×10^{-23} JK^{-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 *реликтово лъчение*, остатък от Големия взрив, изотропно електромагнитно лъчение, пронизващо пространството. Дори само заради реликтовото лъчение, температурата на тяло[б] 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 [12]. 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 принципът на Landauer е валиден, а освен това има и негови експериментални потвърждения [78].

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 [31, 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 време[в] и това винаги ще е така. Ако това беше неизменно вярно, то в недалечно бъдеще щеше да 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, интересуващ от физическите ограничения върху компютърната производителност, може да намери статията на Lloyd [100] 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 $E = mc^2$.

[a] На 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 $(1, 0)$.

[б] Става 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.

[в] Това 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 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.

[†] При машините на Turing (Определение 106), 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 106.

Конвенция 3: 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 12: 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 13: 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 двата са на едни и същи позиции, всички появи на следващия по големина елемент и в двата са на едни и същи позиции, и така нататък, всички появи на максималния елемент и в двата 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 първия—той е 6—е на позиции 2 и 5, а минималният елемент във втория—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 14: Релация $\widehat{R}_n$ на неразличимост върху целочислени редици

За всяко $n \in \mathbb{N}^+$, за всеки $X, Y \in \mathbb{Z}^n$, $X \widehat{R}_n Y$ тогава и само тогава, когато X и Y са съществено подобни.

Тривиално се показва, че за всяко n, $\widehat{R}_n$ е релация на еквиваленост. Нейните класове на еквивалентост 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 еквивалентост 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 TAOCP [86, стр. 194, задача 3]. Като брой те са точно колкото пълните преднаредби (Определение 31 в Допълнение 17); неслучайно формалното определение на задачата СОРТИРАНЕ ползва "пълна преднаредба" (Задача 9).
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 $\left\{ n \atop k \right\}$ за $1 \leqslant k \leqslant n$ е броят на разбиванията на n-елементно множество на точно k подмножества; чете се "числото на Stirling от втори род n-подмножество-k". Повече за тези числа има в Подсекция 12.4.2, където са показани и ефикасни алгоритми за пресмятането им.

И така, дадени са n елемента, но някои може да имат еднакви големини (при което си остават различни елементи). Нека броят на различните големини е k; в екстремните случаи, $k = 1$ означава всички елементи да имат една и съща големина, а $k = n$ означава да няма еднакви големини. За фиксирано k, по $\left\{ n \atop k \right\}$ начина може да бъдат групирани числата в точно k класа, като числата във всеки клас са с еднаква големина. Например, $\left\{ 4 \atop 2 \right\} = 7$, 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 големини:

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_3 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 $14 = \left\{ 4 \atop 2 \right\} 2!$

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 $\widehat{R}_n$ е

$$P_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} k! \tag{2.6}$$

Примерно, $P_4 = 75$.

Само $\sum_{k=1}^{n} \left\{ {n \atop k} \right\}$ 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 $P_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} k!$: защото разглеждаме и наредбите на дяловете на разбиване.

P_n, изразено чрез рекурентно уравнение. В сила е

$$P_n = \begin{cases} 1, & \text{ако } n = 0 \\ \sum_{k=1}^{n} \binom{n}{k} P_{n-k}, & \text{ако } n \geqslant 1 \end{cases}$$

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 $n \geqslant 1$. Слабите наредби на n елемента се разбиват по броя k на елементите с най-малка големина, където $k \in \{1, \ldots, n\}$. 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 $n - k$ елементи, като обаче всички големини на останалите надвишават m.

Независимо от начина на извеждане, $P_0 = 1$, $P_1 = 1$, $P_2 = 3$, $P_3 = 13$, $P_4 = 75$, $P_5 = 541$ и така нататък. Това е редица A000670 в онлайн енциклопедията на целочислените редици.

Да продължим с разсъжденията за броя на входовете на сортиращ 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 изпълнява една и съща инструкция и от трите копия[†]. Защо е така? Защото тези три входа са съществено подобни.

Лесно се вижда, че това остава в сила за всякакви множества от входове, които са съществено подобни.

Наблюдение 8: 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 $\left(2 \times 10^9, 1, 10^{10^{10}}\right)$, алгоритъмът връща $\left(1, 2 \times 10^9, 10^{10^{10}}\right)$.

Наблюдение 9: 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 големина n, които са от един и същи клас на еквивалентност на $\widehat{R}_n$, алгоритъмът работи по един и същи начин. Тогава броят на различните начини да работи алгоритъма е не повече от P_n, дефинирано в Допълнение 13.

Наблюдение 9 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 разглеждаме, 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 15: Сложност по време

Нека Π е изчислителна задача и A е алгоритъм за нея. За всяка големина на входа $n \in \mathbb{N}^+$, нека $\mathfrak{I}(n)$ 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

$$T_A(n) = \max\{f(\kappa) \mid \kappa \in \mathfrak{I}(n)\}$$

сложността по време на A *в най-добрия случай* е

$$P_A(n) = \min\{f(\kappa) \mid \kappa \in \mathfrak{I}(n)\}$$

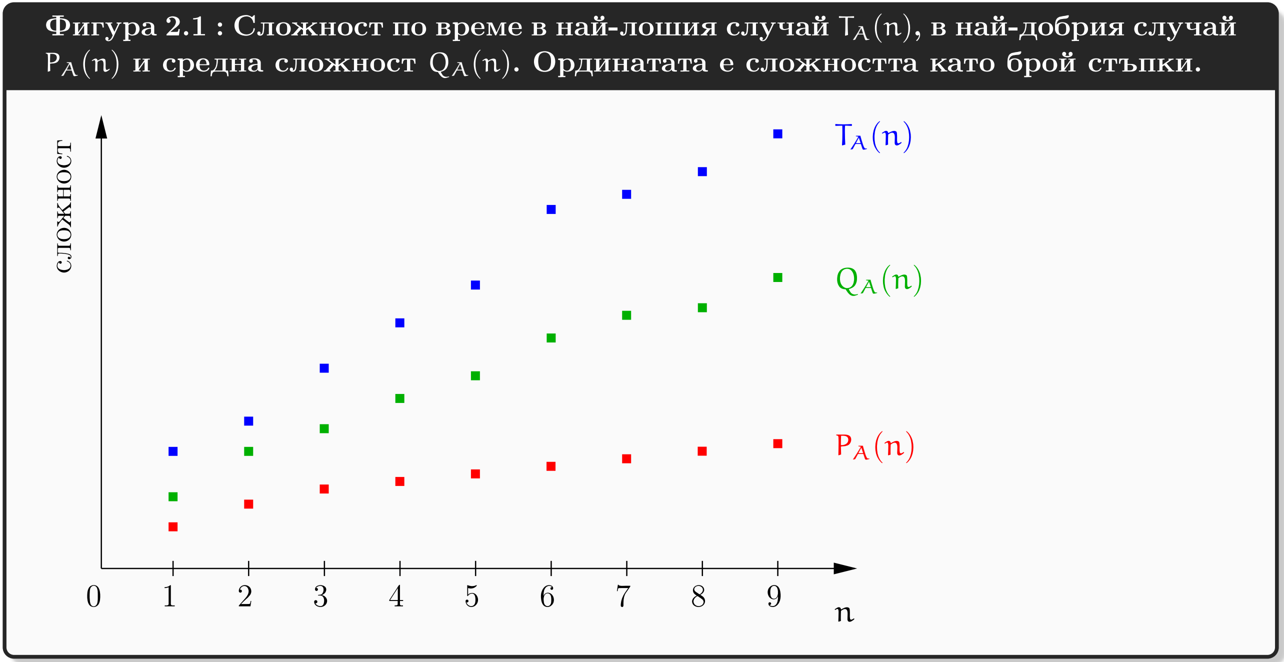
и *средната сложност по време на* A е

$$Q_A(n) = \frac{1}{|\mathfrak{I}(n)|} \sum_{\kappa \in \mathfrak{I}(n)} f(\kappa)$$

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 следваща лекция 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 даде-

на големина са еднакво вероятни. Втората причина е, че резултатът за най-лошия случай е твърда гаранция, че по-лошо не може да бъде.

Сложността в най-добрия случай не се ползва, тъй като практически всеки важен алгоритъм можа да бъде подобрен като скорост за само един вход (по един такъв вход за всяка големина). Например, всеки алгоритъм A за задачата Хамилтонов Път, която ще дефинираме нататък, може да бъде направен много бърз върху един конкретен вход (за всяка големина). По-точно, независимо как точно работи A, той може да бъде модифициран, като в самото му начало се добави тестване на една конкретна пермутация на върховете – дали задава хамилтонов път. Ако се окаже, че тя задава хамилтонов път, то отговорът на въпроса дали има хамилтонов път очевидно е утвърдителен и прекратяваме изпълнението. Ако се окаже, че тя не задава хамилтонов път, от това не следва, че няма такъв, и продължаваме с това, което A прави. С други думи, към всеки алоритъм, дори най-бавния, може да се пришие напълно изкуствено тестване на някакъв конкретен вход. Този трик би подобрил изкуствено сложността в най-добрия случай до теоретично оптималната, без това да ни казва нищо съществено за това, колко е бърз A. В примера с Хамилтонов Път, според преобладаващото мнение, всеки алгоритъм за тази задача е с експоненциална сложност както в най-лошия, така и в средния случай, но чрез споменатия трик може да направим кой да е алгоритъм за тази задача да работи в линейно време в най-добрия случай. Такова изкуствено увеличаване на бързодействието чрез подобрение върху само един вход е напълно безполезно.

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*.

Наблюдение 10: 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, $\forall n : S_A(n) \leqslant T_A(n)$. 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 памет, както е според Наблюдение 10; неравенството $\boxed{\forall n : S_A(n) \leqslant T_A(n)}$ позволява $S_A(n)$ да е колкото искаме по-малко от $T_A(n)$.
В контраст с това, при машините на Turing (Определение 106) 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 107) се определя от състоянието и съдържанието на лентата (и позицията на главата, но да оставим това настрани 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 11

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 10, $T_A(n) \leqslant c_1^{c_2 \cdot S_A(n)}$. Иначе казано, използвайки Θ-нотацията (Подсекция 2.3.3),

$$\Theta(\lg T_A(n)) \leqslant S_A(n) \leqslant T_A(n)$$

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 [85] става ясно, че за 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 $\log_2 k$[†], парадоксът изчезва. Тогава Евклидовият алгоритъм работи във време, пропорционално на квадрата на размера на входа (вж. [85]).

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, съотвените 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. 66. 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 това. Спазваме допусканията, 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]
```

[†] $\log_2 k$ е горе-долу броят на битовете, необходими за записването на n в двоична позиционна бройна система.

6 $\quad\quad j \leftarrow j - 1$
7 $\quad A[j+1] \leftarrow key$

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** $i \leftarrow a$ **to** b се изпълнява точно $b - a + 2$ пъти (ако $a \leqslant b$), защото:

  - тялото на цикъла се изпълнява точно $b - a + 1$ пъти,
  - а **for** $i \leftarrow a$ **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.

- 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 $i \in \{2, 3, \ldots, n\}$. Очевидно $k_i \in \{1, 2, \ldots, i\}$, 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.

[†] При програмиране на истински компютър това не е така. Следният тривиален **for** цикъл на езика C:

```
for (i=2; i <= n; i++);
```

като фрагмент от програма се компилира върху Intel iCore3 процесор, Ubuntu Linux, 64 битов gcc компилатор без оптимизация, в следния код (адресите и отместванията, естествено, варират):

```
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, при истинския компютър инкрементирането с единица `addl`, проверката за край на изпълнението на цикъла `cmp`, и условният скок `jle` в изпълнението, са отделни инструкции.

И така, в нашия опростен модел, изпълнението на INSERTION SORT става в

$$n + 3(n-1) + 3\left(\sum_{i=2}^{n} k_i\right) - 2(n-1) \tag{2.7}$$

тоест,

$$2n - 1 + 3\sum_{i=2}^{n} k_i \tag{2.8}$$

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.

Показахме, че сложността по време в най-добрия случай е

$$P(n) = 2n - 1 + 3\sum_{i=2}^{n} 1 = 2n - 1 + 3n - 3 = 5n - 4 \tag{2.9}$$

и сложността по време в най-лошия случай е

$$T(n) = 2n - 1 + 3\sum_{i=2}^{n}(i-1) = 2n - 1 + \frac{3n(n-1)}{2} = \frac{3n^2+n}{2} - 1 \tag{2.10}$$

Ще анализираме сложността на още един алгоритъм по аналогичен начин.

```
SELECTION SORT(A[1..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 3 се изпълнява общо k пъти за някакво k, такова че $0 \leqslant k \leqslant \frac{n(n-1)}{2}$, в зависимост от конкретния вход. Общо сложността е

$$n + \frac{(n+2)(n-1)}{2} + \frac{n(n-1)}{2} + k = n^2 + n - 1 + k \tag{2.11}$$

Тогава сложността в най-добрия случай е

$$P'(n) = n^2 + n - 1 \tag{2.12}$$

и сложността в най-лошия случай е

$$T'(n) = n^2 + n - 1 + \frac{n(n-1)}{2} = \frac{3n^2+n}{2} - 1 \tag{2.13}$$

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 (2.9), (2.10), (2.12) и (2.13) не са необходими. Нашият модел на сложността е достатъчно далече от реалността, така че събираемото -1, което имаме в (2.13), 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, **събираемото -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 последователности) 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 умението 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 (2.13) 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).

И така, най-същественото за сложностите на Insertion Sort и Selection Sort е, че в най-лошия случай[†] те са квадратични функции.

В нашата редица от допускания, които опростяват нещата и ни позволяват изобщо да правим анализ на сложността на алгоритми, следва допускането, че степента на нарастване на функцията на сложността е всичко, което ни интересува за нея. Защо? Защото се интересуваме само от тенденцията на изменение на сложността при **неограничено нарастване** на големината на входа. Никаква конкретна големина на входа не ни интересува. Никаква нейна конкретна стойност, ако ще да е $10\,000\,000^{10\,000\,000^{10\,000\,000}}$, не е достатъчно голяма, за да спрем до нея. След малко ще дадем прецизно определение на "степен на нарастване", засега ще се задоволим с примера с анализа на двата алгоритъма горе. И на двата функциите на сложността в най-лошия случай са квадратични. Ще смятаме, че сложността им е еднаква: квадратична.

И така, правим още допускания:

Допускане 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 24, разглеждаме само това събираемо, игнорирайки останалите.

Допускане 2. За всеки два алгоритъма, ако техните изрази за сложността са асимптотично еквивалентни функции в смисъла на Определение 22, ще смятаме, че тези алгоритми работят еднакво бързо.

Допускане 1 не винаги е реалистично. Смисълът да се разглежда само тенденция 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

[†]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.

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, $n\sqrt{n}+10\,000\,000^{10\,000\,000^{10\,000\,000}}n$, вторият работи по-бързо, защото неговата водеща функция $n\sqrt{n}$ расте по-бавно от n^2, водещата функция на първия. Очевидно съществува стойност на големината на входа n_0, такава че за всяко $n \geqslant n_0$, вторият алгоритъм печели като бързодействие, но това n_0 е твърде голямо. Както вече стана ясно, то е число, което никога не е големина на вход в реалния свят.

Допускане 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 с една и съща сложност, а именно квадратична, и са еднакво бързи. "Оправданието" ни 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 $10\,000\,000^{10\,000\,000^{10\,000\,000}}$, така че опростяването не е прекалено грубо. В теорията обаче са известни алгоритми, при които тази мултипликативна константа е кула от степени на двойката

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$$

чиято височина е кула от степени на двойката, чиято височина е кула от степени на двойката, и така нататък няколко пъти [77]. Очевидно такъв алгоритъм е непрактичен дори за вход с големина единица.

Да обобщим. Когато разглеждаме тенденцията на нарастването при неограничено нарастване на аргумента n, ние просто правим поредното опростяващо допускане. Причината да го правим е, че **така е по-лесно**. Освен това, практиката показва, че често—но **не винаги**—резултатите, които ни дава това опростяване, са добре корелирани с качествата на реалните програми, реализиращи съответните алгоритми.

По думите на Jack Edmonds [38, 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. Или можем да "опорочим" квадратичен алгоритъм като Insertion Sort, правейки го привидно линеен. 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 [114, 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 A, изразена в Θ-нотация, е по-малка от сложността на алгоритъм B, пак в Θ-нотация, твърде възможно е практическата реализация на алгоритъм A да работи по-бързо от практическата реализация на алгоритъм B.

Определение 16: Асимптотично положителна функция

Нека $f : \mathbb{R}^+ \to \mathbb{R}^+$. Казваме, че f е *асимтотично положителна,* ако

$$\exists n_0 \in \mathbb{R}^+ \, \forall n \geqslant n_0 : f(n) > 0$$

На прост български, иска се функцията да е строго положителна от някоя стойност на аргумента нататък.

Конвенция 4

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 "$\exists n_0 \, \forall n \geqslant n_0$". На прост български се казва "за всички достатъчно големи n".

Определение 17: $\Theta(g(n))$

За всяка функция $g(n)$:

$$\Theta(g(n)) \stackrel{\text{def}}{=} \left\{f(n) \mid \exists c_1, c_2 > 0\ \forall n \nearrow\ :\ 0 \leqslant c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)\right\}$$

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: "$h(n)$ е Тета-голямо от $g(n)$".

Ето пример за педантично доказателство, че една функция е Тета-голямо от друга функция.

Задача 3

Докажете, че $n^2 + 10n + 12 = \Theta(n^2)$, използвайки Определение 17.

Решение: Съгласно определението, трябва да покажем константи $c_1, c_2 > 0$ и стойност n_0 на аргумента, такива че за всяко $n \geqslant n_0$ да е вярно

$$0 \leqslant c_1 n^2 \leqslant n^2 + 10n + 12 \leqslant c_2 n^2$$

Това са всъщност три неравенства. Първото неравенство $0 \leqslant c_1 n^2$ е очевидно: щом $c_1 > 0$, то можем да вземем $n_0 = 1$ и тогава $\forall n \geqslant n_0 : 0 \leqslant c_1 n^2$. Да разгледаме второто неравенство $c_1 n^2 \leqslant n^2 + 10n + 12$. Имаме право да разделим двете страни на n^2, защото разглеждаме само положителни n:

$$c_1 n^2 \leqslant n^2 + 10n + 12 \leftrightarrow c_1 \leqslant 1 + \frac{10}{n} + \frac{12}{n^2}$$

Можем да вземем просто $c_1 = 1$, понеже $\forall n \geqslant n_0 : 1 \leqslant 1 + \frac{10}{n} + \frac{12}{n^2}$. Тук отново n_0 е 1. Да разгледаме третото неравенство $n^2 + 10n + 12 \leqslant c_2 n^2$:

$$n^2 + 10n + 12 \leqslant c_2 n^2 \leftrightarrow 1 + \frac{10}{n} + \frac{12}{n^2} \leqslant c_2$$

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 $\forall n \geqslant n_0 : 1 + \frac{10}{n} + \frac{12}{n^2} \leqslant 100$. Други стойности за c_2 и n_0 също биха свършили работа за доказателството, примерно $c_2 = 3$ и $n_0 = 12$.

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 $n_0 = 12$.

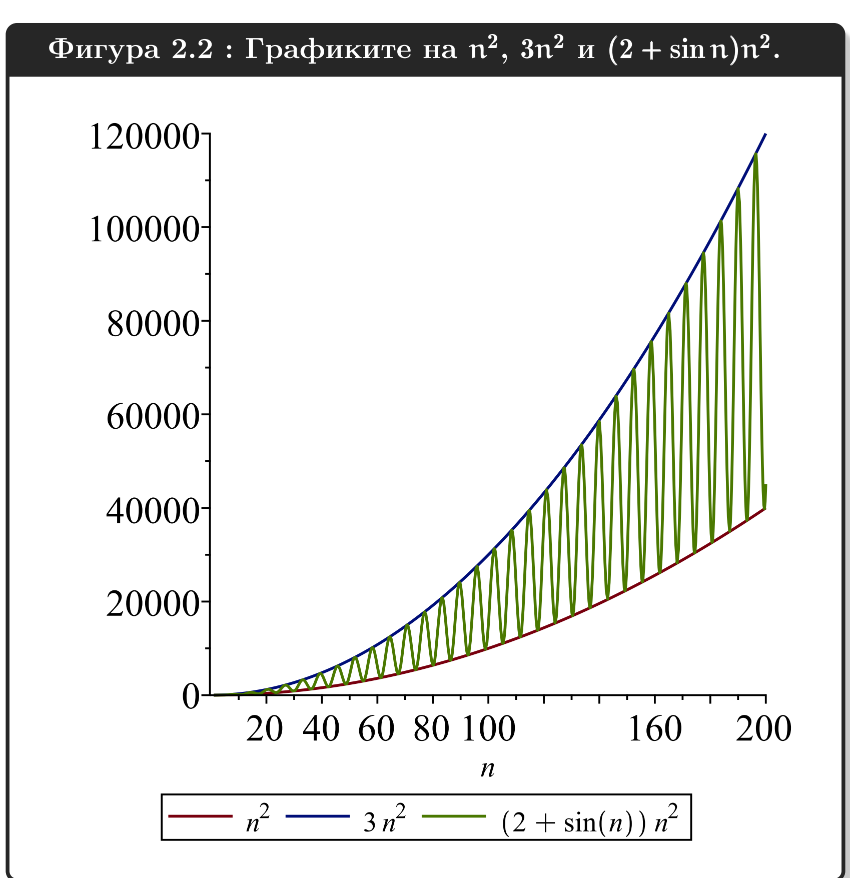
И накрая отбелязваме, че съществуват повече от една наредени тройки (c_1, c_2, n_0), които може да се използват в доказателството. □

Наблюдение 12

Забележете, че “$\lim_{n\to\infty} \frac{h(n)}{g(n)}$ 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 $\lim_{n\to\infty} \frac{h(n)}{g(n)}$ 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 $\lim_{n\to\infty} \frac{(2+\sin n)n^2}{n^2} = \lim_{n\to\infty} 2 + \sin n$ 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”.

Фигура 2.2 : Графиките на n^2, $3n^2$ и $(2+\sin n)n^2$.



Лема 4

За всички функции $f(n)$, $g(n)$ и $h(n)$:

$$f(n) = \Theta(f(n)) \tag{2.14}$$

$$f(n) = \Theta(g(n)) \rightarrow g(n) = \Theta(f(n)) \tag{2.15}$$

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n)) \tag{2.16}$$

† Фигура 2.2 е направена с Maple(TM).

Доказателство: Равенствата 2.14 и 2.16 следват очевидно от рефлексивността и транзитивността на релацията "$\preccurlyeq$" в Определение 17. Да разгледаме равенство 2.15. В термините на Определение 17, то казва:

$$\left(\exists c_1, c_2 > 0 \, \exists n_0 > 0 \, \forall n \geqslant n_0 : 0 \leqslant c_1.g(n) \leqslant f(n) \leqslant c_2.g(n)\right) \rightarrow$$
$$\left(\exists c_1', c_2' > 0 \, \exists n_0' > 0 \, \forall n \geqslant n_0' : 0 \leqslant c_1'.f(n) \leqslant g(n) \leqslant c_2'.f(n)\right)$$

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 3

От равенство 2.15 на Лема 4 веднага следва, че $f(n) = \Theta(g(n))$ тогава и само тогава, когато $g(n) = \Theta(f(n))$.

Определение 18

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 тези алгоритми няма. Те имат квадратична сложност по време и толкова.

Конвенция 5: 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 15: Друга интерпретация на появата на Θ вляво

Според Knuth [83], 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 17; 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 6: 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 7: $\Theta(1)$ 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^ϵ, където ϵ е константа" в Определение 26 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 19, (2.17) 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. 456, 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 19, 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 на стр. 239, 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. 76, 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—понякога 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 19: $O(g(n)), \Omega(g(n)), o(g(n)), \omega(g(n))$

За всяка функция $g(n)$:

$$O(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0\ \forall n \nearrow\ :\ 0 \leqslant f(n) \leqslant c \cdot g(n)\} \tag{2.17}$$

$$\Omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0\ \forall n \nearrow\ :\ 0 \leqslant c \cdot g(n) \leqslant f(n)\} \tag{2.18}$$

$$o(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0\ \forall n \nearrow\ :\ 0 \leqslant f(n) < c \cdot g(n)\} \tag{2.19}$$

$$\omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \forall c > 0\ \forall n \nearrow\ :\ 0 \leqslant c \cdot g(n) < f(n)\} \tag{2.20}$$

Както и при Тета-нотацията, принадлежността към тези множества означаваме не с "$\in$", а с "$=$", примерно пишем $f(n) = O(g(n))$, $h(n) = \omega(\phi(n))$ и така нататък.

Ако $f(n) = O(g(n))$, казваме, че $g(n)$ е *асимптотична горна граница* за $f(n)$. Примерно, вярно е, че:

$$n = O(n^2) \qquad n^2 = O(n^2) \qquad n^2 + 1000n + 10000 = O(n^2) \qquad 10n^2 = O(n^2) \qquad 1 = O(n^2)$$

Съответно, функцията n^2 е асимптотична горна граница за всяка от функциите n, n^2, $n^2 + 1000n + 10000$, $10n^2$ и 1.

От друга страна обаче, $\frac{1}{1\,000\,000}n^3 \neq O(n^2)$. Да видим защо. Да допуснем, че $\frac{1}{1\,000\,000}n^3 = O(n^2)$. Тогава от (2.17) знаем, че съществува положително c, такова че за всички достатъчно големи n е изпълнено:

$$\frac{1}{1\,000\,000}n^3 \leqslant c \cdot n^2$$

Разделяме на n^2 и получаваме еквивалентното неравенство

$$\frac{1}{1\,000\,000}n \leqslant c$$

което е същото като

$$n \leqslant 1\,000\,000\,c$$

Веднага се вижда, че това не може да е вярно. Колкото и голямо c да изберем, съществува n, което е по-голямо от $1\,000\,000c$. Ерго, функцията n^2 не е асимптотична горна граница за функцията $\frac{1}{1\,000\,000}n^3$.

Забележете съществената разлика между дефинициите на $O(g(n))$ и $o(g(n))$. 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. Затова, ако $f(n) = o(g(n))$, казваме, че $g(n)$ е *строга асимптотична горна граница* за $f(n)$.

Дуално, $\Omega(\,)$ и $\omega(\,)$ задават съответно *асимптотична долна граница* и *строга асимптотична долна граница*.

В Лема 5 и Лема 6, нотациите и от двете страни на равенствата означават множества (съгласно дефинициите), а знаците за равенство означават равенства между множества (а не принадлежност към множества).

Лема 5

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 19(2.17) и Определение 19(2.18), това е същото като

$$(\exists c' > 0\, \exists n_0'\, \forall n \geqslant n_0' \,:\, f(n) \leqslant c' \cdot g(n)) \wedge (\exists c'' > 0\, \exists n_0''\, \forall n \geqslant n_0 \,:\, c'' \cdot g(n) \leqslant f(n))$$

Но тогава очевидно съществуват положителни c', c'', такива че за всяко $n \geqslant \max\{n_0', n_0''\}$ е изпълнено

$$c'' \cdot g(n) \leqslant f(n) \leqslant c' \cdot g(n)$$

Съгласно Определение 17, $f(n) \in \Theta(g(n))$. В обратната посока, да допуснем, че $f(n) \in \Theta(g(n))$. Тогава съгласно Определение 17:

$$\exists c_1, c_2 > 0\, \exists n_0\, \forall n \geqslant n_0 \,:\, 0 \leqslant c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$$

Тривиално следва, че

$$(\exists c > 0\, \exists n_0\, \forall n \geqslant n_0 \,: f(n) \leqslant c \cdot g(n)) \wedge (\exists c > 0\, \exists n_0\, \forall n \geqslant n_0 \,: c \cdot g(n) \leqslant f(n))$$

Съгласно Определение 19, това е същото като

$$f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)).$$

□

Лема 6

За всяка функция $g(n)$:

$$o(g(n)) \cap \omega(g(n)) = \emptyset \quad (2.21)$$
$$O(g(n)) \cap \omega(g(n)) = \emptyset \quad (2.22)$$
$$o(g(n)) \cap \Omega(g(n)) = \emptyset \quad (2.23)$$

Доказателство: Ще докажем (2.21). Да разгледаме произволна $f(n) \in o(g(n)) \cap \omega(g(n))$. Съгласно Определение 19(2.19) и Определение 19(2.20), това е същото като

$$(\forall c > 0\, \exists n_0\, \forall n \geqslant n_0 \,:\, f(n) < c \cdot g(n)) \wedge (\forall c > 0\, \exists n_0\, \forall n \geqslant n_0 \,:\, c \cdot g(n) < f(n))$$

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 (2.22). Да разгледаме произволна $f(n) \in O(g(n)) \cap \omega(g(n))$. Съгласно Определение 19(2.17) и Определение 19(2.20), това е същото като

$$(\exists c > 0\, \exists n_0\, \forall n \geqslant n_0\, : f(n) \leqslant c \cdot g(n)) \wedge (\forall c > 0\, \exists n_0\, \forall n \geqslant n_0\, :\, c \cdot g(n) < f(n))$$

Щом изразът вдясно съдържа $\forall c$, в частност той 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 $n > \tilde{n}$:

$$f(n) \leqslant c \cdot g(n) \wedge f(n) > c \cdot g(n)$$

Това очевидно е невъзможно.

Ще докажем (2.23). Да разгледаме произволна $f(n) \in o(g(n)) \cap \Omega(g(n))$. Съгласно Определение 19(2.19) и Определение 19(2.18), това е същото като

$$(\forall c > 0\, \exists n_0\, \forall n \geqslant n_0\, :\, f(n) < c \cdot g(n)) \wedge (\exists c > 0\, \exists n_0\, \forall n \geqslant n_0\, :\, c \cdot g(n) \leqslant f(n))$$

Щом изразът вляво съдържа $\forall c$, в частност той 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 $n > \tilde{n}$:

$$f(n) > c \cdot g(n) \wedge f(n) \leqslant c \cdot g(n)$$

Това очевидно е невъзможно. □

Лема 7

За всички функции $f(n)$, $g(n)$:

$$f(n) = O(g(n)) \leftrightarrow g(n) = \Omega(f(n)) \quad (2.24)$$
$$f(n) = o(g(n)) \leftrightarrow g(n) = \omega(f(n)) \quad (2.25)$$

В [31, стр. 52] тази симетрия е наречена *transpose symmetry*. На български се казва *транспонирана симетрия*. Ерго, O-голямо и Ω-голямо са транспонирано симетрични. O-малко и Ω-малко също са транспонирано симетрични.

Доказателство: Ще докажем (2.24). Наистина, съгласно Определение 19(2.17), $f(n) = O(g(n))$ е същото като

$$\exists c > 0\, \exists n_0\, \forall n \geqslant n_0\, :\, f(n) \leqslant c \cdot g(n)$$

Но $f(n) \leqslant c \cdot g(n)$ е същото като $\frac{1}{c} f(n) \leqslant g(n)$; тъй като $c > 0$, $\frac{1}{c}$ е дефинирано. Заключаваме, че

$$\exists c' > 0\, \exists n_0\, \forall n \geqslant n_0\, :\, c' \cdot f(n) \leqslant g(n)$$

а именно за $c' = \frac{1}{c}$. Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 19(2.18).

Ще докажем (2.25). Наистина, съгласно Определение 19(2.19), $f(n) = o(g(n))$ е същото като

$$\forall c > 0\, \exists n_0\, \forall n \geqslant n_0 \;:\; f(n) < c \cdot g(n)$$

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

$$\forall c' > 0\, \forall n \nearrow \;:\; c' \cdot f(n) < g(n)$$

Но това е същото като $g(n) = \Omega(f(n))$ съгласно Определение 19(2.18). □

Нотациите o и ω може да се дефинират чрез граници.

Лема 8

За всички функции $f(n)$, $g(n)$:

$$f(n) = o(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) = \omega(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Допълнение 16: Нотациите O и o в анализа

В математическия анализ нотациите O и o се ползват *от 19 век*, тоест, много преди модерната теория на алгоритмите. Да развием функцията $\sin(x)$ в ред на Taylor в околността на точка 0:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$$

Често срещан запис на това развиване е:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + O(x^7) \tag{2.26}$$

(2.26) е по-прецизен запис от

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

защото "$\approx$" може да означава какво ли не. (2.26) казва, че $\sin(x)$ е точно равен на сумата на $x - \frac{x^3}{3!} + \frac{x^5}{5!}$ и някаква функция, която в околността на точка 0 намалява, в асимптотичния смисъл, като x^7[a]. Очевидно нотацията "$O(x^7)$" в (2.26) има смисъл, различен от този на Определение 19(2.17). В (2.26) се има предвид не тенденция при неограничено нарастване на аргумента—както е в Определение 19—а тенденция, когато аргументът клони към определена стойност, в случая 0.
Записът "$f(x) = O(g(x))$ при $x \to a$" е кратък запис за следното:

$$\exists c > 0\, \exists \delta > 0 : |f(x)| \leqslant c g(x), \text{ ако } 0 < |x - a| < \delta$$

В (2.26) "$x \to 0$" е изпуснато, но то се подразбира.
За нотацията o нещата са аналогични. Записът "$f(x) = o(g(x))$ при $x \to a$" е кратък запис за следното:

$$\forall c > 0\, \exists \delta > 0 : |f(x)| \leqslant cg(x), \text{ ако } 0 < |x - a| < \delta$$

В заключение, нотациите O и o винаги изразяват тенденции за нарастване (или намаляване) на функции. Ако x е аргументът, тези тенденции са или при $x \to a$ за някакво фиксирано a, или при $x \to \infty$. При изследването на сложността на алгоритми се интересуваме само от тенденцията при $x \to \infty$, поради което асимптотичните нотации за целите на тези лекционни записки винаги са спрямо тенденцията на неограничено нарастване на аргумента.

[a] Тъй като аргументът е по-малък от единица, x^7 изразява намаляване, а не нарастване.

2.3.5 Релации $\asymp$, $\preceq$, $\prec$, $\succeq$ и $\succ$ и някои техни свойства

Определение 20: 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: наместо "$f(n) = \Theta(g(n))$" можем да запишем "$f(n) \asymp g(n)$" и обратно.

Ако ползваме записа с "$\asymp$" наместо "Θ", необходимостта от Конвенция 5 изчезва по очевидни причини: в израза няма "Θ". Също така изчезва и необходимостта от Конвенция 6, понеже двете страни на знака $\asymp$ са равнопоставени. Ерго, следните два записа са еднакво приемливи:

$$10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n \asymp n^2$$
$$n^2 \asymp 10\sqrt{7}n^2 + 15.5n\sqrt{n} + 35n$$

Определение 21: Релации $\preceq$, $\prec$, $\succeq$ и $\succ$

За всички функции $f(n)$ и $g(n)$:

- $f(n) \preceq g(n)$ тогава и само тогава, когато $f(n) = O(g(n))$,
- $f(n) \prec g(n)$ тогава и само тогава, когато $f(n) = o(g(n))$,
- $f(n) \succeq g(n)$ тогава и само тогава, когато $f(n) = \Omega(g(n))$,
- $f(n) \succ g(n)$ тогава и само тогава, когато $f(n) = \omega(g(n))$.

Лема 9 се доказва тривиално с Лема 4 и Следствие 3.

Лема 9

$\asymp$ е релация на еквивалентност.

Определение 22: 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 23: Релации на асимптотични сравнения

Петте релации $\asymp$, $\preceq$, $\prec$, $\succeq$ и $\succ$ 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 6) на $\{\preceq, \succeq\}$ и на $\{\prec, \succ\}$ 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 $\rhd$, е вярно, че $f(n) \rhd g(n)$, то съществува релация измежду петте, да я запишем като $\lhd$, такава че $g(n) \lhd f(n)$.

Съгласно Теорема 12(2.39), асимптотично несравними функции съществуват.

Определение 24: Асимптотично по-бавно и асимптотично по-бързо нарастване

Ако $f(n) \prec g(n)$, казваме, че $f(n)$ *расте асимптотично по-бавно от* $g(n)$. Ако $\phi(n) \succ \psi(n)$, казваме, че $\phi(n)$ *расте асимптотично по-бързо от* $\psi(n)$.

Заради транспонираната симетрия (вж. Следствие 6) на $\{\prec, \succ\}$ имаме право освен това да кажем, че $g(n)$ расте асимптотично по-бързо от $f(n)$ и че $\psi(n)$ расте асимптотично по-бавно от $\phi(n)$.

Лема 5, Определение 20 и Определение 21 влекат непосредствено Следствие 4.

Следствие 4

За всички функции $f(n)$ и $g(n)$, $f(n) \asymp g(n) \leftrightarrow f(n) \preceq g(n) \wedge f(n) \succeq g(n)$.

Лема 6 и Определение 21 влекат непосредствено Следствие 5.

Следствие 5: Несъвместими релации на асимптотични сравнения

Не съществуват функции $f(n)$ и $g(n)$, такива че:

$$f(n) \prec g(n) \wedge f(n) \succ g(n)$$
$$f(n) \preceq g(n) \wedge f(n) \succ g(n)$$
$$f(n) \prec g(n) \wedge f(n) \succeq g(n)$$

Лема 7 и Определение 21 влекат непосредствено Следствие 6.

Следствие 6: Транспонирана симетрия

За всички функции $f(n)$, $g(n)$:

$$f(n) \preceq g(n) \leftrightarrow g(n) \succeq f(n)$$
$$f(n) \prec g(n) \leftrightarrow g(n) \succ f(n)$$

Лема 10 е точно съответствие на Лема 8:

Лема 10

За всички функции $f(n)$, $g(n)$:

$$f(n) \prec g(n) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) \succ g(n) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Лема 11

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \prec g(n) \to f(n) \preceq g(n) \tag{2.27}$$
$$f(n) \succ g(n) \to f(n) \succeq g(n) \tag{2.28}$$

Доказателство: Доказателството е тривиално и в двата случая. За (2.27), 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: $f(n) \leqslant c \cdot g(n)$. За (2.28) доказателството е аналогично. □

Лема 12

Нека $f(n)$ и $g(n)$ са произволни асимптотично положителни функции. Тогава

$$\neg\big(\, f(n) \preceq g(n) \to f(n) \prec g(n) \,\big) \tag{2.29}$$
$$\neg\big(\, f(n) \succeq g(n) \to f(n) \succ g(n) \,\big) \tag{2.30}$$

Доказателство: Като контрапример и за (2.29), и за (2.30) да разгледаме $f(n) = n$ и $g(n) = n$. Очевидно $f(n) \preceq g(n)$ и $f(n) \succeq g(n)$, но $f(n) \nprec g(n)$ и $f(n) \nsucc g(n)$. □

Строгите релации $\prec$ и $\succ$ са несъвместими съответно със $\succeq$ и $\preceq$.

Лема 13

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \succeq g(n) \to f(n) \nprec g(n) \tag{2.31}$$
$$f(n) \prec g(n) \to f(n) \nsucceq g(n) \tag{2.32}$$
$$f(n) \preceq g(n) \to f(n) \nsucc g(n) \tag{2.33}$$
$$f(n) \succ g(n) \to f(n) \npreceq g(n) \tag{2.34}$$

Доказателство: Всяка от тези импликации се доказва тривиално с допускане на противното. Да разгледаме само (2.31). Противното е

$$\neg(f(n) \succeq g(n) \to f(n) \nprec g(n)) \equiv \qquad \text{(* свойства на импликацията *)}$$
$$\neg(\neg(f(n) \succeq g(n)) \vee \neg(f(n) \prec g(n))) \equiv \qquad \text{(* з-н на De Morgan, з-н за дв. отриц. *)}$$
$$f(n) \succeq g(n) \wedge f(n) \prec g(n)$$

Но от Следствие 5 знаем, че такива $f(n)$ и $g(n)$ няма. □

Строгите релации $\prec$ и $\succ$ са несъвместими с асимптотичната еквивалентност $\asymp$.

Лема 14

За всеки две функции $f(n)$ и $g(n)$:

$$f(n) \asymp g(n) \rightarrow f(n) \nprec g(n) \tag{2.35}$$
$$f(n) \prec g(n) \rightarrow f(n) \not\asymp g(n) \tag{2.36}$$
$$f(n) \asymp g(n) \rightarrow f(n) \nsucc g(n) \tag{2.37}$$
$$f(n) \succ g(n) \rightarrow f(n) \not\asymp g(n) \tag{2.38}$$

Доказателство: Следват от Лема 13 и Следствие 4. Да разгледаме само (2.35). От Следствие 4 имаме $f(n) \asymp g(n) \rightarrow f(n) \succeq g(n)$. От Лема 13 имаме $f(n) \succeq g(n) \rightarrow f(n) \nprec g(n)$. Прилагаме правилото хипотетичен силогизъм от съждителната логика $p \rightarrow q \wedge q \rightarrow r \vdash p \rightarrow r$ и получаваме желания 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 12 казва, че тези възможности са точно шест.

Теорема 12: Различните възможности при асимптотично сравняване на функции

За всеки две асимптотично положителни функции $f(n)$ и $g(n)$ в сила е точно едно от следните:

$$f(n) \nprec g(n) \wedge f(n) \npreceq g(n) \wedge f(n) \not\asymp g(n) \wedge f(n) \nsucc g(n) \wedge f(n) \nsucceq g(n) \tag{2.39}$$
$$f(n) \nprec g(n) \wedge \mathbf{f(n) \preceq g(n)} \wedge f(n) \not\asymp g(n) \wedge f(n) \nsucc g(n) \wedge f(n) \nsucceq g(n) \tag{2.40}$$
$$\mathbf{f(n) \prec g(n)} \wedge \mathbf{f(n) \preceq g(n)} \wedge f(n) \not\asymp g(n) \wedge f(n) \nsucc g(n) \wedge f(n) \nsucceq g(n) \tag{2.41}$$
$$f(n) \nprec g(n) \wedge \mathbf{f(n) \preceq g(n)} \wedge \mathbf{f(n) \asymp g(n)} \wedge f(n) \nsucc g(n) \wedge \mathbf{f(n) \succeq g(n)} \tag{2.42}$$
$$f(n) \nprec g(n) \wedge f(n) \npreceq g(n) \wedge f(n) \not\asymp g(n) \wedge \mathbf{f(n) \succ g(n)} \wedge \mathbf{f(n) \succeq g(n)} \tag{2.43}$$
$$f(n) \nprec g(n) \wedge f(n) \npreceq g(n) \wedge f(n) \not\asymp g(n) \wedge f(n) \nsucc g(n) \wedge \mathbf{f(n) \succeq g(n)} \tag{2.44}$$

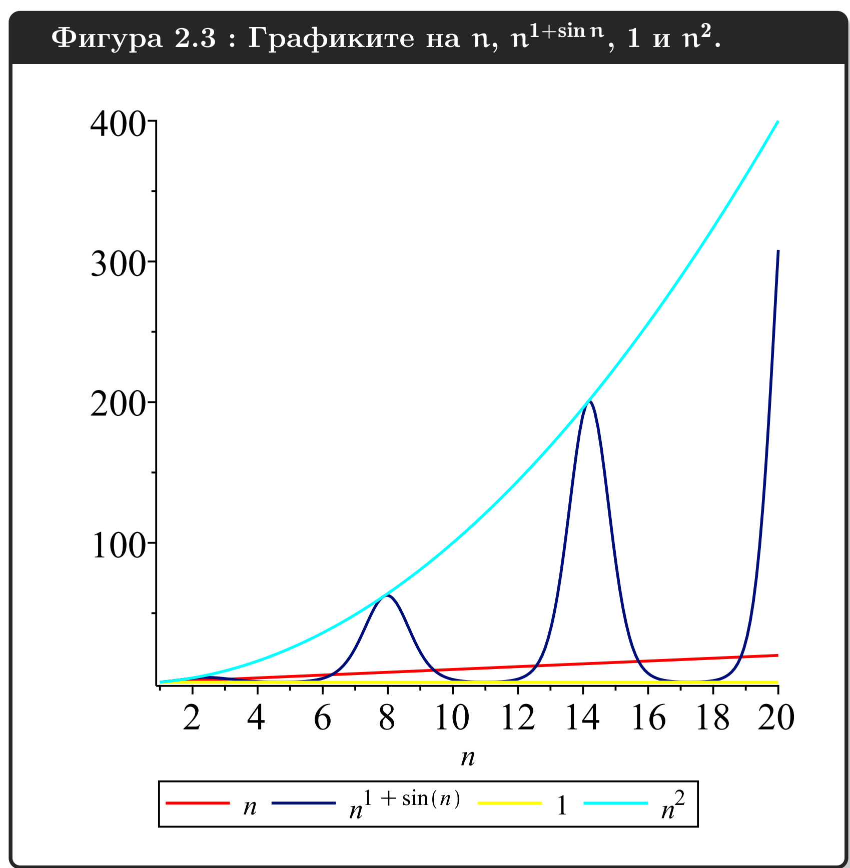
Доказателство: Първо ще покажем по една двойка функции като пример за всяка от тези шест възможности. После ще покажем, че останалите комбинации са невъзможни.

(2.39): Този случай се характеризира с това, че $f(n) \npreceq g(n)$ и $f(n) \nsucceq g(n)$, понеже е достатъчно и $\preceq$, и $\succeq$ 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 4 и контрапозитивните на твърденията от Лема 11.

Следният пример за такива функции е от [31, стр. 52]. Нека $f(n) = n$ и $g(n) = n^{1+\sin n}$. Разгледайте Фигура 2.3[†], която илюстрира графиките на $f(n)$ и $g(n)$, както и на функциите 1 и n^2. Тъй като $1 + \sin n$ осцилира между 0 и 2, то $n^{1+\sin n}$ 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 $f(n') < c \cdot g(n')$ и съществува $n'' > n_0$, такова че $f(n'') > c \cdot g(n'')$. Следователно, нито една от релациите $\preceq$ и $\succeq$ не е в сила.

[†]Фигура 2.3 е направена с Maple(TM).

[‡]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 : Графиките на n, $n^{1+\sin n}$, 1 и n^2.

(2.40): Нека $f(n) = n^{1+\sin n}$ и $g(n) = n^2$ (Фигура 2.3 илюстрира и тези разсъждения). Вярно е, че $f(n) \preceq g(n)$: вземаме $c = 1$ и забелязваме, че за всяко $n \geqslant 1$ е вярно, че $f(n) \leqslant cg(n)$.

От друга страна, $f(n) \not\prec g(n)$, понеже съществува константа $c > 0$, такава че за всяка стойност на аргумента n_0 съществува $n' > n_0$, такова че $f(n') \geqslant c \cdot g(n')$; например, $c = 1$.

Също така е вярно, че $f(n) \not\succeq g(n)$, 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 $f(n) \preceq g(n)$ и $f(n) \not\succeq g(n)$ следва, че $f(n) \not\asymp g(n)$ съгласно Следствие 4.

И накрая, $f(n) \not\succ g(n)$, което следва от това, че $f(n) \preceq g(n)$ и Лема 13.

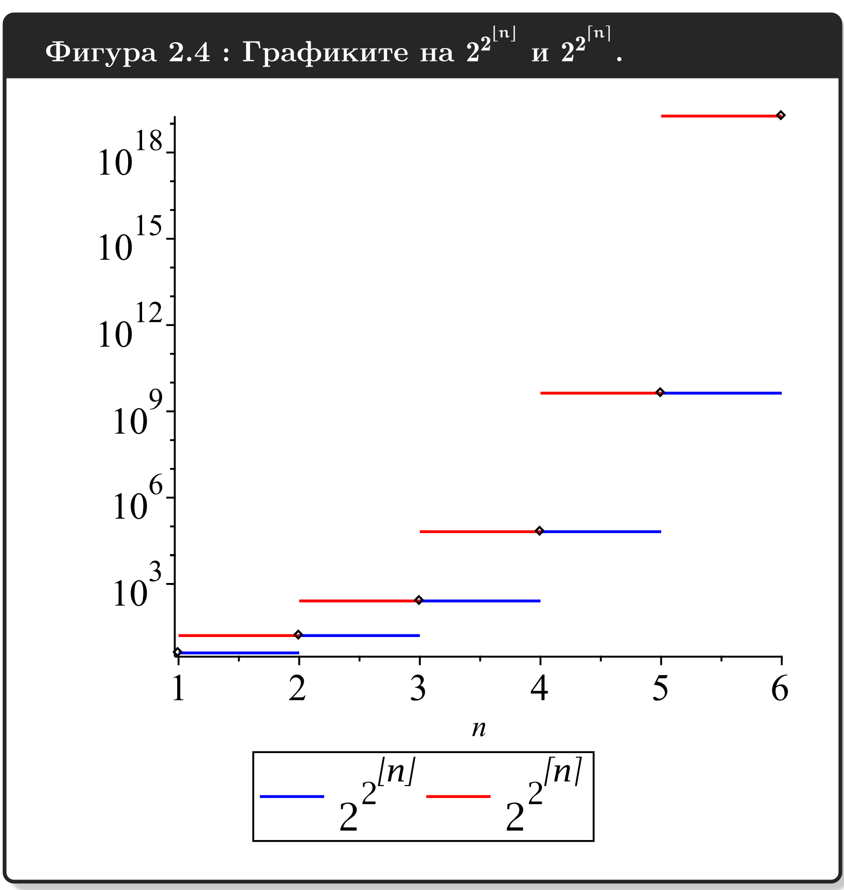
Друга двойка функции, която е подходящ пример, е $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$. Ако $n \in \mathbb{N}$, то $f(n) = g(n)$. Ако обаче $n \in \mathbb{R}^+ \backslash \mathbb{N}$, $f(n) = \sqrt{g(n)}$, понеже

$$n \in \mathbb{R}^+ \backslash \mathbb{N} \;\rightarrow\; \lfloor n \rfloor = \lceil n \rceil - 1 \;\rightarrow\; 2^{\lfloor n \rfloor} = 2^{\lceil n \rceil - 1} \;\rightarrow\; 2^{2^{\lfloor n \rfloor}} = 2^{2^{\lceil n \rceil - 1}} = 2^{\frac{1}{2} 2^{\lceil n \rceil}} = \sqrt{2^{2^{\lceil n \rceil}}}$$

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 $2^{2^{\lceil n \rceil}}$ взема стойност, която е предната ѝ стойност, повдигната на квадрат. Фигурата е с логаритмичен мащаб по ординатата поради изключително бързото нарастване на двойната експонента.

[†]Фигура 2.4 е направена с Maple(TM).

Фигура 2.4 : Графиките на $2^{2^{\lfloor n \rfloor}}$ и $2^{2^{\lceil n \rceil}}$.



(2.41): Пример за такива функции са $f(n) = n$ и $g(n) = n^2$. Това, че $f(n) \prec g(n)$, $f(n) \preceq g(n)$, $f(n) \not\asymp g(n)$, $f(n) \not\succ g(n)$ и $f(n) \not\succeq g(n)$, е очевидно.

(2.42): Пример за такива функции са $f(n) = n$ и $g(n) = n + 1$. Това, че $f(n) \not\prec g(n)$, $f(n) \preceq g(n)$, $f(n) \asymp g(n)$, $f(n) \not\succ g(n)$ и $f(n) \succeq g(n)$, е очевидно.

(2.43): Двата примера от (2.40), но с разменени $f(n)$ и $g(n)$, са валидни за този случай.

(2.44): Пример за такива функции са $f(n) = n^2$ и $g(n) = n$. □

Сега да разгледаме невъзможните комбинации. Всички комбинации с поне едно от следните:

$$f(n) \succ g(n) \wedge f(n) \not\succeq g(n)$$
$$f(n) \prec g(n) \wedge f(n) \not\preceq g(n)$$

са невъзможни, защото $\succ$ влече $\succeq$ и $\prec$ влече $\preceq$ (Лема 11). Всички комбинации с поне едно от следните:

$$f(n) \asymp g(n) \wedge f(n) \not\succeq g(n)$$
$$f(n) \not\preceq g(n) \wedge f(n) \asymp g(n)$$
$$f(n) \preceq g(n) \wedge f(n) \not\asymp g(n) \wedge f(n) \succeq g(n)$$

са невъзможни, защото $\asymp$ е в сила тогава и само тогава, когато и $\succeq$, и $\preceq$ са в сила (Следствие 4). И накрая, всички комбинации с поне едно от следните:

$$f(n) \prec g(n) \wedge f(n) \succeq g(n)$$
$$f(n) \preceq g(n) \wedge f(n) \succ g(n)$$

са невъзможни, защото $\prec$ и $\succeq$ са несъвместими, а също така $\preceq$ и $\succ$ са несъвместими (Следствие 5). □

Следствие 7: $\preceq$ и $\succeq$ 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

$$f(n) \preceq g(n) \text{ или } g(n) \preceq f(n)$$

Заради транспониранатa симетрия имаме право да кажем това по друг начин:

$$f(n) \succeq g(n) \text{ или } g(n) \succeq f(n)$$

Теорема 13: 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

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

за някакви положителни константи c_1 и c_2 и за всички $n \geqslant n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на k-та степен (k е положително) и получаваме

$$c_1^k (g(n))^k \leqslant (f(n))^k \leqslant c_2^k (g(n))^k, \text{ за всяко } n \geqslant n_0$$

Но тогава $(f(n))^k \asymp (g(n))^k$, понеже c_1^k и c_2^k са положителни константи.

В другата посока, нека

$$c_1 (g(n))^k \leqslant (f(n))^k \leqslant c_2 (g(n))^k, \text{ за всяко } n \geqslant n_0$$

за някакви положителни константи c_1 и c_2 и за всички $n \geqslant n_0$, за някакво $n_0 > 0$. Повдигаме двете неравенства на $\frac{1}{k}$-та степен ($\frac{1}{k}$ е положително) и получаваме

$$\sqrt[k]{c_1}\, g(n) \leqslant f(n) \leqslant \sqrt[k]{c_2}\, g(n), \text{ за всяко } n \geqslant n_0$$

Но тогава $f(n) \asymp g(n)$, понеже $\sqrt[k]{c_1}$ и $\sqrt[k]{c_2}$ са положителни константи. □

Теорема 14: 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 $|h(n)| \prec g(n)$. Тогава $f(n) \asymp g(n)$.

Доказателство: Първо ще покажем, че $g(n)$ е асимптотично положителна. Да допуснем противното. Негацията на съждението в Определение 16, по отношение на $g(n)$, е:

$$\forall n_0 \in \mathbb{R}^+ \, \exists n \geqslant n_0 : g(n) \leqslant 0$$

Но тогава е невъзможно да е имаме $|h(n)| = o(g(n))$, понеже това е същото (Определение 19, (2.19)) като за всяка положителна константа c да има стойност на аргумента n_0, такава че за всяко $n \geqslant n_0$ да е вярно, че $0 < |h(n)| < c \cdot g(n)$.

И така, $g(n)$ е асимптотично положителна. Съгласно Определение 16, съществува n_0, такова че $\forall n \geqslant n_0 : g(n) > 0$. Забележете, че е възможно $g(n)+h(n) < g(n) \, \forall n \nearrow$, понеже е възможно $h(n) < 0 \, \forall n \nearrow$. Но

$$\frac{1}{2}g(n) \leqslant g(n) + h(n) \leqslant 2g(n) \, \forall n \nearrow$$

понеже има стойност на аргумента $n' \geqslant n_0$, такава че за $n \geqslant n'$ е изпълнено $|h(n)| < \frac{1}{2}g(n)$. Но $g(n) + h(n) = f(n)$ по условие. Тогава е изпълнено

$$\forall n \geqslant n' : \frac{1}{2}g(n) \leqslant f(n) \leqslant 2g(n)$$

Съгласно Определение 17, в сила е $f(n) \asymp g(n)$. □

Следствие 8

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, инак полиномът не би бил асимптотично положителен.

Доказателство: Нека $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$. Очевидно a_k не е нула, инак полиномът нямаше да е от степен k, и не е отрицателно число, инак полиномът нямаше да е асимптотично положителна функция.

Ясно е, че $|a_{k-1}n^{k-1} + \cdots + a_1 n + a_0| \prec a_k n^k$, понеже $|a_{k-1}n^{k-1} + \cdots + a_1 n + a_0| \leqslant |a_{k-1}|n^{k-1} + \cdots + |a_1|n + |a_0| \, \forall n \nearrow$ и $\lim_{n\to\infty} \frac{|a_{k-1}|n^{k-1} + \cdots + |a_1|n + |a_0|}{a_k n^k} = 0$. Сега прилагаме Теорема 14 с $p(n)$ като $f(n)$, $a_k n^k$ като $g(n)$ и $a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ като $h(n)$. □

Теорема 15 казва, че експоненциалната трансформация върху две функции "изостря" асимптотичните разлики в нарастването им, ако функциите са неограничено нарастващи и имат различно асимптотично нарастване.

Теорема 15: Експонирането запазва $\prec$

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции и $a > 1$ е константа. Тогава

$$f(n) \prec g(n) \rightarrow a^{f(n)} \prec a^{g(n)}$$

Доказателство: Трябва да докажем, че:

$$\forall c > 0 \, \exists n' \, \forall n \geqslant n' : 0 \leqslant a^{f(n)} < c.a^{g(n)} \tag{2.45}$$

Да разгледаме $k = \log_a c$. Тоест, $c = a^k$. Да препишем (2.45) така:

$$\forall k \, \exists n' \, \forall n \geqslant n' : 0 \leqslant a^{f(n)} < a^k a^{g(n)} \tag{2.46}$$

Вземаме логаритъм при основа a от двете неравенства и получаваме:

$$\forall k\ \exists n'\ \forall n \geqslant n' : 0 \leqslant f(n) < k + g(n) \tag{2.47}$$

Трябва да докажем (2.47). По условие имаме:

$$\forall c > 0\ \exists n_0\ \forall n \geqslant n_0 : 0 \leqslant f(n) < c.g(n)$$

Щом това е в сила за всяко $c > 0$, в частност то е в сила за $c = \frac{1}{2}$. И така:

$$\exists n_0\ \forall n \geqslant n_0 : 0 \leqslant f(n) < \frac{g(n)}{2} \tag{2.48}$$

Но $g(n)$ е растяща и неограничена. Тогава:

$$\forall k\ \exists n_1\ \forall n \geqslant n_1 : 0 < k + \frac{g(n)}{2} \tag{2.49}$$

Да препишем (2.49) така:

$$\forall k\ \exists n_1\ \forall n \geqslant n_1 : \frac{g(n)}{2} < k + g(n) \tag{2.50}$$

От (2.48) и (2.50) заключаваме, че:

$$\forall k\ \exists n''\ \forall n \geqslant n'' : 0 \leqslant f(n) < k + g(n) \tag{2.51}$$

Но (2.51) е същото като (2.47). С това доказателството е готово. □

Следствие 9

Нека $f(n)$ и $g(n)$ са растящи и неограничени функции. Тогава

$$\lg f(n) \prec \lg g(n) \;\rightarrow\; f(n) \prec g(n)$$

Теорема 16 в някакъв смисъл е конверсна на Теорема 15. Според Теорема 16, ако образите на две нарастващи и неограничени функции след експоненциална трансформация са асимптотично еквивалентни, то и самите функции са асимптотично еквивалентни.

Теорема 16: 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

$$\exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n > n_0 : c_1 \cdot a^{g(n)} \leqslant a^{f(n)} \leqslant c_2 \cdot a^{g(n)}$$

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:

$$\log_a c_1 + g(n) \leqslant f(n) \leqslant \log_a c_2 + g(n)$$

Забелязваме, че щом $g(n)$ е нарастваща и неограничена, то за всяка константа k_1, такава че $0 < k_1 < 1$, в сила е $k_1 \cdot g(n) \leqslant \log_a c_1 + g(n)$, 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 $\log_a c_2 + g(n) \leqslant k_2.g(n)$ за всички достатъчно големи n. И това е така независимо от това, дали логаритъмът е положителен или отрицателен или е нула. Заключаваме, че съществува n_1, такова че:

$$\forall n \geqslant n_1 : k_1 \cdot g(n) \leqslant f(n) \leqslant k_2 \cdot g(n)$$

Но тогава $f(n) \asymp g(n)$. □

Следствие 10

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 25: 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 26: 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 26, функцията $n^{\frac{1}{2}}$ расте полиномиално.

Определение 27: 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 27, функции като 2^{n^2} и e^{n^5} и $(\sqrt{2})^{n^{\sqrt{2}}}$ също са експоненциално растящи функции.

Теорема 17: $(\log_a f(n))^k \prec (f(n))^c$

Нека $a, k, c \in \mathbb{R}^+$ са произволни константи, като $a > 1$. Нека $f(n)$ е неограничено растяща диференцируема функция. В сила е:

$$(\log_a f(n))^k \prec (f(n))^c$$

Доказателство:

$$\lim_{n\to\infty} \frac{(f(n))^c}{(\log_a f(n))^k} = \qquad // \text{ полагаме } b \leftarrow \frac{c}{k}$$

$$\lim_{n\to\infty} \frac{((f(n))^b)^k}{(\log_a f(n))^k} =$$

$$\lim_{n\to\infty} \left(\frac{(f(n))^b}{\log_a f(n)}\right)^k = \qquad // \text{ } k \text{ е положително}$$

$$\lim_{n\to\infty} \frac{(f(n))^b}{\log_a f(n)} = \qquad // \text{ прилагаме правилото на l'Hôpital}$$

$$\lim_{n\to\infty} \frac{b(f(n))^{b-1}f'(n)}{\left(\frac{1}{\ln a}\right)\left(\frac{1}{(f(n))}\right)f'(n)} =$$

$$\lim_{n\to\infty} (\ln a) b\,(f(n))^b = \infty$$

□

Теорема 18: $(f(n))^k \prec a^{(f(n))^c}$

Нека $a, k, c \in \mathbb{R}^+$ са произволни константи, като $a > 1$. Нека $f(n)$ е неограничено растяща диференцируема функция. В сила е:

$$(f(n))^k \prec a^{(f(n))^c}$$

Доказателство: Да вземем $\log_a$ от двете страни. Лявата страна става $k \log_a f(n)$ и дясната страна става $(f(n))^c$. Но $k \log_a f(n) \prec (f(n))^c$ съгласно Теорема 17. Прилагаме Следствие 9 и получаваме желания резултат. □

Следствие 11 е частен случай на Теорема 17 и Теорема 18, ако $f(n) = n$.

Следствие 11: Полилогаритмично преди полиномиално преди експоненциално

Нека $a, k, \epsilon \in \mathbb{R}^+$ са произволни константи, като $a > 1$. В сила е:

$$(\log_a n)^k \prec n^\epsilon \prec a^{n^k}$$

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 неколкократни 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

$$\begin{aligned} \lg n &\approx 49.82892142 \\ \lg\lg n &\approx 5.638911441 \\ \lg\lg\lg n &\approx 2.495416686 \end{aligned}$$

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 28: Итериран логаритъм

За всяко цяло $i \geqslant 0$ и всяко реално $a > 1$, дефинираме *итерирания логаритъм* $\log_a^{(i)} n$ така:

$$\log_a^{(i)} n = \begin{cases} n, & \text{ако } i = 0 \\ \log_a\left(\log_a^{(i-1)} n\right), & \text{ако } i > 0 \text{ и } \lg^{(i-1)} n > 0 \\ \text{недефинирано}, & \text{ако } i > 0 \text{ и } \log_a^{(i-1)} n < 0 \text{ , или } \log_a^{(i-1)} n \text{ е недефинирано} \end{cases}$$

Забележете, че $\lg^{(i)} n$ **не е** същото като $\lg^i n$. $\lg^{(i)} n$ е итерираният логаритъм

$$\underbrace{\lg\lg\cdots\lg}_{i \text{ пъти}} n$$

докато $\lg^i n$ е $(\lg n)^i$, тоест, $\lg n$, повдигнат на i-та степен. Записът $\boxed{\lg^{(i)} n}$ е по-прецизен от $\boxed{\underbrace{\lg\lg\cdots\lg}_{i \text{ пъти}} n}$, ако искаме да разглеждаме реални функции. Примерно, ако започнем с аргумент 3 и ползваме записа $\lg\cdots\lg 3$, ще получим

$$\begin{aligned} \lg 3 &\approx 1.584962501 \\ \lg\lg 3 &\approx .6644487077 \\ \lg\lg\lg 3 &\approx -.5897702603 \\ \lg\lg\lg\lg 3 &\approx -.7617750199 + 4.532360143 \cdot i \\ \lg\lg\lg\lg\lg 3 &\approx 2.200357409 + 2.506415242 \cdot i \end{aligned}$$

А ако ползваме Определение 28, 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 два записа са еквивалентни.

Теорема 19: Итерирането на логаритъма дава по-малко асимпт. нарастване

Нека $a > 1$ е реална константа, $i \in \mathbb{N}^+$, а $g(n)$ е неограничено растяща диференцируема функция. Тогава

$$\log_a^{(i+1)} g(n) \prec \log_a^{(i)} g(n)$$

Доказателство: Твърдението следва веднага от Теорема 17 с $k = 1$ и $c = 1$, ако $f(n)$ от Теорема 17 е $\log_a^{(i)} g(n)$. □

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 20: Апроксимация на Stirling, слаба версия

В сила е

$$n! \asymp \sqrt{n} \cdot \frac{n^n}{e^n} \tag{2.52}$$

Доказателство Това доказателство е взето от статия на Dan Romik [123]. Ще докажем, че съществуват положителни константи c_1 и c_2, такива че

$$\forall n \nearrow : c_1 \cdot \sqrt{n}\frac{n^n}{e^n} \leqslant n! \leqslant c_2 \cdot \sqrt{n}\frac{n^n}{e^n}$$

Нотация 2: "$\{x\}$" наместо "$x - \lfloor x \rfloor$"

Ползваме следната нотация само в рамките на това доказателство. Нека x е реално число. "$\{x\}$" е кратък запис за $x - \lfloor x \rfloor$. 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 [56, (3.8) на стр. 70].

Нека $n \in \mathbb{N}^+$. Очевидно, в сила е

$$\ln n! = \sum_{k=1}^{n} \ln k = \sum_{k=1}^{n} (\ln k - 0) = \sum_{k=1}^{n} (\ln k - \ln 1) = \sum_{k=1}^{n} \left(\ln x \Big|_1^k \right) = \sum_{k=1}^{n} \int_1^k \frac{1}{x} dx \tag{2.53}$$

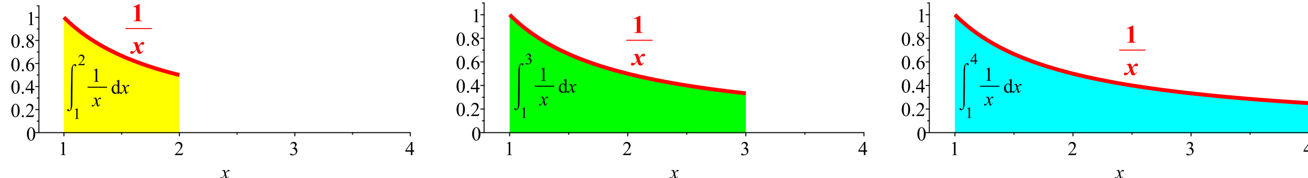
Твърдим, че

$$\sum_{k=1}^{n} \int_1^k \frac{1}{x} dx = \int_1^n \frac{n - \lfloor x \rfloor}{x} dx \tag{2.54}$$

Ще се убедим в истинността на (2.54) с малък пример. Нека $n = 4$. Тогава лявата страна на (2.54) 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 (2.54) може да се запише като сума от три интеграла:

$$\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx \tag{2.55}$$

Ето илюстрация на трите интеграла от (2.55)[†].

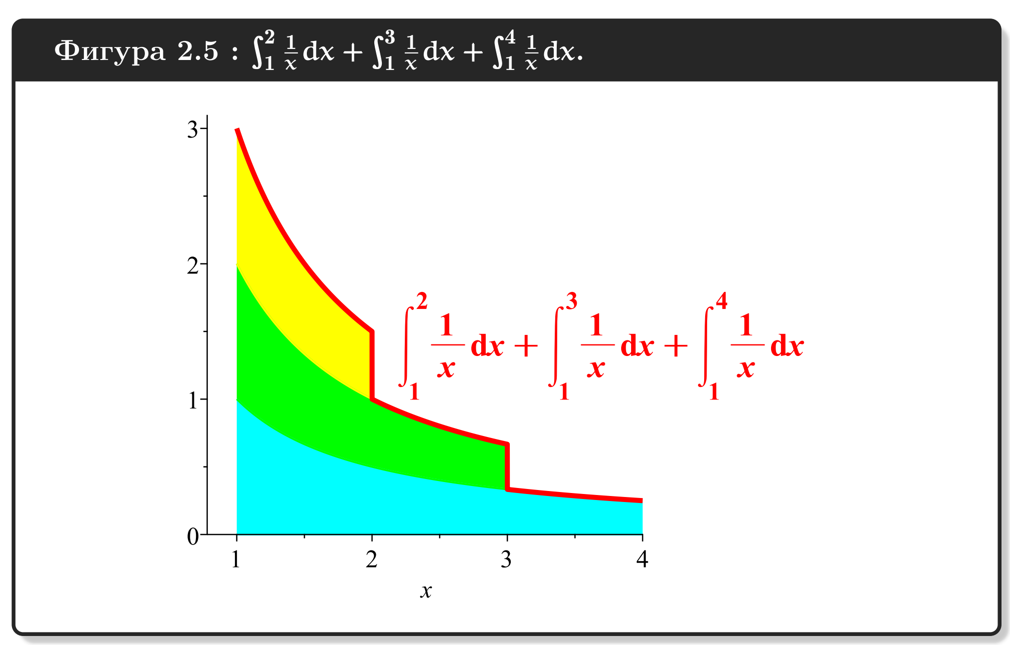


Твърдим, че

$$\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx = \int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx \tag{2.56}$$

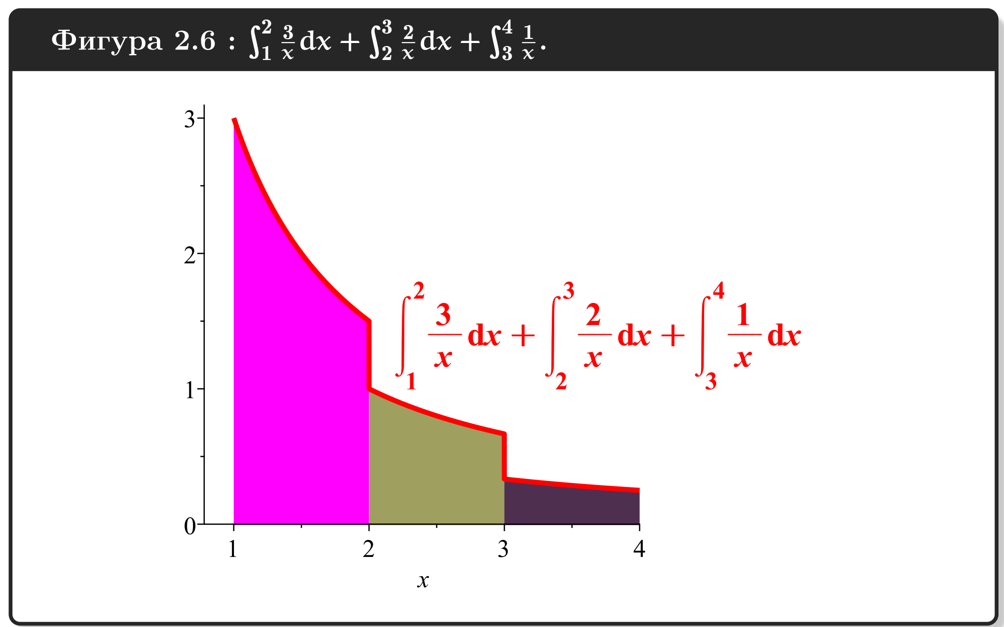
Нагледно неформално доказателство има на Фигура 2.5[‡] и Фигура 2.6[§]. Фигура 2.5 показва $\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx$ като трите площи са наложени една върху друга. Фигура 2.6 показва същия район от равнината, но сега разбит на площи съгласно $\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx$: 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 : $\int_1^2 \frac{1}{x} dx + \int_1^3 \frac{1}{x} dx + \int_1^4 \frac{1}{x} dx$.



[†]Фигурите са направени с Maple(TM).
[‡]Фигура 2.5 е направена с Maple(TM).
[§]Фигура 2.6 е направена с Maple(TM).

Фигура 2.6 : $\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x}$.



Обаче

$$\int_1^2 \frac{3}{x} dx + \int_2^3 \frac{2}{x} dx + \int_3^4 \frac{1}{x} dx = \int_1^4 \frac{4 - \lfloor x \rfloor}{x} dx \tag{2.57}$$

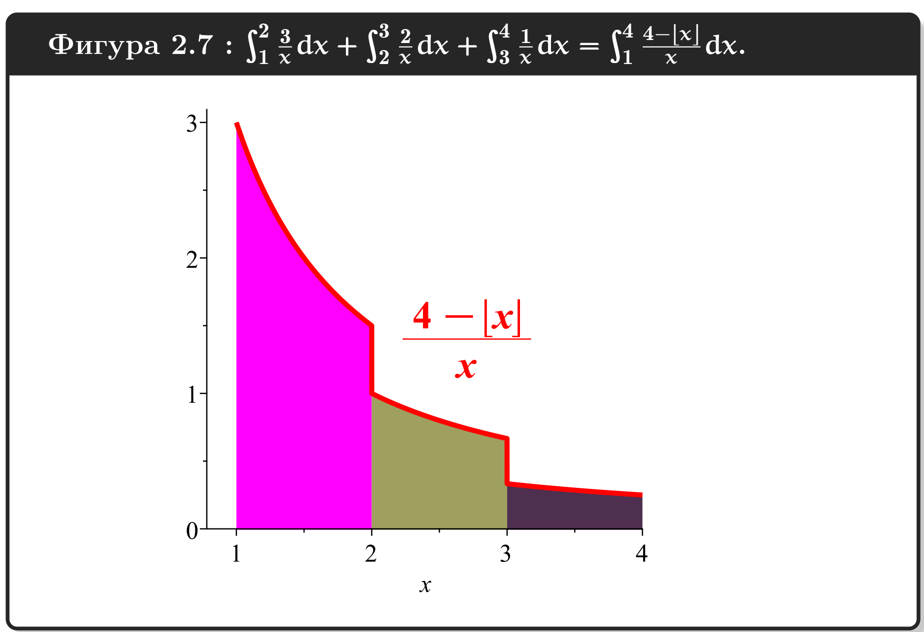
Причината е, че следните две функции $f_1(x)$ и $f_2(x)$ върху интервала $[1, 4]$ съвпадат:

$$f_1(x) = \begin{cases} \frac{3}{x}, & \text{ако } 1 \leqslant x \leqslant 2 \\ \frac{2}{x}, & \text{ако } 2 < x \leqslant 3 \\ \frac{1}{x}, & \text{ако } 3 < x \leqslant 4 \end{cases}$$

$$f_2(x) = \frac{4 - \lfloor x \rfloor}{x}$$

(2.57) е илюстрирано на Фигура 2.7[†].

[†]Фигура 2.7 е направена с Maple(TM).



Фигура 2.7 : $\int_1^2 \frac{3}{x}dx + \int_2^3 \frac{2}{x}dx + \int_3^4 \frac{1}{x}dx = \int_1^4 \frac{4-\lfloor x \rfloor}{x}dx.$

След като сме убедени в истинността на (2.54) и предвид очевидното $-\lfloor x \rfloor = \{x\} - x$, продължаваме така:

$$\ln n! = \int_1^n \frac{n - \lfloor x \rfloor}{x} dx = \int_1^n \frac{n + \{x\} - x}{x} dx \tag{2.58}$$

Ако сега продължим по този начин:

$$\begin{aligned}
&\int_1^n \frac{n + \{x\} - x}{x} dx = \int_1^n \frac{n}{x} dx - \int_1^n \frac{x}{x} dx + \int_1^n \frac{\{x\}}{x} dx = \\
&n \int_1^n \frac{1}{x} dx - \int_1^n 1 \cdot dx + \int_1^n \frac{\{x\}}{x} dx = n \left(\ln x \Big|_1^n \right) - x \Big|_1^n + \int_1^n \frac{\{x\}}{x} dx = \\
&n(\ln n - 0) - (n-1) + \int_1^n \frac{\{x\}}{x} dx = n \ln n - (n-1) + \int_1^n \frac{\{x\}}{x} dx
\end{aligned} \tag{2.59}$$

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 $0 \leqslant \{x\} < 1$; 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 $n \ln n - (n-1) + \int_1^n \frac{\{x\}}{x} dx$ (2.59) 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 21 като леко следствие от Теорема 20. А за да докажем самата Теорема 20, трябва да направим нещо по-умно от извеждането на (2.59).

Продължаваме така:

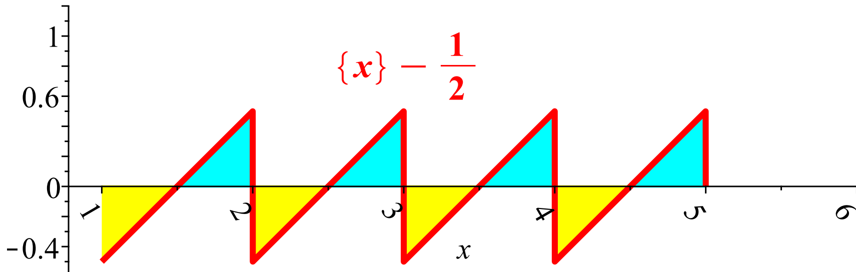
$$\int_1^n \frac{n + \{x\} - x}{x} dx = \int_1^n \frac{(n + \frac{1}{2}) - x + (\{x\} - \frac{1}{2})}{x} dx = \tag{2.60}$$

$$\left(n + \frac{1}{2}\right) \ln n - (n-1) + \int_1^n \frac{\{x\} - \frac{1}{2}}{x} dx \tag{2.61}$$

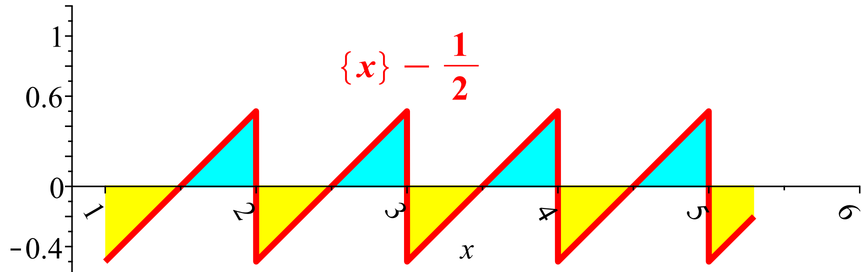
Да разгледаме $\int_1^n \frac{\{x\} - \frac{1}{2}}{x} dx$. Ще покажем, че това събираемо е ограничено отгоре и отдолу от константи. За целта разглеждаме $\int_1^{+\infty} \frac{\{x\} - \frac{1}{2}}{x} dx$. Но $\int_1^{+\infty} \frac{\{x\} - \frac{1}{2}}{x} dx$ е сходящ по критерия на Dirichlet (вижте например [101, Theorem 9.20, стр. 291–292]). Критерият е приложим по следните причини.

- $\int_1^y \left(\{x\} - \frac{1}{2}\right) dx$ 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[‡].

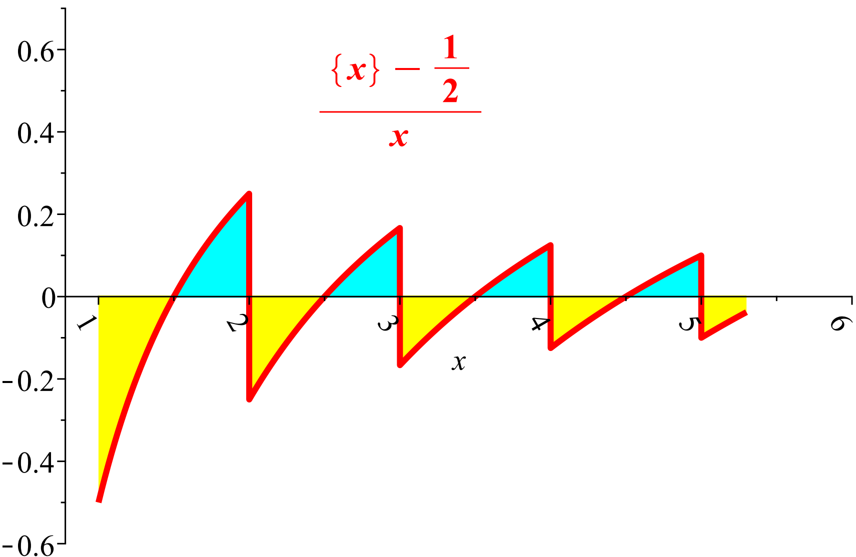


- Функцията $\{x\} - \frac{1}{2}$ е очевидно интегруема по Riemann във всеки интервал $[1, y] \subset [1, +\infty)$.

- 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 (2.61) е ограничен отгоре и отдолу от константи, то съществуват константи c_3 и c_4, такива че

$$\left(n + \frac{1}{2}\right) \ln n - n + c_3 \leqslant \ln n! \leqslant \left(n + \frac{1}{2}\right) \ln n - n + c_4$$

за всички достатъчно големи n. Но това е същото като

$$\left(\ln \frac{n^{n+\frac{1}{2}}}{e^n}\right) + c_3 \leqslant \ln n! \leqslant \left(\ln \frac{n^{n+\frac{1}{2}}}{e^n}\right) + c_4$$

Тогава, за някакви положителни константи c_1 и c_2 е в сила

$$c_1 \cdot \sqrt{n}\frac{n^n}{e^n} \leqslant n! \leqslant c_2 \cdot \sqrt{n}\frac{n^n}{e^n}$$

за всички достатъчно големи n. □

Една забележка върху доказателството на Теорема 20. В (2.60) 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 $y \to \infty$ 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 (2.61).

Теорема 21: Асимптотиката на $\lg n!$

В сила е

$$\lg n! \asymp n \lg n \tag{2.62}$$

Доказателство: От Теорема 20 знаем, че $n! \asymp \sqrt{n}\,\frac{n^n}{e^n}$. Прилагаме Следствие 10 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 $|-n\lg e + \frac{1}{2}\lg n| \prec n\lg n$ и прилагаме Теорема 14. Заключаваме, че $\lg n! \asymp n\lg n$. □

Теорема 22: 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

$$\sum_{k=m}^{n} m\lg m \leqslant \sum_{k=m}^{n} k\lg k \leqslant \sum_{k=m}^{n} n\lg n \quad \leftrightarrow$$
$$(m\lg m)(n-m+1) \leqslant B \leqslant (n\lg n)(n-m+1) \quad \leftrightarrow$$
$$\underbrace{\left(\left\lfloor\frac{n}{2}\right\rfloor \lg \left\lfloor\frac{n}{2}\right\rfloor\right)\left(n - \left\lfloor\frac{n}{2}\right\rfloor + 1\right)}_{C} \leqslant B \leqslant \underbrace{(n\lg n)\left(n - \left\lfloor\frac{n}{2}\right\rfloor + 1\right)}_{D}$$

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 $A \preceq B$. 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, $A \preceq B$. Щом $\sum_{k=1}^{n} k\lg k = A+B$, заключаваме, че $\sum_{k=1}^{n} k\lg k \asymp n^2\lg n$. □

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 23: 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 20 върху факториелите и получаваме

$$\frac{n!}{\left(\frac{n}{2}!\right)^2} \asymp \frac{\sqrt{n}\,\frac{n^n}{e^n}}{\left(\sqrt{\frac{n}{2}}\,\frac{\frac{n}{2}^{\frac{n}{2}}}{e^{\frac{n}{2}}}\right)^2} \asymp \sqrt{n}\,\frac{n^n}{e^n} \times \frac{e^n}{n\,\frac{n^n}{2^n}} = \frac{2^n}{\sqrt{n}}$$ □

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:

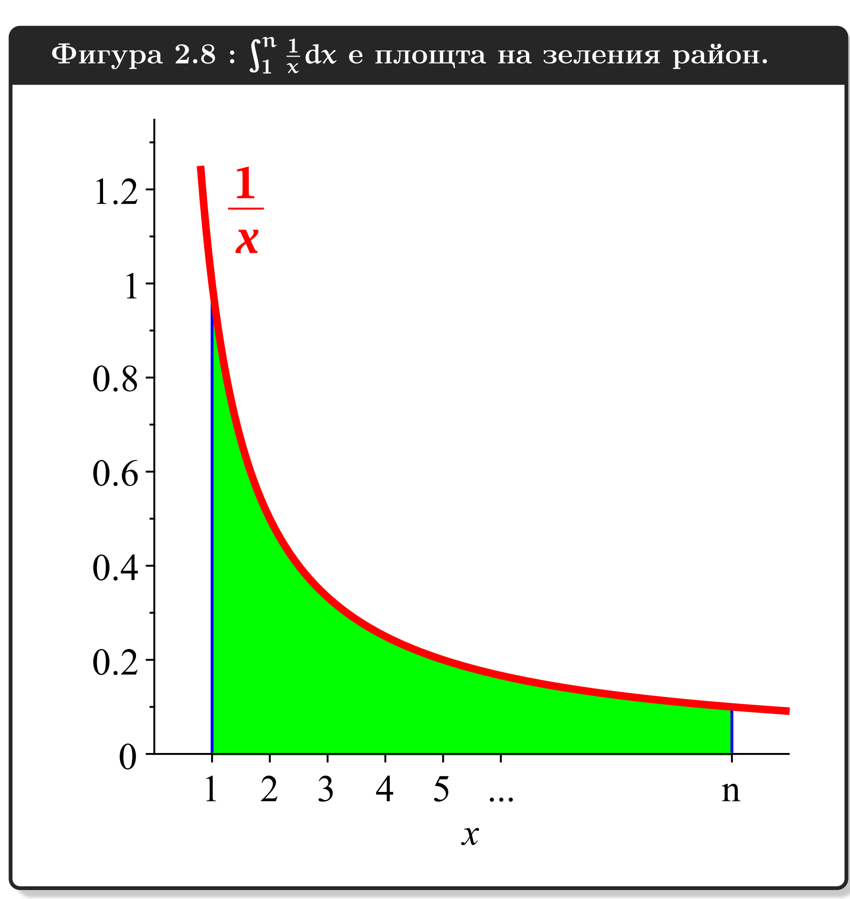
$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Теорема 24: 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[†].



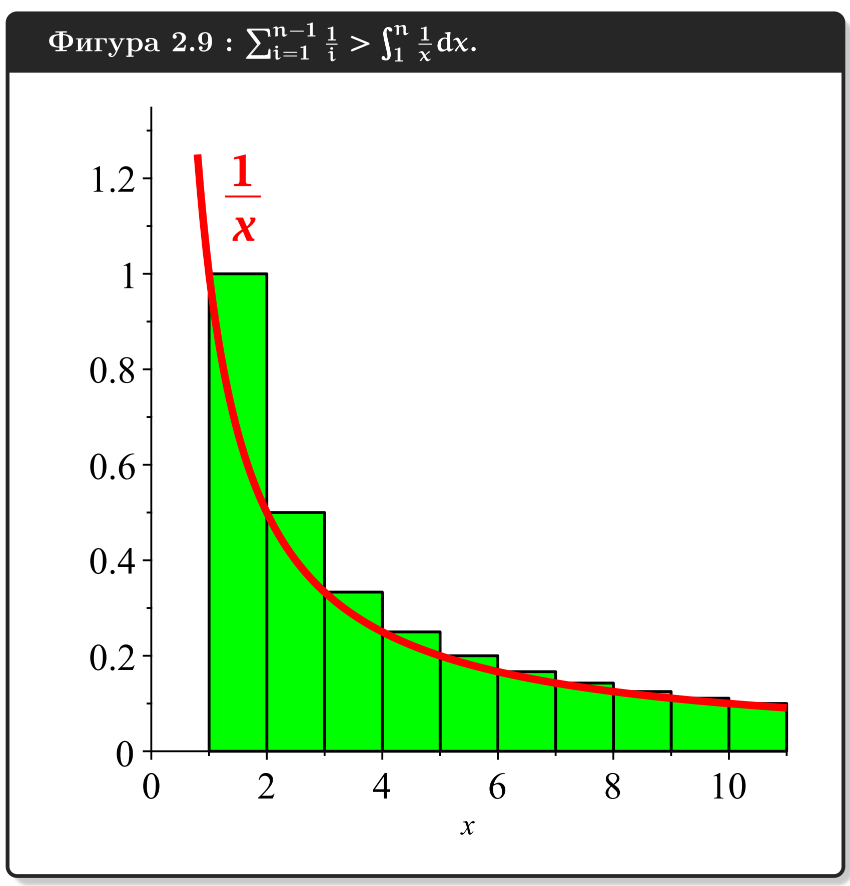
Фигура 2.8 : $\int_1^n \frac{1}{x} dx$ е площта на зеления район.

Да разгледаме стъпаловидната функция $\frac{1}{\lfloor x \rfloor}$ върху интервала $[1, n]$. Вярно е, че $\forall x \in [1, n] : \frac{1}{\lfloor x \rfloor} \geqslant \frac{1}{x}$, понеже $\lfloor x \rfloor \leqslant x$ за всяко x. Забелязваме, че площта на района, ограден от $\frac{1}{\lfloor x \rfloor}$, 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).

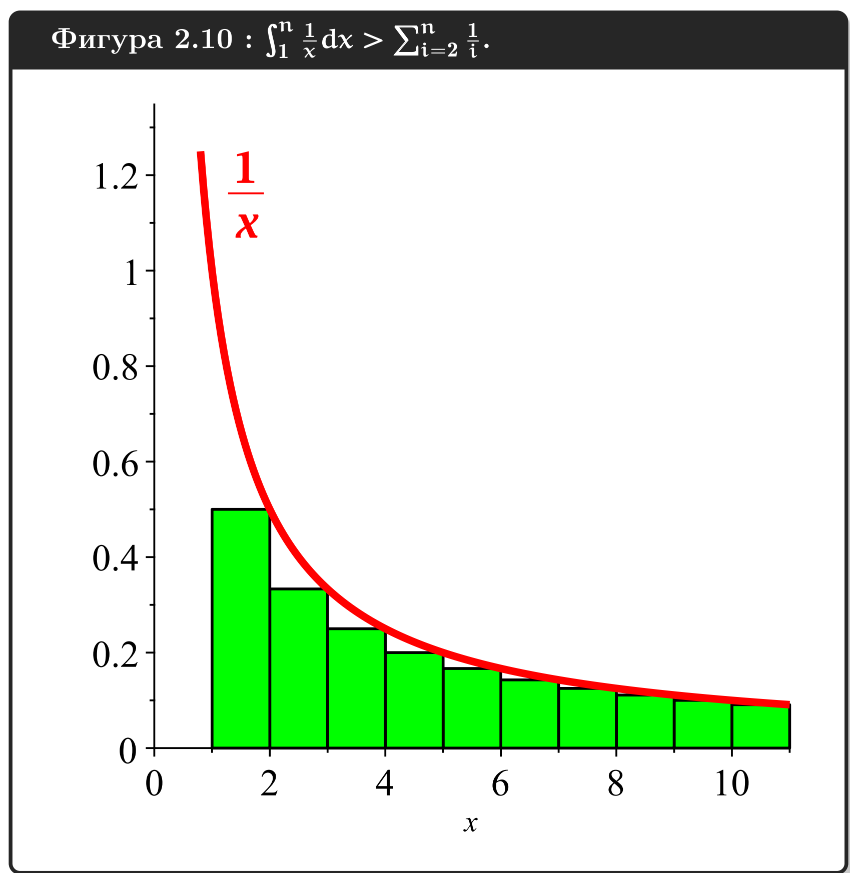
другата е $\frac{1}{i}$ за някое $i \in \{1, 2, \ldots, n-1\}$. Очевидно този район има площ, по-голяма от площта под $\frac{1}{x}$. С други думи, $\sum_{i=1}^{n-1} \frac{1}{i} > \int_1^n \frac{1}{x} dx$. Това е илюстрирано на Фигура 2.9[†].

Фигура 2.9 : $\sum_{i=1}^{n-1} \frac{1}{i} > \int_1^n \frac{1}{x} dx$.



Сега да разгледаме стъпаловидната функция $\frac{1}{\lceil x \rceil}$ върху интервала $[1, n]$. $\forall x \in [1, n] : \frac{1}{\lceil x \rceil} \leqslant \frac{1}{x}$, понеже $\lceil x \rceil \geqslant x$ за всяко x. Забелязваме, че площта на района, ограден от $\frac{1}{\lceil x \rceil}$, 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 $i \in \{2, 3, \ldots, n\}$. 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 : $\int_1^n \frac{1}{x}dx > \sum_{i=2}^n \frac{1}{i}$.

Показахме, 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

$$c_1 \cdot H_n \leqslant \ln n \leqslant c_2 \cdot H_n$$

Например, $c_1 = \frac{1}{2}$, $c_2 = 1$ и $n_0 = 1000$.

Заключаваме, 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 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 17. Ще намерим положителни константи c_1, c_2 и стойност на аргумента n_0, такива че

$$0 \leqslant c_1 \cdot n^b \leqslant (n + a)^b \leqslant c_2 \cdot n^b$$

за всяко $n \geqslant n_0$.

Очевидно

$$n + a \leqslant n + |a| \leqslant 2n, \text{ ако } n \geqslant |a|$$
$$n + a \geqslant n - |a| \geqslant \frac{n}{2}, \text{ ако } \frac{n}{2} \geqslant |a|; \text{ тоест, } n \geqslant 2|a|$$

Тогава

$$\frac{1}{2}n \leqslant n + a \leqslant 2n, \text{ ако } n \geqslant 2|a|$$

Повдигаме на степен b и получаваме

$$\left(\frac{1}{2}\right)^b n^b \leqslant (n + a)^b \leqslant 2^b n^b$$

Вече имаме доказателство със $c_1 = \left(\frac{1}{2}\right)^b$, $c_2 = 2^b$ и $n_0 = \lceil 2|a| \rceil$. □

Задача 5

Докажете или опровергайте, че за всеки две асимптотично положителни функции $f(n)$ и $g(n)$ е в сила

$$\max\left(f(n), g(n)\right) \asymp f(n) + g(n)$$

Решение: Твърдението е вярно. Ще докажем, че съществуван положителни константи c_1 и c_2 и стойност на аргумента n_0, такива че $\forall n \geqslant n_0$:

$$0 \leqslant c_1(f(n) + g(n)) \leqslant \max\left(f(n), g(n)\right) \leqslant c_2(f(n) + g(n))$$

Щом $f(n)$ и $g(n)$ са асимптотично положителни,

$$\exists n_0' \;\; \forall n \geqslant n_0' : f(n) > 0$$
$$\exists n_0'' \;\; \forall n \geqslant n_0'' : g(n) > 0$$

Нека $n_0''' = \max\{n_0', n_0''\}$. Очевидно за всяко $n \geqslant n_0'''$ е вярно

$$0 \leqslant c_1(f(n) + g(n)),$$

ако $c_1 > 0$. Също така е очевидно, че за всяко $n \geqslant n_0'''$:

$$0 \leqslant \frac{1}{2}(f(n) + g(n)) \leqslant \max(f(n), g(n)) \text{ и}$$
$$f(n) + g(n) \geqslant \max(f(n), g(n)),$$

което можем да запишем като

$$0 \leqslant \frac{1}{2}(f(n) + g(n)) \leqslant \max(f(n), g(n)) \leqslant f(n) + g(n)$$

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 $n + 1 \not\asymp 1$. □

Задача 8

Вярно ли е, че за всяка асимптотично положителна функция $f(n)$ е в сила

$$f(n) \preceq (f(n))^2?$$

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

$$\lim_{n\to\infty} \frac{f(n)}{(f(n))^2} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{n^2}} = \lim_{n\to\infty} \frac{n}{1} = \infty$$

Съгласно Лема 10, в сила е $f(n) \succ (f(n))^2$. Тогава, съгласно Лема 13 (2.34), в сила е $f(n) \not\preceq (f(n))^2$.

Ако обаче $f(n)$ е неограничено растяща, твърдението е вярно:

$$\lim_{n\to\infty} \frac{f(n)}{(f(n))^2} = \lim_{n\to\infty} \frac{1}{f(n)} = 0$$

Тогава от Лема 10 имаме $f(n) \prec (f(n))^2$, което, съгласно Лема 11 (2.27), влече $f(n) \preceq (f(n))^2$. □

Задача 9

Нека $f(n)$ е асимптотично положителна функция. Кое от следните твърдения е вярно и кое не е вярно?

$$f\left(\frac{n}{2}\right) \asymp f(n) \tag{2.63}$$

$$f\left(\frac{n}{2}\right) \prec f(n) \tag{2.64}$$

$$f\left(\frac{n}{2}\right) \preceq f(n) \tag{2.65}$$

Решение: Нито едно не е вярно. Като контрапример за (2.63) разгледайте $f(n) = 2^n$. Тогава

$$\lim_{n\to\infty} \frac{f\left(\frac{n}{2}\right)}{f(n)} = \lim_{n\to\infty} \frac{2^{\frac{n}{2}}}{2^n} = \lim_{n\to\infty} \frac{1}{2^{\frac{n}{2}}} = 0$$

така че $f\left(\frac{n}{2}\right) \prec f(n)$. Съгласно Лема 14 (2.36), в сила е $f\left(\frac{n}{2}\right) \not\asymp f(n)$.

Като контрапример за (2.64) разгледайте $f(n) = 2n$. Тогава $f\left(\frac{n}{2}\right) = n$ и очевидно

$$f\left(\frac{n}{2}\right) \asymp f(n)$$

Съгласно Лема 14 (2.35), в сила е $f\left(\frac{n}{2}\right) \not\prec f(n)$.

Като контрапример за (2.65) разгледайте $f(n) = \frac{1}{2^n}$. Тогава $f\left(\frac{n}{2}\right) = \frac{1}{2^{\frac{n}{2}}}$, така че

$$\lim_{n\to\infty} \frac{f\left(\frac{n}{2}\right)}{f(n)} = \lim_{n\to\infty} \frac{\frac{1}{2^{\frac{n}{2}}}}{\frac{1}{2^n}} = \lim_{n\to\infty} \frac{2^n}{2^{\frac{n}{2}}} = \lim_{n\to\infty} 2^{\frac{n}{2}} = \infty$$

така че $f\left(\frac{n}{2}\right) \succ f(n)$. Съгласно Лема 13 (2.34), в сила е $f\left(\frac{n}{2}\right) \not\preceq f(n)$. □

Задача 10

Нека k е положителна константа и n е цяло число. Докажете, че

$$1 + k + k^2 + k^3 + \cdots + k^n \asymp \begin{cases} 1, & \text{if } k < 1 \\ n, & \text{if } k = 1 \\ k^n, & \text{if } k > 1 \end{cases}$$

Решение: Добре известно е, че

$$1 + k + k^2 + k^3 + \cdots + k^n = \begin{cases} \frac{k^{n+1}-1}{k-1}, & \text{ако } k \neq 1 \\ n+1, & \text{ако } k = 1 \end{cases} \tag{2.66}$$

Ако $0 < k < 1$, геометричният ред $\sum_{t=0}^{\infty} k^t$ е сходящ със сума $\frac{1}{1-k}$, така че

$$1 + k + k^2 + k^3 + \ldots + k^n < \sum_{t=0}^{\infty} k^t = \frac{1}{1-k} \asymp 1$$

Ако $k = 1$, имаме

$$1 + k + k^2 + k^3 + \cdots + k^n = n + 1 \asymp n$$

Ако $k > 1$, от (2.66) имаме

$$1 + k + k^2 + k^3 + \cdots + k^n = \frac{k^{n+1} - 1}{k - 1} = \frac{k \cdot k^n}{k - 1} - \frac{1}{k - 1} = \frac{k}{k - 1} k^n + \frac{1}{1 - k}$$

Но $\frac{k}{k-1}$ е положителна константа, а $\left|\frac{1}{1-k}\right| \prec \frac{k}{k-1} k^n$. Прилагаме Теорема 14 и заключаваме, че $1 + k + k^2 + k^3 + \cdots + k^n \asymp k^n$. □

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:

$$\sqrt{f(n)h(n)} \asymp g(n)$$
$$\sqrt{(f(n))^2 + (h(n))^2} \asymp g(n)$$

Решение: И двете твърдения са верни. По условие,

$$\exists c_1 > 0 \, \exists c_2 > 0 \, \exists n_1 > 0 \, \forall n \geqslant n_1 : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$
$$\exists c_3 > 0 \, \exists c_4 > 0 \, \exists n_2 > 0 \, \forall n \geqslant n_2 : c_3 g(n) \leqslant h(n) \leqslant c_4 g(n)$$

Нека $n' = \max\{n_1, n_2\}$. За всяко n, такова че $n \geqslant n'$,

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \tag{2.67}$$
$$c_3 g(n) \leqslant h(n) \leqslant c_4 g(n) \tag{2.68}$$

Ако умножим (2.67) и (2.68), получаваме

$$c_1 c_3 (g(n))^2 \leqslant f(n)h(n) \leqslant c_2 c_4 (g(n))^2$$

Тъй като функциите са асимптотично положителни, БОО можем да допуснем, че и трите са положителни за стойности на аргумента $\geqslant n'$. Тогава имаме право да коренуваме. Коренувайки, получаваме

$$\sqrt{c_1 c_3}\, g(n) \leqslant \sqrt{f(n)h(n)} \leqslant \sqrt{c_2 c_4}\, g(n)$$

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 $n \geqslant n'$:

$$c' g(n) \leqslant \sqrt{f(n)h(n)} \leqslant c'' g(n)$$

Заключаваме, че $\sqrt{f(n)h(n)} \asymp g(n)$.

Сега да повдигнем (2.67) и (2.68) на квадрат. Получаваме

$$c_1^2(g(n))^2 \leqslant (f(n))^2 \leqslant c_2^2(g(n))^2 \tag{2.69}$$
$$c_3^2(g(n))^2 \leqslant (h(n))^2 \leqslant c_4^2(g(n))^2 \tag{2.70}$$

Тогава

$$\begin{aligned}
c_1^2(g(n))^2 + c_3^2(g(n))^2 &\leqslant (f(n))^2 + (h(n))^2 &&\leqslant c_2^2(g(n))^2 + c_4^2(g(n))^2 \leftrightarrow \\
(c_1^2 + c_3^2)(g(n))^2 &\leqslant (f(n))^2 + (h(n))^2 &&\leqslant (c_2^2 + c_4^2)(g(n))^2 \leftrightarrow \\
\sqrt{c_1^2 + c_3^2}g(n) &\leqslant \sqrt{(f(n))^2 + (h(n))^2} &&\leqslant \sqrt{c_2^2 + c_4^2}g(n)
\end{aligned}$$

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 $n \geqslant n'$:

$$c'g(n) \leqslant \sqrt{(f(n))^2 + (h(n))^2} \leqslant c''g(n)$$

Заключаваме, 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 $f(n) \preceq n$, то $g(f(n)) \preceq g(n)$.

Решение: Твърдението не е вярно. Контрапример е $f(n) = 2n$ и $g(n) = 2^n$. Тогава $g(f(n)) = 2^{2n} = 4^n$. Но $4^n \npreceq 2^n$. □

Задача 13

Нека $n \in \mathbb{N}$ и $f(n) \asymp g(n)$. Докажете или опровергайте, че

а) $2f(n) \succeq g(n) + 2$;

б) $2f(n) \succeq g(n + 2)$;

в) $2f(n) \succeq g(n + 2)$ или $2f(n) \preceq g(n + 2)$.

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 $2f(n) \succeq g(n) + 2$. По дефиниция това е кратък запис на

$$\exists c' \; \exists n_0' \; \forall n \geqslant n_0' : c'\frac{1}{n} + 2 \leqslant 2\frac{1}{n}$$

Но тогава

$$\exists c' \, \exists n_0' \, \forall n \geqslant n_0' : c' + 2n \leqslant 2$$

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 $2f(n) \succeq (f(n))^4$ 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, $2^{2^{n+2}} = \left(2^{2^n}\right)^4$.

От една страна, не е вярно, че $2f(n) \succeq g(n+2)$, 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) $n \geqslant n_0$, такива че $2f(n) < cg(n+2)$. Но това е точно логическото отрицание на условието за $2f(n) \succeq g(n+2)$.

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 $n \geqslant n_0$, такива че $2f(n) > cg(n+2)$. Но това е точно логическото отрицание на условието за $2f(n) \preceq g(n+2)$.

Доказахме, че нито $2f(n) \succeq g(n+2)$, нито $2f(n) \preceq g(n+2)$. □

2.3.7.2 Асимптотични сравнения на двойки функции

Определение 29: 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 12—или (2.39), или (2.40), или (2.41), или (2.42), или (2.43), или (2.44)—е в сила.

Наблюдение 13

Ако $f(n) \prec g(n)$, Теорема 12 казва, че задължително $f(n) \preceq g(n)$. Дуално, ако $f(n) \succ g(n)$, то задължително $f(n) \succeq g(n)$. И накрая, ако $f(n) \asymp g(n)$, то задължително $f(n) \preceq g(n)$ и $f(n) \succeq g(n)$. Поради това, за краткост в описанието на решението на задача от вида "Сравнете по асимптотично нарастване $f(n)$ и $g(n)$":

- ако $f(n) \prec g(n)$, не пишем и $f(n) \preceq g(n)$; то се подразбира,
- ако $f(n) \succ g(n)$, не пишем и $f(n) \succeq g(n)$; то се подразбира,
- ако $f(n) \asymp g(n)$, не пишем нито $f(n) \preceq g(n)$, нито $f(n) \succeq g(n)$; 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 $n \cdot \lg \lg n \succ n$, защото $\lg \lg n$ е неограничено растяща. От друга страна, $n \succ \lg^2 n$ от Следствие 11. Тогава $n \cdot \lg \lg n \succ \lg \lg n$ от транзитивността на релацията $\prec$. Тоест, $\lg g(n) \succ \lg f(n)$. Съгласно Следствие 9, в сила е $g(n) \succ f(n)$. □

Задача 15

Сравнете $\lfloor \lg n \rfloor!$ и $\lceil \lg n \rceil!$ по асимптотично нарастване.

Решение: Приели сме, че $n \in \mathbb{R}^+$. Ако $\lg n \in \mathbb{R}^+ \backslash \mathbb{N}^+$, в сила е $\lceil \lg n \rceil = \lfloor \lg n \rfloor + 1$, откъдето

$$\lceil \lg n \rceil! = (\lfloor \lg n \rfloor + 1) \lfloor \lg n \rfloor! \tag{2.71}$$

Тогава $\lfloor \lg n \rfloor! \not\asymp \lceil \lg n \rceil!$, защото

$$\neg(\exists c > 0\ \exists n_0\ \forall n \geqslant n_0 : \lceil \lg n \rceil! \leqslant c \lfloor \lg n \rfloor!) \leftrightarrow$$
$$\forall c > 0\ \forall n_0\ \exists n \geqslant n_0 : \lceil \lg n \rceil! > c \lfloor \lg n \rfloor!$$

Последното е очевидно предвидно предвид (2.71).

От друга страна, $\lfloor \lg n \rfloor! \not\prec \lceil \lg n \rceil!$, защото за всяко n, което е точна степен на двойката, имаме $\lceil \lg n \rceil = \lfloor \lg n \rfloor$.

Със сигурност е вярно, че $\lfloor \lg n \rfloor! \preceq \lceil \lg n \rceil!$, тъй като $\lfloor \lg n \rfloor! \leqslant \lceil \lg n \rceil!$ за всяко n.

Щом $\lfloor \lg n \rfloor! \not\asymp \lceil \lg n \rceil!$ и $\lfloor \lg n \rfloor! \not\prec \lceil \lg n \rceil!$ и $\lfloor \lg n \rfloor! \preceq \lceil \lg n \rceil!$, единствената релация измежду петте релации на асимптотични сравнения, която е в сила за въпросните две функции в този ред, е $\preceq$; това отговаря на (2.40) в Теорема 12. Ситуацията е аналогична на сравнението на $f(n) = 2^{2^{\lfloor n \rfloor}}$ и $g(n) = 2^{2^{\lceil n \rceil}}$ на стр. 96. □

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 $\lceil \lg n \rceil!$ или $\lfloor \lg n \rfloor!$, защото тези две функции не са асимптотично еквивалентни (Задача 15). Ако факториелът в $(\lg n)!$ се апроксимира с апроксимацията на Stirling (Теорема 20), получаваме

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{e^{\lg n}} = \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}} \tag{2.72}$$

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 21, 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 (2.72).

Но $(\lg n) \cdot (\lg \lg n) \succ (\lg n) \cdot (\lg \lg \lg n)$, защото $\lg \lg n \succ \lg \lg \lg n$, което следва от Теорема 19. Тоест, $\lg g(n) \succ \lg f(n)$. Съгласно Следствие 9, в сила е $g(n) \succ f(n)$. □

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

$$\lg f(n) = \lg(n!!)$$
$$\lg g(n) = \lg\left((n-1)!! \cdot ((n-1)!)^{n!}\right) = \lg((n-1)!!) + n! \cdot \lg((n-1)!)$$

Прилагаме Теорема 21 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 $n! \cdot (n-1) \cdot \lg(n-1) \succ (n-1)! \cdot (n-1) \cdot \lg(n-1)!$. Наистина,

$$\lim_{n \to \infty} \frac{n! \cdot (n-1) \cdot \lg(n-1)}{(n-1)! \cdot (n-1) \cdot \lg(n-1)!} = \lim_{n \to \infty} \frac{n \cdot (n-1)! \cdot (n-1) \cdot \lg(n-1)}{(n-1)! \cdot (n-1) \cdot \lg(n-1)!} = \lim_{n \to \infty} n = \infty$$

така че, съгласно Лема 10, наистина $n! \cdot (n-1) \cdot \lg(n-1) \succ (n-1)! \cdot (n-1) \cdot \lg(n-1)!$. Прилагаме Теорема 14 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	$\approx 6 \times 10^{23}$	$\approx 6 \times 10^{198}$	$\approx 2 \times 10^{1\,746}$	$\approx 4 \times 10^{16\,473}$	$\approx 9 \times 10^{22\,228\,103}$
$g(n)$	1	1	128	$\approx 3 \times 10^{21}$	$\approx 2 \times 10^{189}$	$\approx 6 \times 10^{1\,695}$	$\approx 2 \times 10^{16\,147}$	$\approx 2 \times 10^{22\,035\,201}$

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 $f(n) \prec g(n)$?

Да положим $(n-1)! = \nu$. С оглед на тази субституция имаме $n! = n\nu$, така че сравняваме по асимптотично нарастване $\phi(n, \nu) = (n\nu)!$ и $\psi(n, \nu) = \nu! \times \nu^{n\nu}$. Да приложим апроксимацията на Stirling към ϕ и ψ.

$$\phi(n, \nu) \asymp \sqrt{n\nu}\,\frac{(n\nu)^{n\nu}}{e^{n\nu}} = \sqrt{2\pi n\nu}\,\frac{n^{n\nu} \cdot \nu^{n\nu}}{e^{n\nu}}$$

$$\psi(n, \nu) \asymp \sqrt{\nu}\,\frac{\nu^{\nu}}{e^{\nu}} \times \nu^{n\nu}$$

Да разгледаме частното на двете функции в десните страни.

$$\frac{\sqrt{n\nu}\,\frac{n^{n\nu}\cdot\nu^{n\nu}}{e^{n\nu}}}{\sqrt{\nu}\,\frac{\nu^{\nu}}{e^{\nu}} \times \nu^{n\nu}} = \sqrt{n} \cdot \frac{\frac{n^{n\nu}}{e^{n\nu}}}{\frac{\nu^{\nu}}{e^{\nu}}} = \sqrt{n} \cdot \frac{n^{n\nu}}{e^{(n-1)\nu} \cdot \nu^{\nu}}$$

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

$$\frac{n^{n\nu}}{e^{(n-1)\nu} \cdot \nu^{\nu}} = \left(\frac{n^{n}}{e^{(n-1)} \cdot \nu}\right)^{\nu}$$

Да разгледаме основата $\frac{n^n}{e^{(n-1)}\cdot\nu}$. Ще покажем, че това частно клони към безкрайност при неограничено нарастване на n, и така ще получим желания резултат. Забележете, че това е същото като да вземем $\sqrt[\nu]{}$, тоест, $\sqrt[(n-1)!]{}$. За да се освободим от ν в частното, спомняме си, че $\nu = (n-1)!$, и прилагаме апроксимацията на Stirling за $(n-1)!$.

$$\frac{n^n}{e^{(n-1)} \cdot \nu} = \frac{n^n}{e^{(n-1)} \cdot (n-1)!} \asymp \frac{n^n}{e^{(n-1)} \cdot \sqrt{n-1}\,\frac{(n-1)^{n-1}}{e^{n-1}}} = \frac{1}{\sqrt{n-1}} \cdot \frac{n^{n-1} \cdot n}{(n-1)^{n-1}} =$$

$$\frac{n}{\sqrt{n-1}} \cdot \left(\frac{n}{n-1}\right)^{n-1}$$

Но очевидно

$$\lim_{n\to\infty} \frac{n}{\sqrt{n-1}} \cdot \left(\frac{n}{n-1}\right)^{n-1} = \infty$$

откъдето следва, че $f(n) \succ g(n)$. □

2.3.7.3 Подреждане на функции по асимптотично нарастване

Често срещана задача е тази: дадени са функции $f_1(n)$, $f_2(n)$, ..., $f_k(n)$ 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, (2.39) от Теорема 12 не се допуска. Нещо повече, (2.40) и (2.44) от Теорема 12 не се допускат. Съгласно Следствие 7, $\forall i, j \in \{1, \ldots, k\}$ е изпълнено

$$f_i(n) \preceq f_j(n) \;\text{ или }\; f_i(n) \succeq f_j(n)$$

Тъй като не се допуска $f_i(n) \preceq f_j(n) \wedge f_i(n) \nprec f_j(n)$ и не се допуска $f_i(n) \succeq f_j(n) \wedge f_i(n) \nsucc f_j(n)$, в сила е точно едно от следните:

$$f_i(n) \prec f_j(n) \wedge f_i(n) \preceq f_j(n)$$
$$f_i(n) \succ f_j(n) \wedge f_i(n) \succeq f_j(n)$$
$$f_i(n) \preceq f_j(n) \wedge f_i(n) \asymp f_j(n) \wedge f_i(n) \succeq f_j(n)$$

Тъй като $f_i(n) \prec f_j(n)$ влече $f_i(n) \preceq f_j(n)$ и $f_i(n) \succ f_j(n)$ влече $f_i(n) \succeq f_j(n)$ и $f_i(n) \asymp f_j(n)$ влече $f_i(n) \preceq f_j(n) \wedge f_i(n) \succeq f_j(n)$, можем да опростим описанието на трите възможности до:

$$f_i(n) \prec f_j(n)$$
$$f_i(n) \succ f_j(n)$$
$$f_i(n) \asymp f_j(n)$$

В текущия контекст, да бъдат сравнени по асимптотично нарастване $f_i(n)$ и $f_j(n)$ означава да се определи дали $f_i(n) \prec f_j(n)$, или $f_i(n) \succ f_j(n)$, 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 $\pi : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}$, такава че:

$$f_{\pi(1)} \prec f_{\pi(2)} \prec \cdots \prec f_{\pi(k-1)} \prec f_{\pi(k)} \tag{2.73}$$

Естествено, можем да ги подредим и в обратния ред:

$$f_{\pi(k)} \succ f_{\pi(k-1)} \succ \cdots \succ f_{\pi(2)} \succ f_{\pi(1)} \tag{2.74}$$

като (2.73) и (2.74) са еквивалентни заради транспонираната симетрия, но предпочитаме (2.73), защото при сортирането по подразбиране сортираме в нарастващ, или поне ненамаляващ, ред (Задача 9).

В този случай (когато няма асимптотично еквивалентни функции), релацията $\preceq$ върху множеството от дадените функции съвпада, неформално казано, с $\prec$. Строго формално, това не е вярно, понеже $\preceq$ е рефлексивна, а $\prec$ е ирефлексивна, но имаме предвид, че дадените функции може да бъдат линейно наредени чрез $\preceq$ по един единствен начин, а именно

$$f_{\pi(1)} \preceq f_{\pi(2)} \preceq \cdots \preceq f_{\pi(k-1)} \preceq f_{\pi(k)} \tag{2.75}$$

Това е истинска линейна наредба съгласно дефиницията от Дискретни Структури, понеже е рефлексивна, силно антисиметрична и транзитивна. Наредбата от (2.73) е ирефлексивна, силно антисиметрична и транзитивна, като наредбата от (2.75) се явява рефлексивното затваряне на наредбата от (2.73)

Дотук стана ясно какво се иска в задачата: да се намери линейната 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 $1 \leqslant i < j \leqslant n$, 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 (2.73). По този начин се правят само $k - 1$ сравнения, което е същността на Наблюдение 14. Забележете, че Наблюдение 14 е аналогично на Теорема 70 от Лекция 13.

Наблюдение 14

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

$$f_{\pi(1)} \prec f_{\pi(2)} \prec \cdots \prec f_{\pi(k-1)} \prec f_{\pi(k)}$$

За всеки две $f_i(n)$ и $f_j(n)$, които не са непосредствени съседи в (2.73), фактът, че $f_i(n) \prec f_j(n)$, следва от $f_{\pi(1)} \prec f_{\pi(2)} \prec \cdots \prec f_{\pi(k-1)} \prec f_{\pi(k)}$ и транзитивността на релацията $\prec$. 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 14, за да подредим по асимптотично нарастване $f_1(n), \ldots, f_k(n)$, 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 (2.73).

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 $\{f_1(n), \ldots, f_k(n)\}$ се разбива на малък брой подмножества $F_1, \ldots, F_t(n)$, 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 $F_1, \ldots, F_t(n)$, решаването на задачата се свежда до намирането на линейната поднаредба с $\prec$ на функциите във всяко от множествата. След това окончателното решение лесно се "сглобява" от тези вериги. Примерно, ако функциите за сравняване са $\lg n$, $\frac{\lg n}{\lg \lg n}$, $2n^2$, $n^3 + \binom{n}{2}$, 2^n, $3^{\frac{n}{\lg n}}$ и $n!$, 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 с $\prec$ в рамките на всяко от множествата са съответно

$$\frac{\lg n}{\lg \lg n} \prec \lg n$$
$$2n^2 \prec n^3 + \binom{n}{2}$$
$$2^n \prec 3^{\frac{n}{\lg n}} \prec n!$$

откъдето решението е

$$\frac{\lg n}{\lg \lg n} \prec \lg n \prec 2n^2 \prec n^3 + \binom{n}{2} \prec 2^n \prec 3^{\frac{n}{\lg n}} \prec n!$$

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* $\frac{\lg n}{\lg \lg n} \prec \lg n$.

`......някак доказваме, че` $\frac{\lg n}{\lg \lg n} \prec \lg n$

Да разгледаме функциите $\lg n$ *и* $2n^2$. *Ще покажем, че* $\lg n \prec 2n^2$.

`......някак доказваме, че` $\lg n \prec 2n^2$

Да разгледаме функциите $2n^2$ *и* $n^3 + \binom{n}{2}$. *Ще покажем, че* $2n^2 \prec n^3 + \binom{n}{2}$.

`......някак доказваме, че` $2n^2 \prec n^3 + \binom{n}{2}$

`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:

$$\lg n \prec n^2 + n \asymp \binom{n}{2} \prec 2^n$$
$$\lg n \prec \binom{n}{2} \asymp n^2 + n \prec 2^n$$

Ясно е, че в този случай $\preceq$ е преднаредба, която може да не съвпада с рефлексивното затваряне на $\prec$; знаем, че ако $f_i(n) \asymp f_j(n)$, то $f_i(n) \not\prec f_j(n)$ и $f_j(n) \not\prec f_i(n)$ (Лема 14, (2.35)). Съществено е, че $\preceq$ е пълна преднаредба (Определение 31), понеже в текущите допускания всеки две функции са асимптотично сравними.

Тъй като $\asymp$ е релация на еквивалентност (Лема 9), имаме право да разгледаме фактор-релацията $\preceq / \asymp$. Знаем от Лема 15, че $\preceq / \asymp$ е частична наредба. Следователно, в текущите допускания $\preceq / \asymp$ е линейна наредба, тъй като $\preceq$ е пълна.

Класовете на еквивалентност на $\preceq$ са максималните по включване подмножества на $\{f_1(n), \ldots, f_k(n)\}$, всяко от което се състои от асимптотично еквивалентни функции. Примерно, ако множеството от функциите е $\{\lg n, n^2+n, \binom{n}{2}, 2^n, 3\lg n + \lg\lg n\}$, въпросните класове на еквивалентност са $\{\lg n, 3\lg n + \lg\lg n\}$, $\{n^2+n, \binom{n}{2}\}$ и $\{2^n\}$.

Да разгледаме общия случай, в който функциите са $f_1(n), \ldots, f_k(n)$. Нека класовете на еквивалентност на $\preceq$ са m на брой. Да ги наречем $F_1, F_2, \ldots, F_m$. Да кажем, че $|F_1| = k_1$, $|F_2| = k_2, \ldots, |F_m| = k_m$. Очевидно $k_1 + k_2 + \cdots + k_m = k$. 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 "$\prec$" можем да напишем това така

$$F_1 \prec F_2 \prec F_3 \prec \ldots \prec F_{m-1} \prec F_m$$

Решението на задачата да се подредят по асимптотично нарастване $f_1(n), \ldots, f_k(n)$ се състои от следните две дейности.

1. Да се намерят класовете на еквивалентност. За $1 \leqslant i \leqslant m$, 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 $g_1(n), g_2(n), \ldots, g_{k_i}(n)$, достатъчно е да покажем, че

$$\begin{aligned} g_1(n) &\asymp g_2(n) \\ g_2(n) &\asymp g_3(n) \\ &\ldots \\ g_{k_i-1}(n) &\asymp g_{k_i}(n) \end{aligned}$$

   Това са $k_i - 1$ сравнения. Разбира се, сравненията може да са други. Примерно тези

$$\begin{aligned} g_1(n) &\asymp g_2(n) \\ g_1(n) &\asymp g_3(n) \\ &\ldots \\ g_1(n) &\asymp g_{k_i-1}(n) \\ g_1(n) &\asymp g_{k_i}(n) \end{aligned}$$

   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

$g_1(n)$, $g_2(n)$, …, $g_{k_i}(n)$[†]. 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! Искаме 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 примем. 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 представляват покриващо дърво на пълния граф с върхове шестте функции.

2. Да се подредят класовете на еквивалентост чрез $\prec$. 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 $f'(n) \prec f''(n)$, очевидно е вярно, че $h'(n) \prec h''(n)$ 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

$(k_1 - 1) +$ // за установяване на взаимната асимпт. еквивалентност на ф-иите от F_1
$(k_2 - 1) +$ // за установяване на взаимната асимпт. еквивалентност на ф-иите от F_2
…
$(k_m - 1) +$ // за установяване на взаимната асимпт. еквивалентност на ф-иите от F_m
$m - 1$ // за подреждане на класовете в линейна наредба

[†]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 $f_1(n)$, …, $f_k(n)$, така че извършваме $1 - 1 = 0$ сравнения за "намирането"—което е празното действие, понеже не правим нищо—на F_i.

Но тази сума е

$$(k_1 + k_2 + \cdots + k_m - m) + (m - 1) = k_1 + k_2 + \cdots + k_m - 1 = k - 1$$

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 (2.52), 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 (2.62). Тогава $\lg a^n \prec \lg n!$. Тогава $a^n \prec n!$ съгласно Следствие 9.

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 $\lim_{n\to\infty} \frac{n!}{n^n}$. Ще покажем, че границата съществува и е 0. Наистина, след като заместим $n!$ с $\sqrt{n} \cdot \frac{n^n}{e^n}$, получаваме

$$\lim_{n\to\infty} \frac{\sqrt{n} \cdot \frac{n^n}{e^n}}{n^n} = \lim_{n\to\infty} \sqrt{n} \cdot \frac{1}{e^n} = 0$$

Тогава $n! \prec n^n$ съгласно (10).

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 $n \lg n \prec n^2$, а $n^2 \prec a^n$ съгласно Следствие 11, в сила е $n \lg n \prec a^n$. Съгласно Следствие 9, в сила е $n^n \prec a^{a^n}$.

Окончателната наредба е

$$a^n \prec n! \prec n^n \prec a^{a^n}$$ □

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

$$\lim_{n\to\infty} \frac{(n+1)^n}{n^n} = \lim_{n\to\infty} \left(\frac{n+1}{n}\right)^n = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$$

Заключаваме, 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 $n^n \prec n^{(n+1)}$.

Да сравним $n^{(n+1)}$ с $(n+1)^{(n+1)}$. Да разгледаме границата

$$\lim_{n\to\infty} \frac{(n+1)^{n+1}}{n^{n+1}} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^{n+1} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \cdot \lim_{n\to\infty} \left(1 + \frac{1}{n}\right) = e \cdot 1 = e$$

Тогава $n^{(n+1)} \asymp (n+1)^{(n+1)}$.

Окончателната наредба е

$$n^n \asymp (n+1)^n \prec n^{(n+1)} \asymp (n+1)^{(n+1)}$$ □

Задача 20

Наредете по асимптотично нарастване следните функции. Обосновете отговорите си. Напишете в явен вид наредбата. *Забележка: всички логаритми са с основа* 2.

$$f_1(n) = 2n, \quad f_2(n) = n^{n!}, \quad f_3(n) = 2^{(\lg n)^2}, \quad f_4(n) = 2^{\lg(n^2)}, \quad f_5(n) = \frac{\sqrt{n}}{\lg \lg n},$$

$$f_6(n) = \binom{2n}{2}^2, \quad f_7(n) = \lg \binom{2n}{n}, \quad f_8(n) = \sqrt[3]{n} \lg \lg n, \quad f_9(n) = n^{\lg \lg n}, \quad f_{10}(n) = n!^n$$

Решение: Ще извършим девет сравнения на функции.

(i) Ще покажем, че $f_8 \prec f_5$. Наистина,

$$f_8(n) = \sqrt[3]{n} \lg \lg n = n^{\frac{1}{3}} \lg \lg n \prec \quad // \text{ Теорема } 19$$
$$n^{\frac{1}{3}} \lg n \prec \quad // \text{ Следствие } 11$$
$$n^{\frac{1}{3}} n^{\frac{1}{12}} = n^{\frac{5}{12}} \prec \quad // \text{ Следствие } 11$$
$$n^{\frac{5}{12}} \cdot \frac{n^{\frac{1}{12}}}{\lg \lg n} = \frac{n^{\frac{6}{12}}}{\lg \lg n} = \frac{\sqrt{n}}{\lg \lg n} = f_5(n)$$

(ii) Ще покажем, че $f_5 \prec f_1$. Но това е напълно очевидно, понеже $\sqrt{n} \prec n$, така че $\frac{\sqrt{n}}{\lg \lg n} \prec 2n$.

(iii) Ще покажем, че $f_1 \asymp f_7$. Използваме Теорема 23, 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 $f_1 \prec f_4$. Веднага се вижда, че

$$f_4(n) = 2^{\lg(n^2)} = n^2$$

Това, че $n \prec n^2$, е очевидно.

(v) Ще покажем, че $f_4 \prec f_6$. 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 $f_6 \prec f_9$. Помним, че $f_6(n) \asymp n^4$. От друга страна, двойният логаритъм е растяща функция на n, така че $n^4 \prec n^{\lg \lg n}$.

(vii) Ще покажем, че $f_9 \prec f_3$. Първо преобразуваме f_3 така:

$$f_3(n) = 2^{(\lg n)^2} = 2^{(\lg n)\cdot(\lg n)} = 2^{\lg\left(n^{\lg n}\right)} = n^{\lg n}$$

Това, че $n^{\lg\lg n} \prec n^{\lg n}$, се вижда веднага, понеже логаритъмът е растяща функция.

(viii) Ще покажем, че $f_3 \prec f_{10}$. Логаритмуваме двете функции:

$$\lg\left(n^{\lg n}\right) = (\lg n)(\lg n) = (\lg n)^2$$
$$\lg\left(n!^n\right) = n \lg n! \asymp n^2 \lg n \quad // \text{ съгласно Теорема 21}$$

Но $(\lg n)^2 \prec n^2 \lg n$ съгласно Следствие 11. Заключаваме, че $n^{\lg n} \prec n!^n$.

(ix) Ще покажем, че $f_{10} \prec f_2$. Вече видяхме, че $\lg f_{10}(n) \asymp n^2 \lg n$. От друга страна, $\lg\left(n^{n!}\right) \asymp n! \lg n$. Тъй като факториелът очевидно расте асимптотично по-бързо от всяка полиномиална функция, заключаваме, че $\lg f_{10}(n) \prec \lg f_2(n)$. Оттук веднага следва $f_{10} \prec f_2$.

Окончателната наредба е:

$$f_8 \prec f_5 \prec f_1 \asymp f_7 \prec f_4 \prec f_6 \prec f_9 \prec f_3 \prec f_{10} \prec f_2$$ □

Задача 21

Наредете по асимптотично нарастване следните функции. Обосновете отговорите си. Напишете в явен вид наредбата. *Забележка: всички логаритми са с основа* 2.

$f_1 = (\lg n)^{\lg n}$, $\quad f_2 = \left(\lg\left(\sqrt{n}\right)\right)^{\lg\left(\sqrt{n}\right)}$, $\quad f_3 = (\lg n)^{(\lg n)^{\lg n}}$

$f_4 = 2^{2^{n-1}}$, $\quad f_5 = \sum_{k=0}^{n} \binom{n}{k}$, $\quad f_6 = \binom{2n}{n}$

Решение: Ще извършим пет сравнения на функции.

(i) Ще докажем, че $f_2 \prec f_1$. Не е добра идея да се логаритмуват двете функции, защото образите след логаритмичната трансформация са асимптотично еквивалентни. Действаме така:

$$f_2 = \left(\lg\left(\sqrt{n}\right)\right)^{\lg\left(\sqrt{n}\right)} = \left(\frac{1}{2}\lg n\right)^{\frac{1}{2}\lg n} \leqslant (\lg n)^{\frac{1}{2}\lg n} = \left((\lg n)^{\frac{1}{2}}\right)^{\lg n} = \left(\sqrt{\lg n}\right)^{\lg n}$$

Но $\left(\sqrt{\lg n}\right)^{\lg n} \prec (\lg n)^{\lg n}$, понеже

$$\lim_{n\to\infty} \frac{\left(\sqrt{\lg n}\right)^{\lg n}}{(\lg n)^{\lg n}} = \lim_{n\to\infty} \left(\frac{\sqrt{\lg n}}{\lg n}\right)^{\lg n} = \lim_{n\to\infty} \left(\frac{1}{\sqrt{\lg n}}\right)^{\lg n} = \lim_{n\to\infty} \frac{1}{\left(\sqrt{\lg n}\right)^{\lg n}} = 0$$

Тогава наистина $f_2 \prec f_1$.

(ii) Ще докажем, че $f_1 \prec f_5$. От Теоремата на Newton следва, че $f_5 = 2^n$. Сравняваме $(\lg n)^{\lg n}$ с 2^n. Логаритмуваме двете функции. Сега сравнямаме $\lg f_1(n) = (\lg n)(\lg\lg n)$ с

$\lg f_5(n) = n$. Но всяка полилогаритмично нарастваща функция расте асимптотично по-бавно от всяка полиномиално растяща функция (Следствие 11), така че $(\lg n)^2 \prec n$. Имайки предвид, че $(\lg n)(\lg\lg n) \preceq (\lg n)^2$ (Теорема 19), заключаваме, че $(\lg n)(\lg\lg n) \prec n$. Съгласно Следствие 9, в сила е $f_1 \prec f_5$.

(iii) Ще докажем, че $f_5 \prec f_6$. От Теоремата на Newton следва, че $f_5 = 2^n$. Теорема 23 казва, че $f_6 \asymp \frac{4^n}{\sqrt{n}}$. Тривиално се доказва, че $\lim_{n\to\infty} \frac{\frac{4^n}{\sqrt{n}}}{2^n} = \infty$. Доказахме, че $f_5 \prec f_6$.

(iv) Ще докажем, че $f_6 \prec f_3$. Както вече видяхме, $f_6 \asymp \frac{4^n}{\sqrt{n}}$. Сравняваме $\frac{4^n}{\sqrt{n}}$ и f_3, логаритмувайки и двете. Очевидно

$$\lg f_6 = \lg \frac{4^n}{\sqrt{n}} \asymp n \tag{2.76}$$

$$\lg f_3 = \left((\lg n)^{\lg n}\right)(\lg n) \tag{2.77}$$

Но $n \prec (\lg n)^{\lg n}$. 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 $\lg n \prec (\lg n)(\lg\lg n)$, понеже $\lg\lg n$ е неограничено растяща. Заключаваме, че наистина $n \prec (\lg n)^{\lg n}$.

От това и (2.76) следва, че $\lg f_6 \prec (\lg n)^{\lg n}$. Но $(\lg n)^{\lg n} \prec \left((\lg n)^{\lg n}\right)(\lg n)$, понеже $\lg n$ е неограничено растяща. Тогава $\lg f_6 \prec \left((\lg n)^{\lg n}\right)(\lg n)$. От това и (2.77) следва, че $\lg f_6 \prec \lg f_3$. Съгласно Следствие 9, $f_6 \prec f_3$.

(v) Ще докажем, че $f_3 \prec f_4$. 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:

$$f_2 \prec f_1 \prec f_5 \prec f_6 \prec f_3 \prec f_4$$ □

Сега ще подредим по асимптотично нарастване прочутите тридесет функции от учебника CLR [31, зад. 3-3, стр. 61]. Първо една дефиниция, която ползва итерирания логаритъм от Определение 28.

Определение 30: функция $\lg^* n$ [31, стр. 58]

Тогава

$$\lg^* n = \min\left\{i \geqslant 0 \mid \lg^{(i)} n \leqslant 1\right\}$$

"$\lg^* n$" се чете *log-star* на английски. На български ще казваме "лог-звезда".

Да видим стойностите на функцията лог-звезда за някои стойности на аргумента.

$$
\begin{aligned}
\lg^* 2 &= 1, \text{ понеже } \lg^{(0)} 2 = 2 \text{ и } \lg^{(1)} 2 = \lg\left(\lg^{(0)} 2\right) = \lg(2) = 1 \\
\lg^* 3 &= 2, \text{ понеже } \lg^{(0)} 3 = 3 \text{ и } \lg\left(\lg^{(0)} 3\right) = \lg(\lg 3) = 0.6644\ldots \\
\lg^* 4 &= 2 \\
\lg^* 5 &= 3 \\
&\ldots \\
\lg^* 16 &= 3 \\
\lg^* 17 &= 4 \\
&\ldots \\
\lg^* 65536 &= 4 \\
\lg^* 65537 &= 5 \\
&\ldots \\
\lg^* 2^{65536} &= 5 \\
\lg^* \left(2^{65536} + 1\right) &= 6 \\
\lg^* 2^{2^{65536}} &= 6 \\
&\ldots \\
\lg^* \left(2^{2^{65536}} + 1\right) &= 7
\end{aligned}
$$

Очевидно, всяко реално число $x \geqslant 2$ може да се представи като кула от двойки така:

$$x = 2^{2^{\cdot^{\cdot^{\cdot^{2^{\alpha}}}}}}$$

където α е реално число, такова че $1 < \alpha \leqslant 2$. *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	$2^{1.5849625007\ldots}$	2	2
4	2^2	2	2
5	$2^{2^{1.2153232957\ldots}}$	3	3
16	2^{2^2}	3	3
17	$2^{2^{2^{1.0223362884\ldots}}}$	4	4
65536	$2^{2^{2^2}}$	4	4
65537	$2^{2^{2^{2^{1.00000051642167\ldots}}}}$	5	5

Лесно се вижда, че $\lg^* n$ е височината на кулата на n.

Задача 22: [31, зад. 3-3, стр. 61]

Подредете по асимптотично нарастване следните тридесет функции.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$\left(\sqrt{2}\right)^{\lg n}$	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 $1 \prec \lg(\lg^* n)$, 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 $\lg f(n) \prec f(n)$ (Теорема 19). В частност, $\lg(\lg^* n) \prec \lg^* n$.

(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) Сравяваме $\lg^* n$ и $2^{\lg^* n}$. Но за всяка неограничено растяща функция $f(n)$ е вярно, че $f(n) \prec 2^{f(n)}$ (леко следствие от Теорема 19). В частност, $\lg^* n \prec 2^{\lg^* n}$.

(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 $\lg\left(2^{\lg^* n}\right) = (\lg^* n) \cdot (\lg 2) = \lg^* n$ и $\lg \lg \lg n = \lg^{(3)} n$. Очевидно $\lg^* n \prec \lg^{(k)} n$ за всяка константа $k \geqslant 0$, така че $\lg^* n \prec \lg^{(3)} n$. Тогава $2^{\lg^* n} \prec \lg \lg n$. Тогава $2^{\lg^* n} \prec \ln \ln n$.

(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 19, $\lg \lg \lg n \prec \frac{1}{2} \lg \lg n$. Тогава $\lg \lg n \prec \sqrt{\lg n}$. Тогава $\ln \ln n \prec \sqrt{\lg n}$.

(viii) Сравняваме $\sqrt{\lg n}$ и $\ln n$. Тъй като $\ln n \asymp \lg n$, ще сравним $\sqrt{\lg n}$ и $\lg n$. Очевидно $\lim_{n \to \infty} \frac{\sqrt{\lg n}}{\lg n} = 0$. Тогава $\sqrt{\lg n} \prec \lg n$. Тогава $\sqrt{\lg n} \prec \ln n$.

(ix) Сравняваме $\ln n$ и $\lg^2 n$. Тъй като $\ln n \asymp \lg n$, ще сравним $\lg n$ и $\lg^2 n$. Очевидно $\lim_{n\to\infty} \frac{\lg n}{\lg^2 n} = 0$. Тогава $\lg n \prec \lg^2 n$. Тогава $\ln n \prec \lg^2 n$.

(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 $\lg\lg n \prec \sqrt{\lg n}$ (Следствие 11). Тогава $2\lg\lg n \prec (\sqrt{2\lg n}) \cdot \lg 2$. Тогава $\lg^2 n \prec 2^{\sqrt{2\lg n}}$.

(xi) Сравняваме $2^{\sqrt{2\lg n}}$ и $(\sqrt{2})^{\lg n}$. Да разгледаме логаритмите на двете функции. Те са съответно $\lg\left(2^{\sqrt{2\lg n}}\right) = (\sqrt{2\lg n})\cdot\sqrt{2} = \sqrt{2}\sqrt{\lg n}$ и $\lg\left((\sqrt{2})^{\lg n}\right) = (\lg n)\cdot\left(\lg\sqrt{2}\right)$. Но $\sqrt{\lg n} \prec \lg n$ (Следствие 11). Тогава $\lg\left(2^{\sqrt{2\lg n}}\right) \prec \lg\left((\sqrt{2})^{\lg n}\right)$. Тогава $2^{\sqrt{2\lg n}} \prec (\sqrt{2})^{\lg n}$.

(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 $\sqrt{n} \prec n$. Тогава $(\sqrt{2})^{\lg n} \prec n$.

(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 $n \prec n\lg n$.

(xv) Да сравним $n\lg n$ и $\lg n!$. Теорема 21 казва, че $n\lg n \asymp \lg n!$.

(xvi) Да сравним $n\lg n$ и n^2. В сила е $\lim_{n\to\infty} \frac{n\lg n}{n^2} = \lim_{n\to\infty} \frac{\lg n}{n} = 0$, понеже $\lg n \prec n$ съгласно Следствие 11. Тогава $n\lg n \prec n^2$.

(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 $\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$. Тогава $n^2 \prec n^3$.

(xix) Да сравним n^3 с $(\lg n)!$. Вече видяхме (Задача 16, (2.72)), че

$$(\lg n)! \asymp \sqrt{\lg n} \cdot \frac{(\lg n)^{\lg n}}{n^{\lg e}}$$

ако ползваме апроксимацията на Stirling (Теорема 20). 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

$$\lim_{n\to\infty} \frac{\sqrt{\lg n} \cdot \frac{n^{\lg\lg n}}{n^{\lg e}}}{n^3} = \lim_{n\to\infty} \sqrt{\lg n} \cdot \frac{n^{\lg\lg n}}{n^{3+\lg e}} = \infty$$

понеже $\lg\lg n$ е неограничено растяща функция, а $3 + \lg e$ е константа. Заключаваме, че $n^3 \prec (\lg n)!$.

(xx) Да сравним $(\lg n)!$ и $(\lg n)^{\lg n}$. От (Задача 16, (2.72)) 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

$$\lim_{n\to\infty} \frac{\frac{\sqrt{\lg n}}{n^{\lg e}} \cdot (\lg n)^{\lg n}}{(\lg n)^{\lg n}} = \lim_{n\to\infty} \frac{\sqrt{\lg n}}{n^{\lg e}} = 0$$

тъй като $\sqrt{\lg n} \prec n^{\lg e}$ (Следствие 11). Заключаваме, че $(\lg n)! \prec (\lg n)^{\lg n}$.

(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 $(\lg n)(\lg \lg n) \prec (\lg n)^2$, а $(\lg n)^2 \prec n \lg\left(\frac{3}{2}\right)$ съгласно Следствие 11. Тогава $(\lg n)(\lg \lg n) \prec n \lg\left(\frac{3}{2}\right)$. Тогава $n^{\lg \lg n} \prec \left(\frac{3}{2}\right)^n$ (Следствие 9).

(xxiii) Да сравним $\left(\frac{3}{2}\right)^n$ и 2^n. Забелязваме, че

$$\lim_{n\to\infty} \frac{\left(\frac{3}{2}\right)^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{4}\right)^n = 0$$

Тогава $\left(\frac{3}{2}\right)^n \prec 2^n$ (Лема 10).

(xxiv) Да сравним 2^n и $n \cdot 2^n$. Очевидно $2^n \prec n \cdot 2^n$, понеже n е неограничено растяща функция.

(xxv) Да сравним $n \cdot 2^n$ и e^n.

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{e^n} = \lim_{n\to\infty} \frac{n}{\frac{e^n}{2^n}} = \lim_{n\to\infty} \frac{n}{\left(\frac{e}{2}\right)^n} = 0$$

понеже $n \prec \left(\frac{e}{2}\right)^n$ (Следствие 11). Тогава $n \cdot 2^n \prec e^n$ (Лема 10).

(xxvi) Да сравним e^n и $n!$. От Задача 18 знаем, че $e^n \prec n!$.

(xxvii) Да сравним $n!$ и $(n+1)!$. Но $(n+1)! = (n+1) \cdot n!$. Очевидно $n! \prec (n+1) \cdot n!$, понеже $n+1$ е неограничено растяща функция. Тогава $n! \prec (n+1)!$.

(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 $n\lg n \prec n^2$, а $n^2 \prec 2^n$ съгласно Следствие 11. Тогава $\Theta((n+1)\lg(n+1)) \prec 2^n$. Тогава $(n+1)! \prec 2^{2^n}$ (Следствие 9).

(xxix) Да сравним 2^{2^n} и $2^{2^{n+1}}$. Очевидно $2^{2^{n+1}} = 2^{2.2^n} = \left(2^{2^n}\right)^2$. Тогава

$$\lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{n+1}}} = \lim_{n\to\infty} \frac{2^{2^n}}{\left(2^{2^n}\right)^2} = \lim_{n\to\infty} \frac{1}{2^{2^n}} = 0$$

Съгласно Лема 10, в сила е $2^{2^n} \prec 2^{2^{n+1}}$.

Окончателната наредба е

$$\begin{array}{lclclclclclcl}
1 & \asymp & n^{\frac{1}{\lg n}} & \prec & \lg(\lg^* n) & \prec & \lg^* n & \asymp & \lg^*(\lg n) & \prec & 2^{\lg^* n} & \prec \\
\ln\ln n & \prec & \sqrt{\lg n} & \prec & \ln n & \prec & \lg^2 n & \prec & 2^{\sqrt{2\lg n}} & \prec & (\sqrt{2})^{\lg n} & \prec \\
n & \asymp & 2^{\lg n} & \prec & n\lg n & \asymp & \lg n! & \prec & n^2 & \asymp & 4^{\lg n} & \prec \\
n^3 & \prec & (\lg n)! & \prec & (\lg n)^{\lg n} & \asymp & n^{\lg\lg n} & \prec & \left(\frac{3}{2}\right)^n & \prec & 2^n & \prec \\
n\cdot 2^n & \prec & e^n & \prec & n! & \prec & (n+1)! & \prec & 2^{2^n} & \prec & 2^{2^{n+1}} &
\end{array}$$

□

2.3.8 Релациите на асимптотични сравнения при целочислена променлива

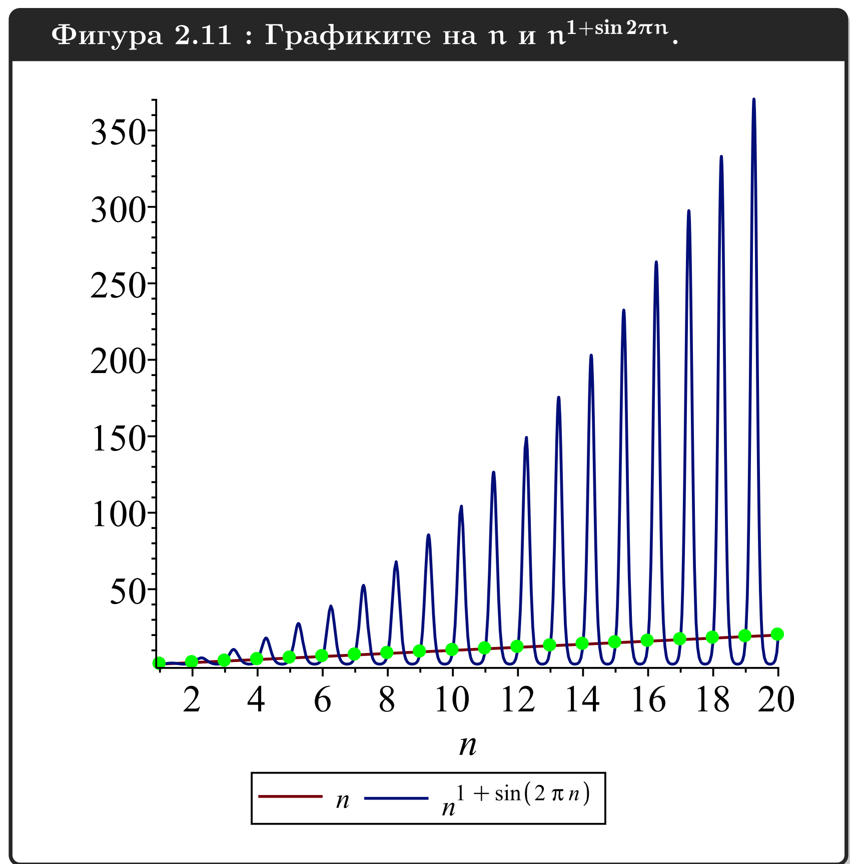
Вече приехме, че разглеждаме функции с домейн $\mathbb{R}^+$ (вж. Конвенция 4). Дали нещо ще се промени, ако преминем към домейн $\mathbb{N}^+$? От гледна точка на алгоритмичния анализ, функциите са с домейн именно $\mathbb{N}^+$, понеже големината на входа е цяло положително число.

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, $n^2 \asymp n^2 + n$, $2^n \succ n$, $n2^n \prec 3^n$ и ако ги разглеждаме върху $\mathbb{R}^+$, и ако ги разглеждаме върху $\mathbb{N}^+$.

Дори функцията да може да връща не-цяло число при целочислен аргумент, примерно $\frac{n}{5}$, $\sqrt{n}$ или $\log_2 n$, същото остава в сила, стига функцията да е "добре държаща се". Посочените функции са именно такива, така че, примерно, $\frac{n}{5} \succeq \sqrt{n}$ 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. 95. 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.

Фигура 2.11 : Графиките на n и $n^{1+\sin 2\pi n}$.



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. 96. 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 [83], 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 5).

Knuth в [83] 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-O нотация, а с релацията $\preceq$, 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 $\preceq$ и $\succeq$ като преднаредби

Както видяхме в Лема 9, $\asymp$ е релация на еквивалентност, подобно на релацията на равенство върху числата. Да разгледаме $\preceq$ и $\succeq$. Съгласно Следствие 7, това са фундаменталните релации на асимптотични сравнения. Те не са антисиметрични, защото може за различни функции $f(n)$ и $g(n)$ да е изпълнено $f(n) \preceq g(n)$ и $g(n) \preceq f(n)$, примерно $n^2 \preceq n^2 + n$ и $n^2 + n \preceq n^2$. Следователно $\preceq$ и $\succeq$ не са релации на частична наредба, а са по-общите **релации на преднаредба**. Освен това, както вече видяхме в доказателството на Теорема 12(2.39), $\preceq$ и $\succeq$ не са пълни, защото може за различни функции $f(n)$ и $g(n)$ да не е изпълнено нито $f(n) \preceq g(n)$, нито $g(n) \preceq f(n)$. Следният допълнителен материал съдържа достатъчно, за да бъде осмислено това.

Допълнение 17: 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 $\preceq$ и $\succeq$. Да си припомним някои дефиниции за релациите по принцип. Сравнете това изложение с изложението в [118, стр. 49–50].

Определение 31: 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.

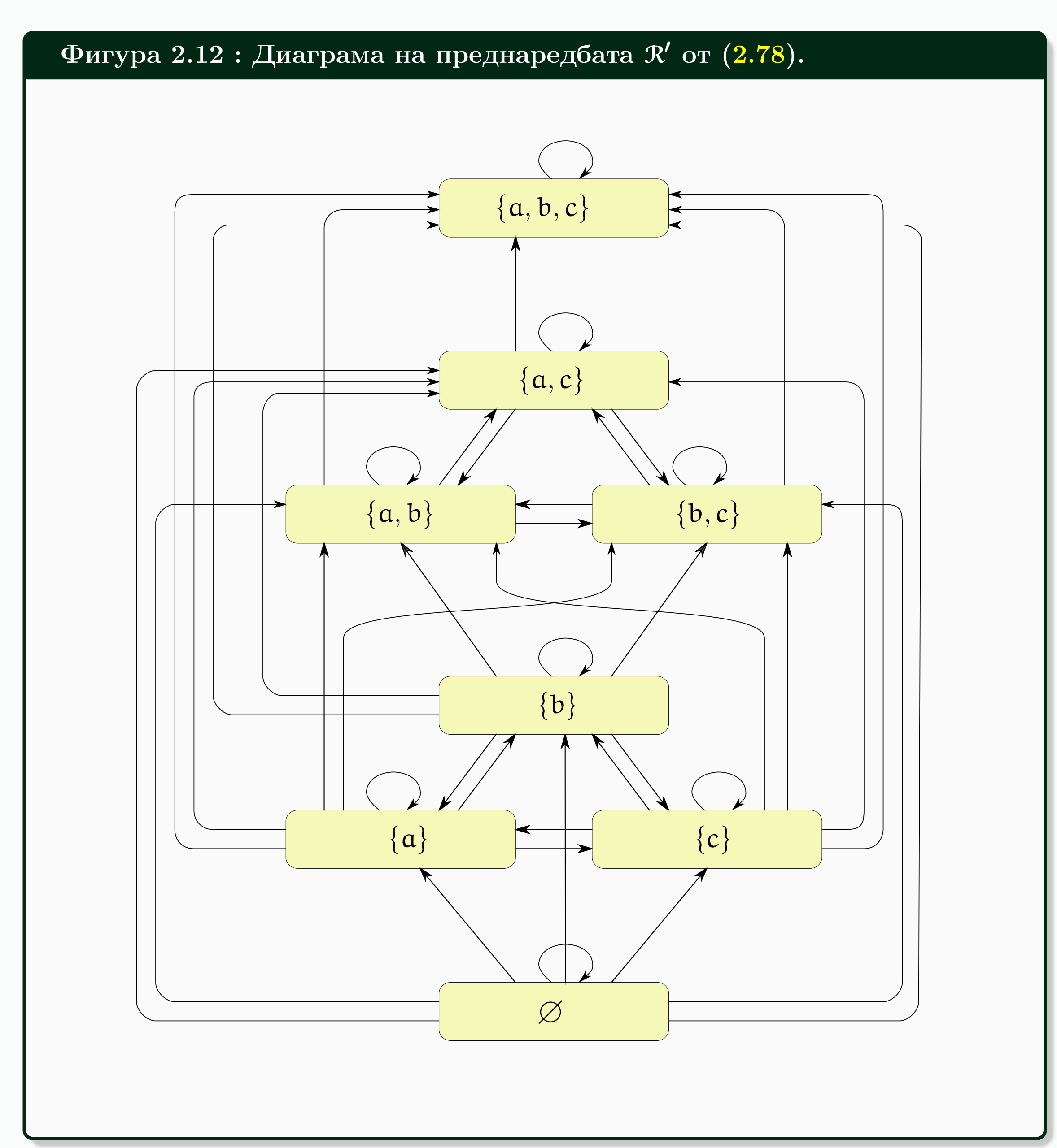
[a] На английски, *preorder* или *quasi-oder*.

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:

$$\forall X \in 2^S \; \forall Y \in 2^S : (X, Y) \in \mathcal{R}' \leftrightarrow |X| \leqslant |Y| \tag{2.78}$$

Очевидно $\mathcal{R}'$ е рефлексивна и транзитивна – това следва от свойствата на релацията $\leqslant$ 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 (2.78).



Следното определение и лема са Lemma 1 в книгата на Birkhoff [19, стр. 21].

Определение 32: 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 $\mathfrak{X}$ е множеството от класовете на еквивалентност на $\simeq$. *Фактор-релацията спрямо* R[б] е релацията $\mathfrak{S} \subseteq \mathfrak{X} \times \mathfrak{X}$, дефинирана така:

$$\forall P, Q \in \mathfrak{X} : (P, Q) \in \mathfrak{S} \leftrightarrow \exists a \in P \, \exists b \in Q : aRb$$

[a] Оригиналният символ е $\sim$, а не $\simeq$. Тук ползваме $\simeq$, за да няма объркване с релацията за асимптотична еквивалентност от Подсекция 2.3.12.

[б] В книгата на Birkhoff се използва терминът *quotient set*.

Лема 15

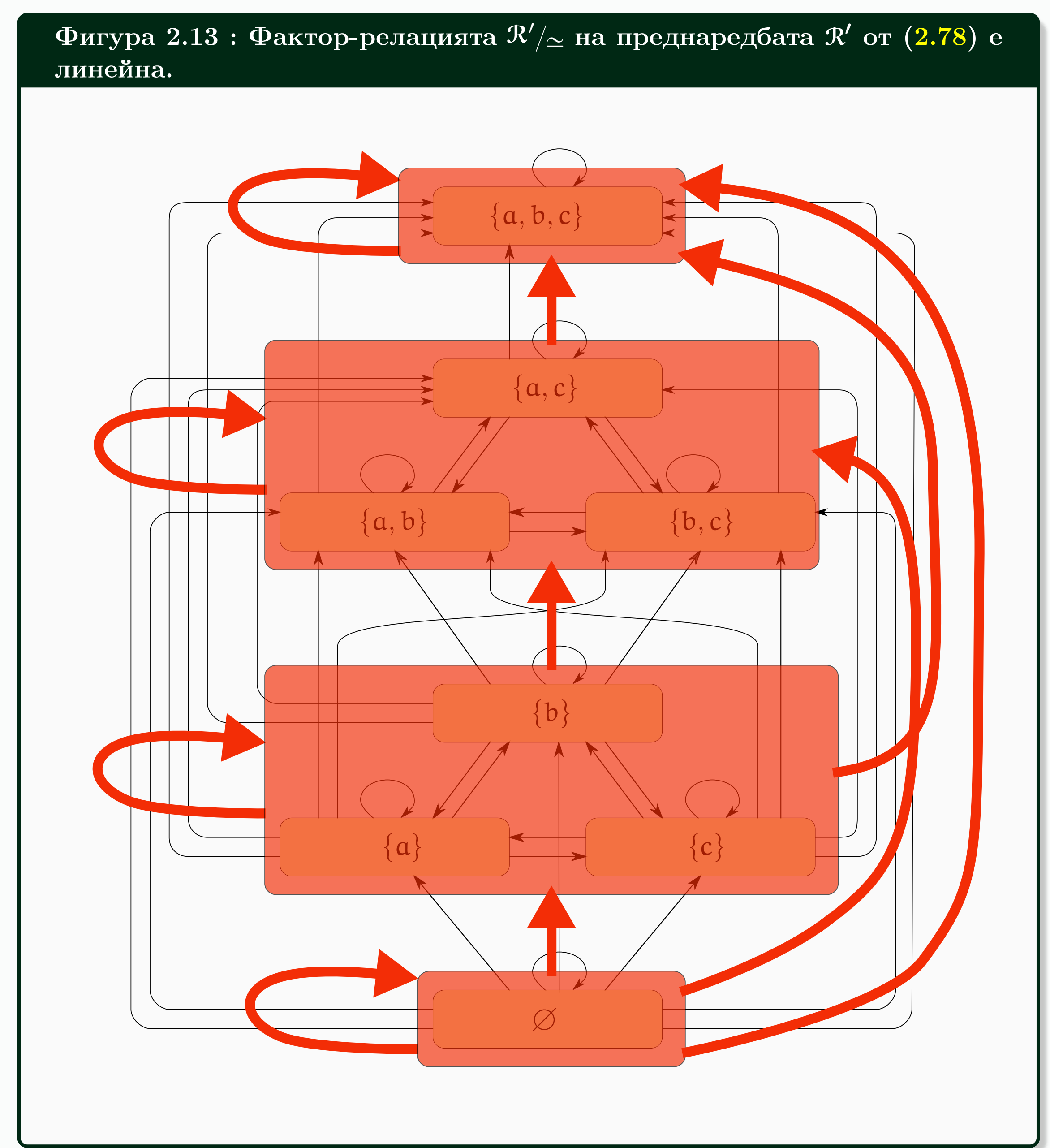
Фактор-релацията е релация на частична наредба.

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 (2.78) 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 (2.78) е линейна.



Пример за преднаредба, която не е пълна. Нека B е следното множество от триелементни редици от естествени числа:

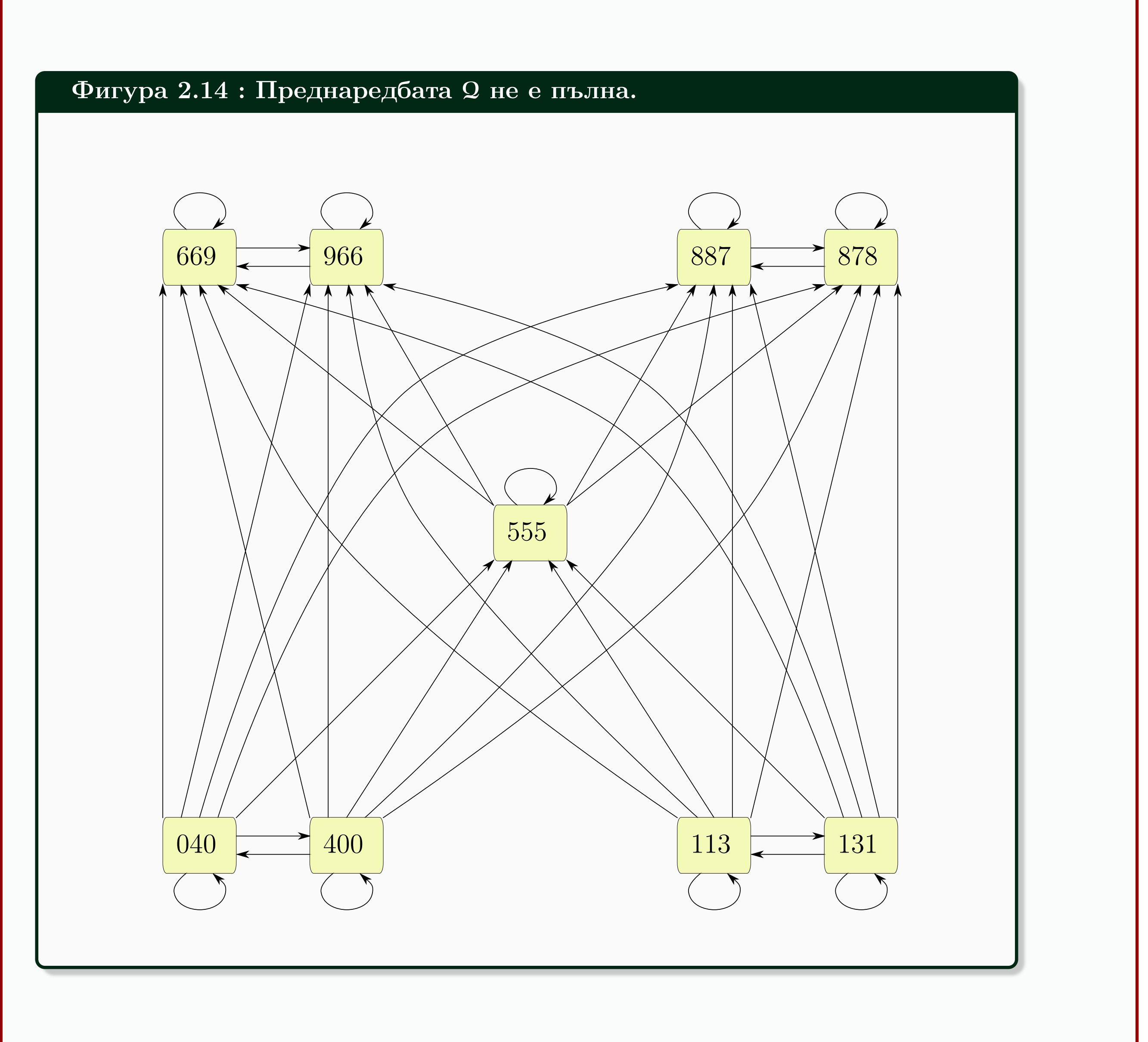
$$B = \{040, 400, 113, 131, 555, 779, 977, 886, 868\}$$

Нека $\mathcal{Q}$ е следната релация над B:

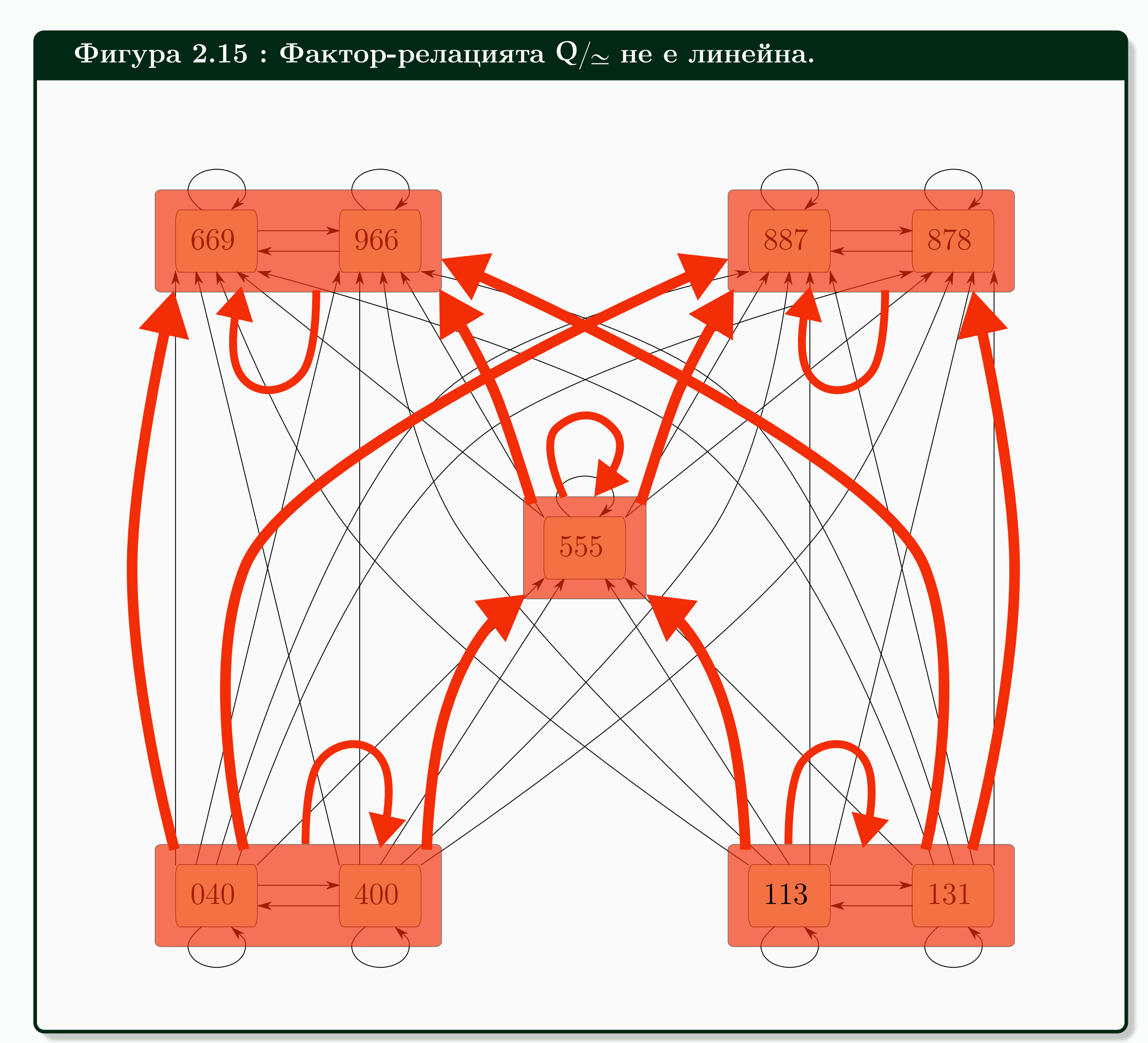
$$\forall x, y \in B : (x, y) \in \mathcal{Q} \quad \leftrightarrow \quad \sum x = \sum y \text{ или } (x_i \leqslant y_i, \text{ за } i \in \{1, 2, 3\})$$

където $\sum x$ означава сумата от елементите на редицата x (аналогично и за y), а x_i е i-ият елемент на x (аналогично и за y). Диаграмата на $\mathcal{Q}$ е показана на Фигура 2.14. Фигура 2.15 показва фактор-релацията.

Фигура 2.14 : Преднаредбата Ω не е пълна.



Фигура 2.15 : Фактор-релацията $Q/_{\simeq}$ не е линейна.



Наблюдение 15

Релациите $\prec$ и $\succ$ са ирефлексивни и транзитивни, а $\preceq$ и $\succeq$ са рефлексивни и транзитивни. Тогава $\prec$ и $\succ$ са строги преднаредби, а $\preceq$ и $\succeq$ са преднаредби върху множеството на функциите.
Преднаредбите $\preceq$ и $\succeq$ не са антисиметрични.
Преднаредбите $\preceq$ и $\succeq$ не са пълни.
Тъй като $\asymp$ е релация на еквивалентност, имаме право да говорим за фактор-релациите $\preceq/\asymp$ и $\succeq/\asymp$. 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 $\preceq/\asymp$ и $\succeq/\asymp$ са "истински" частични наредби, в които има несравними елементи.
Ако ограничим домейна на релациите до ненамаляващите положителни функции, $\preceq/\asymp$ има минимален елемент, който е множеството от константите (положителните реални числа). Максимален елемент, естествено, няма – винаги има по-бързо растяща функция. Дуално, $\succeq/\asymp$ 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 33 е обобщение на Определение 17 за функции на две променливи.

Определение 33: $\Theta(g(n, m))$

За всяка функция $g(n, m)$:

$$\Theta(g(n,m)) \stackrel{\text{def}}{=} \big\{f(n,m) \mid \exists c_1, c_2 > 0\ \exists n_0, m_0\ \forall n \geqslant n_0\ \forall m \geqslant m_0 : \\ 0 \leqslant c_1 \cdot g(n,m) \leqslant f(n,m) \leqslant c_2 \cdot g(n,m)\big\}$$

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 34: Релация $\sim$ над множеството на функциите

За всички функции $f(n)$ и $g(n)$, $f(n) \sim g(n)$ тогава и само тогава, когато $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

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 12. От него следва, че $\sim$ е по-“силна” от $\asymp$ в смисъл, че:

$$f(n) \sim g(n) \ \rightarrow f(n) \asymp g(n)$$
$$\neg\big(\, f(n) \asymp g(n) \ \rightarrow f(n) \sim g(n)\,\big)$$

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 17 (вижте Определение 32) 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 (2.7) 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 $\boxed{s \leftarrow s + A[i]}$. 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** $i \leftarrow c_1$ **to** $f(n)$
>   *«тяло-на-цикъла»*
> *«завършване»*

където и *«инициализация»*, и *«тяло-на-цикъла»*, 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-цикъла»* 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** $i \leftarrow c_1$ **to** $f(n)$
>   **for** $j \leftarrow c_2$ **to** $g(n)$
>   *«тяло-на-цикъла»*
> *«завършване»*

където и *«инициализация»*, и *«тяло-на-цикъла»*, и *«завършване»* имат сложност $\Theta(1)$, а c_1 и c_2 са някакви константи. С разсъждения, напълно аналогични на тези 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-цикъла»* 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-цикъла»
«завършване»
```

където и *«инициализация»*, и *«тяло-на-цикъла»*, 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-цикъла»* 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. 80, 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 (2.11) – 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 ← s + 1
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 16

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,

$$\sum_{i=t}^{n} 1 = \begin{cases} n-t+1, & \text{ако } t \leqslant n \\ 0, & \text{в противен случай} \end{cases}$$

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,

$$\sum_{k=i+j-1}^{n} 1 = \begin{cases} n-i-j+2, & \text{ако } i+j-1 \leqslant n, \text{ тоест } j \leqslant n-i+1 \\ 0, & \text{в противен случай} \end{cases}$$

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 $i > \left\lfloor \frac{n}{2} \right\rfloor$. Следователно,

максималната стойност на i, която трябва да разгледаме, е $\left\lfloor \frac{n}{2} \right\rfloor$:

$$
\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) - \\
&\quad \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) - \\
&\quad \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 - \\
&\quad \frac{1}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left((n+1)(n+2) - i(2n+3) + i^2 - i^2 - i \right) = \\
&(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 - \\
&\quad \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{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} + 2 \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{6} - \\
&\quad \frac{1}{2} \left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) + (n+2) \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} = \\
&\frac{\lfloor \frac{n}{2} \rfloor (n+1)(n+2)}{2} - \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)(2n+3)}{2} + \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{3}
\end{aligned}
$$

Ако n е четно, тоест $n = 2k$ за някое $k \in \mathbb{N}^+$, то $\left\lfloor \frac{n}{2} \right\rfloor = k$, 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 $\left\lfloor \frac{n}{2} \right\rfloor = k$, така че

$$s = \frac{k(2k+2)(2k+3)}{2} - \frac{k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$\frac{k(k+1)(4k+6) - k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$k(k+1)\left(\frac{1}{2} + \frac{2k+1}{3}\right) = \frac{k(k+1)(4k+5)}{6}$$

□

От Лема 16 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.

```
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 ← s + 1
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 $n + i + j - 3 \leqslant n$, тоест, $i + j \leqslant 3$, тоест,

- $i = 1$ и $j = 1, 2$;
- $i = 2$ и $j = 1$.

И така, AlgY връща $s = 3$.

Дали обаче върнатата стойност дава асимптотиката на сложността по време? В примерите, които разгледахме 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**-цикъл на редове 3–5. Ерго, изпълнението на AlgY не е по-бързо от изпълнението на следния алгоритъм.

```
AlgZ(A[1..n]: масив от цели числа)
1  for i ← 1 to n
```

2 for $j \leftarrow i$ to n
3 *прави нещо в констнантно 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 алгоритъма за двоично търсене в сортиран масив в итеративния вариант, който наричаме BinSearchIter и чийто псевдокод е на стр. 239. Добре известно е, че сложността му в най-лошия случай е логаритмична и това може да се покаже полуформално 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 17

Сложността по време на BinSearchIter на стр. 239 е $O(\lg n)$.

Доказателство: Следното твърдение е инвариант за **while** цикъла (редове 5–12).

Инвариант 2: Цикълът на BinSearchIter

При достигане номер t на ред 5 е вярно, че:

$$h - \ell + 1 \leqslant \frac{n}{2^{t-1}} \qquad (2.79)$$

База. Нека $t = 1$. Тогава $\ell = 1$ и $h = n$ заради редове 3 и 4. В сила е

$$n - 1 + 1 \leqslant \frac{n}{2^{1-1}} \leftrightarrow n \leqslant \frac{n}{1} \qquad \checkmark$$

Поддръжка. Нека изпълнението е на ред 5 и предстои поне още едно изпълнение на **while**-цикъла. Игнорираме възможността $A[mid] = key$, защото, ако това е вярно, изпълнението стига до ред 8 и спира, поради което ред 5 няма да бъде достигнат повече. И така, разглеждаме само следните два случая:

- key $< A[\text{mid}]$. Тогава изпълнението достига ред 10, 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 5. Трябва да покажем, че по отношение на новите ℓ и h (новото ℓ е старото ℓ, но така се казва), неравенство (2.79) остава в сила. Започваме от индуктивното предположение (2.79) за старите стойности:

$$
\begin{aligned}
h - \ell + 1 &\leqslant \frac{n}{2^{t-1}} \qquad // \text{ делим на } 2\\
\frac{h-\ell}{2} + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
\frac{h+\ell-2\ell}{2} + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
\frac{h+\ell}{2} - \ell + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
\frac{h+\ell}{2} - \ell + \frac{1}{2} - 1 + 1 &\leqslant \frac{n}{2^t}\\
\frac{h+\ell}{2} - 1 - \ell + \frac{1}{2} + 1 &\leqslant \frac{n}{2^t}\\
\frac{h+\ell}{2} - 1 - \ell + 1 &\leqslant \frac{n}{2^t} \qquad // \text{ понеже } \frac{1}{2} > 0\\
\underbrace{\left\lfloor \frac{h+\ell}{2} \right\rfloor - 1}_{\text{новото } h} - \ell + 1 &\leqslant \frac{n}{2^t} \qquad // \text{ понеже } \lfloor m \rfloor \leqslant m,\ \forall m \in \mathbb{R}^+
\end{aligned}
$$

- key $> A[\text{mid}]$. Тогава изпълнението достига ред 12, 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 5. Трябва да покажем, че по отношение на новите ℓ и h (новото h е старото h), неравенство (2.79) остава в сила. Започваме от индуктивното предположение (2.79) за старите стойности:

$$
\begin{aligned}
h - \ell + 1 &\leqslant \frac{n}{2^{t-1}} \qquad // \text{ делим на } 2\\
\frac{h-\ell}{2} + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
\frac{2h-h-\ell}{2} + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
h - \frac{h+\ell}{2} + \frac{1}{2} &\leqslant \frac{n}{2^t}\\
h - \frac{\ell+h}{2} + \frac{1}{2} - 1 + 1 &\leqslant \frac{n}{2^t}\\
h - \frac{\ell+h}{2} - \frac{1}{2} + 1 &\leqslant \frac{n}{2^t}\\
h - \left(\frac{\ell+h}{2} + \frac{1}{2}\right) + 1 &\leqslant \frac{n}{2^t}\\
h - \underbrace{\left(\left\lfloor \frac{\ell+h}{2} \right\rfloor + 1\right)}_{\text{новото } \ell} + 1 &\leqslant \frac{n}{2^t} \qquad // \text{ понеже } \left\lfloor \frac{\ell+h}{2} \right\rfloor + 1 \geqslant \frac{\ell+h}{2} + \frac{1}{2},\ \forall \ell, h \in \mathbb{N}^+
\end{aligned}
$$

Доказахме инварианта (2.79). Да наречем “$t_{\max}$” стойността на t при достигането на ред 5,

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 инварианта (2.79), получаваме

$$0 + 1 \leqslant \frac{n}{2^{t_{max}-1}} \leftrightarrow 2^{t_{max}} \leqslant 2n \leftrightarrow t_{max} \leqslant 1 + \log_2 n$$

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 $O(\lg n)$. □

Да си припомним, че разглеждаме сложността по време в **най-лошия** случай.

Лема 18

Сложността по време на BinSearchIter на стр. 239 е $\Omega(\lg n)$.

Доказателство: Следното твърдение е инвариант за **while** цикъла (редове 5–12).

Инвариант 3: Цикълът на BinSearchIter

При достигане номер t на ред 5 е вярно, че:

$$\frac{n}{2^t} - 4 < h - \ell \tag{2.80}$$

База. Нека $t = 1$. Тогава $\ell = 1$ и $h = n$ заради редове 3 и 4. В сила е

$$\frac{n}{2^{1-1}} - 4 < n - 1 \leftrightarrow \frac{n}{1} - 4 < n - 1 \qquad \checkmark$$

Поддръжка. Нека изпълнението е на ред 5 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 8 и спира, поради което ред 5 няма да бъде достигнат повече. И така, разглеждаме само следните два случая:

- $\text{key} < A[\text{mid}]$. Тогава изпълнението достига ред 10, 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 5. Трябва да покажем, че по отношение на новите ℓ и h (новото ℓ е старото ℓ), неравенство (2.80) остава в сила. Започваме от индуктивното предположение (2.80) за старите стойности:

$$\frac{n}{2^t} - 4 < h - \ell \qquad // \text{ делим на } 2$$
$$\frac{n}{2^{t+1}} - 2 < \frac{h - \ell}{2}$$
$$\frac{n}{2^{t+1}} - 2 < \frac{h + \ell - 2\ell}{2}$$
$$\frac{n}{2^{t+1}} - 2 < \frac{h + \ell}{2} - \ell$$
$$\frac{n}{2^{t+1}} - 4 < \frac{h + \ell}{2} - 2 - \ell$$
$$\frac{n}{2^{t+1}} - 4 < \underbrace{\left\lfloor \frac{h + \ell}{2} \right\rfloor - 1}_{\text{новото } h} - \ell \qquad // \text{ понеже } m - 2 \leqslant \lfloor m \rfloor - 1,\ \forall m \in \mathbb{R}^+$$

[†]Ред 5 се достига още един път.

- $\mathrm{key} > A[\mathrm{mid}]$. Тогава изпълнението достига ред 12, където ℓ става $\mathrm{mid}+1$, 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 5. Трябва да покажем, че по отношение на новите ℓ и h (новото h е старото h), неравенство (2.80) остава в сила. Започваме от индуктивното предположение (2.80) за старите стойности:

$$
\begin{aligned}
&\frac{n}{2^t} - 4 < h - \ell \qquad // \text{ делим на } 2\\
&\frac{n}{2^{t+1}} - 2 < \frac{h-\ell}{2}\\
&\frac{n}{2^{t+1}} - 2 < \frac{2h-h-\ell}{2}\\
&\frac{n}{2^{t+1}} - 2 < h - \frac{h+\ell}{2}\\
&\frac{n}{2^{t+1}} - 4 < h - \frac{h+\ell}{2} - 2\\
&\frac{n}{2^{t+1}} - 4 < h - \left(\frac{h+\ell}{2} + 2\right)\\
&\frac{n}{2^{t+1}} - 4 < h - \underbrace{\left(\left\lfloor \frac{h+\ell}{2} \right\rfloor + 1\right)}_{\text{новото } \ell} \qquad // \text{ понеже } m + 2 \geqslant \lfloor m \rfloor + 1,\ \forall m \in \mathbb{R}^+
\end{aligned}
$$

Доказахме инварианта (2.80). Да наречем “t_{max}” стойността на t при достигането на ред 5, 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 (2.80), 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 25: Сложността на двоичното търсене е логаритмична

Сложността по време на BINSEARCHITER на стр. 239 в най-лошия случай е $\Theta(\lg n)$.

Доказателство: Следва веднага от Лема 17 и Лема 18.

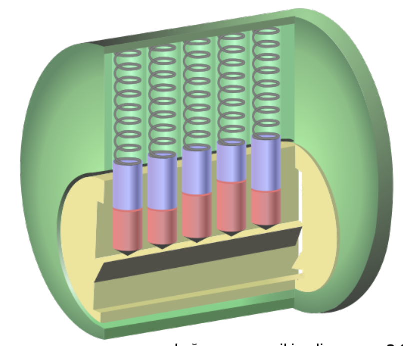
2.5 Винаги ли е лоша високата сложност

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 алторитми—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 проклятие.

[†] Ред 5 се достига още един път.

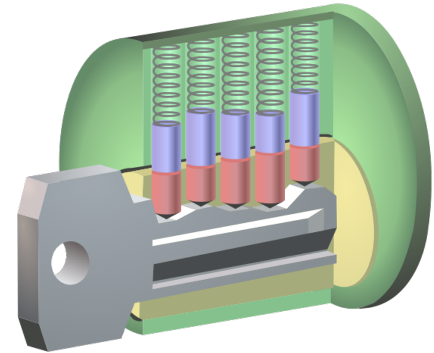
Има обаче друга гледна точка, от която високата сложност е благословия. Това е гледната точка на сигурността. Първо да разгледаме съвсем обикновен пример: ключ от секретна брава като този

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 щифтовене пречат на цилиндъра да се върти.



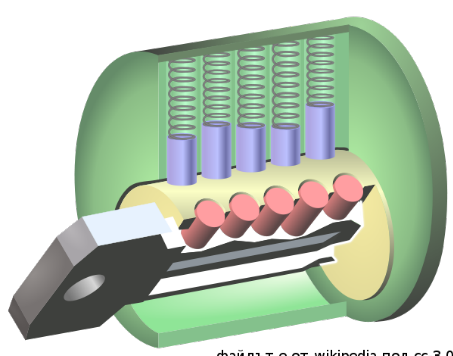
файлът е от wikipedia под cc-3.0

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



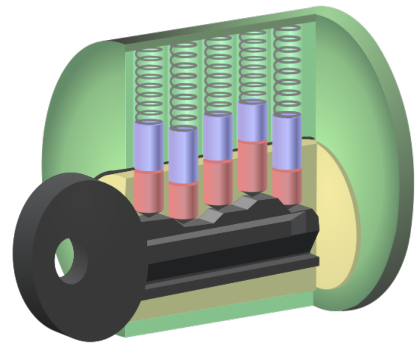
файлът е от wikipedia под cc-3.0

и вече нищо не пречи на цилиндъра да бъде завъртян, което води до издърпване на езика на бравата и вратата се отваря.



файлът е от wikipedia под cc-3.0

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



файлът е от wikipedia под cc-3.0

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 $\{1, 2, \ldots, m\}^n$. 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 A иска да прати съобщение на потребител B, тя взема публичния ключ на B, шифрира с него съобщението си и го праща на B. На свой ред B, използвайки секретния си ключ, дешифрира съобщението. Съобщението може да се дешифрира само чрез секретния ключ на B. Подслушвачът вижда шифрирания вариант, но от него не научава нищо, защото не разполага със секретния ключ на B. 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 [47] и Impagliazzo [74]. Изключително детайлно въведение в криптографията е култовата книга Applied Cryptography на Bruce Schneier [126].

[†]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 Akra-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 "$T\left(\frac{n}{2}\right)$". 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:

$$
\begin{aligned}
&F_0 = 0 \\
&F_1 = 1 \\
&F_n = F_{n-1} + F_{n-2} \text{ за } n \geqslant 2
\end{aligned}
\tag{3.1}
$$

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 16).

Добре дефинирано рекурентно уравнение ще наричаме рекурентно уравнение, което задава коректен (рекурсивен) алгоритъм, който алгоритъм пресмята елементи на някаква безкрайната 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:

$$T(0) = 1$$
$$T(n) = 2T(n-3) \text{ за } n \geqslant 1$$

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, ...

$$T(0) = 0$$
$$T(1) = 1$$
$$T(2) = 1$$
$$T(n) = 2T(n-1) - T(n-3) \quad \text{за } n \geqslant 3 \tag{3.2}$$

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 $n > 1$.

коефициент може да се дефинира чрез рекурентното уравнение на две променливи

$$\binom{n}{k} = \begin{cases} 0, & \text{ако } k > n \\ 1, & \text{ако } k = 0 \text{ или } k = n \\ \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{в противен случай} \end{cases}$$

или числото на Stirling от втори род може да се дефинира чрез рекурентното уравнение на две променливи

$$\left\{ {n \atop k} \right\} = \begin{cases} 0, & \text{ако } k > n \\ 0, & \text{ако } k = 0 \text{ и } n > 0 \\ 1, & \text{ако } k = n \\ \left\{ {n \atop k} \right\} = k \cdot \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}, & \text{в противен случай} \end{cases}$$

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.

Може да бъдат дадени няколко взаимно зависими рекурентни уравнения – това е проявление на *взаимна рекурсия*. Пример за такива рекурентни уравнения има в [88, 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 списания за числа на Fibonacci, където е дадена задачата, да се докаже, че H_{n-1} и G_n са последователни числа на Fibonacci $\forall n \geqslant 1$, ако F_n са числата на Fibonacci:

$$\begin{aligned} G_1 &= 1 \\ H_0 &= 0 \\ G_n &= F_{n+1} G_{n-1} + F_n H_{n-2}, \quad n \geqslant 2 \\ H_n &= F_{n+1} G_n + F_n H_{n-1}, \quad n \geqslant 1 \end{aligned}$$

3.1.3 Рекурентни уравнения и рекурсивни алгоритми

Връзката между рекурентните уравнения и рекурсивните алгоритми е трикова. От една страна, всяко рекурентно уравнение **<u>е</u>** рекурсивен алгоритъм, който за всеки достатъчно голям вход-число връща изход-число след пресмятане, което включва рекурсия (викане на себе си върху по-малко число). 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 рекурентното уравнение, където $n \in \mathbb{N}$:

$$T(n) = \begin{cases} 1, & \text{ако } n = 0 \\ nT(n-1) + 1, & \text{в противен случай} \end{cases} \tag{3.3}$$

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 16

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

$$T(n) = 2T(n-1) + 1 \tag{3.4}$$

а може да се появява много пъти с различни стойности на аргумента си, например

$$T(n) = T(n-1) + T(n-3) + T(n-5) \tag{3.5}$$

[†]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

$$T(n) = T(n-1) + T(n-2) + \cdots + T(1) + T(0), \tag{3.6}$$

което се нарича *рекурентно уравнение с пълна история.*

Наблюдение 17

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, $T(n) = T(n - \lfloor \lg n \rfloor)$. Екстремен 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 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. Намаляване чрез изваждане се среща често (макар и не само там) в рекурентните уравнения, описващи сложността на алгоритми, изградени по схемата **Пълно изчерпване**.

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 [39].

- *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

  $$T(n) = T(n-1) + T(n-2) + \cdots + T(n-k) + n^2$$

  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 18: 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, $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$ може да се запише като $T(n) = T\left(n - \left\lceil \frac{n}{2} \right\rceil\right) + 1$; $S(n) = S(\lfloor \sqrt{n} \rfloor) + 1$ 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, в което дясната страна е линейна функция на функцията на уравнението. Например, следните три рекурентни уравнения са линейни:

$$A(n) = A(n-1) + 1$$
$$B(n) = nB(n-2) + n^2B(n-4) + \left\lfloor \frac{n \sin n}{\sqrt{n^3 + \lg n}} \right\rfloor$$
$$C(n) = \sum_{k=0}^{n-1} 2^k C(k)$$

докато тези четири **<u>не са</u>** линейни:

$$A(n) = A(n-1)A(n-2) + 1$$
$$B(n) = \left\lceil \frac{1}{1 + B(n-1)} \right\rceil$$
$$C(n) = \left\lfloor \sqrt{C(n-2)} \right\rfloor$$
$$D(n) = \lfloor \lg (D(n-1) + D(n-2)) \rfloor$$

- Различаваме рекурентни уравнения с константни коефициенти, например

$$P(n) = 1P(n-1)P(n-2) + 3P(n-3)$$
$$S(n) = \left\lfloor \sum_{k=1}^{\infty} \frac{1}{k^2} \right\rfloor S(n-2) + n$$
$$T(n) = 2T(n-1) + 5T(n-3)$$

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,

$$T(n) = c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) \tag{3.7}$$

където k и c_1, ..., c_k са константи. При нехомогенните има събираеми, които са от съвсем друг вид, които не съдържат функцията от лявата страна. Примерно,

$$T(n) = c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) + f(n) \tag{3.8}$$

където k и c_1, ..., 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. [130, стр. 56])

$$a_0 = 1$$
$$a_n = \frac{1}{1 + a_{n-1}}$$

определя редицата от дроби 1, $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{5}$, $\frac{5}{8}$, ..., 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. [130, 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

$$F_n = \begin{cases} n, & \text{ако } n \in [0, 2) \\ F_{n-1} + F_{n-2}, & \text{ако } n > 2 \end{cases} \tag{3.9}$$

Математически погледнато, този алгоритъм е безукорно ефективен. Нещо повече, тъй като реалната променлива намалява с вадене на цели числа, алгоритъмът на (3.9) може да бъде имплементиран ефикасно по схемата **Динамично Програмиране**, подобно на изчислението на числа на Fibonacci в Подсекция 12.1.2. При други видове намалявания вдясно, примерно деление на константа или кореннуване, идеята на динамичното програмиране не е приложима и освен това съществува опасност от загуба на точност при практическата имплементация, ако работим с числа с плаваща запетая, но, математически погледнато, алгоритъмът винаги е безукорно ефективен, ако интервалът-начално условие е удачно подбран.

Наблюдение 19

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.

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

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{3.10}$$

  описващо сложността по време на MERGESORT (Секция 6.3), може да бъде написано по-прецизно като

$$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) + n \tag{3.11}$$

  Това е смислено, защото масив с нечетна големина не може да бъде разделен на две равни части; най-доброто, на което можем да се надяваме, е едната от тези части да е с не повече от единица по-малка от другата, което функциите $\lfloor\rfloor$ и $\lceil\rceil$ ни гарантират. Но този подход е тромав и на практика, ако $\lfloor\rfloor$ и $\lceil\rceil$ се срещат в рекурентно уравнение, което искаме да решим, първата ни работа е да се освободим от тези функции, апроксимирайки решението с друго, подходящо рекурентно уравнение без тях.

- Да допуснем, че променливата може да е дробно число. Това налага началните условия да са върху цял интервал, а не само върху няколко дискретни точки, но това е без значение. Ние не посочваме началните условия така или иначе, защото, за целите на анализа на алгоритъма, се интересуваме само от асимптотиката на решението на рекурентното уравнение. Добре известно е, че за рекурентни уравнения, които типично се появяват при изследването на сложността по време на рекурсивни алгоритми, асимптотиката на решението за уравнението с целочислена променлива и за уравнението с дробна променлива е една и съща. В примера с MERGESORT, уравнение (3.10) има решение с асимптотика $\Theta(n \lg n)$ и уравнение (3.11) има решение с асимптотика $\Theta(n \lg n)$.

Според [90], 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 вляво 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 23). Какво точно е затворена формула зависи от това, какви операции са разрешени. Решението "$\frac{n(n+1)}{2}$" е валидно, ако са разрешени 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, кореннуване и степенуване, то уравнението (3.13), колкото и да е странно, няма решение. При изброените ограничения, $T(n) = \prod_{i=1}^{n} i$ или $T(n) = n \cdot (n-1) \cdots 2 \cdot 1$ 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.

Ползите 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

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \tag{3.14}$$

[†]Един милиард за движението надолу до достигане на началното условие и още един милиард за движението нагоре, от началното условие до $T(1\,000\,000\,000)$.

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 канцеliranията, 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 някакви математически гаранции за не-загуба на точност.

- 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 уравнение ще разискваме в детайли в Секция 3.2. Тук само ще споменем, че универсален **формален** метод за решаване на рекурентни уравнения **няма**.

3.1.7 Рекурентни уравнения в изследването на алгоритми

Рекурентните уравнения се използват широко в теорията на алгоритмите, защото сложността на даден рекурсивен алгоритъм се описва с подходящо рекурентно уравнение. Правилото за съставяне на подходящо рекурентно уравнение е просто: лявата страна е, да речем, $T(n)$ и изразява работата на алгоритъма върху вход с размер n, а дясната страна има по едно T за всяко рекурсивно викане, като аргументът е функцията, с която намалява входа при даденото рекурсивно викане, плюс израз, който отразява работата на алгоритъма извън рекурсивните викания. Ще видим достатъчно примери за конструиране на рекурентни уравнения, които описват сложността на рекурсивни алгоритми.

При изследването на алгоритмите не се интересуваме от точен израз за сложността – както се каза в минала лекция, това не е нито възможно, нито е необходимо. Интересува ни асимптотиката. Съответно няма да се опитваме да решаваме рекурентните уравнения точно, а само ще търсим асимптотиката на решението. Основно наблюдение, което оставяме без доказателство е, че асимптотиката на решението не зависи от началните условия. Разбира се, има примери за обратното – асимптотиката на

$$T(n) = T(n-1)^{T(n-2)}$$

зависи от началните условия. Но никой естествен алгоритъм няма сложност по време, която се описва от това уравнение. Алгоритмите, които ще разглеждаме в този курс, са такива, че съотвентите им рекурентни уравнения **не зависят** от началните стойности в смисъл, че асимптотиката е една и съща, каквито и да са началните стойности. Поради това отсега нататък ще изпускаме началните условия от описанията на рекурентните уравнения. Под "рекурентно уравнение" ще разбираме само "същината", игнорирайки началните условия.

Тъй като се интересуваме само от асимптотиката на решението, при дискретни рекурентни уравнения, чиято дясна страна съдържа функциите $\lfloor\,\rfloor$ и $\lceil\,\rceil$, ще игнорираме тези функции, по този начин преминавайки към континуални рекурентни уравнения. Това не променя асимптотиката на решението. Примерно, както вече отбелязахме в Подсекция 3.1.5, ако ни е дадено

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

ще решим наместо него това уравнение

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

чието решение има същата асимптотика.

И още едно опростяване, което не променя асимптотиката на функцията, което ще покажем с пример. Ако последното рекурентно уравнение описва работата на алгоритъм, който прави две рекурсивни викания и извършва освен това линейна работа, събираемото, които отразява тази линейна работа, би трябвало да е "$\Theta(n)$", така че рекурентното уравнение би трябвало да е:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Но асимптотиката на решението не се променя, ако наместо "$\Theta(n)$" напишем просто "n", тоест най-простия за записване представител на класа от функции $\Theta(n)$. И така, записът, който ще ползваме, е:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

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 23

Решете рекурентното уравнение

$$\begin{aligned} &T(0) = 0 \\ &T(n) = T(n-1) + n \text{ за } n > 0 \end{aligned} \tag{3.15}$$

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 (3.15), с "$T(n-2) + n - 1$", получаваме

$$T(n) = T(n-2) + (n-1) + n \tag{3.16}$$

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 (3.16), получаваме

$$T(n) = T(n-3) + (n-2) + (n-1) + n \tag{3.17}$$

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

$$T(n) = T(0) + 1 + 2 + \cdots + (n-2) + (n-1) + n \tag{3.18}$$

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 нарасват с единица.

Сега прилагаме два известни факта. Първо, че $T(0) = 0$, което знаем от началното условие, и второ, че $1 + 2 + \cdots n = \frac{n(n+1)}{2}$. Заествайки в (3.18), 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 (3.17) към (3.18) 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), теорема на Akra-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 Fibonacci (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 19

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 $a \leqslant b - k$,

такъв че за всяко достатъчно голямо n съществува $c \in [a, b]$, такова че итерирането

$$n \mapsto n - k \mapsto n - 2k \mapsto n - 3k \mapsto \cdots$$

завършва в c. С други думи, че итерирането е

$$\underbrace{n \mapsto n - k \mapsto n - 2k \mapsto n - 3k \mapsto \cdots \mapsto n - (m-1)k}_{\text{извън } [a,b]} \mapsto \underbrace{n - mk}_{\text{в } [a,b]}$$

за някое m, такова че $n - mk \leqslant b$, 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

$$n \leqslant km + b$$
$$n \geqslant km - k + b$$

така че $m \asymp n$. □

Следствие 12

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

$$n \mapsto \frac{n}{k} \mapsto \frac{n}{k^2} \mapsto \frac{n}{k^3} \mapsto \cdots \mapsto c$$

където c е елемент от предварително избран, подходящ за k, реален интервал, какъвто беше $[a, b]$ в доказателството на Лема 19. Нека $\ell = \log_k n$. Заместваме n с k^ℓ. Тогава итерирането става

$$k^\ell \mapsto \frac{k^\ell}{k} \mapsto \frac{k^\ell}{k^2} \mapsto \frac{k^\ell}{k^3} \mapsto \cdots \mapsto c$$

което е същото като

$$k^\ell \mapsto k^{\ell-1} \mapsto k^{\ell-2} \mapsto k^{\ell-3} \mapsto \cdots \mapsto c$$

Сега прилагаме Лема 19 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

$$n \mapsto \sqrt[k]{n} \mapsto \sqrt[k]{\sqrt[k]{n}} \mapsto \sqrt[k]{\sqrt[k]{\sqrt[k]{n}}} \mapsto \cdots \mapsto c$$

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

$$k^{k^\ell} \mapsto \sqrt[k]{k^{k^\ell}} \mapsto \sqrt[k]{\sqrt[k]{k^{k^\ell}}} \mapsto \sqrt[k]{\sqrt[k]{\sqrt[k]{k^{k^\ell}}}} \mapsto \cdots \mapsto c$$

което е същото като

$$k^{k^\ell} \mapsto k^{k^{\ell-1}} \mapsto k^{k^{\ell-2}} \mapsto k^{k^{\ell-3}} \mapsto \cdots \mapsto c$$

Сега прилагаме Лема 19 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. □

Следствие 13

Лема 19 и Следствие 12 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 съвършени 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 24

Решете уравнението

$$T(n) = n^2 T\left(\frac{n}{2}\right) + 1$$

чрез развиване.

Решение: В сила е

$$
\begin{aligned}
T(n) &= n^2\, T\left(\frac{n}{2}\right) + 1 \\
&= n^2 \left(\frac{n^2}{4} T\left(\frac{n}{4}\right) + 1\right) + 1 \\
&= \frac{n^4}{2^2} T\left(\frac{n}{2^2}\right) + n^2 + 1 \\
&= \frac{n^4}{2^2} \left(\frac{n^2}{2^4} T\left(\frac{n}{2^3}\right) + 1\right) + n^2 + 1 \\
&= \frac{n^6}{2^6} T\left(\frac{n}{2^3}\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^6}{2^6} \left(\frac{n^2}{2^6} T\left(\frac{n}{2^4}\right) + 1\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}} T\left(\frac{n}{2^4}\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}} \left(\frac{n^2}{2^8} T\left(\frac{n}{2^5}\right) + 1\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + \frac{n^2}{2^0} + \frac{n^0}{2^0} \\
&= \frac{n^{10}}{2^{5.4}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}} \\
&\dots \\
&= \underbrace{\frac{n^{2i}}{2^{i(i-1)}} T\left(\frac{n}{2^i}\right)}_{A} + \underbrace{\frac{n^{2(i-1)}}{2^{(i-1)(i-2)}} + \frac{n^{2(i-2)}}{2^{(i-2)(i-3)}} + \dots + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}}}_{\text{B: сума с } i \text{ събираеми}}
\end{aligned}
$$

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 12. 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 $T(1) = \Theta(1)$.

$$A = \frac{n^{2\lg n}}{2^{\lg^2 n - \lg n}} T(1) \asymp \frac{2^{\lg n} n^{2\lg n}}{2^{\lg^2 n}} = \frac{n.n^{2\lg n}}{2^{\lg^2 n}}$$

Но

$$2^{\lg^2 n} = 2^{\lg n . \lg n} = 2^{\lg\left(n^{\lg n}\right)} = n^{\lg n} \tag{3.19}$$

Тогава

$$A \asymp \frac{n.n^{2\lg n}}{n^{\lg n}} \asymp n^{1+\lg n} \tag{3.20}$$

Сега да разгледаме B. Очевидно, можем да го представим като сума по следния начин:

$$
\begin{aligned}
B &= \sum_{j=1}^{\lg n} \frac{n^{2((\lg n)-j)}}{2^{((\lg n)-j)((\lg n)-j-1)}} \\
&= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{n^{2\lg n}}{2^{(\lg^2 n - j\lg n - j\lg n + j^2 - \lg n + j)}} \\
&= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(2^{2j\lg n}\right)\left(2^{\lg n}\right)}{\left(2^{\lg^2 n}\right)\left(2^{j^2+j}\right)} \qquad (3.21)
\end{aligned}
$$

Но

$$2^{2j\lg n} = 2^{\lg\left(n^{2j}\right)} = n^{2j} \qquad (3.22)$$

и

$$2^{\lg n} = n \qquad (3.23)$$

Прилагаме (3.19), (3.22) и (3.23) върху (3.21) и получаваме

$$
\begin{aligned}
B &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(n^{2j}\right)(n)}{\left(n^{\lg n}\right)\left(2^{j^2+j}\right)} \\
&= \sum_{j=1}^{\lg n} \frac{n^{1+\lg n}}{2^{j^2+j}} \\
&= \left(n^{1+\lg n}\right) \sum_{j=1}^{\lg n} \frac{1}{2^{j^2+j}} \\
&\leqslant \left(n^{1+\lg n}\right) \sum_{j=1}^{\infty} \frac{1}{2^{j^2+j}} && \text{// тъй като знаем, че } \sum_{j=1}^{\infty} \frac{1}{2^j} = 1 \\
&\leqslant n^{1+\lg n} \qquad (3.24)
\end{aligned}
$$

От (3.20) и (3.24) следва, че

$$T(n) \asymp n^{1+\lg n}$$

Допълнение 18: Няколко рекурентни уравнения чрез развиване

Задача 25

Решете рекурентното уравнение

$$T(n) = T(n-2) + 2\lg n$$

чрез развиване.

Решение Да развием уравнението:

$$\begin{aligned}
T(n) &= T(n-2) + 2\lg n \\
&= T(n-4) + 2\lg(n-2) + 2\lg n \\
&= T(n-6) + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \\
&= \dots \\
&= \Theta(1) + \dots + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \\
&\asymp \lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \dots + \lg 3^2 + \lg 1^2 \\
&= \lg\left(n^2 \cdot (n-2)^2 \cdot (n-4)^2 \cdot \dots \cdot 3^2 \cdot 1\right) \\
&= \lg\big(\underbrace{n \cdot n \cdot (n-2) \cdot (n-2) \cdot (n-4) \cdot (n-4) \cdot \dots \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 1}_{n \text{ множителя}}\big)
\end{aligned}$$

Нека

$$\begin{aligned}
T(n) &= \lg\big(\underbrace{n \cdot n \cdot (n-2) \cdot (n-2) \cdot (n-4) \cdot (n-4) \cdot \dots \cdot 5 \cdot 5 \cdot 3 \cdot 3 \cdot 1}_{n \text{ множителя}}\big) \\
X(n) &= \lg\big(\underbrace{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1}_{n \text{ множителя}}\big) \\
Y(n) &= \lg\big(\underbrace{(n+1) \cdot n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 4 \cdot 3 \cdot 2}_{n \text{ множителя}}\big)
\end{aligned}$$

Очевидно

$$X(n) \leqslant T(n) \leqslant Y(n)$$

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, $T(n) \asymp n\lg n$. □

Задача 26

Решете уравнението

$$T(n) = \frac{n}{n+1}T(n-1) + 1$$

чрез развиване.

Решение В сила е

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1)+1\\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2)+1\right)+1\\
&= \frac{n-1}{n+1}T(n-2)+\frac{n}{n+1}+1\\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3)+1\right)+\frac{n}{n+1}+1\\
&= \frac{n-2}{n+1}T(n-3)+\frac{n-1}{n+1}+\frac{n}{n+1}+1\\
&= \frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4)+1\right)+\frac{n-1}{n+1}+\frac{n}{n+1}+1\\
&= \frac{n-3}{n+1}T(n-4)+\frac{n-2}{n+1}+\frac{n-1}{n+1}+\frac{n}{n+1}+1 \qquad (3.25)
\end{aligned}$$

Ако продължим така до $T(1)$, развиваме (3.25) до

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1)+\frac{3}{n+1}+\frac{4}{n+1}+\cdots+\frac{n-2}{n+1}+\frac{n-1}{n+1}+\frac{n}{n+1}+1\\
&= \frac{2}{n+1}T(1)+\frac{3}{n+1}+\frac{4}{n+1}+\cdots+\frac{n-2}{n+1}+\frac{n-1}{n+1}+\frac{n}{n+1}+\frac{n+1}{n+1}\\
&= \underbrace{\frac{\Theta(1)}{n+1}}_{O(1)}+\frac{1}{n+1}\sum_{i=3}^{n+1} i\\
&\asymp \frac{1}{n+1}\cdot\frac{(n-1)(n+4)}{2}\\
&\asymp \Theta(n)
\end{aligned}$$

И така, $T(n) \asymp n$. □

Задача 27

Решете рекурентното уравнение

$$T(n) = \frac{n}{n+1}T(n-1)+\lg n$$

чрез развиване.

Решение В сила е

$$
\begin{aligned}
&\frac{n}{n+1}T(n-1) + \lg n = \\
&\frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + \lg(n-1)\right) + \lg n = \\
&\frac{n-1}{n+1}T(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n = \\
&\frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + \lg(n-2)\right) + \frac{n}{n+1}\lg(n-1) + \lg n = \\
&\frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n = \\
&\quad \ldots = \\
&\underbrace{\frac{2}{n+1}T(1)}_{O(1)} + \frac{3}{n+1}\lg 2 + \frac{4}{n+1}\lg 3 + \cdots + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n \asymp \\
&\frac{(2+1)\lg 2}{n+1} + \frac{(3+1)\lg 3}{n+1} + \cdots + \frac{(n-1+1)\lg(n-1)}{n+1} + \frac{(n+1)\lg n}{n+1} = \\
&\frac{1}{n+1}\left(\sum_{k=2}^{n} k\lg k + \sum_{k=2}^{n}\lg k\right)
\end{aligned}
$$

Но $\sum_{k=2}^{n} k\lg k \asymp n^2 \lg n$ съгласно Теорема 22, а $\sum_{k=2}^{n} \lg k = \lg n! \asymp n\lg n$ съгласно Теорема 21, откъдето $\frac{1}{n+1}\left(\sum_{k=2}^{n} k\lg k + \sum_{k=2}^{n}\lg k\right) \asymp n\lg n$, така че $T(n) \asymp n\lg n$. □

Задача 28

Решете рекурентното уравнение

$$T(n) = nT(n-1) + 1$$

чрез развиване.

Решение

$$
\begin{aligned}
T(n) &= nT(n-1) + 1 \\
&= n((n-1)T(n-2) + 1) + 1 \\
&= n(n-1)T(n-2) + n + 1 \\
&= n(n-1)((n-2)T(n-3) + 1) + n + 1 \\
&= n(n-1)(n-2)T(n-3) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)((n-3)T(n-4) + 1) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2) + n(n-1) + n + 1 \\
&\quad \ldots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!}{(n-i+1)!} + \frac{n!}{(n-i+2)!} + \cdots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&\quad \ldots \\
&= \frac{n!}{1!}T(1) + \frac{n!}{2!} + \frac{n!}{3!} + \cdots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&= n! \times \underbrace{\left(\frac{\Theta(1)}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-1)!} + \frac{1}{n!}\right)}_{A}
\end{aligned}
$$

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, $T(n) \asymp n!$. □

Задача 29

Решете рекурентното уравнение

$$T(n) = nT(n-1) + 2^n$$

чрез развиване.

Решение

$$
\begin{aligned}
T(n) &= nT(n-1) + 2^n \\
&= n((n-1)T(n-2) + 2^{n-1}) + 2^n \\
&= n(n-1)T(n-2) + n2^{n-1} + 2^n \\
&= n(n-1)((n-2)T(n-3) + 2^{n-2}) + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)T(n-3) + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)((n-3)T(n-4) + 2^{n-3}) + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2)2^{n-3} + n(n-1)2^{n-2} + n2^{n-1} + 2^n \\
&\quad \ldots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!\,2^{n-i+1}}{(n-i+1)!} + \frac{n!\,2^{n-i+2}}{(n-i+2)!} + \cdots + \frac{n!\,2^{n-1}}{(n-1)!} + \frac{n!\,2^n}{n!} \\
&\quad \ldots \\
&= \frac{n!}{1!}T(1) + \frac{n!\,2^2}{2!} + \frac{n!\,2^3}{3!} + \cdots + \frac{n!2^{n-1}}{(n-1)!} + \frac{n!\,2^n}{n!} \\
&= n! \times \left(\Theta(1) + \sum_{k=2}^{n} \frac{2^k}{k!}\right)
\end{aligned}
$$

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 30

Решете рекурентното уравнение

$$T(n) = 2T(n-1) + n!$$

чрез развиване.

Решение

$$
\begin{aligned}
T(n) &= 2T(n-1) + n! \\
&= 2(2T(n-1) + (n-1)!) + n! \\
&= 4T(n-1) + 2(n-1)! + n! \\
&= 4(2T(n-2) + (n-2)!) + 2(n-1)! + n! \\
&= 8T(n-2) + 4(n-2)! + 2(n-1)! + n! && (3.26) \\
&= 8(2T(n-3) + (n-3)!) + 4(n-2)! + 2(n-1)! + n! \\
&= 16T(n-3) + 8(n-2)! + 4(n-2)! + 2(n-1)! + n! \\
&\quad \ldots && (3.27) \\
&= \underbrace{2^{n-1}T(1) + 2^{n-2}2! + 2^{n-3}3! + \cdots 2^2(n-2)! + 2(n-1)!}_{A} + n! && (3.28)
\end{aligned}
$$

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 20.

Лема 20

$\forall n \geqslant 2 : 2(n-1)! + 2^2(n-2)! + 2^3(n-3)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! < 2n!$.

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 $n \geqslant 2$ е вярно, че

$$2(n-1)! + 2^2(n-2)! + 2^3(n-3)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! < 2n!$$

Индуктивна стъпка: Ще покажем, че

$$2n! + 2^2(n-1)! + 2^3(n-2)! + \cdots + 2^{n-2}3! + 2^{n-1}2! + 2^n 1! < 2(n+1)!$$

Но това е същото като

$$2n! + 2\big(2(n-1)! + 2^2(n-2)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!\big) < 2 \cdot (n+1) \cdot n!$$

което е същото като

$$\begin{aligned}
&2n! + 2\big(2(n-1)! + 2^2(n-2)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!\big) < 2 \cdot n \cdot n! + 2n! \;\leftrightarrow \\
&2\big(2(n-1)! + 2^2(n-2)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1!\big) < 2 \cdot n \cdot n! \;\leftrightarrow \\
&2(n-1)! + 2^2(n-2)! + \cdots + 2^{n-3}3! + 2^{n-2}2! + 2^{n-1}1! < n \cdot n!
\end{aligned}$$

Това следва веднага от индуктивното предположение и допускането, че $n \geqslant 2$. □

Има алтернативно, по-просто и елегантно доказателство, посочено на автора на лекционните записки от Добромир Кралчев. Представяме израза от (3.28) така:

$$n!\left(1 + \underbrace{\frac{2}{n} + \frac{2^2}{n(n-1)} + \frac{2^3}{n(n-1)(n-2)} + \cdots + \frac{2^{n-2}}{n(n-1)\cdot \cdots \cdot 3} + \frac{2^{n-1}}{n(n-1)\cdot \cdots \cdot 2}}_{B}\right)$$

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:

$$\frac{1}{\frac{n}{2}} + \frac{1}{\frac{n}{2}\cdot\frac{n-1}{2}} + \underbrace{\frac{1}{\frac{n}{2}\cdot\frac{n-1}{2}\cdot\frac{n-2}{2}} + \cdots + \frac{1}{\frac{n}{2}\cdot\frac{n-1}{2}\cdot \cdots \cdot \frac{2}{2}}}_{C}$$

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 (3.28) 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 извикванията на рекурсивния алгоритъм, съобразяваме колко работа се извършва във всеки връх от дървото (тоест, във всяко индивидуално изпълнение на рекурсивния алгоритъм), и сумираме тези количества по нива в дървото. След което търсим закономерност в сумите по нива. Виждаме, че метода с дървото е подобен на метода с развиването по това, че не е формален и за да даде резултати, трябва човекът, който го ползва, да прояви достатъчно добра интуиция. Тук ще направим илюстрация с пример. Ще решим рекурентното уравнение

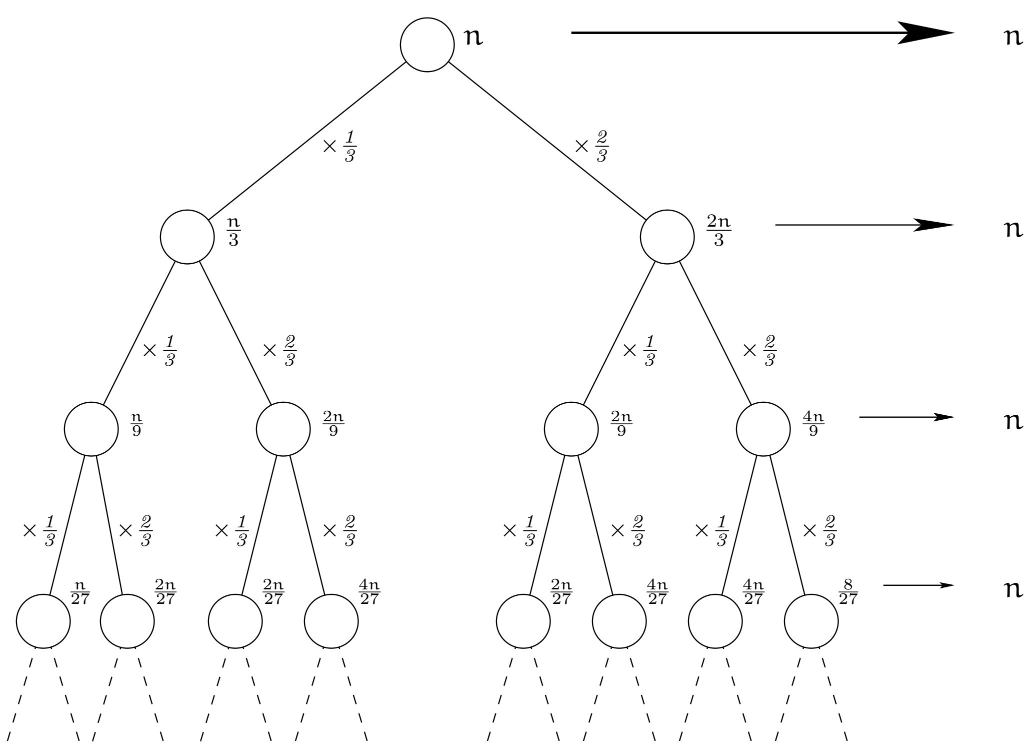
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \tag{3.29}$$

използвайки метода с дървото на рекурсията. Дървото на рекурсията—по-точно, само "горната" му част—е изобразено на Фигура 3.1. Всеки вътрешен връх представлява едно изпълнение на алгоритъма, което прави рекурсивни викания. Листата, които не са нарисувани, отговарят на изпълнения, които не правят рекурсивни викания; с други думи, листата са изпълненията, които задействат "спирачката" на рекурсията. Ребрата са маркирани с множителите, с които входът намалява. До всеки връх е написана работата, която алгоритъмът извършва в това индивидуално изпълнение извън рекурсивните викания. Върховете от едно и също ниво в дървото са нарисувани на едно и също ниво на фигурата.

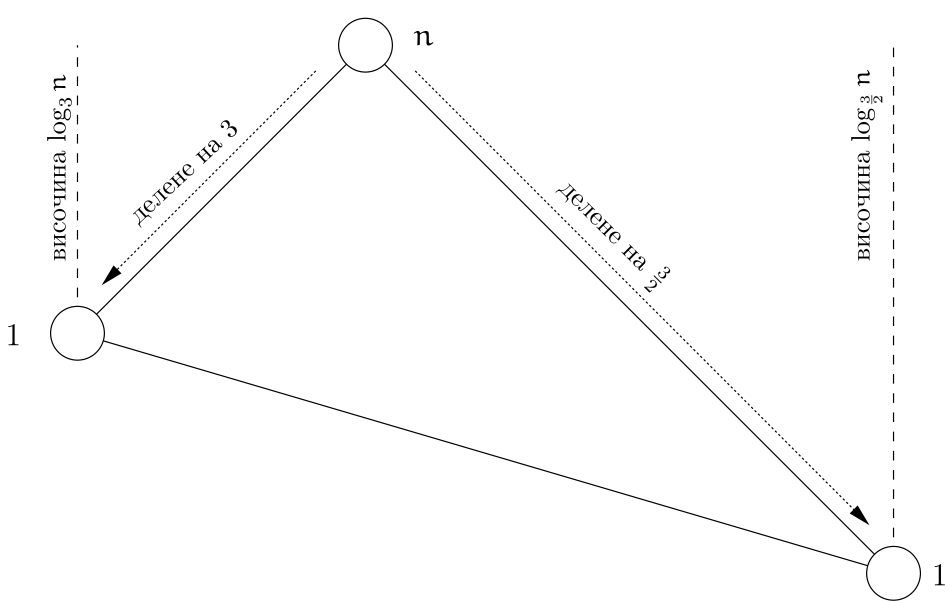
Коренът е първоначалното изпълнение. В него алгоритъмът извършва работа n извън двете рекурсивни викания, затова вдясно от корена е написано "n". Едното рекурсивно викане се върши върху вход, който е с големина $\frac{1}{3}$ от големината на началния, а другото върху вход, който е с големина $\frac{2}{3}$ от големината на началния. Затова двете ребра, свързващи корена с децата му, са маркирани с $\frac{1}{3}$ и $\frac{2}{3}$. До всяко от двете деца е написана работата, която се извършва от съответното извикване на алгоритъма. 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). Стойността, асоциирана с лявото дете на даден връх, е стойността, асоциирана с родителя, умножена по $\frac{1}{3}$. Стойността, асоциирана с дясното дете, е стойността, асоциирана с родителя, умножена по $\frac{2}{3}$. С други думи, наляво има деление на 3, а надясно, деление на $\frac{3}{2}$. Това, че разглеждаме дървото като наредено дърво, има значение единствено за по-лесното решаване на задачата. 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 краят на пътя, който започва от корена и във всеки вътрешен връх продължава наляво, а най-отдалеченото от корена листо е краят на пътя, който започва от корена и във всеки вътрешен връх продължава надясно. Образно казано, дървото има формата, показана на Фигура 3.2. 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 достигаме 1 по-бързо, отколкото с деление на $\frac{3}{2}$.

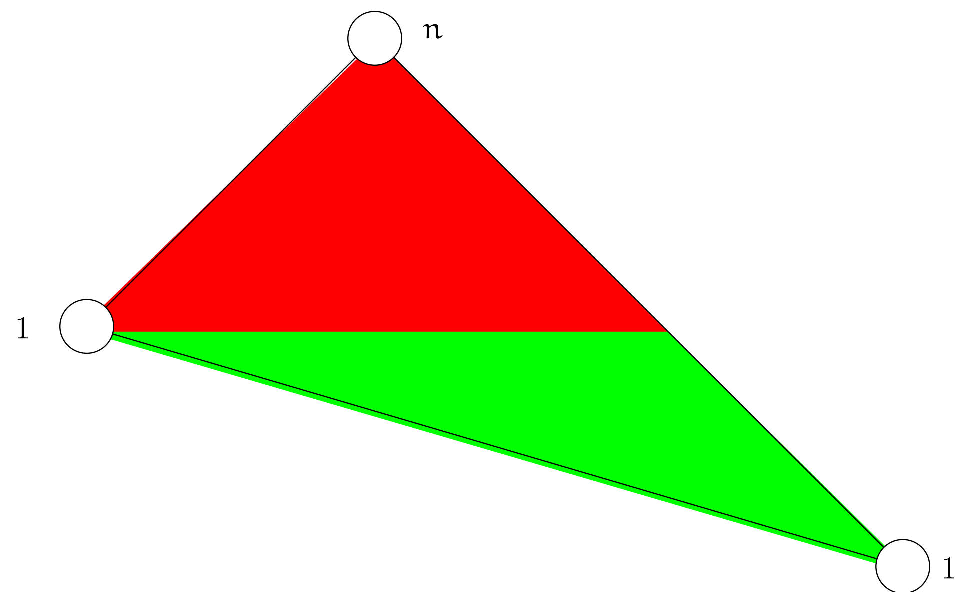
Тривиално е да се докаже, че нива с номер от 0 (коренът) до $\log_3 n$ (най-лявото дете) са



Фигура 3.1: Дървото на рекурсията, отговарящо на $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$. За всяко от изобразените нива, сумата от стойностите на върховете в него е n.



Фигура 3.2: 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 $\log_{\frac{3}{2}} n$.



Фигура 3.3: Друга схема на дървото при $\mathsf{T}(n) = \mathsf{T}\left(\frac{n}{3}\right) + \mathsf{T}\left(\frac{2n}{3}\right) + n$. Червеното поддърво е съвършено, понеже нивата му са пълни. Следващите нива, нарисувани в зелено, са непълни.

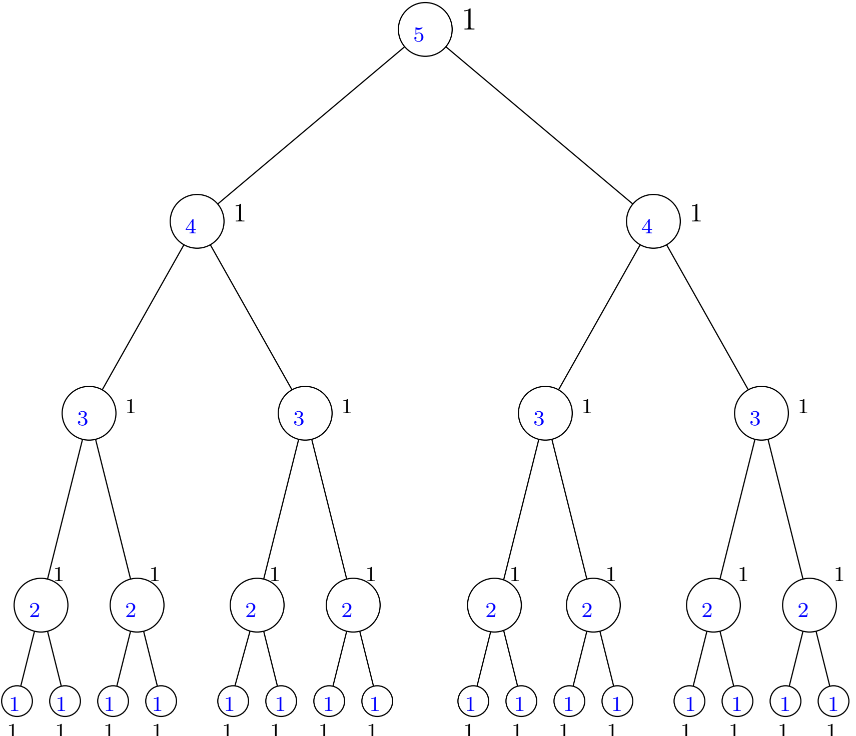
пълни[†] и във всяко от тях, сумата от стойностите е n. Фигура 3.3 илюстрира пълните нива и останалите: пълните нива са в червено, останалите са в зелено. Да намерим асимптотиката на рекурентното уравнение е същото като да оценим сумата от асоциираните стойности по всички върхове на дървото. За червеното поддърво (Фигура 3.3), 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 $n \log_{\frac{3}{2}} n$ е горна граница за същата сума, понеже $\log_{\frac{3}{2}}$ е височината на цялото дърво и никое ниво няма сума повече от n. Но $n \log_3 n \asymp n \lg n$ и $n \log_{\frac{3}{2}} n \asymp n \lg n$. Щом $\mathsf{T}(n) \preceq n \lg n$ и $\mathsf{T}(n) \succeq n \lg n$, следва, че $\mathsf{T}(n) \asymp n \lg n$.

От този пример става ясно, че методът с дървото на рекурсията е подобен на метода с развиването, само че тук групираме събираемите по различен начин, търсейки закономерност. В последния пример изчислихме сумата от всички “нехомогенни части” по всички извиквания. И по-точно казано, не я изчислихме, а само я оценихме като асимптотика. Под “нехомогенни части” имаме предвид събираеми като n в (3.29). Такова събираемо отговаря на работата, която алгоритъмът извършва извън рекурсивните викания. Ако наречем такова събираемо, “нехомогенната част” по аналогия с (3.8), то, прилагайки метода с дървото на рекурсията, ние всъщност сумирахме нехомогенните части[‡]. Истинската работа, която върши алгоритъмът, е сумата от тези “нехомогенни части” и броят на върховете на дървото. Но само сумата от “нехомогенните части” дава асимптотиката, така че можем да игнорираме броят на върховете и да не го добавяме към сумата от “нехомогенните части”. Ако обаче трябва да решаваме хомогенно рекурентно уравнение като (3.1), трябва да намерим броя на върховете, или поне неговата асимптотика.

От друга страна, очевидно е, че всяко рекурентно уравнение, което описва сложност на алгоритъм, има **ненулева** нехомогенна част, защото алгоритъмът няма как да не върши поне константна работа извън рекурсивните викания. Така че, по отношение на изследването на

[†] В смисъл, че в ниво k има 2^k върха.

[‡] По-точно, сумирахме нехомогенните части и началните условия, но ние игнорираме началните условия при асимптотичния анализ.



Фигура 3.4: Дървото на рекурсията при $T(1) = 1$, $T(n) = 2T(n-1)+1$, когато $n = 5$. Във върховете, в син цвят са написани съответните стойности на n.

алгоритми, напълно достатъчно е да сумираме въпросните "нехомогенни 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.

Сега ще дадем пример за това, колко е важно да открием закономерност в сумите при този метод. Да разгледаме рекурентното уравнение

$$
\begin{aligned}
&T(1) = 1 \\
&T(n) = 2T(n-1) + 1, \; n \geqslant 2
\end{aligned}
\tag{3.30}
$$

и съответното дърво на рекурсията на Фигура 3.4 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 (3.30) е $T(n) \asymp 2^n$.

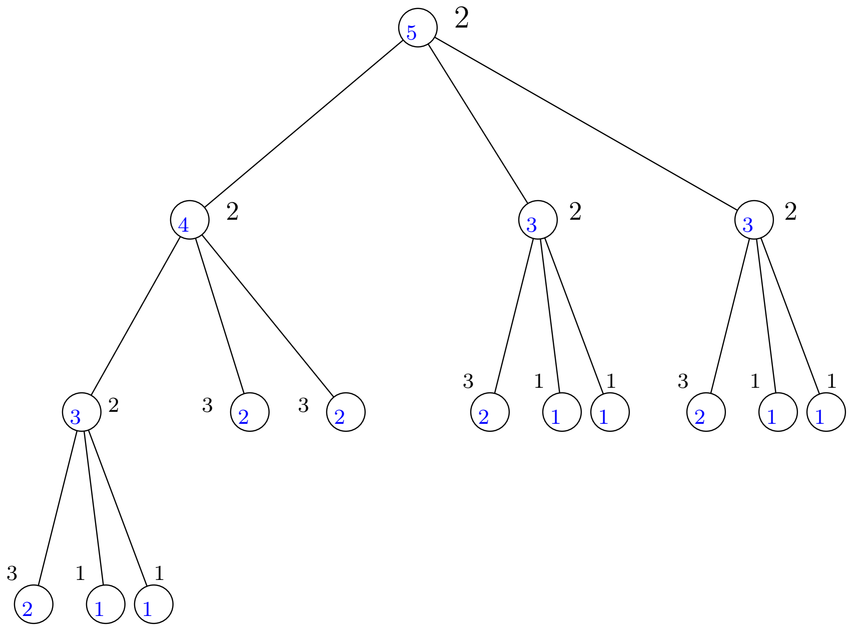
Сега да преобразуваме (3.30) ето така:

$$T(n) = 2T(n-1) + 1 \;\leftrightarrow\; T(n) = T(n-1) + T(n-1) + 1$$

Очевидно $T(n-1) = 2T(n-2) + 1$

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



Фигура 3.5: 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.

добавим и начално условия $T(2) = 3$. Рекурентното уравнение става:

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 3 \\ T(n) &= T(n-1) + 2T(n-2) + 2, \ n \geqslant 3 \end{aligned} \tag{3.31}$$

Рекурентни уравнение (3.30) и (3.31) са еквивалентни. Но техните съответни дървета на рекурсията нито са еднакви, нито са изоморфни. Фигура 3.5 показва дървото на рекурсията, съответстващо на (3.31), за $n = 5$. Дървото от Фигура 3.5 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:

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \tag{3.32}$$

Ще докажем, че $T(n) = \Theta(n \lg n)$ по индукция по n. За да постигнем това, ще докажем поотделно, че $T(n) = O(n \lg n)$ и $T(n) = \Omega(n \lg n)$. В тези доказателство база няма, тъй като игнорираме началните условия.

Част i: Доказателство, 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 $n \geqslant n_0$:

$$T(n) \leqslant cn \lg n \tag{3.33}$$

Индуктивното предположение е, че

$$\mathsf{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) \leqslant c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \tag{3.34}$$

От (3.32) и (3.34):

$$\begin{aligned}
\mathsf{T}(n) &\leqslant 2c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\
&\leqslant 2c \left(\frac{n}{2} + 1\right) \lg \left\lceil \frac{n}{2} \right\rceil + n & (3.35) \\
&\leqslant 2c \left(\frac{n}{2} + 1\right) \lg \left(\frac{3}{4} n\right) + n \qquad // \text{ защото } \forall n \geqslant 2 : \frac{3}{4} n \geqslant \left\lceil \frac{n}{2} \right\rceil & (3.36) \\
&= c(n+2) \left(\lg n + \lg \frac{3}{4}\right) + n \\
&= cn \lg n + cn \lg \frac{3}{4} + 2c \lg n + 2c \lg \frac{3}{4} + n
\end{aligned}$$

Иска се да докажем, че полученото $cn \lg n + cn \lg \frac{3}{4} + 2c \lg n + 2c \lg \frac{3}{4} + n$ не надхвърля $cn \lg n$, за някаква константа c. Това е вярно, ако $cn \lg \frac{3}{4} + 2c \lg n + 2c \lg \frac{3}{4} + n \leqslant 0$. Да разгледаме

$$cn \lg \frac{3}{4} + 2c \lg n + 2c(\lg 3 - 2) + n = \left(c \lg \frac{3}{4} + 1\right) n + 2c \lg n + 2c \lg \frac{3}{4}$$

Асимптотиката на нарастването се определя от събираемото $\left(c \lg \frac{3}{4} + 1\right) n$. 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

$$cn \lg n + cn \lg \frac{3}{4} + 2c \lg n + 2c \lg \frac{3}{4} + n \leqslant cn \lg n$$

за всички достатъчно големи стойности на n. С други думи, за по-кратко решение, няма да определяме n_0, бидейки убедени, че такава начална стойност има, а само ще определим подходящо c.

За да бъде изпълнено $c \lg \frac{3}{4} + 1 < 0$, трябва да е вярно, че $c > \frac{1}{\lg \frac{4}{3}}$. Численo, дясната страна е по-малка от 2.41. Следователно, всяко $c > 2.41$ "върши работа" за нашето доказателство.

Наблюдение 20

В (3.35) ние заместваме $\left\lceil \frac{n}{2} \right\rceil$ 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 "$\leqslant$" между (3.35) и (3.36) е в сила.
- А защо трябва да е изпълнено $\alpha < 1$? Да допуснем, че $\alpha = 1$; с други думи, заместваме $\left\lceil \frac{n}{2} \right\rceil$ с n. Тогава (3.36) 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: Доказателство, 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 $n \geqslant n_1$:

$$T(n) \geqslant dn \lg n \tag{3.37}$$

Индуктивното предположение е, че

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) \geqslant d \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil \tag{3.38}$$

От (3.32) и (3.38):

$$\begin{aligned} T(n) &\geqslant 2d \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\ &\geqslant 2d \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) + n \\ &= dn(\lg n - 1) + n \\ &= dn \lg n + (1-d)n \\ &\geqslant dn \lg n \qquad \text{при условие, че } (1-d)n \geqslant 0 \end{aligned} \tag{3.39}$$

Очевидно всяко d, такова че $0 < d \leqslant 1$, "върши работа" за нашето доказателство.

С това доказателството е приключено. 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 $T(n) \leqslant cn$ за някаква константа c. Индуктивното предположение е, че $T(n-1) \leqslant c(n-1)$. Тогава $T(n) \leqslant 2c(n-1) + 1$. Тогава $T(n) \leqslant 2cn - 2c + 1$. 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* $n \geqslant n_0$*,* $T(n) \leqslant cn$. Естествено, това твърдение е лъжа, така че няма как да го докажем, но важното е, че **<u>това</u>** 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 **<u>друга</u>** 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 индукционното предположение. Ще докажем по индукция, че решението на

  $$T(n) = 2T(n-1) + n \tag{3.40}$$

  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

$$T(n) \leqslant c2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leqslant 2c2^{n-1} + n \\ &= c2^n + n \\ &\nleqslant c2^n \text{ за никоя положителна } c \end{aligned}$$

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

$$T(n) \leqslant c2^n - b$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leqslant 2(c2^{n-1} - b) + n \\ &= c2^n - 2b + n \\ &\nleqslant c2^n - b \text{ за никои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Трети опит. Допускаме, че съществуват положителни константи b и c, такива че за всички достатъчно големи n е вярно, че

$$T(n) \leqslant c2^{n-b}$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leqslant 2(c2^{n-b-1}) + n \\ &= c2^{n-b} + n \\ &\nleqslant c2^{n-b} \text{ за никои положителни } b, c \end{aligned}$$

Ще засилим още твърдението.

Четвърти опит. Допускаме, че съществува положителна константа c, такава че за всички достатъчно големи n е вярно, че

$$T(n) \leqslant c2^n - n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leqslant 2(c2^{n-1} - (n-1)) + n \\ &= c2^n - n + 2 \\ &\nleqslant c2^n - n \text{ за никоя положителна } c \end{aligned}$$

Пети опит. Допускаме, че съществуват положителни константи b и c, такива че за всички достатъчно големи n е вярно, че

$$T(n) \leqslant c2^n - bn$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\leqslant 2(c2^{n-1} - b(n-1)) + n \\ &= c2^n - 2bn + 2b + n \\ &= c2^n - bn + (1-b)n + 2b \\ &\leqslant c2^n - bn \text{ за всеки избор на } c > 0 \text{ и } b > 1 \end{aligned}$$

Успех! Чак сега успяхме да формулираме индукционно 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

$$T(n) \geqslant d2^n$$

От индуктивното предположение имаме

$$\begin{aligned} T(n) &\geqslant 2(d2^{n-1}) + n \\ &= d2^n + n \\ &\geqslant d2^n \end{aligned}$$

Успех! Видяхме, че в този случай—а това е вярно по принцип при доказателства по индукция на решения на рекурентни уравнения—засилване на индукционното предположение се иска само за едното от O и Ω доказателствата.

Следната задача ползва термина *целочислено разбиване*, на английски *integer partition*.

Определение 35: Целочислено разбиване

Целочислено разбиване на цяло положително число k е изреждане на цели положителни числа в ненарастващ ред, които се сумират до k.

Това, че събираемите да се изреждат в ненарастващ ред, е само за по-лесно възприемане; формално, няма наредба между тях, така че две изреждания на едни и същи събираеми са едно и също разбиване; за да бъдат различни две разбивания, трябва мултимножествата от събираемите да са различни. Записът става със знака "+" между дяловете на разбиването, а

не с нотация за мултимножество. Като пример, числото 4 има пет целочислени разбивания:

$$
\begin{aligned}
4 &= 4 \\
4 &= 3 + 1 \\
4 &= 2 + 2 \\
4 &= 2 + 1 + 1 \\
4 &= 1 + 1 + 1 + 1
\end{aligned}
$$

В Задача 31 доказваме само O за $T(n)$; това, че $T(n) = \Omega(n)$, е доста очевидно. А "m" не означава брой на ребра—за каквито в тази задача не става дума—а е буква за броя на дяловете на някакво целочислено разбиване.

Задача 31

Докажете, че рекурентното уравнение

$$T(n) = \sum_{i=1}^{m} T(n_i) + \Theta(m)$$

за произволно целочислено разбиване $n_1+n_2+\cdots+n_m$ на $n-1$ има решение $T(n) = O(n)$.

Решение: С индукция по n. Да допуснем, че съществуват положителни константи b и c, такива че $T(n) \leqslant c \cdot n - b$. Да допуснем, че k е достатъчно голяма положителна константа; в частност, която е по-голяма или равна на скритата мултипликативна константа в израза "$\Theta(m)$". От индуктивното предположение:

$$
\begin{aligned}
T(n) &\leqslant \sum_{i=1}^{m}(c \cdot n_i - b) + k \cdot m \\
&= c\left(\sum_{i=1}^{m} n_i\right) - b \cdot m + k \cdot m \qquad // \text{ но } n_1 + n_2 + \cdots + n_m = n - 1 \\
&= c(n-1) + m(k-b) \\
&= c \cdot n - c + m(k-b) \\
&\leqslant c \cdot n - b, \quad \text{ако } k - b < 0 \text{ и } c > b
\end{aligned}
$$

□

Задача 31 е полезна при оценяване на сложността на алгоритми по схемите **Разделяй-и-Владей** или **Динамично Програмиране**, каквито ще видим в Подсекция 12.9.1 и Подсекция 12.9.2, които се изпълняват върху коренови дървета и при които сложността върху дърво с n върха е

- сумата от сложностите на виканията върху поддърветата, вкоренени в децата на корена—ние не знаем точно колко са тези деца, може да са от 1 до $n-1$ на брой, така че казваме, че има m деца, като $1 \leqslant m \leqslant n-1$—като тези деца имат $n_1, \ldots, n_m$ върха, като $n_1 + n_2 + \cdots + n_m$ се явява целочислено разбиване на $n-1$,

- плюс сложността на алгоритъма извън рекурсивните викания, за която допускаме, че е линейна в m.

Сложността извън рекурсивните викания е поне линейна в m, защото всяко от извършените m рекурсивни викания връща $\Omega(1)$ информация и всичко това трябва да бъде обработено от текущото рекурсивно викане, което дава долна граница $\Omega(m)$.

Допълнение 19: Няколко рекурентни уравнения по индукция

Следното нелеко за решаване рекурентно уравнение е от сборника на Parberry [116, pp. 40, Problem 248].

Задача 32

Решете рекурентното уравнение

$$T(n) = T\left(\left\lfloor \frac{n}{\lg n} \right\rfloor\right) + 1. \tag{3.41}$$

Решение: Уравнението е с единствена поява вдясно, поради което решението му, в асимптотичния смисъл, е броят пъти, който се изпълнява итераторът

$$n \to \left\lfloor \frac{n}{\lg n} \right\rfloor \tag{3.42}$$

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 13, че решението е $S(n) = \log_m n$. Да видим какво става, когато m не е константа. Дори тогава можем да приложим Следствие 13, 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 очевидно е

$$S(n) = \Theta(\log_{\lg n} n) = \Theta\left(\frac{\lg n}{\lg \lg n}\right)$$

Обаче в (3.42), n намалява при итерирането, така че и $\lg n$ намалява, така че делим на все по и по-малко нещо, което клони към константа.
От тези съображения можем да сложим следните граници за $T(n)$:

$$\frac{\lg n}{\lg \lg n} \preceq T(n) \preceq \lg n \tag{3.43}$$

Това вече е нещо. $\lg \lg n$ е доста близо до константа (макар да е неограничено растяща функция), така че (3.43) 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 (3.41) и $\mathsf{T}(n)$ се определя чрез

$$\mathsf{T}(n) = \mathsf{T}\left(\frac{n}{\lg n}\right) + 1 \tag{3.44}$$

Част i: Ще докажем, че $\mathsf{T}(n) = \mathsf{O}\left(\frac{\lg n}{\lg \lg n}\right)$. Тоест, че има положителна константа c и стойност на аргумента n_0, такива че за всяко $n \geqslant n_0$,

$$\mathsf{T}(n) \leqslant c \frac{\lg n}{\lg \lg n} \tag{3.45}$$

Индуктивното предположение е

$$\mathsf{T}\left(\frac{n}{\lg n}\right) \leqslant c \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg \left(\frac{n}{\lg n}\right)} \tag{3.46}$$

Заместваийки (3.46) в (3.44), получаваме

$$\begin{aligned} \mathsf{T}(n) &\leqslant c \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg \left(\frac{n}{\lg n}\right)} + 1 \\ &= c \frac{\lg n - \lg \lg n}{\lg\left(\lg n - \lg \lg n\right)} + 1 \end{aligned} \tag{3.47}$$

Заместваме $\lg n$ с 2^m, така че заместваме $\lg \lg n$ с m, в (3.47) и получаваме

$$\begin{aligned} &c \frac{2^m - m}{\lg\left(2^m - m\right)} + 1 \leqslant c \frac{2^m - m}{\lg 1.9^m} + 1 = c \frac{2^m - m}{m \lg 1.9} + 1 = \frac{c}{1.9} \frac{2^m - m}{m} + 1 = \\ &\frac{c}{1.9} \frac{2^m}{m} - \frac{c}{1.9} + 1 \leqslant c \frac{2^m}{m} = c \frac{\lg n}{\lg \lg n} \end{aligned} \tag{3.48}$$

От (3.47) и (3.48) заключаваме, че

$$\mathsf{T}(n) \leqslant c \frac{\lg n}{\lg \lg n}$$

за всяко положително c, да кажем $c = 1$, и всяко достатъчно голямо n.

Част ii: Ще докажем, че $\mathsf{T}(n) = \Omega\left(\frac{\lg n}{\lg \lg n}\right)$. Тоест, че има положителна константа d и стойност на аргумента n_0, такива че за всяко $n \geqslant n_0$,

$$\mathsf{T}(n) \geqslant d \frac{\lg n}{\lg \lg n} \tag{3.49}$$

Индуктивното предположение е

$$\mathsf{T}\left(\frac{n}{\lg n}\right) \geqslant d \frac{\lg\left(\frac{n}{\lg n}\right)}{\lg \lg \left(\frac{n}{\lg n}\right)} \tag{3.50}$$

Замествайки (3.50) в (3.44), получаваме

$$
\begin{aligned}
T(n) &\geqslant d\frac{\lg\left(\frac{n}{\lg n}\right)}{\lg\lg\left(\frac{n}{\lg n}\right)} + 1 \\
&= d\frac{\lg n - \lg\lg n}{\lg(\lg n - \lg\lg n)} + 1
\end{aligned} \tag{3.51}
$$

Заместваме $\lg n$ с 2^m, така че заместваме $\lg\lg n$ с m, в (3.51) и получаваме

$$
d\frac{2^m - m}{\lg(2^m - m)} + 1 \geqslant d\frac{2^m - m}{\lg 2^m} + 1 = d\frac{2^m - m}{m} + 1 = d\frac{2^m}{m} - d + 1 \geqslant d\frac{2^m}{m} \tag{3.52}
$$

Това е вярно за, да кажем, $d = 1$. Връщаме се към променливата n. От (3.51) и (3.52) виждаме, че

$$
T(n) \geqslant d\frac{\lg n}{\lg\lg n}
$$

за $d = 1$ и всяко достатъчно голямо n. □

Ние вече решихме $T(n) = T(n-2) + 2\lg n$ в Задача 25, 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: С 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,

$$
T(n) \leqslant cn\lg n \tag{3.53}
$$

В сила е

$$
\begin{aligned}
T(n) &\leqslant c(n-2)\lg(n-2) + 2\lg n \qquad // \text{ от индукционното предположение} \\
&\leqslant c(n-2)\lg n + 2\lg n \\
&= cn\lg n - 2c\lg n + 2\lg n \\
&\leqslant cn\lg n, \text{ ако } -2c\lg n + 2\lg n \leqslant 0 \leftrightarrow c \geqslant 1
\end{aligned}
$$

Part ii: Ще докажем, че $T(n) = \Omega(n\lg n)$. Тоест, че има положителна константа d, такава че за всяко достатъчно голямо n,

$$
T(n) \geqslant dn\lg n \tag{3.54}
$$

В сила е

$$\begin{aligned} T(n) &\geqslant d(n-2)\lg(n-2) + 2\lg n \qquad // \text{ от индукционното предположение} \\ &= (dn - 2d)\lg(n-2) + 2\lg n \\ &= dn\lg(n-2) + 2(\lg n - d\lg(n-2)) \end{aligned} \tag{3.55}$$

Предвид (3.54) and (3.55), целта ни е да покажем, че

$$\begin{aligned} dn\lg(n-2) + 2(\lg n - d\lg(n-2)) \geqslant dn\lg n \leftrightarrow \\ dn\lg(n-2) - dn\lg n + 2(\lg n - d\lg(n-2)) \geqslant 0 \leftrightarrow \\ \underbrace{d\lg\left(\frac{n-2}{n}\right)^n}_{A} + \underbrace{2\lg\frac{n}{(n-2)^d}}_{B} \geqslant 0 \end{aligned} \tag{3.56}$$

Първо ще оценим A при неограничено нарастване на n:

$$\lim_{n\to\infty} d\lg\left(\frac{n-2}{n}\right)^n = d\lim_{n\to\infty}\lg\left(1+\frac{-2}{n}\right)^n = d\lg\lim_{n\to\infty}\left(1+\frac{-2}{n}\right)^n = d\lg e^{-2} = -2d\lg e$$

Да разгледаме B при неограничено нарастване на n:

$$\lim_{n\to\infty} 2\lg\frac{n}{(n-2)^d} = 2\lg\lim_{n\to\infty}\frac{n}{(n-2)^d} \tag{3.57}$$

Забележете, че при всяко $d \in (0,1)$, (3.57) е $+\infty$. Например, за $d = \frac{1}{2}$, (3.57) става

$$\begin{aligned} &2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\frac{n^{\frac{1}{2}}}{(n-2)^{\frac{1}{2}}}\right) = \\ &2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\left(\frac{n}{n-2}\right)^{\frac{1}{2}}\right) = \\ &2\lg\left(\left(\lim_{n\to\infty}n^{\frac{1}{2}}\right)\underbrace{\left(\lim_{n\to\infty}\left(\frac{1}{1-\frac{2}{n}}\right)^{\frac{1}{2}}\right)}_{1}\right) = +\infty \end{aligned}$$

Следва, че (3.56) е вярно за всеки избор на d от $(0,1)$; примерно, $d = \frac{1}{2}$, понеже A по абсолютна стойност е ограничена от константа, а B расте неограничено. □

Наблюдение 21

Доказателството по индукция в **Част ii** е триково. Разгледайте (3.55):

$$dn \lg (n-2) + 2(\lg n - d \lg (n-2))$$

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. [15]. 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 26: 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 [15] 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 [31]. Без съмнение то идва оттам, че въпросната теорема

се явява обобщение на множество теореми като Теорема 26, 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 уравнение. По отношение на линейните рекурентни уравнения с константни коефициенти, при които намаляването е чрез деление с константа, каквито уравнения третира Теорема 27, е известна значително по-мощна и обща Теорема на Akra-Bazzi [4] (Подсекция 3.2.5). И така, Теорема 27, 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. [31, стр. 61].

Теорема 27: Мастър теорема (Theorem 4.1 от [31, стр. 94])

Нека $a \geqslant 1$ и $b > 1$ са константи и нека $f(n)$ е положителна функция. Нека

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3.58}$$

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 $f(n) = O\left(n^{\log_b a}/n^{\epsilon}\right)$ за някоя положителна константа ϵ, то $T(n) = \Theta(n^{\log_b a})$.

Случай 2 Ако $f(n) = \Theta(n^{\log_b a})$, тогава $T(n) = \Theta\left(n^{\log_b a}. \lg n\right)$. С други думи, $T(n) = \Theta(f(n). \lg n)$.

Случай 3 Ако са изпълнени следните условия:

1. $f(n) = \Omega\left(n^{\log_b a} \cdot n^{\epsilon}\right)$ за някоя положителна константа ϵ, и
2. съществува константа c, такава че $0 < c < 1$ и $\forall n \nearrow : a \cdot f\left(\frac{n}{b}\right) \leqslant c \cdot f(n)$,

то $T(n) = \Theta(f(n))$. □

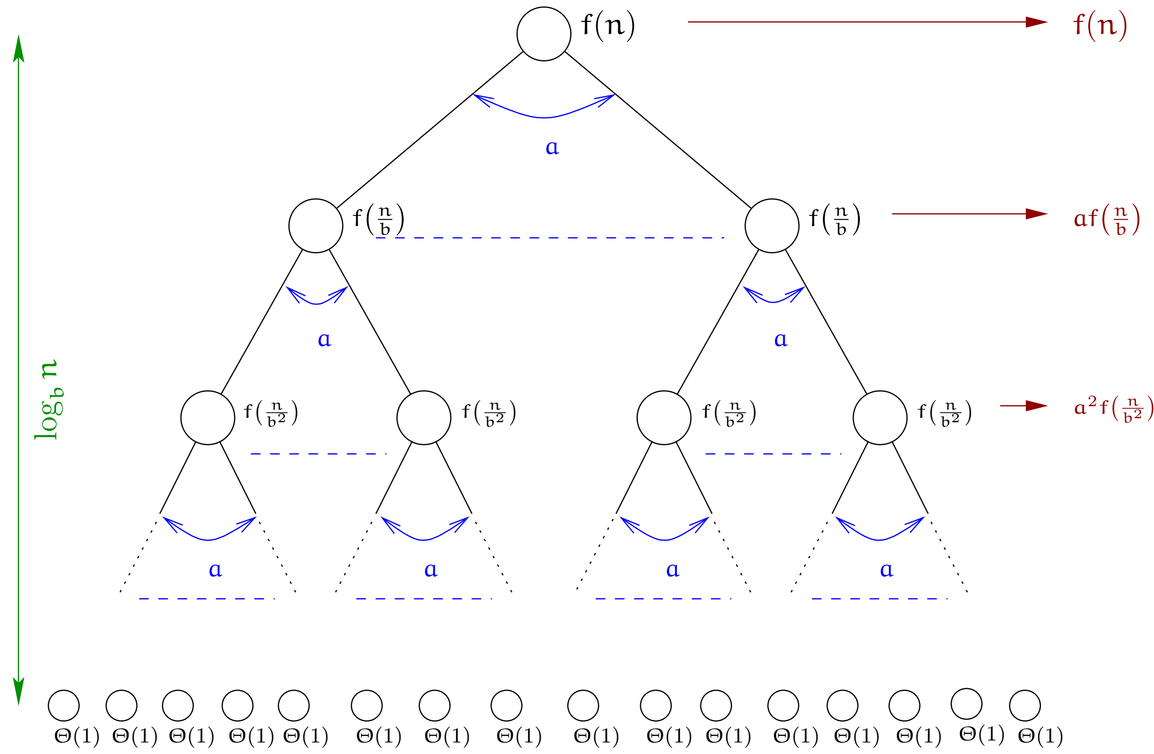
Накратко ще наричаме тази теорема "МТ".

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 22. Читателят може да види пълно доказателство на МТ в [31]. Доказателство има и в статията [90]. 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 $b \geqslant 2$, то (3.58) 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 3.6. Всеки връх на дървото има разклоненост a, и входът намалява с деление на b. Сумите по нивата на дървото на нехомогенните части са $f(n)$, $af\left(\frac{n}{b}\right)$, $a^2 f\left(\frac{n}{b^2}\right)$, и така нататък. Височината на дървото е $\log_b n$. Броят

†В [31] е доказано, че същите съображения остават в сила дори когато a и b не са цели.



Фигура 3.6: 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.

на върховете от ниво[†] k очевидно е a^k, така че броят на листата е 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 $a = 1$, дървото има точно едно листо, а височината продължава да е $\log_b n$, но да игнорираме изключението $a = 1$.

Наблюдение 22

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.

[†]Припомняме си, че върховете от ниво k са точно върховете на разстояние k от корена.

Наблюдение 23

Сравняването на големината на дървото с нехомогенната част има смисъл и при други видове рекурентни уравнения. Например, в Задача 28 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 29 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 30 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 $n!$. В някакъв смисъл, това е аналог на третия случай на МТ.

Решенито 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 неговия логаритъм.

Пояснение 2. Трите случаи на МТ **не са разбиване** на всички възможни случаи. Както вече видяхме в Теорема 12(2.39), 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 $f(n) \neq O\left(n^{\log_b a}/n^{\epsilon}\right)$ 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 20: Решение на $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$

Ще решим $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$. Първо да опитаме с Теорема 27. Използвайки терминологията на Теорема 27, $a = b = 2$, 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 $\frac{n}{\lg n} = \Omega(n^1.n^\epsilon)$ 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 27. Нещо повече: не можем да ползваме и Теорема 28 на стр. 206, защото $\forall t > 0 : \frac{n}{\lg n} \neq \Theta(n^{\log_2 2} \times \lg^t(n))$.
Ще решим задачата с развиване:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{\frac{n}{2}}{\lg\frac{n}{2}}\right) + \frac{n}{\lg n} \\
&= 4T\left(\frac{n}{4}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= 4\left(2T\left(\frac{n}{8}\right) + \frac{\frac{n}{4}}{\lg\frac{n}{4}}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= 8T\left(\frac{n}{8}\right) + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&\dots \\
&= nT(1) + \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \dots + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= nT(1) + \underbrace{n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{(\lg n) - 2} + \frac{1}{(\lg n) - 1} + \frac{1}{\lg n}\right)}_{B}
\end{aligned}$$

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 $f(n) = O\left(n^{\log_b a}/n^\epsilon\right)$ в първия случай на МТ е по-силно от $f(n) = o\left(n^{\log_b a}\right)$, а условието $f(n) = \Omega\left(n^{\log_b a}.n^\epsilon\right)$ в третия случай е по-силно от $f(n) = \omega\left(n^{\log_b a}\right)$. За удобство да направим следните дефиниции.

Определение 36: 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* $\psi(n)$.

$\boxed{f(n) = O\left(n^{\log_b a}/n^\epsilon\right)}$, 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 $\boxed{f(n) = o\left(n^{\log_b a}\right)}$. Аналогично, $\boxed{f(n) = \Omega\left(n^{\log_b a} \cdot n^{\epsilon}\right)}$ 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 $\boxed{f(n) = \omega\left(n^{\log_b a}\right)}$.

Наблюдение 24

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 24 ни дава право да кажем следното.

$$\begin{aligned}
f(n) = O(n^k/n^{\epsilon}) &\rightarrow f(n) = o(n^k) \\
f(n) = o(n^k) &\nrightarrow f(n) = O(n^k/n^{\epsilon}) \\
f(n) = \Omega(n^k.n^{\epsilon}) &\rightarrow f(n) = \omega(n^k) \\
f(n) = \omega(n^k) &\nrightarrow f(n) = \Omega(n^k.n^{\epsilon})
\end{aligned}$$

Наблюдение 25

Би било грешка да се опитаме да формулираме първия случай на МТ чрез $\boxed{f(n) = o\left(n^{\log_b a}\right)}$ или третия случай чрез $\boxed{f(n) = \omega\left(n^{\log_b a}\right)}$.

Пояснение 4. Условието за регулярност в третия случай е по-силно от условието $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ в смисъл, че

$$\exists c, 0 < c < 1, \forall n \nearrow : a \cdot f\left(\frac{n}{b}\right) \leqslant c \cdot f(n) \rightarrow \exists \epsilon > 0 : f(n) = \Omega\left(n^{(\log_b a) + \epsilon}\right) \tag{3.59}$$

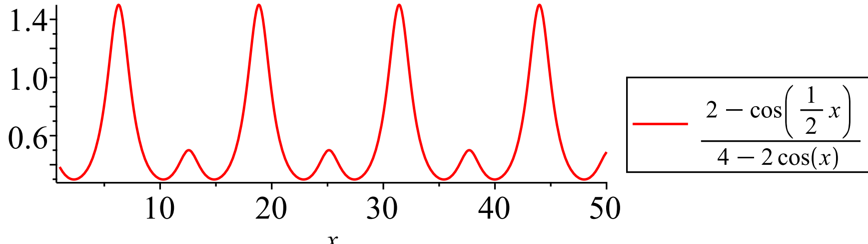
$$\exists \epsilon > 0 : f(n) = \Omega\left(n^{(\log_b a) + \epsilon}\right) \nrightarrow \exists c, 0 < c < 1, \forall n \nearrow : a \cdot f\left(\frac{n}{b}\right) \leqslant c \cdot f(n) \tag{3.60}$$

Ето пример за $f(n)$ като в (3.60), 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 опитаме 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 $f(n) = \Omega\left(n^{(\log_b a) + \epsilon}\right)$. 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

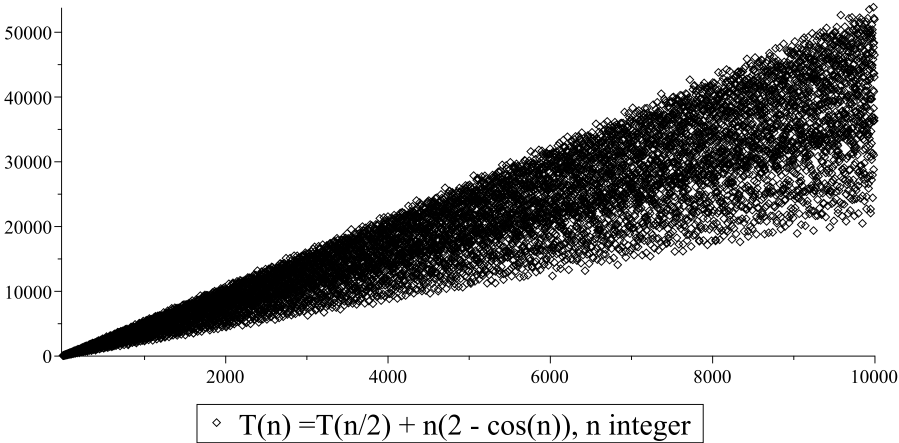
$a \cdot f\left(\frac{n}{b}\right) \leqslant c \cdot f(n)$? Очевидно

$$
\begin{aligned}
& a \cdot f\left(\frac{n}{b}\right) \leqslant c \cdot f(n) \leftrightarrow \\
& 1 \cdot \frac{n}{2}\left(2 - \cos\left(\frac{n}{2}\right)\right) \leqslant c \cdot n(2 - \cos(n)) \leftrightarrow \\
& 2 - \cos\left(\frac{n}{2}\right) \leqslant 2c(2 - \cos(n)) \\
& c \geqslant \frac{2 - \cos\left(\frac{n}{2}\right)}{2(2 - \cos(n))}
\end{aligned}
$$

Лесно се вижда, че за всяко цяла положително n_0 съществува цяло $n \geqslant n_0$, такова 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 $n = 10\,000$.



Импликацията (3.59) 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 21. 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.

†Фигурата е направена с Maple(TM).
‡Фигурата е направена с Maple(TM).

Допълнение 21: Условието за регулярност в МТ влече Случай 3

Задача 4.4-3 в [31] е: покажете, че Случай 3 на МТ съдържа излишък в смисъл, че условието за регулярност $af\left(\frac{n}{b}\right) \leqslant cf(n)$ 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:

$$f(n) \geqslant \frac{a}{c} f\left(\frac{n}{b}\right)$$

Тъй като $a \geqslant 1$ и $c < 1$, то $\frac{a}{c} > 1$. За краткост, ще означаваме $\frac{a}{c}$ с буквата s. Тогава в сила е:

$$f(n) \geqslant s f\left(\frac{n}{b}\right)$$

Тогава

$$\begin{aligned}
f(n) &\geqslant s^2 f\left(\frac{n}{b^2}\right)\\
f(n) &\geqslant s^3 f\left(\frac{n}{b^3}\right)\\
&\dots\\
f(n) &\geqslant s^t f\left(\frac{n}{b^t}\right) \qquad (3.61)
\end{aligned}$$

Съществува някаква стойност n_0 на променливата n, за която този процес ще спре. Тогава $\frac{n}{b^t} = n_0$, следователно $t = \log_b\left(\frac{n}{n_0}\right) = \log_b n - \log_b n_0$. Заместваме t и $\frac{n}{b^t}$ и c в (3.61) и получаваме:

$$f(n) \geqslant s^{\log_b n - \log_b n_0} \times f(n_0) = \frac{s^{\log_b n}}{s^{\log_b n_0}} \times f(n_0)$$

Забележете, че $\frac{1}{s^{\log_b n_0}} \times f(n_0)$ е константа. За краткост, да наречем тази константа β. Тогава:

$$f(n) \geqslant s^{\log_b n} \times \beta$$

Да се върнем към началните имена a и c, като полагаме $z = \frac{1}{c}$, за да е изпълнено $z > 1$. Имаме $s = a \times z$.

$$f(n) \geqslant a^{\log_b n} \times z^{\log_b n} \times \beta$$

Имайки предвид, че $a^{\log_b n} = n^{\log_b a}$ and $z^{\log_b n} = n^{\log_b z}$, извеждаме:

$$f(n) \geqslant \beta \times n^{\log_b a} \times n^{\log_b z}$$

Тъй като $z > 1$, то $\log_b z > 0$. Нека $\epsilon = \log_b z$. Извеждаме желания резултат: за всички достатъчно големи n и някаква константа β:

$$f(n) \geqslant \beta n^{(\log_b a)+\epsilon} \Rightarrow f(n) = \Omega(n^{(\log_b a)+\epsilon})$$

□

Ит така, **Случай 3** на МТ би могъл да бъде формулиран само с условието за регулярност, а условието $f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$ е излишно. Това е и краят на поясненията върху МТ.

Допълнение 22: 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, $m = \log_b n$.

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \leftrightarrow \\ T(b^m) &= aT\left(\frac{b^m}{b}\right) + f(b^m) \leftrightarrow \\ T(b^m) &= aT\left(b^{m-1}\right) + f(b^m) \end{aligned}$$

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 19 и Следствие 12): 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

$$g(m) = p_1(m)c_1^m + \cdots + p_t(m)c_t^m \tag{3.62}$$

където p_i са полиноми, а c_i са константи, такива че $i \neq j \rightarrow c_i \neq c_j$. Да допуснем, че (3.62) 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

$$\{\underbrace{c_1, \ldots, c_1}_{\deg(p_1)+1}, \ldots, \underbrace{c_t, \ldots, c_t}_{\deg(p_t)+1}\}_M$$

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 (3.78), решението е

$$S(m) \asymp \begin{cases} a^m, & \text{ако } a > c_1 \\ m^d \cdot a^m, & \text{ако } a = c_1 \\ m^{d-1} \cdot c_1^m. & \text{ако } a < c_1 \end{cases} \tag{3.63}$$

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

$$S(m) \asymp \begin{cases} a^m, & \text{ако } a > c_1 \\ m^d \cdot a^m, & \text{ако } a = c_1 \\ g(m). & \text{ако } a < c_1 \end{cases} \tag{3.64}$$

Да се върнем към оригиналната променлива n и функция $T(n)$, помнейки, че $m = \log_b n$, $S(m) = T(n)$ и $g(m) = f(n)$:

$$T(n) \asymp \begin{cases} a^{\log_b n}, & \text{ако } a > c_1 \\ (\log_b n)^d \cdot a^{\log_b n}, & \text{ако } a = c_1 \\ f(n). & \text{ако } a < c_1 \end{cases} \tag{3.65}$$

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 27), на която вторият случай е засилен чрез обобщението на МТ (Теорема 28), но **при допускането**, че $f(n)$ е такава, че

$$f(n) = p_1(\log_b n) \cdot c_1^{\log_b n} + \cdots + p_t(\log_b n) \cdot c_t^{\log_b n} \tag{3.66}$$

С други думи, ако $f(n)$ сума от функции, всяка от които е произведение от полиномиална функция на n и полилогаритмична функция на n. За да се убедим, че (3.62) и (3.66) казват едно и също, да си припомним, че $m = \log_b n$ и $g(m) = f(b^m) = f(n)$.

Предвид това, че събираемото с най-голямата константа c_i задава асимптотиката на $f(n)$, веднага виждаме, че случаите на МТ (Теорема 27) 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 (3.66), така че няма нужда да го споменаваме в (3).
Нещо повече. Дори $f(n)$ да е по-бързо растяща функция от това, което (3.66) казва—с други думи, ако $f(n)$ не е от вида (3.66), но е суперполиномиална—асимптотиката на $T(n)$ пак се дава от $f(n)$. Така че резултата от Допълнение 22 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 (3.66), 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:

$$\exists c \text{ такова че } 0 < c < 1 \text{ и } 3\frac{n}{4} \lg \frac{n}{4} \leqslant cn \lg n \text{ за всички достатъчно големи } n$$

Очевидно това е вярно за, да речем, $c = \frac{3}{4}$, така че третият случай на МТ е приложим. Съгласно него, $T(n) \asymp n \lg n$. □

Задача 35

Решете чрез МТ $T(n) = T\left(\frac{2n}{3}\right) + 1$.

Решение: Преписваме условието така:

$$T(n) = 1.T\left(\frac{n}{\frac{3}{2}}\right) + 1$$

за да е по-ясно какви са a и b. Виджаме, 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, $T(n) \asymp \lg n$. □

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, $T(n) \asymp n^2$. □

Допълнение 23: Едно разширение на Теорема 27

Теорема 28: Разширение на МТ

При допусканията на Теорема 27, нека

$$f(n) = \Theta(n^{\log_b a} \lg^t n) \tag{3.67}$$

за някое $t \in \mathbb{N}^+$. Тогава $T(n) = \Theta(n^{\log_b a} \lg^{t+1} n)$.

Доказателство: Теорема 27 не е директно приложима, защото рекурентното уравнение с тази функция $f(n)$ при $t \geqslant 1$ не може да бъде класифицирано в нито един от трите случая. Решаваме задачата с развиване. За простота допускаме, 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 (3.67) е с основа b. Забелязваме, че всичко в Θ-нотацията в дясната страна на (3.67) може да се запише така:

$$n^{\log_b a} \log_b^t n = b^{(m \log_b a)}\, m^t = b^{(\log_b a^m)}\, m^t = a^m m^t \tag{3.68}$$

Тогава (3.67) е еквивалентно на

$$c_1 a^m m^t \leqslant f(b^m) \leqslant c_2 a^m m^t$$

за някакви положителни константи c_1 и c_2 и всички достатъчно големи стойности на m. За простота, ще допуснем, че

$$f(b^m) = a^m m^t \tag{3.69}$$

Не е трудно да се направи 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 (3.69), записваме:

$$\begin{aligned}
T(b^m) &= aT\left(\frac{b^m}{b}\right) + a^m m^t \\
&= aT(b^{m-1}) + a^m m^t \;\leftrightarrow \\
S(m) &= aS(m-1) + a^m m^t \qquad \text{замествайки } T(b^m) \text{ със } S(m) \\
&= a\big(aS(m-2) + a^{m-1}(m-1)^t\big) + a^m m^t \\
&= a^2 S(m-2) + a^m (m-1)^t + a^m m^t \\
&= a^2\big(aS(m-3) + a^{m-2}(m-2)^t\big) + a^m(m-1)^t + a^m m^t \\
&= a^3 S(m-3) + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\
&\ldots \\
&= a^{m-1}S(1) + a^m 2^t + a^m 3^t + \cdots + a^m(m-2)^t + a^m(m-1)^t + a^m m^t \\
&= a^{m-1}S(1) + a^m(2^t + 3^t + \cdots + (m-2)^t + (m-1)^t + m^t)
\end{aligned}$$

Но $2^t + 3^t + \cdots + (m-2)^t + (m-1)^t + m^t = \Theta(m^{t+1})$; за извеждането на това вижте [56, (6.98) на стр. 288]. Тогава в сила е

$$\begin{aligned}
&a^{m-1}S(1) + a^m(2^t + 3^t + \cdots + (m-2)^t + (m-1)^t + m^t) = \\
&a^{m-1}S(1) + a^m\Theta(m^{t+1}) = \\
&\Theta(a^m m^{t+1})
\end{aligned}$$

Изведохме:

$$S(m) = \Theta(a^m m^{t+1}) \tag{3.70}$$

Да се върнем към началните T и n. (3.70) става

$$T(n) = \Theta\left(a^{\log_b n}(\log_b n)^{t+1}\right)$$

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 чрез $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$.

Решение: Да опитаме с МТ. Имаме $a = 2$, $b = 2$, така че $\log_b a$ е $\log_2 2 = 1$, следователно $n^{\log_b a}$ е $n^1 = n$. 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 $n \lg n = \Omega(n^1.n^\epsilon)$ за някое $\epsilon > 0$? Не. Вижте Теорема 17.

Следователно, тази задача не може да се реши с МТ. Ще я решим чрез Теорема 28, 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 по различни начини като например SELECT в Подсекция 6.2.2, не се поддават на изследване с МТ. Рекурентното уравнение (6.6), което описва сложността на SELECT, е решено по индукция в [31, стр. 222], защото [31] 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 24, понеже е сравнително сложно. Теоремата, както и доказателството ѝ, са представени съгласно статията на Leighton [96][†]. Заслужава да се спомене непубликуваната статия на Neville Campbell [26] от над 300 (триста!) страници, която твърди, че доказателствата на Leighton съдържа дребни грешки от непрецизност, които тя (твърди, че) поправя.

Теорема 29: Теорема на Akra-Bazzi (Theorem 1 от [96])

Нека

$$T(x) = \begin{cases} \Theta(1), & \text{ако } 1 \leqslant x \leqslant x_0 \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x), & \text{ако } x > x_0 \end{cases} \tag{3.71}$$

където

1. $x \geqslant 1$ е реално число,
2. x_0 е константа, такава че $x_0 \geqslant \frac{1}{b_i}$ и $x_0 \geqslant \frac{1}{1-b_i}$, за $1 \leqslant i \leqslant k$,
3. $a_i > 0$ е константа за $1 \leqslant i \leqslant k$,
4. $b_i \in (0,1)$ е константа за $1 \leqslant i \leqslant k$,
5. $k \geqslant 1$ 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

$$T(x) \asymp x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\, du\right)$$

[†] Книгата на Cormen, Leiserson, Rivest [31] споменава теоремата на Akra-Bazzi, но само в края на главата за МТ (стр. 112–113), без да са решени никакви задачи с нея.

Определение 37: Условието за полиномиалното нарастване от [96]

В контекста на Теорема 29 казваме, че $g(x)$ удовлетворява *условието за полиномиалното нарастване*, ако съществуват положителни константи c_1 и c_2, такива че за всяко $x \geqslant 1$, за всяко $1 \leqslant i \leqslant k$, за всяко $u \in [b_i x, x]$:

$$c_1 g(x) \leqslant g(u) \leqslant c_2 g(x)$$

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 $\alpha x^{-k} \leqslant f(x) \leqslant \beta x^k$ [26, Lemma 2.34], но конверсното не е непременно вярно [26, 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 [26, Lemma 4.1]. Сумата и произведението на функции, удовлетворяващи условието за полиномиалното нарастване, удовлетворяват условието за полиномиалното нарастване [26, Corollary 4.3]. Също така и частното, стига в знаменателя да има положителна функция. Композицията на функции също запазва условието за полиномиалното нарастване [26, 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 29, защото в такъв случай $g(x)$ не удовлетворява условието за полиномиалното нарастване.

Допълнение 24: Доказателство на теоремата на Akra-Bazzi

Лема 21: Lemma 1 от [96]

Ако $g(x)$ е неотрицателна функция, удовлетворяваща условието за полиномиалното нарастване, то съществуват положителни константи c_3 и c_4, такива че за всяко $1 \leqslant i \leqslant k$ и всяко $x \geqslant 1$ е вярно, че

$$c_3 g(x) \leqslant x^p \int_{b_i x}^{x} \frac{g(u)}{u^{p+1}}\, du \leqslant c_4 g(x)$$

Доказателство: От условието за полиномиалното нарастване имаме:

$$\int_{b_i x}^{x} \frac{g(u)}{u^{p+1}}\, du \leqslant x^p (x - b_i x) \frac{c_2 g(x)}{\min\{(b_i x)^{p+1}, x^{p+1}\}}$$
$$= \frac{(1-b_i)c_2 g(x)}{\min\{1, b_i^{p+1}\}}$$
$$\leqslant c_4 g(x)$$

където c_4 е константа, такава че

$$c_4 \geqslant \frac{(1-b_i)c_2}{\min\{1, b_i^{p+1}\}}$$

за $1 \leqslant i \leqslant k$.

Аналогично,

$$\int_{b_i x}^{x} \frac{g(u)}{u^{p+1}}\, du \geqslant x^p (x - b_i x) \frac{c_1 g(x)}{\max\{(b_i x)^{p+1}, x^{p+1}\}}$$
$$= \frac{(1-b_i)c_1 g(x)}{\max\{1, b_i^{p+1}\}}$$
$$\geqslant c_3 g(x)$$

където c_3 е константа, такава че

$$c_3 \leqslant \frac{(1-b_i)c_1}{\min\{1, b_i^{p+1}\}}$$

за $1 \leqslant i \leqslant k$.

□

Ето отново теоремата.

Теорема: Theorem 1 от [96]

Дадено е рекурентното уравнение (3.71)

$$T(x) = \begin{cases} \Theta(1), & \text{ако } 1 \leqslant x \leqslant x_0 \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x), & \text{ако } x > x_0 \end{cases}$$

с ограниченията и имената, посочени след (3.71). 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

$$I_0 = [1, x_0]$$
$$I_j = (x_0 + j - 1, x_0 + j] \quad \text{за } j \geqslant 1$$

Ако $x \in I_j$ за някое $j \geqslant 1$, то за всяко i, такова че $1 \leqslant i \leqslant k$ е вярно, че $b_i x \in I_{j'}$ за някое $j' < j$. Това е така, понеже $b_i x > b_i(x_0 + j - 1) \geqslant b_i x_0 \geqslant 1$, и освен това $b_i x \leqslant b_i(x_0 + j) \leqslant x_0 + j - (1 - b_i)x_0 \leqslant x_0 + j - 1$. 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

$$T(x) \geqslant c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du\right)$$

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:

$$\begin{aligned}
T(x) &= \sum_{i=1}^{k} a_i T(b_i x) + g(x) \quad // \text{ от индуктивното предположение} \\
&\geqslant \sum_{i=1}^{k} a_i c_5 (b_i x)^p \left(1 + \int_1^{b_i x} \frac{g(u)}{u^{p+1}}\,du\right) + g(x) \\
&= c_5 x^p \sum_{i=1}^{k} a_i b_i^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du - \int_{b_i x}^x \frac{g(u)}{u^{p+1}}\,du\right) + g(x) \quad // \text{ от Лема 21} \\
&\geqslant c_5 x^p \sum_{i=1}^{k} a_i b_i^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du - \frac{c_4 g(x)}{x^p}\right) + g(x) \\
&= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du - \frac{c_4 g(x)}{x^p}\right)\left(\sum_{i=1}^{k} a_i b_i^p\right) + g(x) \quad // \text{ защото } \sum_{i=1}^{k} a_i b_i^p = 1 \\
&= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du - \frac{c_4 g(x)}{x^p}\right) + g(x) \\
&= c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du\right) + g(x)(1 - c_4 c_5) \\
&\geqslant c_5 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du\right), \quad \text{ако } (c_4 c_5 < 1)
\end{aligned}$$

Доказателството, че има положителна константа c_6, такава че за всяко $x > x_0$ е вярно, че

$$T(x) \leqslant c_6 x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\,du\right),$$

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 $c_3 c_6 \geqslant 1$.
Това е и края на доказателството. □

Задача 38: от [4]

Решете чрез метода на Akra-Bazzi $T(n) = \frac{4}{3}T\left(\frac{n}{2}\right) + 3T\left(\frac{n}{3}\right) + \frac{16}{3}T\left(\frac{n}{4}\right) + \Theta(n^2 \lg\lg n)$.

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 [96]

Решете чрез метода на Akra-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 Akra-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 30: Решение на линейно хомогенно рекурентно уравнение

Нека редицата $\mathbf{a} = (a_0, a_1, a_2, \ldots)$ е зададена с линейното рекурентно уравнение

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_{r-1} a_{n-(r-1)} + c_r a_{n-r},\ n \geqslant r \tag{3.72}$$

и α_1, α_2, ..., α_s са различните комплексни корени на *характеристичното уравнение*

$$x^r - c_1 x^{r-1} - c_2 x^{r-2} - \cdots - c_{r-1} x - c_r = 0 \tag{3.73}$$

Нека α_i има кратност k_i. Тогава

$$a_n = P_1(n)\alpha_1^n + P_2(n)\alpha_2^n + \cdots + P_s(n)\alpha_s^n \tag{3.74}$$

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 $\mathbf{a}$. □

Забележете, че $k_1 + k_2 + \cdots + k_s = r$, което следва веднага от основната теорема на алгебрата.

В текущата терминология, (3.72) изглежда така:

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \cdots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \tag{3.75}$$

Както вече казахме, това е линейно хомогенно рекурентно уравнение с константни коефициенти и крайна история. 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. Ще разглеждаме нехомогенни рекурентни уравнения от този вид:

$$\begin{aligned} T(n) = {} & c_1 T(n-1) + c_2 T(n-2) + \cdots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \\ & + \beta_1^n Q_1(n) + \beta_2^n Q_2(n) + \cdots + \beta_m^n Q_m(n) \end{aligned} \tag{3.76}$$

β_1, β_2, ..., β_m са две по две различни положителни константи. $Q_1(n)$, $Q_2(n)$, ..., $Q_m(n)$ са полиноми от степени съответно d_1, d_2, ..., d_m, а m е цяла положителна константа.

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 38: Мултимножество: нотация, кратност, обединение, кардиналност

Със нотацията "$\{\quad\}_M$" означаваме мултимножества. Например, $\{1, 1, 2, 3, 3, 3\}_M$ е мултимножеството, състоящо се от две единици, една двойка и две тройки. За всеки елемент a по отношение на някакво мултимножество A, с $\#(a, A)$ означаваме броя на появите на a в A, което 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:

$$A \cup B = \{x \mid (x \in A \text{ или } x \in B) \text{ и } (\#(x, A \cup B) = \#(x, A) + \#(x, B))\}_M$$

Кардиналност на мултимножеството A е сумата от броевете на появите на елементите му и се означава с $|A|$. Например, $|\{1, 1, 2, 3, 3, 3\}_M| = 6$.

Теорема 31: Решение на линейно нехомогенно рекурентно уравнение

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 $1 \leqslant i \leqslant m$. Нека $Y = A \cup B$. Очевидно, $|Y| = r + \sum_{i=1}^{m}(d_i + 1)$. Нека преименуваме различните елементи на Y като $y_1, y_2, \ldots, y_t$ и да дефинираме, че $z_i = \#(y_i, Y)$, за $1 \leqslant i \leqslant t$. Тогава общото решение на (3.76) е:

$$\begin{aligned} T(n) = &\gamma_{1,1}\, y_1^n + \gamma_{1,2}\, n\, y_1^n + \gamma_{1,3}\, n^2\, y_1^n + \cdots + \gamma_{1,z_1}\, n^{z_1-1}\, y_1^n + \\ &\gamma_{2,1}\, y_2^n + \gamma_{2,2}\, n\, y_2^n + \gamma_{2,3}\, n^2\, y_2^n + \cdots + \gamma_{2,z_2}\, n^{z_2-1}\, y_2^n + \\ &\ldots \\ &\gamma_{t,1}\, y_t^n + \gamma_{t,2}\, n\, y_t^n + \gamma_{t,3}\, n^2\, y_t^n + \cdots + \gamma_{t,z_t}\, n^{z_t-1}\, y_t^n, \end{aligned} \tag{3.77}$$

където $\gamma_{i,j}$, за $1 \leqslant i \leqslant t$ и $1 \leqslant j \leqslant z_t$, 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 14: Асимптотиката на общото решение от Теорема 31

Понеже се интересуваме само от 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, а именно, най-голямото y_i, умножено по най-голямата възможна степен на n. Накратко, ако $y_i = \max\{y_1, \ldots, y_t\}$, то

$$T(n) \asymp n^{z_i-1}\, y_i^n \tag{3.78}$$

Забележете, че ако $T(n)$ описва сложността на алгоритъм, най-голямото по абсолютна стойност y_i задължително е положително.

Допълнение 25: Извеждане на решението на линейните рек. у-ния

Ще изведем решението съгласно учебника *An Introduction to Difference Equations* на Elaydi [39, 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 фунции

$$\mathcal{G} = \{g \mid g : \mathbb{N} \to \mathbb{R}\}$$

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 $\mathbf{\Delta}$ (оператор разлика) и $\mathbf{E}$ (оператор изместване):

$$\mathbf{\Delta}x(n) = x(n+1) - x(n) \tag{3.79}$$
$$\mathbf{E}x(n) = x(n+1) \tag{3.80}$$

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	...
$\mathbf{\Delta}x(n)$	1	1	1	1	...	1	...
$\mathbf{E}x(n)$	1	2	3	4	...	$n+1$	...

Въвеждаме и съответните итерираните *итерирани оператори*:

$$\mathbf{\Delta}^k x(n) = \begin{cases} x(n), & \text{ако } k = 0 \\ \mathbf{\Delta}(\mathbf{\Delta}^{k-1}x(n)), & \text{ако } k > 0 \end{cases}$$

$$\mathbf{E}^k x(n) = \begin{cases} x(n), & \text{ако } k = 0 \\ \mathbf{E}(\mathbf{E}^{k-1}x(n)), & \text{ако } k > 0 \end{cases}$$

Ето как изглеждат $\mathbf{\Delta}^2$, $\mathbf{\Delta}^3$, $\mathbf{E}^2$ и $\mathbf{E}^3$ за $x(n) = n$:

n	0	1	2	3	...	n	...
$x(n)$	0	1	2	3	...	n	...
$\mathbf{\Delta}x(n)$	1	1	1	1	...	1	...
$\mathbf{\Delta}^2x(n)$	0	0	0	0	...	0	...
$\mathbf{\Delta}^3x(n)$	0	0	0	0	...	0	...
$\mathbf{E}x(n)$	1	2	3	4	...	$n+1$	...
$\mathbf{E}^2x(n)$	2	3	4	5	...	$n+2$	...
$\mathbf{E}^3x(n)$	3	4	5	6	...	$n+3$	...

В някакъв смисъл, операторът $\mathbf{\Delta}$ е аналог на оператора $\mathbf{D}$ (производна) от математическия анализ:

$$\mathbf{D}f(y) = \lim_{h \to 0} \frac{f(y+h) - f(y)}{h}$$

Също както в континуалния анализ, ако $f(y) = y$, то $\mathbf{D}f(y) = 1$, а $\mathbf{D}^2f(y) = 0$, в дискретния анализ е вярно, че, ако $x(n) = n$, то $\mathbf{\Delta}x(n) = 1$, а $\mathbf{\Delta}^2x(n) = 0$.

И операторът разлика, и операторът изместване са линейни оператори. Ако a и b са

константи, в сила са:

$$\begin{aligned}\mathbf{\Delta}(ax(n) + by(n)) &= a\mathbf{\Delta}x(n) + b\mathbf{\Delta}y(n)\\ \mathbf{E}(ax(n) + by(n)) &= a\mathbf{E}x(n) + b\mathbf{E}y(n)\end{aligned}$$

По дефиниция, $\mathbf{I}$ е операторът идентитет:

$$\mathbf{I}x = x$$

Вярно е, че

$$\begin{aligned}\mathbf{E} &= \mathbf{\Delta} + \mathbf{I}\\ \mathbf{\Delta} &= \mathbf{E} - \mathbf{I}\end{aligned} \tag{3.81}$$

В общия случай, 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 $\mathbf{\Delta}^2$ имаме:

$$\begin{aligned}\mathbf{\Delta}^2 x(n) &= \mathbf{\Delta}(\mathbf{\Delta}x(n))\\ &= \mathbf{\Delta}(x(n + 1) - x(n))\\ &= \mathbf{\Delta}x(n + 1) - \mathbf{\Delta}x(n)\\ &= x(n + 2) - x(n + 1) - x(n + 1) + x(n)\\ &= x(n + 2) - 2x(n + 1) + x(n)\end{aligned}$$

За $\mathbf{\Delta}^3$ имаме:

$$\begin{aligned}\mathbf{\Delta}^3 x(n) &= \mathbf{\Delta}(\mathbf{\Delta}^2 x(n))\\ &= \mathbf{\Delta}(x(n + 2) - 2x(n + 1) + x(n))\\ &= \mathbf{\Delta}x(n + 2) - 2\mathbf{\Delta}x(n + 1) + \mathbf{\Delta}x(n)\\ &= (x(n + 3) - x(n + 2)) - 2(x(n + 2) - x(n + 1)) + (x(n + 1) - x(n))\\ &= x(n + 3) - 3x(n + 2) + 3x(n + 1) - x(n)\end{aligned}$$

Можем да покажем по индукция, че

$$\mathbf{\Delta}^k x(n) = \sum_{i=0}^{k} (-1)^i \binom{k}{i} x(n + k - i)$$

Действието на оператора $\mathbf{\Delta}$ върху полиноми е аналогично на действието на оператора $\mathbf{D}$ от континуалния анализ: ако полиномът е от степен $k \geqslant 1$, то еднократното прилагане на $\mathbf{\Delta}$ "сваля" степента с единица, k-кратното му прилагане дава $a_k k!$, където a_k е коефициентът пред старшия член, а оттам $(k+i)$-кратното му прилагане за $i \geqslant 1$ дава

нула. Наистина, нека

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots a_1 n + a_0$$

където $k \geqslant 1$ и $a_k, \ldots, a_0$ са константи; ерго, $\deg p(n) = k$. Тогава

$$\begin{aligned}
\Delta p(n) &= \big(a_k(n+1)^k + a_{k-1}(n+1)^{k-1} + \cdots + a_1(n+1) + a_0\big) \\
&\quad - \big(a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0\big) = \\
&= \underbrace{a_k n^k + a_k k n^{k-1} + \cdots + a_k}_{a_k(n+1)^k} + \underbrace{a_{k-1}(n+1)^{k-1} + \cdots + a_1(n+1) + a_0}_{\text{от степен } k-1} \\
&\quad - a_k n^k - a_{k-1}n^{k-1} - \cdots - a_1 n - a_0 = \\
&= a_k k n^{k-1} + \underbrace{\cdots + a_k}_{\text{от степен } <k-1} + a_{k-1}(n+1)^{k-1} + \underbrace{\cdots + a_1(n+1) + a_0}_{\text{от степен } <k-1} \\
&\quad - a_{k-1}n^{k-1} - \underbrace{\cdots - a_1 n - a_0}_{\text{от степен } <k-1} = \\
&= a_k k n^{k-1} + \underbrace{\cdots + a_k}_{\text{от степен } <k-1} \\
&\quad + a_{k-1}n^{k-1} + \underbrace{\cdots}_{\text{от степен } <k-1} + \underbrace{\cdots + a_1(n+1) + a_0}_{\text{от степен } <k-1} \\
&\quad - a_{k-1}n^{k-1} - \underbrace{\cdots - a_1 n - a_0}_{\text{от степен } <k-1} = \\
&= a_k k n^{k-1} + \underbrace{\cdots}_{\text{от степен } <k-1}
\end{aligned}$$

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 $k \geqslant 2$. Лесно се вижда, че

$$\Delta^k p(n) = a_k k!$$

и

$$i \geqslant 1 \rightarrow \Delta^{k+i} p(n) = 0 \tag{3.82}$$

Дефинираме *оператор полином* от степен k така. Нека k е константа и е даден следният полином от степен k на $\mathbf{E}$:

$$\mathbf{p}(\mathbf{E}) = a_k \cdot \mathbf{E}^k + a_{k-1} \cdot \mathbf{E}^{k-1} + \cdots + a_1 \cdot \mathbf{E} + a_0 \cdot \mathbf{I}$$

Да разгледаме действието на $\mathbf{p}(\mathbf{E})$ върху $b^n y(n)$, където b е константа, а $y(n)$ е някаква

функция.

$$\begin{aligned}\mathbf{p}(\mathbf{E})b^n y(n) &= (a_k \cdot \mathbf{E}^k + a_{k-1} \cdot \mathbf{E}^{k-1} + \cdots + a_1 \cdot \mathbf{E} + a_0 \cdot \mathbf{I})b^n y(n)\\ &= a_k \mathbf{E}^k b^n y(n) + a_{k-1}\mathbf{E}^{k-1} b^n y(n) + \cdots a_1 \mathbf{E} b^n y(n) + a_0 b^n y(n)\\ &= b^n \big(a_0 b^k y(n+k) + a_1 b^{k-1} y(n+k-1) + \cdots + a_{k-1} b y(n+1) + a_k y(n)\big)\\ &= b^n \mathbf{p}(b \cdot \mathbf{E}) y(n)\end{aligned}$$

И така,

$$\mathbf{p}(\mathbf{E})b^n y(n) = b^n \mathbf{p}(b \cdot \mathbf{E}) y(n) \tag{3.83}$$

Анихилатор за функцията $y(n)$ е оператор полином $\mathbf{N}(\mathbf{E})$, такъв че за всяко n е изпълнено

$$\mathbf{N}(\mathbf{E})y(n) = 0 \tag{3.84}$$

Може да се каже, че $\mathbf{N}(\mathbf{E})$ е анихилатор за $y(n)$, ако $y(n)$ е решение на (3.84). Следват няколко примера.

- Нека $y(n) = 2^n$. Тогава $\mathbf{N}(\mathbf{E}) = \mathbf{E} - 2$ е анихилатор за $y(n)$, защото $(\mathbf{E} - 2)2^n = 2^{n+1} - 2 \cdot 2^n = 0$.
- И изобщо, ако $y(n) = a^n$, то $\mathbf{N}(\mathbf{E}) = \mathbf{E} - a$ е анихилатор за $y(n)$, защото $(\mathbf{E} - a)a^n = a^{n+1} - a \cdot a^n = 0$.
- Нека $y(n) = n2^n$. Тогава $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - 2)^2$ е анихилатор за $y(n)$, защото

$$\begin{aligned}(\mathbf{E} - 2)^2 n 2^n &= (\mathbf{E}^2 - 4\mathbf{E} + 4)n2^n\\ &= (n+2)2^{n+2} - 4(n+1)2^{n+1} + 4n2^n\\ &= 4n2^n + 8 \cdot 2^n - 8n2^n - 8 \cdot 2^n + 4n2^n = 0\end{aligned}$$

- И изобщо, ако $y(n) = na^n$, то $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - a)^2$ е анихилатор за $y(n)$.
- Лесно се вижда, че ако $y(n) = n^2 a^n$, то $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - a)^3$ е анихилатор за $y(n)$.
- И изобщо, ако $y(n) = n^b a^n$, то $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - a)^{b+1}$ е анихилатор за $y(n)$.
- Лесно се вижда, че ако $y(n) = n^{b_1} a_1^n + n^{b_2} a_2^n$, то $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - a_1)^{b_1+1}(\mathbf{E} - a_2)^{b_2+1}$ е анихилатор за $y(n)$.
- И изобщо, ако $y(n) = n^{b_1} a_1^n + \cdots + n^{b_t} a_t^n$, то $\mathbf{N}(\mathbf{E}) = (\mathbf{E} - a_1)^{b_1+1} \cdot \cdots \cdot (\mathbf{E} - a_t)^{b_t+1}$ е анихилатор за $y(n)$.

Второ понятие: фундаментално множество от решения. Разглеждаме общи хомогенни линейни рекурентни уравнения с константни коефициенти и крайна история

$$x(n+k) + c_1 x(n+k-1) + c_2 x(n+k-2) + \cdots + c_{k-1} x(n+1) + c_k x(n) = 0 \tag{3.85}$$

където $c_1, \ldots, c_k$ са реални константи и $c_k \neq 0$. Казваме, че функциите $f_1(n), \ldots, f_r(n)$ са *линейно независими* за $n \geqslant n_0$, ако

$$\forall n \geqslant n_0 : a_1 f_1(n) + \cdots + a_r f_r(n) = 0$$

тстк $a_1 = \cdots = a_r = 0$.

Определение 39

Всяко множество от k линейно независими решения на (3.85) се нарича *фундаментално множество от решения*.

Има сравнително ефикасен начин да се провери дали множество от решения е линейно независимо, тоест, фундаментално. За целта ще ни помогне понятието *детерминанта на Casorati*[a], на английски *Casoratian*, която е дискретния аналог на детерминантата на Wronski (на английски, *Wronskian*) от континуалната математика.

Определение 40

Детерминантата на Casorati за решенията $x_1(n), \ldots, x_r(n)$ е

$$W(n) = \det \begin{bmatrix} x_1(n) & x_2(n) & \cdots & x_r(n) \\ x_1(n+1) & x_2(n+1) & \cdots & x_r(n+1) \\ x_1(n+2) & x_2(n+2) & \cdots & x_r(n+2) \\ \vdots & \vdots & \cdots & \vdots \\ x_1(n+r-1) & x_2(n+r-1) & \cdots & x_r(n+r-1) \end{bmatrix} \tag{3.86}$$

Лема 22: Lemma 2.13 от [39, стр. 68]

Нека $x_1(n), \ldots, x_r(n)$ са решения на (3.85). Нека $W(n)$ е детерминантата на Casorati за тях. Тогава, за всяко $n \geqslant n_0$:

$$W(n) = (-1)^{k(n-n_0)} c_k^{n-n_0} W(n_0).$$

Доказателство: Ще докажем за частния случай $n = 3$. Доказателството за общия случай може да се направи аналогично.
И така, нека $n = 3$. От (3.86) имаме

$$W(n+1) = \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ x_1(n+3) & x_2(n+3) & x_3(n+3) \end{bmatrix} \tag{3.87}$$

От (3.85) имаме:

$$x_1(n+3) = -c_3 x_1(n) - (c_1 x_1(n+2) + c_2 x_1(n+1))$$
$$x_2(n+3) = -c_3 x_2(n) - (c_1 x_2(n+2) + c_2 x_2(n+1))$$
$$x_3(n+3) = -c_3 x_3(n) - (c_1 x_3(n+2) + c_2 x_3(n+1))$$

Заместваме в (3.87) и получаваме

$$W(n+1) = \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ A & B & C \end{bmatrix} \tag{3.88}$$

където

$$\begin{aligned} A &= -c_3 x_1(n) - (c_2 x_1(n+1) + c_1 x_1(n+2)) \\ B &= -c_3 x_2(n) - (c_2 x_2(n+1) + c_1 x_2(n+2)) \\ C &= -c_3 x_3(n) - (c_2 x_3(n+1) + c_1 x_3(n+2)) \end{aligned}$$

Съгласно линейната алгебра, (3.88) може да се запише просто като

$$\begin{aligned} W(n+1) &= \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ -c_3 x_1(n) & -c_3 x_2(n) & -c_3 x_3(n) \end{bmatrix} \\ &= -c_3 \det \begin{bmatrix} x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \\ x_1(n) & x_2(n) & x_3(n) \end{bmatrix} \\ &= -c_3(-1)^2 \det \begin{bmatrix} x_1(n) & x_2(n) & x_3(n) \\ x_1(n+1) & x_2(n+1) & x_3(n+1) \\ x_1(n+2) & x_2(n+2) & x_3(n+2) \end{bmatrix} \end{aligned}$$

Изразихме $W(n+1)$ чрез $W(n)$:

$$W(n+1) = (-1)^3 c_3 W(n) \tag{3.89}$$

което е линейно рекурентно уравнение с една поява вдясно (от първи ред). С обикновено развиване получаваме

$$W(n+1) = (-1)^{3(n-n_0)} c_3^{n-n_0} W(n_0) \tag{3.90}$$

Което е и краят на доказателството за $n = 3$. □

Следствие 15

Тъй като $c_k \neq 0$ по условие, вярно е, че детерминантата на Casorati $W(n)$ е ненулева за всяко $n \geqslant n_0$ тстк детерминантата на Casorati $W(n_0)$ е ненулева. Ерго, или $W(n)$ е тъждествено нула при $n \geqslant n_0$, или никога не е нула при $n \geqslant n_0$.

Ще покажем, че множество от k решения на (3.85) е фундаментално (тоест, линейно независимо) тстк $W(n)$ никога не е нула. Разглеждаме k решения $x_1(n), \ldots, x_k(n)$ на (3.85). Да кажем, че за някакви константи $\alpha_1, \ldots, \alpha_k$ е вярно, че

$$\forall n \geqslant n_0 : \alpha_1 x_1(n) + \cdots + \alpha_k x_k(n) = 0$$

Тогава също така е вярно, че

$$\begin{aligned}
&\alpha_1 x_1(n+1) + \cdots \quad + \alpha_k x_k(n+1) \quad = 0 \\
&\ldots \\
&\alpha_1 x_1(n+k-1) + \cdots + \alpha_k x_k(n+k-1) = 0
\end{aligned}$$

Всичко това може да се запише кратко ето така:

$$X(n)\xi = \mathbf{0} \tag{3.91}$$

където

$$X(n) = \begin{bmatrix} x_1(n) & \cdots & x_k(n) \\ \vdots & \ldots & \vdots \\ x_1(n+k-1) & \cdots & x_k(n+k-1) \end{bmatrix}$$

и

$$\xi = \begin{bmatrix} \alpha_1 \\ \cdots \\ \alpha_k \end{bmatrix}$$

Очевидно $\det X(n) = W(n)$. Съгласно линейната алгебра, (3.91) има само тривиалното решение $\alpha_1 = \cdots = \alpha_k = 0$ тстк $X(n)$ е несингулярна, тоест, $\det X(n) \neq 0$, тоест, $W(n) \neq 0$ за $n \geqslant n_0$. От тези и предишните съображения лесно следва Теорема 32.

Теорема 32: Множество решения е фундаментално тстк дет. на Casorati е ненулева

Множеството $\{x_1(n), \ldots, x_k(n)\}$ от решения на (3.85) е фундаментално тстк за някое $n_0 \in \mathbb{N}^+$ е вярно, че $W(n_0) \neq 0$.

Задача 40

Покажете, че 2^n, $(-2)^n$ и $(-3)^n$ образуват множество от фундаментални решения на

$$x(n+3) + 3x(n+2) - 4x(n+1) - 12x(n) = 0 \tag{3.92}$$

Решение: Лесно може да се убедим, че и трите функции са решения, замествайки в (3.92). Примерно,

$$2^{n+3} + 3 \cdot 2^{n+2} - 4 \cdot 2^{n+1} - 12 \cdot 2^n = 8 \cdot 2^n + 12 \cdot 2^n - 8 \cdot 2^n - 12 \cdot 2^n = 0$$

Сега да се убедим, че те са фундаментални решения. Построяваме детерминантата на

Casorati:

$$W(n) = \det \begin{bmatrix} 2^n & (-2)^n & (-3)^n \\ 2^{n+1} & (-2)^{n+1} & (-3)^{n+1} \\ 2^{n+2} & (-2)^{n+2} & (-3)^{n+2} \end{bmatrix}$$

Тогава

$$W(0) = \det \begin{bmatrix} 2^0 & (-2)^0 & (-3)^0 \\ 2^1 & (-2)^1 & (-3)^1 \\ 2^2 & (-2)^2 & (-3)^2 \end{bmatrix} = \det \begin{bmatrix} 1 & 1 & 1 \\ 2 & -2 & -3 \\ 4 & 4 & 9 \end{bmatrix} = -20 \neq 0.$$

□

Теорема 33: Theorem 2.18 от [39, стр. 72]: съществуване на фунд. множество от решения

Уравнение (3.85) има фундаментално множество от решения за $n \geqslant 0$.

Доказателство: Очевидно (3.85) има "единични" решения $x_1(n)$, $x_2(n)$, ..., $x_k(n)$:

$$\begin{aligned} &x_1(0) = 1, x_1(1) = 0, x_1(2) = 0, \ldots, x_1(k-1) = 0 \\ &x_2(0) = 0, x_2(1) = 1, x_2(2) = 0, \ldots, x_2(k-1) = 0 \\ &x_3(0) = 0, x_3(1) = 0, x_3(2) = 1, \ldots, x_3(k-1) = 0 \\ &\ldots \\ &x_k(0) = 0, x_k(1) = 0, x_k(2) = 0, \ldots, x_k(k-1) = 1 \end{aligned}$$

Веднага се вижда, че $W(0) = 1$. Щом детерминантата на Casorati е ненулева, то от Теорема 32 следва, че множеството от "единичните" решения е фундаментално множество от решения на (3.85). □

Уравнение (3.85) има безброй много фундаментални множества от решения. Сега ще видим как се генерират фундаментални решения от известни решения.

Лема 23

Нека $x_1(n)$ и $x_2(n)$ са решения на (3.85). Тогава всяка линейна комбинация на $x_1(n)$ и $x_2(n)$ е решение на (3.85).

Доказателство: Ако за $n \geqslant k$ имаме

$$\begin{aligned} &x_1(n+k) + c_1 x_1(n+k-1) + c_2 x_1(n+k-2) + \cdots + c_{k-1} x_1(n+1) + c_k x_1(n) = 0 \\ &x_2(n+k) + c_1 x_2(n+k-1) + c_2 x_2(n+k-2) + \cdots + c_{k-1} x_2(n+1) + c_k x_2(n) = 0 \end{aligned}$$

то очевидно

$$\begin{aligned} &\big(x_1(n+k) + x_2(n+k)\big) + c_1\big(x_1(n+k-1) + x_2(n+k-1)\big) + \cdots + \\ &\qquad c_{k-1}\big(x_1(n+1) + x_2(n+1)\big) + c_k\big(x_1(n) + x_2(n)\big) = 0 \\ &\alpha x_1(n+k) + \alpha c_1 x_1(n+k-1) + \cdots + \alpha c_{k-1} x_2(n+1) + \alpha p_k x_2(n) = 0 \end{aligned}$$

за произволна константа α. □

Чрез Лема 23 лесно можем да докажем следния важен резултат.

Теорема 34: Принцип на суперпозицията

Ако x_1, x_2, ..., $x_r(n)$ са решения на (3.85), то

$$x(n) = \alpha_1 x_1(n) + \alpha_2 x_2(n) + \cdots + \alpha_r x_r(n)$$

също е решение на на (3.85), където α_1, α_2, ..., α_r са произволни константи. □

Теорема 35: Фундаментално множество от решения и конкретно решение

Нека $\{x_1(n), \ldots, x_k(n)\}$ е фундаментално множество от решения на (3.85). Тогава за всяко конкретно решение $x(n)$ на (3.85) съществуват константи α_1, α_2, ..., α_k, такива че

$$x(n) = \sum_{i=1}^{k} \alpha_i x_i(n)$$

Доказателство: Ползвайки нотацията на (3.91), записваме

$$X(n)\xi = \widehat{x}(n)$$

където

$$\widehat{x}(n) = \begin{bmatrix} x(n) \\ x(n+1) \\ \vdots \\ x(n+k-1) \end{bmatrix}$$

Тъй като $X(n)$ е инвертируема, то

$$\xi = X^{-1}(n)\widehat{x}(n)$$

В частност,

$$\xi = X^{-1}(n_0)\widehat{x}(n_0)$$

и

$$\xi = X^{-1}(0)\widehat{x}(0)$$

□

Теорема 35 ни дава право да направим следното определение.

Определение 41: Общото решение на хомогенно линейно рекурентно уравнение

Нека $\{x_1(n), \ldots, x_k(n)\}$ е фундаментално множество от решения на (3.85). Тогава *общото решение* на (3.85) е

$$x(n) = \sum_{i=1}^{k} \alpha_i x_i(n)$$

за произволни константи α_1, α_2, ..., α_k.

Всяко (конкретно) решение на (3.85) се получава от общото с подходящ избор на константи α_1, α_2, ..., α_k.

Теорема 36: Решенията на хомогенно лин. рек. у-ние образуват линейно пространство

Нека S е множеството от всички решения на (3.85). 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 23. Базис може да е фундаменталното множество от решения от Теорема 33. □

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 (3.85). Ще намерим фундаментално решение и оттам ще намерим общото решение.

Допускаме, че решенията на (3.85) са от вида λ^n, където λ е комплексно число. Замествайки в (3.85), получаваме

$$\lambda^k + c_1\lambda^{k-1} + c_2\lambda^{k-2} + \cdots + c_{k-1}\lambda + c_k = 0 \qquad (3.93)$$

Уравнение (3.93) е *характеристичното уравнение* на рекурентното уравнение (3.85). Съгласно основната теорема на алгебрата, (3.93) 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 тях[б]. Нека характеристичните корени са λ_1, ..., λ_k.

Случай i Характеристичните корени са два по два различни. Ще покажем, че в този случай множеството $\{\lambda_1^n, \ldots, \lambda_k^n\}$ е фундаментално множество от решения. За да

се убедим, че е така, да разгледаме детерминантата на Casorati $W(0)$:

$$W(0) = \det \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 & \cdots & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \lambda_3^2 & \cdots & \lambda_k^2 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \lambda_1^{k-1} & \lambda_2^{k-1} & \lambda_3^{k-1} & \cdots & \lambda_k^{k-1} \end{bmatrix} \quad (3.94)$$

Това е пример за детерминанта на Vandermonde, за която има добре известно решение. А именно,

$$W(0) = \prod_{1 \leqslant i < j \leqslant k} (\lambda_i - \lambda_j)$$

Щом $\lambda_i \neq \lambda_j$ при $i \neq j$, очевидно $W(0) \neq 0$. Следователно, $\{\lambda_1^n, \lambda_2^n, \ldots, \lambda_k^n\}$ е фундаментално множество от решения на (3.85). Оттук следва, че общото решение е

$$x(n) = \sum_{i=1}^{k} \alpha_i \lambda_i^n$$

От гледна точка на анализа на алгоритми по-точно решение не ни интересува. БОО, нека най-голямата от λ_i е λ_1, като това е и най-голямата по абсолютна стойност ламбда. Тогава решение е $T(n) \asymp \lambda_1^n$.

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 λ_1, λ_2, ..., λ_r с кратности съответно m_1, m_2, ..., m_r. Очевидно $\sum_{i=1}^{r} m_i = k$.
Можем да запишем (3.85) като

$$(\mathsf{E}^k + c_1\mathsf{E}^{k-1} + c_2\mathsf{E}^{k-2} + \cdots + c_{k-1}\mathsf{E} + c_k\mathbf{I})x(n) = 0$$

което можем да запишем като

$$(\mathsf{E} - \lambda_1)^{m_1}(\mathsf{E} - \lambda_2)^{m_2} \cdot \cdots \cdot (\mathsf{E} - \lambda_r)^{m_r} x(n) = 0 \quad (3.95)$$

факторизирайки $\boxed{\mathsf{E}^k + c_1\mathsf{E}^{k-1} + \cdots + c_k}$ до $\boxed{(\mathsf{E} - \lambda_1)^{m_1}(\mathsf{E} - \lambda_2)^{m_2} \cdot \cdots \cdot (\mathsf{E} - \lambda_r)^{m_r}}$.
Да разгледаме само

$$(\mathsf{E} - \lambda_i)^{m_i} x(n) = 0 \quad (3.96)$$

за кое да е $i \in \{1, 2, \ldots, r\}$. Твърдим, че всяка функция $h(n)$, която е решение на (3.96), е решение и на (3.85). Наистина, ако $h(n)$ е такова решение, то $(\mathsf{E} - \lambda_i)^{m_i} h(n) = 0$.

Тогава

$$\begin{aligned}
&(E-\lambda_1)^{m_1}(E-\lambda_2)^{m_2}\cdot\dots\cdot(E-\lambda_r)^{m_r}h(n) = \\
&(E-\lambda_1)^{m_1}\cdot\dots\cdot(E-\lambda_{i-1})^{m_{i-1}} \\
&\quad (E-\lambda_{i+1})^{m_{i+1}}\cdot\dots\cdot(E-\lambda_r)^{m_r}\underbrace{(E-\lambda_i)^{m_i}h(n)}_{0} = \\
&(E-\lambda_1)^{m_1}\cdot\dots\cdot(E-\lambda_{i-1})^{m_{i-1}} \\
&\quad (E-\lambda_{i+1})^{m_{i+1}}\cdot\dots\cdot(E-\lambda_r)^{m_r}0 = 0
\end{aligned}$$

Да допуснем, че сме намерили множество от фундаментални решения за уравненията $(E-\lambda_i)^{m_i}x(n)$, за всяко $i \in \{1, 2, \dots, r\}$. Ще покажем, че тяхното обединение е множество от фундаментални решения за (3.95).

Лема 24

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 (3.96).

Доказателство: Съгласно Следствие 15, достатъчно е да покажем, че $W(0) \neq 0$, където $W(0)$ е следната детерминантата на Casorati:

$$\begin{vmatrix}
\lambda_i^0 & \binom{0}{1}\lambda_i^{0-1} & \binom{0}{2}\lambda_i^{0-2} & \dots & \binom{0}{m_i-2}\lambda_i^{0-m_i+2} & \binom{0}{m_i-1}\lambda_i^{0-m_i+1} \\
\lambda_i^1 & \binom{1}{1}\lambda_i^{1-1} & \binom{1}{2}\lambda_i^{1-2} & \dots & \binom{1}{m_i-2}\lambda_i^{1-m_i+2} & \binom{1}{m_i-1}\lambda_i^{1-m_i+1} \\
\lambda_i^2 & \binom{2}{1}\lambda_i^{2-1} & \binom{2}{2}\lambda_i^{2-2} & \dots & \binom{2}{m_i-2}\lambda_i^{2-m_i+2} & \binom{2}{m_i-1}\lambda_i^{2-m_i+1} \\
\vdots & \vdots & \vdots & \dots & \vdots & \vdots \\
\lambda_i^{m_i-2} & \binom{m_i-2}{1}\lambda_i^{m_i-2-1} & \binom{m_i-2}{2}\lambda_i^{m_i-2-2} & \dots & \binom{m_i-2}{m_i-2}\lambda_i^{m_i-2-m_i+2} & \binom{m_i-2}{m_i-1}\lambda_i^{m_i-2-m_i+1} \\
\lambda_i^{m_i-1} & \binom{m_i-1}{1}\lambda_i^{m_i-1-1} & \binom{m_i-1}{2}\lambda_i^{m_i-1-2} & \dots & \binom{m_i-1}{m_i-2}\lambda_i^{m_i-1-m_i+2} & \binom{m_i-1}{m_i-1}\lambda_i^{m_i-1-m_i+1}
\end{vmatrix}$$

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:

$$\begin{vmatrix}
1 & 0 & 0 & \dots & 0 & 0 \\
\lambda_i^1 & 1 & 0 & \dots & 0 & 0 \\
\lambda_i^2 & \binom{2}{1}\lambda_i^{2-1} & 1 & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \dots & \vdots & \vdots \\
\lambda_i^{m_i-2} & \binom{m_i-2}{1}\lambda_i^{m_i-2-1} & \binom{m_i-2}{2}\lambda_i^{m_i-2-2} & \dots & 1 & 0 \\
\lambda_i^{m_i-1} & \binom{m_i-1}{1}\lambda_i^{m_i-1-1} & \binom{m_i-1}{2}\lambda_i^{m_i-1-2} & \dots & \binom{m_i-1}{m_i-2}\lambda_i^{m_i-1-m_i+2} & 1
\end{vmatrix}$$

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 $G_i = \left\{\lambda_i^n, \binom{n}{1}\lambda_i^{n-1}, \binom{n}{2}\lambda_i^{n-2}, \ldots, \binom{n}{m_i-1}\lambda_i^{n-m_i+1}\right\}$ е фундаментално множество от решения.
Остава да покажем, че $\binom{n}{t}\lambda_i^{n-t}$ е решение на (3.96) за всяко $t \in \{0, 1, \ldots, m_i - 1\}$. Наистина, имайки предвид, че $\binom{n}{t}$ е полином на n от степен t, в сила е:

$$
\begin{aligned}
&(\mathbf{E} - \lambda_i)^{m_i}\binom{n}{t}\lambda_i^{n-t} = && \text{// съгласно (3.83)}\\
&\lambda_i^{n-t}(\lambda_i\mathbf{E} - \lambda_i)^{m_i}\binom{n}{t} = \\
&\lambda_i^{n-t+m_i}(\mathbf{E} - \mathbf{I})^{m_i}\binom{n}{t} = && \text{// съгласно (3.81)}\\
&\lambda_i^{n-t+m_i}\mathbf{\Delta}^{m_i}\binom{n}{t} = && \text{// понеже } t < m_i \text{ и съгласно (3.82)}\\
&0
\end{aligned}
$$

□

Теорема 37: Theorem 2.23 от [39, стр. 77]: фунд. м-во от решения при кратни корени

Множеството $G = \bigcup_{i=1}^{r} G_i$ е фундаментално множество от решения на (3.95).

Доказателство: Съгласно Лема 24, функциите от G са решения на (3.95). 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. [39, стр. 77]), да разгледаме конкретен пример. Да кажем, че $k = 6$, $r = 2$, $m_1 = 4$ и $m_2 = 2$. Тогава

$$
W(0) = \det \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0\\
\lambda_1 & 1 & 0 & 0 & \lambda_2 & 1\\
\lambda_1^2 & 2\lambda_1 & 1 & 0 & \lambda_2^2 & 2\lambda_2\\
\lambda_1^3 & 3\lambda_1^2 & 3\lambda_1^1 & 1 & \lambda_2^3 & 3\lambda_2^2\\
\lambda_1^4 & 4\lambda_1^3 & 6\lambda_1^2 & 4\lambda_1 & \lambda_2^4 & 4\lambda_2^3\\
\lambda_1^5 & 5\lambda_1^4 & 10\lambda_1^3 & 10\lambda_1^2 & \lambda_2^5 & 5\lambda_2^4
\end{bmatrix} \tag{3.97}
$$

Сравнете (3.97) с детерминантата от (3.94). В (3.94) на всяко λ_i съответства точно една колона, в която, отгоре надолу, са записани степените на това λ_i, започвайки от нула. В (3.97), на всяко λ_i съответстват точно m_i колони, оформени като блок; в (3.97) 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 извес-

тно решение [80]. А именно,

$$W(0) = \prod_{1 \leqslant i < j \leqslant r} (\lambda_i - \lambda_j)^{m_i \cdot m_j}$$

Тъй като $\lambda_i \neq \lambda_j$ при $i \neq j$, то $W(0) \neq 0$. Тогава G е множество от фундаментални решения на (3.95). □

Следствие 16 на Теорема 37 е математическа обосновка на метода с характеристичното уравнение, що се отнася до хомогенните уравнения.

Следствие 16

Общото решение на (3.95) се дава от

$$x(n) = \sum_{i=1}^{r} \left(\alpha_{i,0} + \alpha_{i,1} n + \cdots + \alpha_{i,m_i-1} n^{m_i-1}\right) \lambda_i^n$$

където $\alpha_{i,j}$ за $1 \leqslant i \leqslant r$, $0 \leqslant j \leqslant m_i - 1$ са константи.

Извеждане на решението за нехомогенните уравнения. За съжаление, Следствие 16 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 (3.85) не може да описва сложност на алгоритъм. Сега разглеждаме съответно нехомогенно уравнение в общ вид:

$$x(n+k) + c_1 x(n+k-1) + c_2 x(n+k-2) + \cdots + c_{k-1} x(n+1) + c_k x(n) = g(n) \quad (3.98)$$

където $g(n)$ е нехомогенната част. *Съответното хомогенно уравнение* е

$$x(n+k) + c_1 x(n+k-1) + c_2 x(n+k-2) + \cdots + c_{k-1} x(n+1) + c_k x(n) = 0 \quad (3.99)$$

Забележете, че то е същото като (3.85); тук е записано отново за прегледност.

Задача 41

Разгледайте нехомогенното уравнение

$$x(n+2) - x(n+1) - 6x(n) = 5 \cdot 3^n \tag{3.100}$$

1. Намерете общото решение на съответното хомогенно уравнение.
2. Проверете дали 3^n и $(-2)^n$ са решения на (3.100).
3. Проверете дали $x_1(n) = n3^{n-1}$ и $x_2(n) = (1+n)3^{n-1}$ са решения на (3.100).
4. Покажете, че $x_2(n) - x_1(n)$ не е решение на (3.100).
5. Покажете, че не е вярно, че $c \cdot x_1(n)$ за всяка константа c е решение на (3.100).

Решение: По 1: съответното хомогенно уравнение е

$$x(n+2) - x(n+1) - 6x(n) = 0 \tag{3.101}$$

Характеристичното уравнение е

$$\lambda^2 - \lambda - 6 = 0$$

Корените му са $\lambda_1 = 3$ и $\lambda_2 = -2$. Общото му решение, съгласно Следствие 16, е

$$x(n) = \alpha_1 3^n + \alpha_2 (-2)^n$$

Естествено, 3^n и $(-2)^n$ са решения на (3.101):

$$3^{n+2} - 3^{n+1} - 6 \cdot 3^n = 9 \cdot 3^n - 3 \cdot 3^n - 6 \cdot 3^n = 0$$
$$(-2)^{n+2} - (-2)^{n+1} - 6 \cdot (-2)^n = 4 \cdot (-2)^n - (-2)(-2)^n - 6 \cdot (-2)^n = 0$$

По 2: нито 3^n, нито $(-2)^n$ са решения на (3.100). Току-що видяхме, че заместването на всяко от тях в лявата страна на (3.100) дава нула, а дясната страна на (3.100) не е нула.

По 3: $n3^{n-1}$ е решение. Заместваме $x(n)$ с $n3^{n-1}$ в лявата страна на (3.100):

$$\begin{aligned}(n+2)3^{n+1} - (n+1)3^n - 6n3^{n-1} &= 3^{n-1}\big((2+n)3^2 - (1+n)3^1 - 6n\big)\\ &= 3^{n-1}\big(18 + 9n - 3 - 3n - 6n\big)\\ &= 3^{n-1}\big(15\big)\\ &= 5 \cdot 3^n\end{aligned}$$

$(1+n)3^{n-1}$ също е решение. Пак заместваме $x(n)$ вляво, този път с $(1+n)3^{n-1}$:

$$\begin{aligned}(1+n+2)3^{n+1} - (1+n+1)3^n - 6(1+n)3^{n-1} &= 3^{n-1}\big((3+n)3^2 - (2+n)3^1 - 6(1+n)\big)\\ &= 3^{n-1}\big(27+9n-6-3n-6-6n\big)\\ &= 3^{n-1}(15)\big)\\ &= 5\cdot 3^n\end{aligned}$$

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 (3.100):

$$\begin{aligned}c(n+2)3^{n+1} - c(n+1)3^n - 6cn3^{n-1} &= c3^{n-1}\big((2+n)3^2 - (1+n)3^1 - 6n\big)\\ &= c3^{n-1}\big(18+9n-3-3n-6n\big)\\ &= c3^{n-1}\big(15\big)\\ &= 5c3^n\end{aligned}$$

Ако $c \neq 1$, то $5c3^n$ не е същото като $5 \cdot 3^n$. □

Задача 41 показва, че, за разлика от хомогенните уравнения (вж. Теорема 36), 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 41 ни навежда и на мисълта, че разликата от две решения на нехомогенното уравнение (3.98) е решение на съответното хомогенно уравнение (3.99). Ще оформим това в отделна теорема, въпреки че доказателството е твърде просто.

Теорема 38: Разликата м-у две реш. на нехомогенното е реш. на хомогенното у-ние

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 (3.98). Тогава

$$\begin{aligned}x_1(n+k) + c_1x_1(n+k-1) + \cdots + c_{k-1}x_1(n+1) + c_kx_1(n) = g(n)\\ x_2(n+k) + c_1x_2(n+k-1) + \cdots + c_{k-1}x_2(n+1) + c_kx_2(n) = g(n)\end{aligned}$$

Изваждаме втория ред от първи и получаваме:

$$\begin{aligned}&\big(x_1(n+k) - x_2(n+k)\big) + c_1\big(x_1(n+k-1) - x_2(n+k-1)\big) + \cdots +\\ &\qquad c_{k-1}\big(x_1(n+1) - x_2(n+1)\big) + c_k\big(x_1(n) - x_2(n)\big) = 0\end{aligned}$$

Виждаме, че $x_1(n) - x_2(n)$ е решение на (3.99). □

Да кажем, че общото решение на хомогенното (3.99) е $x_h(n)$. Да кажем, че $x_p(n)$ означава кое е да е решение[a] на нехомогенното (3.98). Ще наричаме $x_p(n)$, "частно решение".

Следната теорема ни казва как може да конструираме всички решения на нехомогенното (3.98).

Теорема 39: Theorem 2.40 от [39, стр. 84]: общото решение на нехомогенното уравнение

Всяко решение на нехомогенното (3.98) може да бъде записано като

$$x(n) = x_p(n) + \sum_{i=1}^{k} \alpha_i x_i(n)$$

където $\{x_1(n), \ldots, x_k(n)\}$ е фундаментално множество от решения на хомогенното (3.99), а $x_p(n)$ е частно решение.

Доказателство: Според Теорема 39, разликата $x(n) - x_p(n)$ е решение на хомогенното (3.99). Съгласно Теорема 35, това решение е от вида $\sum_{i=1}^{k} \alpha_i x_i(n)$, където $x_1(n)$, ..., $x_k(n)$ са елементите на някое фундаментално множество от решения на въпросното хомогенно уравнение. □

Теорема 39 ни дава право да направим следното определение.

Определение 42: Общото решение на нехомогенно линейно рекурентно уравнение

Нека $x_h(n)$ е общото решение на хомогенното уравнение (3.99). Общото решение на нехомогенното уравнение (3.98) е $x_h(n) + x_p(n)$, където $x_p(n)$ е частно решение на (3.98).

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* ([39, 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 (3.76) нехомогенната част е точно такава.

Да запишем (3.98) така:

$$\mathbf{p}(\mathbf{E})x(n) = g(n) \tag{3.102}$$

където $\mathbf{p}(\mathbf{E}) = \mathbf{E}^k + c_1\mathbf{E}^{k-1} + c_2\mathbf{E}^{k-2} + \cdots + c_{k-1}\mathbf{E} + c_k\mathbf{I}$. Нека $\mathbf{N}(\mathbf{E})$ е анихилатор за $g(n)$ от (3.102). Тогава прилагаме анихилатора върху двете страни на уравнението и получаваме

$$\mathbf{N}(\mathbf{E})\,\mathbf{p}(\mathbf{E})\,x(n) = 0 \tag{3.103}$$

Тъй като търсим само асимптотиката на решението, а не истинското решение с точните константи, (3.103) е напълно достатъчно. В [39, 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 (3.103) по начин, напълно аналогичен на начина, по който третирахме (3.95). Уравнение (3.95) бе получено от хомогенното (3.85), прилагайки оператора полином, факторизиран по характеристичните корени. Уравнение (3.103) бе получено от нехомогенното (3.98). Очевидно лявата страна на (3.103) може да бъде факторизирана по корените на алгебрично уравнение, което се получава от композицията на оператора полином, който "идва" от хомогенната част, и анихилатора, 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 $\mathbf{N}(\mathbf{E})x(n) = 0$.

Да разгледаме отново примера (3.100). Съотвентото хомогенно е $x(n+2) - x(n+1) - 6x(n) = 0$ с мултимножество от корените $\{3, -2\}$, така че операторът полином е $\mathbf{p}(\mathbf{E}) = (\mathbf{E}-3)(\mathbf{E}+2)$. Анихилаторът на $5 \cdot 3^n$ е $\mathbf{E} - 3$ (петицата е без значение за анихилатора и оттам, за общото решение). Тогава операторът от дясната страна на (3.103) е $(\mathbf{E} - 3)^2(\mathbf{E}+2)$. Очевидно е в какъв смисъл коренът 3, "идващ" от дясната страна чрез $(\mathbf{E}-3)x(n) = 0$, "влиза" в мултимножеството от корените. Асимптотиката на решението е $\Theta(n3^n)$.

---

[a] Felice Casorati е италиански математик.

[б] Читателят може да възрази, че в общия случай алгебрични уравнения от степен $\geqslant 5$ 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 $\geqslant 5$, 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 $\mathsf{T}(n) \asymp \lambda^n$, очевидно никоя апроксимация $\widetilde{\lambda}$, която е различна от λ, няма да даде правилната асимптотика на решението, понеже или $\lambda^n \prec \widetilde{\lambda}^n$, или $\lambda^n \succ \widetilde{\lambda}^n$.

Но това е само от гледна точка на теоретичен перфекционизъм. На практика, намирането на $\widetilde{\lambda}$ 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 събираимо в израза за $\mathsf{T}(n)$ 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.

[в] В книгата на Elaydi [39, стр. 76] има очевидна грешка в крайния етап на извеждането на $W(0)$.

[г] Субскриптът "p" в "$x_p(n)$" идва от "particular".

Ще разгледаме три примера за решаване на рекурентни уравнения с метода с характеристичното уравнение.

Задача 42

Решете рекурентното уравнение

$$\mathsf{T}(n) = \mathsf{T}(n-1) + 1$$

Решение: Преписваме рекурентното уравнение така: $\mathsf{T}(n) = \mathsf{T}(n-1) + 1^n n^0$, за да сме

сигурни, че фóрмата му удовлетворява (3.76). 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 (3.76), $m = 1$, $\beta_1 = 1$, 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 $T(n) = \Theta(n)$. □

Задача 43

Решете рекурентното уравнение

$$T(n) = 4T(n-3) + 1 \tag{3.104}$$

Решение: Характеристичното уравнение е

$$x^3 - 4 = 0$$

Корените му са

$$\begin{aligned} x_1 &= \sqrt[3]{4} \\ x_2 &= \sqrt[3]{4}\,e^{i\frac{2\pi}{3}} \\ x_3 &= \sqrt[3]{4}\,e^{i\frac{-2\pi}{3}} \end{aligned}$$

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:

$$\begin{aligned} T(n) &= A\left(\sqrt[3]{4}\right)^n + B\left(\sqrt[3]{4}\right)^n e^{\frac{2\pi n i}{3}} + C\left(\sqrt[3]{4}\right)^n e^{\frac{-2\pi n i}{3}} + D1^n = \\ &= A\left(\sqrt[3]{4}\right)^n + B\left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{2\pi n}{3}\right) + i\sin\left(\frac{2\pi n}{3}\right)\right) + \\ &\quad C\left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{-2\pi n}{3}\right) + i\sin\left(\frac{-2\pi n}{3}\right)\right) + D \\ &= A\left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2\pi n}{3}\right)(B + C) + \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2\pi n}{3}\right)(B - C)i + D \end{aligned} \tag{3.105}$$

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 задават цели положителни числа на T. Причината е много проста: няма как (3.104) да "произведе" дори едно число, което не е цяло положително, ако стартира с цяли положителни числи. И така, от (3.105) извеждаме веднага, че $T(n) \asymp \left(\sqrt[3]{4}\right)^n$. □

Допълнение 26: Когато има комплексни характеристични корени

Разсъждаваме върху точното решение на

$$T(n) = 4T(n-3) + 1$$

имайки предвид общото решение (3.105).

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 23), ако имаме линейно рекурентно уравнение и знаем, че някакви функции g_i, където $1 \leqslant i \leqslant k$, са решения, то всяка тяхна линейна комбинация също е решение. Тогава имаме общо решение

$$T(n) = A_1\left(\sqrt[3]{4}\right)^n + A_2\left(\sqrt[3]{4}\right)^n \cos\left(\frac{2\pi n}{3}\right) + A_3\left(\sqrt[3]{4}\right)^n \sin\left(\frac{2\pi n}{3}\right) + A_4 \qquad (3.106)$$

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 (3.104) 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 числа, то константите $A_1, \ldots, A_4$ в (3.106) трябва да са такива, че $T(n)$ е цяло число за всяко $n \in \mathbb{N}^+$.

Първо да разгледаме общото решение (3.106) 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 (3.106). 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

(3.106) като за **три** израза:

$$T(n) = \begin{cases} A_1\left(\sqrt[3]{4}\right)^n - A_2\frac{\left(\sqrt[3]{4}\right)^n}{2} + A_3\frac{\left(\sqrt[3]{4}\right)^n\sqrt{3}}{2} + A_4, & \text{ако } n \equiv 1 \mod 3 \\ A_1\left(\sqrt[3]{4}\right)^n - A_2\frac{\left(\sqrt[3]{4}\right)^n}{2} - A_3\frac{\left(\sqrt[3]{4}\right)^n\sqrt{3}}{2} + A_4, & \text{ако } n \equiv 2 \mod 3 \\ A_1\left(\sqrt[3]{4}\right)^n + A_2\left(\sqrt[3]{4}\right)^n + A_4, & \text{ако } n \equiv 0 \mod 3 \end{cases} \tag{3.107}$$

За да намерим $A_1, \ldots, A_4$, в (3.106) последователно заместваме n с 1, 2, 3 и 4, като за $T(1), \ldots, T(4)$ ползваме началните условия, и така получаваме следната система:

$$\begin{aligned} 1 &= A_1\,4^{1/3} - A_2\frac{4^{1/3}}{2} + A_3\frac{4^{1/3}\sqrt{3}}{2} + A_4 \\ 2 &= A_1\,4^{2/3} - A_2\frac{4^{2/3}}{2} - A_3\frac{4^{2/3}\sqrt{3}}{2} + A_4 \\ 3 &= A_1 + A_2\,4 + A_4 \\ 4 &= A_1\,4^{4/3} - A_2\,2\cdot 4^{1/3} + A_3\,2\sqrt{3}\,4^{1/3} + A_4 \end{aligned}$$

Решението на системата е

$$\begin{aligned} A_1 &= \frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6} \\ A_2 &= -\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2} \\ A_3 &= \frac{4^{2/3}\sqrt{3}}{12} - \frac{4^{1/3}\sqrt{3}}{6} \\ A_4 &= 0 \end{aligned}$$

Тогава точното решение е

$$\begin{aligned} T(n) = &\left(\frac{1}{4} + \frac{4^{2/3}}{12} + \frac{4^{1/3}}{6}\right)4^{n/3} + \\ &\left(-\frac{4^{2/3}}{12} - \frac{4^{1/3}}{6} + \frac{1}{2}\right)4^{n/3}\cos(2\pi n/3) + \\ &\left(\frac{4^{2/3}\sqrt{3}}{12} - \frac{4^{1/3}\sqrt{3}}{6}\right)4^{n/3}\sin(2\pi n/3) \end{aligned} \tag{3.108}$$

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 (3.108) като за три отделни израза, всеки с константни коефициенти – също както (3.107).

Задача 44

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 $\{1,1,1,1,2,2,2,2,3,4,4,5,5\}_M$. Най-големият елемент е 5 с кратност 2. Решението е $T(n) \asymp n5^n$. □

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 17.

Изч. Задача 9: Сортиране

екземпляр: Редица $X = (a_1, a_2, \ldots, a_n)$, чиито елементи са от някакво множество A, което е подходящо за сортиране и върху които е дефинирана релация на пълна преднаредба $\lessdot \subseteq A \times A$.
решение: пермутация $(a'_1, a'_2, \ldots, a'_n)$ на елементите на X, такава че $a'_1 \lessdot a'_2 \lessdot \cdots \lessdot a'_n$.

Подчертаваме, че екземплярът на задачата е **редицата** $(a_1, a_2, \ldots, a_n)$, а не (мулти)множеството $\{a_1, a_2, \ldots, a_n\}_M$.

Елементите a_1, ..., 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 $\lessdot$ 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 $\lessdot$ ще бъде добре познатата "по-малко или равно" $\leqslant$.

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 $\lessdot$ да е дефинирана смислено.

на долни граници (Глава 13). Примерно, в [112, стр. 6] задачата е дефинирана по този начин. Да се ограничим само върху реални числа.

Изч. Задача 10: Сортиране, абстрактна дефиниция

екземпляр: Мултимножество $\{a_1, a_2, \ldots, a_n\}_M$ от реални числа.
решение: За всяка двойка (i, j), такава че $1 \leqslant i < j \leqslant n$, отговор дали $a_i < a_j$.

Изглежда, че авторът на [112] 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 [112, 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 настрани 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 фиксираме 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 [133, стр. 106].

Наблюдение 26

Предимството на сортираните данни над несортираните се корени в транзитивността на релацията $\prec$. Ако редицата $(a_1, a_2, \ldots, a_n)$ е сортирана и е известно, че $a_i \prec x$ за някое i и някакво x, то от транзитивността на $\prec$ заключаваме, че $a_j \prec x$ за всяко $j < i$.

4.2.1 Двоично търсене

В простия си вариант като задача за разпознаване, Търсене се дефинира така.

Изч. Задача 11: Търсене, версия за разпознаване

екземпляр: $a_1, a_2, \ldots, a_n$, key: обекти от един и същи тип.
въпрос: Дали има елемент измежду $a_1, a_2, \ldots, a_n$, 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 12: Търсене, версия за търсене

екземпляр: $a_1, a_2, \ldots, a_n$, key: обекти от един и същи тип.
решение: Ако има елемент измежду $a_1, a_2, \ldots, a_n$, който е равен на key, индексът на един такъв елемент. В противен случай индикация, че такъв елемент няма.

Ако входът на произволен алгоритъм за Търсене е произволна редица $a_1, a_2, \ldots, a_n$, то единствената възможност за действие е проверяване дали $a_i \stackrel{?}{=} \text{key}$ по всички $i \in \{1, 2, \ldots, n\}$. Това се нарича *последователно търсене*. Най-лошият случай по отношение на сложността е, key да не е нито един от $a_1, a_2, \ldots, a_n$. Тогава всеки алгоритъм би трябвало да направи n сравнения в най-лошия 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 $a_1, a_2, \ldots, a_n$ 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.

```
BinSearchIter(A[1..n], key)
 1  (* итеративен вариант *)
 2  (* A е сортиран *)
 3  ℓ ← 1
 4  h ← n
 5  while ℓ ≤ h do
 6      mid ← ⌊(ℓ+h)/2⌋
 7      if A[mid] = key
 8          return mid
 9      else if key < A[mid]
10          h ← mid − 1
11      else
12          ℓ ← mid + 1
13  return −1
```

```
BinSearchRec(A[ℓ..h], key)
 1  (* рекурсивен вариант *)
 2  (* A е сортиран *)
 3  if ℓ > h
 4      return −1
 5  else
 6      mid ← ⌊(ℓ+h)/2⌋
 7      if A[mid] = key
 8          return mid
 9      else if key < A[mid]
10          return
11              BinSearchRec(A[ℓ..mid − 1], key)
12      else
13          return
14              BinSearchRec(A[mid + 1..h], key)
```

Началното викане на рекурсивната версия е BinSearchRec($A[1..n]$, key). Посочените два алгоритъма решават Търсене във версия за търсене. Накратко, тяхната коректност се обосновава така: ако изобщо елементът key се намира в масива A, той се намира в подмасива $A[\ell..h]$. Това твърдение остава вярно след промените в индексите ℓ и h само защото масивът е сортиран предварително. Пълна обосновка има в Допълнение 27.

Сложността по време и на двата алгоритъма е $\Theta(\lg n)$ в най-лошия случай, което изведохме много подробно за итеративната версия в Подсекция 2.4.3. Нещо повече, точната сложност като брой сравнения $A[mid] \stackrel{?}{=} key$ е $\lfloor \lg n \rfloor + 1$ в най-лошия 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 с 7 000 000 000!

Допълнение 27: Коректността на двоичното търсене

Първо ще докажем коректността на итеративната версия.

Теорема 40: Коректността на BinSearchIter

За всяко изпълнение на BinSearchIter($A[1\,..\,n]$, key), ако key се среща в A, то BinSearchIter връща индекса на key в A, в противен случай BinSearchIter връща -1.

Доказателство: Следното твърдение е инвариант за **while**-цикъла на редове 5–12.

Инвариант 4: Цикълът на BinSearchIter

Всеки път, когато изпълнението на BinSearchIter е на ред 5, ако е вярно, че

① $A\left[\left\lfloor \frac{\ell+h}{2} \right\rfloor\right] \neq key$ и

② $\ell \leq h$

то, ако $key \in A[1\,..\,n]$, то $key \in A[\ell\,..\,h]$.

База. Разглеждаме първия път, когато изпълнението достигне ред 5. Заради присвояванията на редове 3 и 4, съждението "ако $key \in A[1\,..\,n]$, то $key \in A[\ell\,..\,h]$" е тавтология. Базата е вярна независимо от това дали ① е вярно или не.

Поддръжка. Да допуснем, че изпълнението е на ред 5 и ① и ② са в сила и, ако key се среща в $A[1\,..\,n]$, то key непременно се намира в $A[\ell\,..\,h]$; казано по друг начин, ако key се среща в $A[1\,..\,n]$, то key не може да е нито в $A[1\,..\,\ell-1]$, нито в $A[h+1\,..\,n]$. Това че ① и ② са в сила влече, че тялото на цикъла ще се изпълни поне още веднъж, като изпълнението пак ще е на ред 5 и по този начин ще се убедим, че инвариантът се запазва.
На ред 6, на mid се присвоява $\left\lfloor \frac{\ell+h}{2} \right\rfloor$. Съгласно допускането ①, $A[mid] \neq key$, така че следните два случая са изчерпателни.

Случай I $key < A[mid]$. Щом $key < A[mid]$, то $key < A[j]$ за всяко $j \in \{mid+1, mid+2, \ldots, n\}$; това е заради транзитивността на релацията $<$ и факта, че $A[1\,..\,n]$ е сортиран. Но тогава key не може да се намира в $A[mid\,..\,h]$, така че, ако key се среща в A, key се намира в $A[\ell\,..\,mid-1]$.
Очевидно условието на ред 9 е истина и на ред 10, на h се присвоява $mid - 1$. Спрямо новото h е вярно, че при следващото достигане на ред 5, ако $key \in A[1\,..\,n]$, то $key \in A[\ell\,..\,h]$. С други думи, инвариантът се запазва.
Това, че инвариантът е импликация, не води до проблеми. Ако някое от ① или ② е лъжа, импликацията е истина, тъй като антецедентът ѝ е лъжа.

Случай II $\mathsf{key} > \mathsf{A}[\mathrm{mid}]$. Щом $\mathsf{key} > \mathsf{A}[\mathrm{mid}]$, то $\mathsf{key} > \mathsf{A}[j]$ за всяко $j \in \{\ell, \ell+1, \ldots, \mathrm{mid}-1\}$; това е заради транзитивността на релацията $>$ и факта, че $\mathsf{A}[1..n]$ е сортиран. Но тогава key не може да се намира в $\mathsf{A}[\ell..\mathrm{mid}-1]$, така че, ако key се среща в A, key се намира в $\mathsf{A}[\mathrm{mid}+1..h]$.
Очевидно условието на ред 9 е лъжа и на ред 12, на ℓ се присвоява $\mathrm{mid}+1$. Спрямо новото ℓ е вярно, че при следващото достигане на ред 5, ако $\mathsf{key} \in \mathsf{A}[1..n]$, то $\mathsf{key} \in \mathsf{A}[\ell..h]$. С други думи, инвариантът се запазва.

Терминация. Да допуснем, че изпълнението е на ред 5 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 $\mathsf{A}[1..n]$, то key се намира в $\mathsf{A}[\ell..h]$. В този случай обаче $\mathsf{A}[\ell..h]$ е празният масив, така че входният масив не съдържа key. От друга страна, изпълнението отива на ред 13, така че алгоритъмът коректно индикира отсъствието на key във входния масив, връщайки -1.

2. $\mathsf{A}\left[\left\lfloor \frac{\ell+h}{2} \right\rfloor\right] = \mathsf{key}$. Тогава очевидно изпълнението отива на ред 8 и алгоритъмът коректно връща индекса на key във входния масив. □

Сега ще докажем коректността на рекурсивната версия.

Теорема 41: Коректността на BinSearchRec

За всяко изпълнение на BinSearchRec($\mathsf{A}[\ell..h]$, key), ако key се среща в A, то BinSearchRec връща индекса на key в A, в противен случай BinSearchRec 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 3, а самата база е ред 4. Наистина, ако ℓ е по-голямо от h, масивът $\mathsf{A}[\ell..h]$ е празен, така че key не се среща в него и алгоритъмът връща -1 на ред 4.

Разглеждаме работата на алгоритъма на дадено ниво в рекурсията.

Индуктивно предположение. Допускаме, че ако бъде направено кое да е от рекурсивните викания на ред 10 или ред 13, 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 $\mathsf{A}[\mathrm{mid}] = \mathsf{key}$. В този случай, от една страна е вярно, че key се среща в масива, а от друга страна на ред 8, алгоритъмът връща индекса на key.

Случай II $\mathsf{key} < \mathsf{A}[\mathrm{mid}]$. В този случай, $\mathsf{key} < \mathsf{A}[j]$ за всяко $j \in \{\mathrm{mid}+1, \mathrm{mid}+2, \ldots, h\}$; това е заради транзитивността на релацията $<$ и факта, че $\mathsf{A}[1..n]$ е сортиран. Но тогава key не може да се намира в $\mathsf{A}[\mathrm{mid}..h]$, така че, ако key се среща в $\mathsf{A}[\ell..h]$, key се намира в $\mathsf{A}[\ell..\mathrm{mid}-1]$.

Булевото условие на ред 9 е истина и рекурсивното викане на ред 10 се извършва. Съгласно индуктивното предположение, викането върху $A[\ell \mathinner{..} \mathrm{mid}-1]$ на ред 10 работи коректно, откъдето следва, че викането върху $A[\ell \mathinner{..} h]$ работи коректно.

Случай III $\mathrm{key} > A[\mathrm{mid}]$. В този случай, $\mathrm{key} > A[j]$ за всяко $j \in \{\ell, \ell+1, \ldots, \mathrm{mid}-1\}$; това е заради транзитивността на релацията $<$ и факта, че $A[1 \mathinner{..} n]$ е сортиран. Но тогава key не може да се намира в $A[\ell \mathinner{..} \mathrm{mid}]$, така че, ако key се среща в $A[\ell \mathinner{..} h]$, key се намира в $A[\mathrm{mid}+1 \mathinner{..} h]$.
Булевото условие на ред 9 е лъжа, поради което се извършва рекурсивното викане на ред 13. Съгласно индуктивното предположение, викането върху $A[\mathrm{mid}+1 \mathinner{..} h]$ на ред 13 работи коректно, откъдето следва, че викането върху $A[\ell \mathinner{..} h]$ работи коректно. □

Следствие 17

Началното викане BINSEARCHREC($A[1 \mathinner{..} n]$, key) работи коректно: ако key се намира в $A[1 \mathinner{..} n]$, 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 изражда 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 13: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ОЛЕКОТНИЯТ ВАРИАНТ

екземпляр: $a_1, a_2, \ldots, a_n$: цели числа; $n \geqslant 2$
решение: $\min \{|a_i - a_j| : 1 \leqslant i < j \leqslant n\}$

Изч. Задача 14: НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, ТЕЖКИЯТ ВАРИАНТ

екземпляр: $a_1, a_2, \ldots, a_n$: цели числа; $n \geqslant 2$
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 двуелементи 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 кои да два елемента от входа е δ, то след сортирането задържително има съседни елементи, разстоянието между които е δ; в противен случай би имало два елемента от входа, разстоянието между които е дори по-малко.

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

Изч. Задача 15: Уникалност на елементите (Element Uniqueness / EU)

екземпляр: $a_1, a_2, \ldots, a_n$: цели числа; $n \geqslant 2$.
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 ← 1 to n − 1
 2      for j ← 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 ← 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 TAOCP на Knuth [86, 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. Интересен е случаят, в който $m \asymp n$. Тогава наивният подход води до квадратичен алгоритъм. Следният алгоритъм е по-добър.

```
ALG MATCHING IN ARRAYS(A[1..n], B[1..m])
 1  SORT(A)
 2  SORT(B)
 3  i ← 1, j ← 1
 4  while i ⩽ n and j ⩽ 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 е 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 срещана**та** 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 ← m + 1
 8          else
 9              s ← s + 1
10              if s > m
11                  mode ← A[i]
12                  m ← s
13      else
14          s ← 1
15  return mode
```

Допълнение 28: Коректността на COMPUTE MODE

Както вече казахме, модата не е непременно уникална. Алгоритъмът връща *най-малката мода*. За краткост, навсякъде до края на Допълнение 28, когато казваме "мода" имаме предвид най-малката мода.

Следното определение е в сила **само в рамките на това доказателство**.

Определение 43: 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 кой да е от нейните елементи.
По отношение на даден индекс $i \in \{2, 3, \ldots, n\}$, 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 $A[1..i]$. □

Забележете, че ако масивът е сортиран и $A[i]$ започва нова група, то $A[i]$ не може да

е мода при $i \geqslant 2$.

Наблюдение 27

Нека $A[1..n]$ 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 28

За всяко $i \in \{2, 3, \ldots, n\}$, има точно четири, две по две изключващи се, възможности за $A[i]$. Ще бележим тези възможности с буквите $Q_1, \ldots, Q_4$.

Q_1: $A[i]$ разширява група-мода.

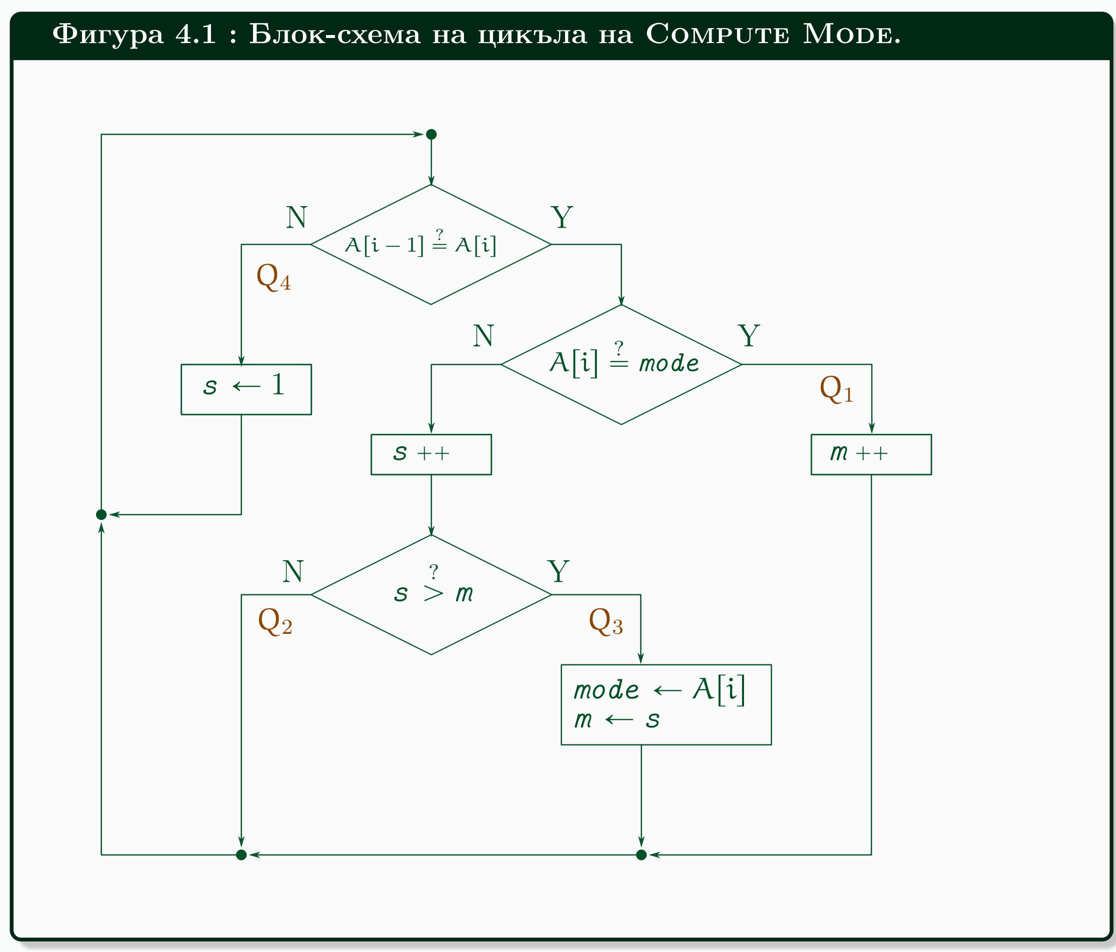
Q_2: $A[i]$ разширява група-немода.

Q_3: $A[i]$ превръща група-немода в група-мода.

Q_4: $A[i]$ започва нова група.

Удобно е да мислим за $Q_1, \ldots, Q_4$ 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 състояние 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 5: Compute Mode

При всяко достигане на ред 4 на Compute Mode:

① $mode$ съдържа стойността на модата на $A[1..i-1]$.

② m съдържа броя на нейните появи в $A[1..i-1]$.

③ Ако $A[i-1]$ не е мода на $A[1..i-1]$, то s съдържа дължината на най-дясната група в $A[1..i-1]$.

База. Нека изпълнението е на ред 4 за първи път. Тогава $i = 2$ и подмасивът $A[1..i-1]$ 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 $A[1..i-1]$. 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 $A[1..i-1]$. След инкрементирането на *s* на ред 9 новата стойност на *s* съдържа дължината на най-дясната група в $A[1..i]$, понеже $A[i-1] = A[i]$. Изпълнението отива на ред 10.

Случай I.2.а $A[i]$ не е мода на $A[1..i]$. В текущия контекст, това е същото като да кажем, че дължината на най-дясната група на $A[1..i]$ не е по-голяма от дължина от броя на появите на модата на $A[1..i]$. Но видяхме, че дължината на най-дясната група на $A[1..i]$ се съдържа в променливата *s*. От друга страна, броя на появите на модата на $A[1..i]$ се съдържа в променлива *m* от индуктивното предположение. Заключаваме, че $s \leqslant m$, а **Случай I.2.а** отговаря на Q_2: $A[i]$ разширява група-немода.
Щом $s \leqslant m$, 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 $A[1..i-1]$, а *m* съдържа броя на появите ѝ. Показахме, че части ① и ② на инварианта се запазват.

Случай I.2.б $A[i]$ е мода на $A[1..i]$. В текущия контекст, това е същото като да кажем, че най-дясната група на $A[1..i]$, чиято дължина се съдържа в променливата *s*, има по-голяма дължина от броя на появите на модата на $A[1..i-1]$, който брой се съдържа в променлива *m* (от индуктивното предположение). Тогава $s > m$, а *s* съдържа броя на появите на модата на $A[1..i]$. И така, $A[i-1]$ не е мода на $A[1..i-1]$ в **Случай I.2**, но същата стойност е мода на $A[1..i]$, понеже $A[i]$ е равна на $A[i-1]$ в **Случай I**. Заключаваме, че **Случай I.2.б** 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 $A[1..i]$, а *m* съдържа броя на появите ѝ. После променливата i се инкрементира. Спрямо новата ѝ стойност, *mode* съдържа модата на $A[1..i-1]$, 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 $A[1..i]$. Следователно, модата на $A[1..i]$ е същата като на $A[1..i-1]$, и броят на появите ѝ в $A[1..i]$ е същият като в $A[1..i-1]$. Съгласно индуктивното предположение, *mode* съдържа модата на $A[1..i-1]$, а *m* съдържа броя на появите ѝ в $A[1..i-1]$. Заключаваме, че в този момент *mode* съдържа модата на $A[1..i]$, а *m* съдържа броя на появите ѝ в $A[1..i]$.
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 $A[1..i-1]$. ✓

② *m* съдържа броя на появите на модата на $A[1..i-1]$. ✓

③ От една страна, $A[i-1]$ не е мода на $A[1..i-1]$, от друга страна най-дясната група на $A[1..i-1]$ 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, ..., 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 44: 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 [31, стр. 213].

Като изчислителна задача това е задача за търсене.

Изч. Задача 16: Медиана

екземпляр: a_1, a_2, ..., a_n: цели числа.
решение: медианата на a_1, a_2, ..., 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 последователност 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]
```

Заслужава да се отбележи, че медианата може да бъде изчислена в $\Theta(n)$ 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 [31], 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 17: kSUM, версия за разпознаване

екземпляр: Множество от реални числа S.
въпрос: Дали има $a_1, \ldots, a_k \in S$, такива че $a_1 + \cdots + a_k = 0$?

Естествено обобщение е следната задача за търсене.

Изч. Задача 18: 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 $\Theta(n^k)$[†], 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 ALG2SUM. 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.

ALG2SUM 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 ALG3SUMCOUNT 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

$$a_1 + a_2 + a_3 = m \leftrightarrow (a_1 - m/3) + (a_2 - m/3) + (a_3 - m/3) = 0$$

Дали става дума за различни елементи от S? Тук имаме предвид задачата във версия за разпознаване. Възниква въпросът, дали $a_1, \dots, a_k$ са непременно два по два различни елементи на S. Това **не е** същият въпрос като въпросът дали S е мултимножество или обикновено множество. Дори S да е обикновено множество, ако мислим за $a_1, \dots, a_k$ като за променливи, изразът "$a_1, \dots, a_k \in S$" казва, че те вземат стойности от S и нищо не пречи няколко или дори всички от тях да имат една и съща стойност. При тази интерпретация, ако S съдържа 0, то отговорът на въпроса в kSUM е тривиално ДА, понеже, ако $a_1 = \dots = a_k = 0$, очевидно $a_1 + \dots + a_k = 0$. Примерно, при $k = 3$, ако настояваме $a_1 \neq a_2 \neq a_3 \neq a_1$, то отговорът за екземпляра $\{0, 1, 2\}$ е НЕ, но без това ограничение отговорът е ДА заради възможността $a_1 = a_2 = a_3 = 0$.

На пръв поглед е много важно да се уточни дали въпросните a_i са уникални или не, защото това променя задачата. По-внимателно разглеждане показва, че същината на задачата е при уникални a_i в смисъл, че дори да допускаме различни a_i-та да имат като стойност един и същи елемент от S, това не добавя значителна сложност към алгоритъма, който дава отговора. Примерно:

[†] Сложността на наивния алгоритъм е $\Theta(n^k)$ само ако смятаме, че k е константа. В противен случай сложността е $\Omega\left(k\binom{n}{k}\right)$, понеже подмножествата с мощност k са точно $\binom{n}{k}$, а всяко от тях бива тествано във време $\Theta(k)$.

- Ако $k = 2$, да разрешим a_1 и a_2 да съдържат един и същи елемент от S може да даде възможност за различен отговор (ДА вместо НЕ) само ако S съдържа 0. А това дали S съдържа 0 може да се провери в линейно време, така че асимптотиката на алгоритъма не се променя.

- Ако $k = 3$, да разрешим a_1, a_2 и a_3 да не са уникални елементи от S може да даде възможност за различен отговор (ДА вместо НЕ) в следните случаи:

  - S съдържа 0 и тогава отговорът е тривиално ДА.
  - S съдържа $c \neq 0$ и $d \neq 0$, като $c = -d/2$. Тогава отговорът е ДА, понеже за $a_1 = d$ и $a_2 = a_3 = c$ е вярно, че $a_1 + a_2 + a_3 = 0$. Наличието на елементи с обратни знаци, единият от които по абсолютна стойност е половината от другия, може да се установи в линейно време от алгоритъм, подобен на ALG2SUM, ако числата са сортирани предварително.

Ако S е мултимножество? Тук допускаме, че $a_1, \ldots, a_k$ са два по два различни елементи от входа, но входът може да е мултимножество. Пояснение: ако, примерно, мултимножеството е $\{s_1, s_2, s_3\}_M$ и $s_1 = s_2 = 0$, а $s_3 = 1$, то s_1 и s_2 са различни елементи, въпреки че имат една и съща стойност; ерго, по отношение на 3SUM във версия за разпознаване, отговорът за екземпляра $\{0, 0, 1\}_M$ е НЕ[†].

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 ALG3SUMCOUNT от Подподсекция 2.1.3.5 не работи върху мултимножества: при всяко откриване на триада на ред 6 алгоритъмът инкрементира 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 ALG3SUMCOUNT 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.

Заслужава да се отбележи, че 3SUM във версия за търсене може да се реши и с квадратичен алгоритъм дори върху екземпляри-мултимножества, ако решението е само бройката на триадите, а не множеството от триадите. 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 ALG3SUMCOUNT с незначителната разлика, че при откриване на триада икнрементираме d така: $\boxed{d \leftarrow d + \alpha \cdot \beta \cdot \gamma}$, където α е броят срещания на $A[i]$, β е броят срещания на $A[p]$, а γ е броят срещания на $A[q]$.

Този трик обаче открива само броя на триадите, в които няма повторения. Към този брой трябва да добавим следните две числа.

[†] Ако обаче разрешавахме $a_1, \ldots, a_k$ да не са непременно два по два различни елементи от входа, то отговорът за $\{0, 0, 1\}_M$ щеше да е ДА.

- 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 $|s| = |t|/2$. Ако 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. Откриването на тези двойки може да стане в линейно време върху сортирания, кондензиран масив с идея, подобна на идеята на ALG2SUM.

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 [50]:

- 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 транслации 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 [86]. Knuth означава с N броя на елементите във входа, а сортираната пермутация с “$K_{p(1)} \leqslant K_{p(2)} \leqslant \cdots \leqslant K_{p(N)}$”.

Задача 45: задача 2 на стр. 5 в [86]

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:

$$K_{p(1)} = \cdots = K_{p(i_1)} < K_{p(i_1+1)} = \cdots = K_{p(i_2)} < \cdots < K_{p(i_{n-1}+1)} = \cdots = K_{p(i_n)},$$

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 последователност 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 коректна лексикографска сортировка, разбира се, искаме началото да бъде

$$(1, 2), (1, 4), \ldots$$

но може да се получи и

$$(1, 4), (1, 2), \ldots$$

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 45: 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 клочюве, които елементи са все пак различими (заради, примерно, вторични ключове), този, който е вляво от другия **преди** сортирането, задължително ще е вляво от него и **след** сортирането.

Наблюдение 29

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 45. 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.

[a] Това 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 картоиграч 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 подпоследователността 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

```
INSERTION SORT(A[1..n]: array of integers)
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 ← j − 1
7      A[j + 1] ← 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 29: 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 25 и Теорема 42 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 $A[1..i-1]$ се състои от същите елементи като входния подмасив $A[1..i-1]$, но в сортиран вид
- и подмасивът $A[i..n]$ е същият като входния $A[i..n]$[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 $A[1..i]$ се състои от същите елементи като входния $A[1..i]$, но в сортиран вид,
- и подмасивът $A[i+1..n]$ е същият като входния $A[i+1..n]$.

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 $A[1..i-1]$ е сортиран и се състои точно от елементите, които са били на позиции $1, \ldots, i-1$ във входа. В този момент $A[i]$ може да "не си е на мястото" в $A[1..i]$, защото $A[i]$ може да е по-малък от някакви елементи на $A[1..i-1]$. 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. И така, в t_2:

- $A[1..j]$ е същият като $A[1..j]$ в t_1 и се състои точно от тези елементи на $A[1..i-1]$ в t_1, които са по-малки или равни на *key*, в сортиран вид;
- $A[j+2..i]$ е същият като $A[j+1..i-1]$ в t_1 и се състои точно от тези елементи на $A[1..i-1]$ в 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 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 $A[1..i]$, в момента t_2, $j+1$ е "правилното място" на елемента, който се е намирал в $A[i]$ в t_1. "Правилното място" е по отношение на елементите, които са по-малки или равни на него. Те се намират в $A[1..j]$ в t_1 и не се местят от вътрешния цикъл[б]. Вътрешният цикъл пресмята това $j+1$, като освен това премества с единица надясно точно тези елементи на $A[1..i]$, които са по-големи от елемента $A[i]$ в 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':

- $A[j + 2 \,..\, i]$ е сортиран и всеки елемент от него е по-голям от *key* и
- $A[1 \,..\, j]$ е сортиран и е същият като в t_1 и
- клетка $A[j + 1]$ може да се счита за празна

то в t'':

- $A[j + 1 \,..\, i]$ е сортиран и всеки елемент от него е по-голям от *key* и
- $A[1 \,..\, j - 1]$ 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.

[а] Това е абсолютно очевидно от кода на алгоритъма, но за пълнота го формулираме отделно.

[б] Забележете, че $j + 1$ е правилното място за $A[i]$ именно по отношение на елементите, които са по-малки или равни на на него. По отношение на елементите, които са по-големи от него, правилното място е j, а не $j + 1$. Но това няма значение, защото сме избрали да местим елементите, които са по-големи от $A[i]$, а да оставяме на място тези, които са по-малки или равни.

Навсякъде в доказателствата долу допускаме, че $n > 1$. Използваме израза "клетка номер x на масива Y може да бъде смятана за свободна" – това означава, че съдържанието ѝ е запазено другаде и може да бъде презаписана, без първоначално намиращият се там елемент да бъде унищожен безвъзвратно.

Лема 25: Коректността на вътрешния цикъл на Insertion Sort

Спрямо <u>**едно единствено**</u> изпълнение на **for** цикъла на Insertion Sort (редове 1–7), нека наричаме масива A непосредствено преди изпълнението на **while** цикъла (редове 4–6) с името A'. Да допуснем, че подмасивът $A'[1 \,..\, i-1]$ е сортиран. Тогава в момента на приключване на **while** цикъла е вярно, че:

① $A[1 \,..\, j]$ се състои точно от елементите на $A'[1 \,..\, i - 1]$, които са по-малки или равни на *key*, в сортиран вид;

② $A[j + 2 \,..\, i]$ се състои точно от елементите на $A'[1 \,..\, i - 1]$, които са по-големи от *key*, в сортиран вид;

③ клетката $A[j + 1]$ е свободна;

④ $A[i + 1 \,..\, n]$ е същият като $A'[i + 1 \,..\, n]$.

Доказателство: Следното твърдение е инвариант за **while** цикъла.

Инвариант 6: Вътрешният цикъл на Insertion Sort

Всеки път, когато изпълнението е на ред 4:

❶ $A[1..j] = A'[1..j]$;

❷ $A[j+2..i] = A'[j+1..i-1]$;

❸ $\forall x \in A[j+2..i] : x > key$;

❹ $A[i+1..n] = A'[i+1..n]$.

С учебна цел, да повторим нещо, което вече видяхме на стр. 47. Доказателството на инварианта е доказателство по индукция. Започва с ***база****, в която верността на предиката просто се проверява. В обикновените доказателства по индукция следващите две фази са: индуктивно предположение и индуктивна стъпка. Тук ги сливаме в една фаза от доказателството, която наричаме* ***поддръжка****. И накрая има последна фаза* ***терминация****, която няма еквивалент при "обикновените" доказателства по индукция, които са върху* безкрайни *индуктивни дефинирани множества. При доказателствата за коректност на алгоритми винаги доказваме твърдение върху* крайно *индуктивно генерирано множество, тъй като алгоритмите винаги терминират. Терминацията се отнася до последното достигане на условието на цикъла – когато тялото няма да се изпълни нито веднъж повече. Когато заместим стойността на управляващата променлива от този момент в инварианта, трябва да получим точно това твърдение, което ни трябва за доказателство на коректността.*

База. Ще покажем ❶. Когато изпълнението достигне до ред 4 за първи път, текущият масив A е A' по определение, така че наистина $A[1..j] = A'[1..j]$.

Ще покажем ❷. Когато изпълнението достигне до ред 4 за първи път, $j = i - 1$ заради присвояването на ред 3. Тогава $A[j+2..i]$ всъщност е $A[i+1..i]$, който подмасив е празен, а $A'[j+1..i-1]$ всъщност е $A'[i..i-1]$, който подмасив също е празен.

❸ е изпълнено, защото $A[j+2..i]$ е празен, а за всеки елемент на празното множество е изпълнен всеки предикат. ❹ е очевидно.

Поддръжка. Да допуснем, че твърдението е в сила в даден момент t, 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 t. Нека t_{next} означава момента на следващото достигане на ред 4.

Ще покажем, че ❶ е в сила в t_{next}. Първо разглеждаме присвояването на ред 5. То не променя нищо в подмасива $A[1..j-1]$ и той остава равен на $A'[1..j-1]$. Сега разглеждаме декрементирането на ред 6. Непосредствено след него, спрямо новата стойност на j, същият този подмасив може да бъде означен с "$A[1..j]$", а също така и с "$A'[1..j]$". И така, подмасивът $A[1..j]$ в t_{next} е същият като $A'[1..j]$ и е сортиран.

Ще покажем, че ❷ е в сила в t_{next}. Първо разглеждаме присвояването на ред 5. От индуктивното предположение ❷ знаем, че $A[j+2..i] = A'[j+1..i-1]$. От индуктивното предположение ❶ следва, че $A[j] = A'[j]$. Тогава след $A[j+1] \leftarrow A[j]$ на ред 5 е вярно, че $A[j+1..i] = A'[j..i-1]$. Обаче подмасивът $A'[j..i-1]$ е сортиран по допускане, така че текущият подмасив $A[j+1..i]$ също е сортиран. Сега разглеждаме декрементирането на ред 6. Непосредствено след него, спрямо новата стойност на j е вярно, че $A[j+2..i] = A'[j+1..i-1]$ и това е сортиран подмасив. В t_{next} това остава вярно.

Ще покажем, че ❸ е в сила в t_{next}. Допуснали сме, че $\forall x \in A[j+2..i] : x > key$ в t. Знаем, че $A[j] > key$, иначе тази итерация на **while**-а не би се изпълнявала. Първо разглеждаме присвояването на ред 5. Непосредствено след него е вярно, че $\forall x \in A[j+1..i] : x > key$. Сега разглеждаме декрементирането на ред 6. След него, спрямо новата стойност на j е вярно, че $\forall x \in A[j+2..i] : x > key$ при следващото достигане на ред 4.

Това, че ❹ е в сила при следващото достигане на ред 4, е очевидно.

Терминация. Да разгледаме момента, в който изпълнението е на ред 4 и условието там е FALSE. Тогава $j \leqslant 0$ или $key \geqslant A[j]$.

Случай 1: $j \leqslant 0$. Тъй като j намалява с единица, то не може да стане отрицателно, следователно $j = 0$. Заместваме j с 0 в инварианта и получаваме:

❶ $A[1..0] = A'[1..0]$;

❷ $A[2..i] = A'[1..i-1]$;

❸ $\forall x \in A[2..i] : x > key$;

❹ $A[i+1..n] = A'[i+1..n]$.

Очевидно в този случай в $A'[1..i-1]$ няма елементи, по-малки или равни на key. Но тогава е вярно, че

① $A[1..0]$ се състои точно от елементите на $A'[1..i-1]$, които са по-малки или равни на key, в сортиран вид; // става дума за празен подмасив

② $A[2..i]$ се състои точно от елементите на $A'[1..i-1]$, които са по-големи от key, в сортиран вид; // става дума за целия $A'[1..i-1]$

③ клетката $A[1]$ е свободна;

④ $A[i+1..n]$ е същият като $A'[i+1..n]$.

Което е точно твърдението на лемата при $j = 0$.

Случай 2: Сега да допуснем, че $j > 0$. Тогава $key \geqslant A[j]$. Да разгледаме отново инварианта. Той е:

❶ $A[1..j] = A'[1..j]$;

❷ $A[j+2..i] = A'[j+1..i-1]$;

❸ $\forall x \in A[j+2..i] : x > key$;

❹ $A[i+1..n] = A'[i+1..n]$.

От ❶ следва, че $A[j] = A'[j]$. Но тогава $key \geqslant A'[j]$. Щом $A[1..j] = A'[1..j]$ и $A[1..i-1]$ е сортиран, то $A[1..j]$ е сортиран. Щом $A[1..j]$ е сортиран и $key \geqslant A'[j]$, то $\forall x \in A[1..j] : x \leqslant key$. От това и ❷ и ❸ следва, че

① $A[1..j]$ се състои точно от елементите на $A'[1..i-1]$, които са по-малки или равни на key, в сортиран вид.

Да разгледаме пак ❷ и ❸. От тях и ① и фактът, че $A'[1..i-1]$ е сортиран следва, че

② $A[j+2\,..\,i]$ се състои точно от елементите на $A'[1\,..\,i-1]$, които са по-големи от *key*, в сортиран вид.

От ❶, ❷ и присвояването на ред 2 следва, че

③ клетката $A[j+1]$ е свободна.

Това, че ④ е вярно, е очевидно. Доказахме лемата. □

Теорема 42: Коректността на INSERTION SORT

INSERTION SORT е сортиращ алгоритъм.

Доказателство: Да наречем входа A''. Следното твърдение е инвариант за **for** цикъла:

Инвариант 7: Външният цикъл на INSERTION SORT

Всеки път, когато изпълнението на INSERTION SORT е на ред 1, текущият подмасив $A[1\,..\,i-1]$ се състои точно от същите елементи като $A''[1\,..\,i-1]$, но в сортиран вид. Освен това, текущият подмасив $A[i\,..\,n]$ се състои от точно от същите елементи като $A''[i\,..\,n]$.

База. Първият път, когато изпълнението достигне ред 1 е вярно, че $i = 2$. Текущият подмасив $A[1\,..\,1]$ се състои от един единствен елемент $A''[1]$ и е тривиално сортиран. Освен това, текущият подмасив $A[2\,..\,n]$ се състои от точно от същите елементи като $A''[2\,..\,n]$. ✓

Поддръжка. Да допуснем, че твърдението е изпълнено при някое достигане на ред 1 и **for** цикълът ще бъде изпълнен още един път. Нека $\tilde{A}$ е името на A при **това достигане** на ред 1. Индуктивното допускане, изразено чрез $\tilde{A}$, е:

$\tilde{A}[1\,..\,i-1]$ се състои точно от същите елементи като $A''[1\,..\,i-1]$, но в сортиран вид, и $\tilde{A}[i\,..\,n]$ се състои от точно от същите елементи като $A''[i\,..\,n]$.

Изпълнява се ред 2 и сега *key* съдържа $\tilde{A}[i]$. Съгласно индуктивното допускане, това е еквивалентно на твърдението, че сега *key* съдържа $A''[i]$. Следва $j \leftarrow i-1$. После се изпълнява **while**-цикълът. Изпълнението на **while**-цикъла терминира. В момента, в който **while** цикълът терминира, съгласно Лема 25 са в сила следните твърдения:

① $A[1\,..\,j]$ се състои точно от елементите на $\tilde{A}[1\,..\,i-1]$, които са по-малки или равни на *key*, в сортиран вид;

② $A[j+2\,..\,i]$ се състои точно от елементите на $\tilde{A}[j+1\,..\,i-1]$, които са по-големи от *key*, в сортиран вид;

③ клетката $A[j+1]$ е свободна;

④ $A[i+1\,..\,n]$ е същият като $\tilde{A}[i+1\,..\,n]$.

Имайки предвид индуктивното предположение, тези твърдения стават:

① $A[1\,..\,j]$ се състои точно от елементите на $A''[1\,..\,i-1]$, които са по-малки или равни на *key*, в сортиран вид;

② $A[j+2\,..\,i]$ се състои точно от елементите на $A''[1\,..\,i-1]$, които са по-големи от *key*, в сортиран вид;

③ клетката $A[j+1]$ е свободна;

④ $A[i+1..n]$ е същият като $A''[i+1..n]$.

Тъй като в *key* вече се намира съдържанието на $A''[i]$, можем да твърдим, че в момента след присвояването на ред 7, текущият $A[1..i]$ се състои от същите елементи като $A''[1..i]$, но в сортиран вид. Изпълнението отива отново на ред 1 и i се инкрементира. Спрямо новата стойност на i е вярно, че:

> текущият $A[1..i]$ се състои от същите елементи като $\widetilde{A}[1..i]$, а $A[1..i-1]$ се състои от същите елементи като $A''[1..i-1]$.

И така, инвариантът се запазва.

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 $A''[1..(n+1)-1]$, но в сортиран вид". Накракто, "текущият подмасив $A[1..n]$ се състои от същите елементи като $A''[1..n]$, но в сортиран вид". □

Сложност по време на Insertion Sort Вече показахме в (2.10) на стр. 80, 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 $\geqslant$ на това място, алгоритъмът нямаше да е стабилен.

4.4.2 Selection Sort

Да си припомним псевдокода на Selection Sort.

```
Selection Sort(A[1..n]: array of integers)
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])
```

Коректността на Selection Sort. Теорема 43 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 разгледахме. При Selection Sort не беше абсолютно очевидно, че елементите в края са точно тези, които са били в началото,

защото неговата реализацията 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 26

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 $A[i \mathinner{..} n]$.

Доказателство:
По отношение на едно изпълнение на външния **for** цикъл, следното твърдение е инвариант за вътрешния **for** цикъл:

Инвариант 8: Вътрешният цикъл на Selection Sort

Всеки път, когато изпълнението достигне ред 2, текущият $A[i]$ е минимален елемент в $A[i \mathinner{..} j-1]$.

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 $A[i \mathinner{..} (i+1) - 1]$.

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 $A[i \mathinner{..} j-1]$. Тъй като $A[j] < A[i]$, съгласно транзитивността на релацията $<$, $A[j]$ е минималният елемент в подмасива $A[i \mathinner{..} j]$ преди размяната. Тогава $A[i]$ е минималният елемент в $A[i \mathinner{..} j]$ 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 $A[i \mathinner{..} j-1]$.

Случай 2. $A[j] \not< A[i]$. Тогава условието на ред 3 е False и размяната на ред 4 не се случва. Съгласно предположението, $A[i]$ е минимален елемент в $A[i \mathinner{..} j-1]$. Тъй като $A[i] \leqslant A[j]$, очевидно $A[i]$ е минимален елемент в $A[i \mathinner{..} j]$. 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 $A[i \mathinner{..} j-1]$.

Терминация. Да разгледаме момента, в който изпълннението 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 $A[i \mathinner{..} (n+1) - 1]$". □

Теорема 43: Коректността на Selection Sort

Selection Sort е сортиращ алгоритъм.

Доказателство:
Да наречем оригиналния масив $A'[1 \mathinner{..} n]$. Следното е инвариант за външния **for** цикъл (редове 1–4):

Инвариант 9: Външният цикъл на Selection Sort

Всеки път, когато изпълнението на Selection Sort е на ред 1, текущият подмасив $A[1..i-1]$ се състои от $i-1$ на брой най-малки елементи на $A'[1..n]$ в сортиран вид.

База. При първото достигане на ред 1 е вярно, че $i = 1$. Текущият подмасив $A[1..i-1]$ е празен и се състои от нула на брой най-малки елементи от $A'[1..n]$ 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 26, ефектът на вътрешния **for** цикъл е, че поставя на i-тата позиция най-малка стойност от $A''[i..n]$. От друга страна, съгласно индуктивното допускане, $A''[1..i-1]$ се състои от $i-1$ на брой най-малки елементи на $A'[1..n]$ в сортиран вид. Заключаваме, че в края на изпълнението на външния **for** цикъл, текущият $A[1..i]$ се състои от i на брой най-малки елемента от $A'[1..n]$ 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 $A[1..i-1]$ съдържа $i-1$ на брой най-малки елемента от $A'[1..n]$ 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 $A[1..n-1]$ се състои от $n-1$ на брой най-малки елемента на $A'[1..n]$ в сортиран вид". Но тогава текущият $A[n]$ е максимален елемент от $A'[1..n]$. С това доказателството за коректност на Selection Sort приключва. □

Сложност по време на Selection Sort. Вече показахме в (2.13) на стр. 80, 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 сортирация 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 децата на вътрешните върхове са наредени с линейна наредба. Следните две дървета T_1 и T_2 са различни като наредени дървета (наредбата е отляво надясно), 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 46: Височина, дълбочина, нивá

Височина на връх u в кореново дърво е максималното разстояние между u и кое да е листо в $T[\![u]\!]$. *Височина на дървото* е височината на корена. *Дълбочина 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 нивá* са нивá, 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 47: попълнено двоично дърво, не-индуктивно определение

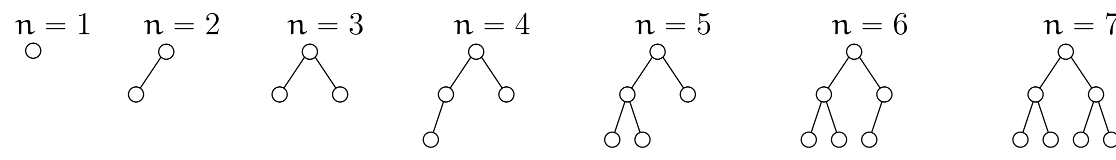
Попълнено двоично дърво (complete binary tree) е наредено двоично дърво, в което:

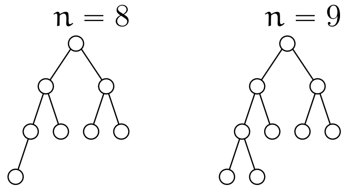
- всички нива, с изключение може би на последното, имат максималния възможен брой[а] върхове[б].
- ако листата са на две нива, листата на последното ниво са максимално вляво. □

[а] Ниво номер k може да има най-много 2^k върха в себе си.
[б] Следствие от това е, че всички листа са или в едно ниво, или в две нивá, които са съседни.

На английски терминът е complete tree.

За всяко $n \in \mathbb{N}^+$ има едно единствено попълнено дърво с n върха:





Определение 48: 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 $h \geqslant 0$. Нека r е връх извън T' и T''. Тогава

$$\big(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}\big)$$

  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 49: попълнено двоично дърво, индуктивно определение

База:

1. $(\{u\}, \emptyset)$ е попълнено двоично дърво с корен u, множество от листá $\{u\}$, 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 листá $\{v\}$, 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 листá W', и височина $h - 1 \geqslant 0$. Нека T'' е попълнено кореново дърво с корен u'', множество от листá W'', и височина височина $h - 1$. Нека r е връх извън T' и T''. Тогава

$$\left(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}\right),$$

където u' е лявото дете на r, а u'' е дясното дете на r, е попълнено кореново дърво с корен r, множество от листá $W' \cup W''$ и височина h.

4. Нека T' е попълнено кореново дърво с корен u', множество от листá W', и височина $h - 1 \geqslant 1$. Нека T'' е съвършено кореново дърво с корен u'', множество от листá W'', и височина $h - 2$. Нека r е връх извън T' и T''. Тогава

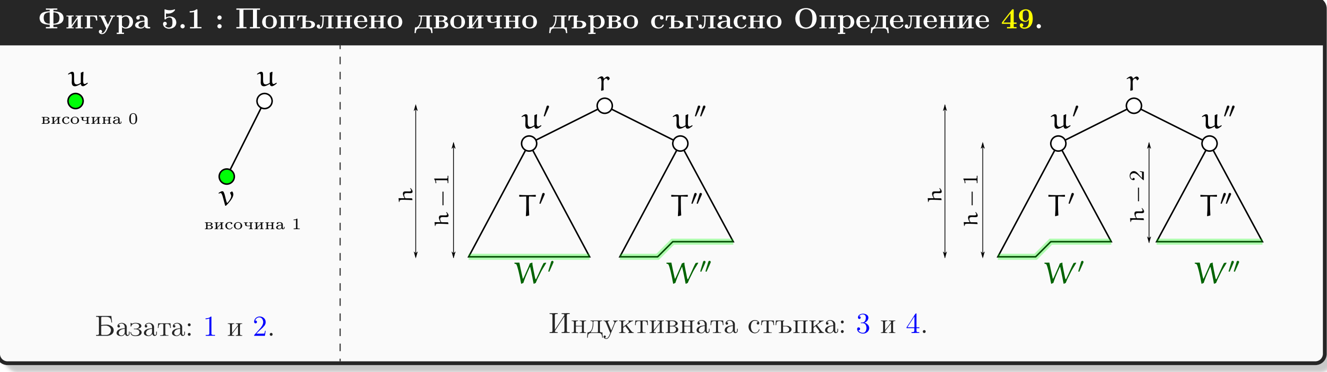
$$\left(V(T') \cup V(T'') \cup \{r\}, E(T') \cup E(T'') \cup \{(r, u'), (r, u'')\}\right),$$

където u' е лявото дете на r, а u'' е дясното дете на r, е попълнено кореново дърво с корен r, множество от листá $W' \cup W''$ и височина h.

Определение 47 е еквивалентно на Определение 49.

Фигура 5.1 илюстрира Определение 49. Листата са в зелено.

Фигура 5.1 : Попълнено двоично дърво съгласно Определение 49.



Доказателството за еквивалентност на Определение 47 и Определение 49 остават за читателя.

Както обикновено, пишейки $\lg n$, имаме предвид $\log_2 n$.

Лема 27

За всяко попълнено двоично дърво с n върха, за височината h е изпълнено $h = \lfloor \lg n \rfloor$.

Доказателство: Чрез структурна индукция съгласно Определение 49. 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:

$$h - 1 \leqslant \lg p < h$$
$$h - 1 \leqslant \lg q < h$$

Тогава,

$$2^{h-1} \leqslant p < 2^h$$
$$2^{h-1} \leqslant q < 2^h$$

Но p, q и h са цели числа и 2^h също е цяло число. Имаме право да кажем, че

$$2^{h-1} \leqslant p < 2^h - \frac{1}{2}$$
$$2^{h-1} \leqslant q < 2^h - \frac{1}{2}$$

Тогава

$$2 \times 2^{h-1} \leqslant p + q < 2 \times 2^h - 1 \quad \leftrightarrow$$
$$2^h \leqslant p + q < 2^{h+1} - 1 \quad \leftrightarrow$$
$$2^h < p + q + 1 < 2^{h+1}$$

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 $h-2 = \lfloor \lg q \rfloor$. Но q е число от вида 2^k-1 за някое $k \geqslant 0$, защото T'' е съвършено дърво. Тогава $q + 1 = 2^k$. Следователно, $\lfloor \lg (q+1) \rfloor = \lfloor \lg q \rfloor + 1$, така че $h - 1 = \lfloor \lg (q+1) \rfloor$. С разсъждения, аналогични на предния случай, получаваме

$$h - 1 \leqslant \lg p < h$$
$$h - 1 \leqslant \lg (q + 1) < h$$

Тогава,

$$2^{h-1} \leqslant p < 2^h$$
$$2^{h-1} \leqslant q + 1 < 2^h$$

Оттук имаме

$$2^h \leqslant p + q + 1 < 2^{h+1} \quad \leftrightarrow$$
$$2^h \leqslant n < 2^{h+1}$$

Тогава,

$$h \leqslant \lg n < h + 1$$

Следователно, $h = \lfloor \lg n \rfloor$. ✓ □

Лема 28

Всяко попълнено двоично дърво T има точно $\left\lceil \frac{n}{2} \right\rceil$ листа́.

Доказателство: Чрез структурна индукция съгласно Определение 49. За двата базови случаи твърдението е вярно: в случай 1, $n = 1$ и наистина има $\left\lceil \frac{1}{2} \right\rceil = 1$ листа́, а в случай 2, $n = 2$ и наистина има $\left\lceil \frac{2}{2} \right\rceil = 1$ листа́. ✓

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 тях казва, че те имат съответно $\left\lceil \frac{p}{2} \right\rceil$ и $\left\lceil \frac{q}{2} \right\rceil$ листа́. Наготово използваме факта, че T' има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $p = 2k + 1$. Съгласно Определение 49, множеството от листата на T се разбива на множествата от листата на T' и T'', така че броят на листата на T е:

$$\left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = \left\lceil \frac{2k+1}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = k + 1 + \left\lceil \frac{q}{2} \right\rceil = \left\lceil k + 1 + \frac{q}{2} \right\rceil = \left\lceil \frac{2k+2+q}{2} \right\rceil =$$
$$= \left\lceil \frac{2k+1+q+1}{2} \right\rceil = \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad ✓$$

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 тях казват, че те имат съответно $\left\lceil \frac{p}{2} \right\rceil$ и $\left\lceil \frac{q}{2} \right\rceil$ листа́. Наготово използваме факта, че T'' има нечетен брой върхове, понеже е съвършено двоично дърво. Нека $q = 2k + 1$. Съгласно Определение 49, множеството от листата на T се разбива на множествата от листата на T' и T'', така че броят на листата на T е:

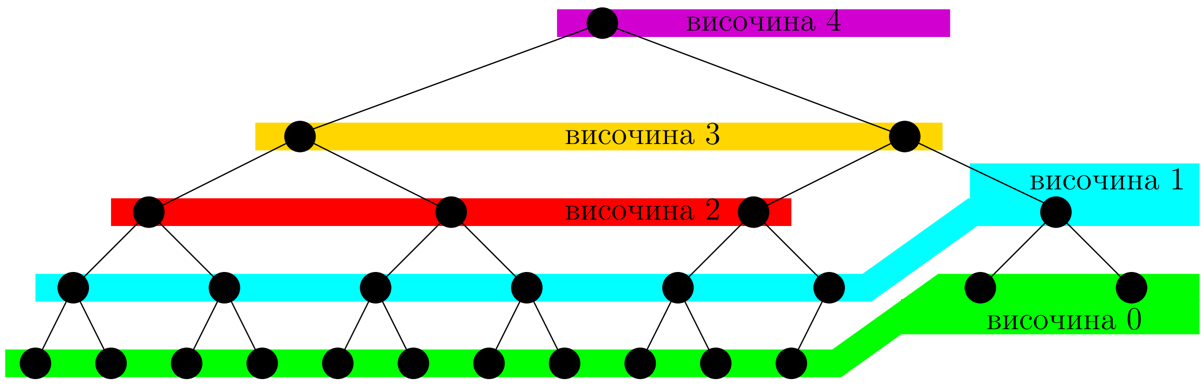
$$\left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{q}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + \left\lceil \frac{2k+1}{2} \right\rceil = \left\lceil \frac{p}{2} \right\rceil + k + 1 = \left\lceil \frac{p}{2} + k + 1 \right\rceil = \left\lceil \frac{p+2k+2}{2} \right\rceil =$$
$$= \left\lceil \frac{p+2k+1+1}{2} \right\rceil = \left\lceil \frac{p+q+1}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil \quad ✓$$

□

Следствие 18

Вътрешните върхове на произволно попълнено двоично дърво с n върха са $\left\lfloor \frac{n}{2} \right\rfloor$.

Доказателство: Следва веднага от Лема 28 и факта, че $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$ за всяко цяло n. □



Фигура 5.2: Височините на върховете в попълнено двоично дърво T.

Лема 29: от страница 71 в [56]

Нека $f(x)$ е произволна реална непрекъсната монотонно растяща функция със свойството:

$$f(x) \text{ е цяло } \rightarrow \ x \text{ е цяло}$$

Тогава $\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$ и $\lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$.

От Лема 29 веднага имаме тези следствия.

Следствие 19

$$\forall x \in \mathbb{R}^+ \ \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \quad \text{и} \quad \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil \right)$$

Доказателство: Прилагаме Лема 29 с $f(x) = \frac{x}{b}$. □

Следствие 20

$$\forall x \in \mathbb{R}^+ \ \forall a \in \mathbb{N}^+ \ \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \quad \text{и} \quad \left\lceil \frac{\lceil \frac{x}{a} \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \right)$$

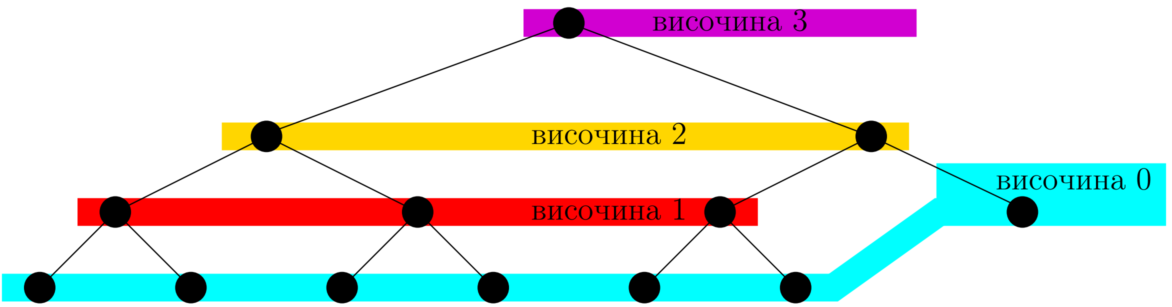
Доказателство:
Прилагаме Следствие 19 с $\frac{x}{a}$ наместо x. □

Лема 30

Във всяко попълнено двоично дърво T с n върха има точно $\left\lceil \frac{n}{2^k} \right\rceil$ върха с височина $\geq k$.

Доказателство: Да си припомним, че *височина на връх* u е максималното разстояние от него до кое да е листо в поддървото $T[\![u]\!]$. Пример за височините на върхове има на Фигура 5.2. Доказателството е по индукция по k.

База. $k = 0$. Върховете с височина ≥ 0 са точно върховете на T, които са n на брой. Забележете, че $n = \left\lceil \frac{n}{2^0} \right\rceil$. ✓



Фигура 5.3: Попълненото дърво T', което се получава от дървото T от Фигура 5.2 след изтриване на всички върхове с височина < 1, тоест с височина 0, тоест листата. Това изтриване намалява с точно единица височините на останалите върхове.

Индуктивно предположение. Нека твърдението е в сила за някаква височина k, която не е максималната.

Индуктивна стъпка. Да изтрием всички върхове с височина $< k$ от T. Нека полученото дърво е T' и нека n' е броят на неговите върхове. Листата на T', тоест върховете с височина 0 в T', очевидно са точно върховете с височина k в T. Всички върхове в T' са точно върховете с височина $\geqslant k$ в T. Съгласно индуктивното предположение, $n' = \left\lfloor \frac{n}{2^k} \right\rfloor$. Примерно, да разгледаме дървото T от Фигура 5.2 с $k = 1$: след изтриването на всички върхове с височина < 1 получаваме дървотоT' от Фигура 5.3.

Съгласно Следствие 18, има точно $\left\lceil \frac{n'}{2} \right\rceil$ вътрешни върхове в T'. Но вътрешните върхове на T' са точно върховете с височина $\geqslant k+1$ в T. Следователно, има $\left\lceil \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rceil$ върха с височина $\geqslant k+1$ в T. Съгласно Лема 29, $\left\lceil \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rceil = \left\lfloor \frac{n}{2 \times 2^k} \right\rfloor$, и очевидно $\left\lfloor \frac{n}{2 \times 2^k} \right\rfloor = \left\lfloor \frac{n}{2^{k+1}} \right\rfloor$. □

Теорема 44: Броят на върховете с височина k в попълнено дърво.

Във всяко попълнено двоично дърво T с n върха има точно $\left\lceil \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rceil$ върха с височина k.

Доказателство: Съгласно Лема 30, има точно $\left\lfloor \frac{n}{2^k} \right\rfloor$ върха с височина $\geqslant k$. Върховете с височина k са точно листата в поддървото на T, индуцирано от върховете с височина $\geqslant k$. Съгласно Лема 28, тези листа са точно $\left\lceil \frac{\left\lfloor \frac{n}{2^k} \right\rfloor}{2} \right\rceil$. □

5.1.2 Адреси в попълнено двоично дърво

Следното определение задава процедура, която дава адреси-стрингове на върховете на попълнено двоично дърво, използвайки Определение 49. С “ϵ” означаваме празния стринг.

Определение 50: Адреси на върхове в попълнено двоично дърво

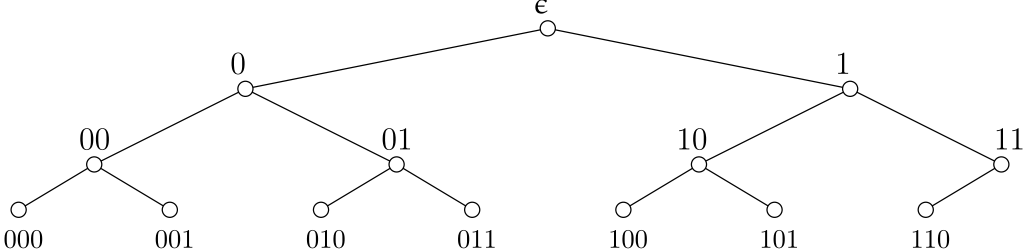
Всеки връх x в попълненото дърво има *адрес*, който бележим с $\text{addr}(x)$. Адресите се конструират от следната процедура от две фази.

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:

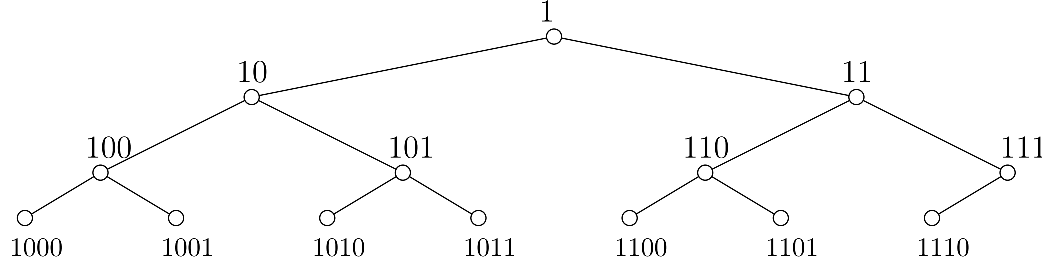
- $\text{addr}(u) = \epsilon$,
- за всеки връх x от T', новият адрес на x се получава от стария с конкатенация на нула в левия край.
- за всеки връх x от T'', новият адрес на x се получава от стария с конкатенация на единица в левия край.

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 50 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 31

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 49, и съгласно Определение 50, $\text{addr}(u) = 1$ 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 $\text{addr}(v) = \text{addr}(u)1$, 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. □

За всяко $n \in \mathbb{N}^+$, $\mathsf{bin}(n)$ 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 $\mathsf{bin}_k(n)$ се дефинира така:

$$\mathsf{bin}_k(n) = \begin{cases} 0^{k-m}\,\mathsf{bin}(n), & \text{ако } k > m \\ \mathsf{bin}(n), & \text{ако } k = m \\ \text{суфиксът с дължина } k \text{ на } \mathsf{bin}(n), & \text{ако } k < m. \end{cases}$$

Забележете, че в случая m е дължината на представянето на n в двоична позиционна бройна система. Също така забележете, че $\mathsf{bin}(n)$ и $\mathsf{bin}_k(n)$ 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-малка 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 32

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 $0 \leqslant d \leqslant h$. Тогава

(а) За $0 \leqslant d \leqslant h - 1$, множеството от адресите на върховете от V_d е множеството от всички булеви стрингове с дължина $d + 1$ и водеща единица.

(б) Множеството от адресите на върховете от V_h е множеството от булевите стрингове с дължина $h + 1$ и водеща единица от 10^h до $\mathsf{bin}(n)$ в лексикографската наредба.

(в) 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 $0 \leqslant d \leqslant h-1$, 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 $\mathsf{bin}_h(n)$ в лексикографската наредба.

Това ще докажем чрез структурна индукция съгласно Определение 49 на попълнено дърво, използвайки Определение 50 на адреси на върховете на попълнено дърво.

База. В случай 1 на Определение 49, 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 $\mathsf{bin}_1(2) = 0$. ✓

Индуктивна стъпка Да разгледаме случай 3.

- На ниво 0, коренът r наистина получава адрес ϵ.
- Да разгледаме произволно ниво d, където $1 \leqslant d \leqslant h-1$. 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 $0 \leqslant d-1 \leqslant h-2$, 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

$$\underbrace{00\ldots0}_{\text{дължина } h-1}, \quad \underbrace{00\ldots01}_{\text{дължина } h-1}, \ldots, \underbrace{\mathsf{bin}_{h-1}(p)}_{\text{дължина } h-1}$$

  Но $p = 2^{h-1+1} - 1 = 2^h - 1$, защото T' е съвършено дърво, така че $\mathsf{bin}_{h-1}(p) = 1^{h-1}$ единици. Тогава адресите в ниво $h-1$ на T' са, отляво надясно,

$$\underbrace{00\ldots0}_{\text{дължина } h-1}, \quad \underbrace{00\ldots01}_{\text{дължина } h-1}, \ldots, \underbrace{11\ldots1}_{\text{дължина } h-1}$$

  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

$$\underbrace{00\ldots0}_{\text{дължина } h-1}, \quad \underbrace{00\ldots01}_{\text{дължина } h-1}, \ldots, \underbrace{\mathsf{bin}_{h-1}(q)}_{\text{дължина } h-1}$$

Следователно, в цялото дърво T, адресите на върховете от ниво h са, отляво надясно

$$\underbrace{00\ldots0}_{\text{дължина } h},\ \underbrace{00\ldots01}_{\text{дължина } h},\ldots,\underbrace{011\ldots1}_{\text{дължина } h},\ \underbrace{10\ldots0}_{\text{дължина } h},\ \underbrace{10\ldots01}_{\text{дължина } h},\ldots,\underbrace{1\,\mathsf{bin}_{h-1}(q)}_{\text{дължина } h}$$

Тъй като $n = p+q+1$ и $p = 2^h-1$, то $n = 2^h+q$. Но тъй като $q \leqslant p$, дължината на $\mathsf{bin}(q)$ също е по-малка от h и $\mathsf{bin}_{h-1}(q)$ се получава от $\mathsf{bin}(q)$ с добавяне на $\geqslant 0$ на брой нули вляво. Имайки предвид, че $\mathsf{bin}(2^h) = 10^h$, заключаваме, че стрингът $1\,\mathsf{bin}_{1-h}(q)$ е равен на $\mathsf{bin}(n)$. Заключаваме, че адресите от последното ниво на T' отляво надясно, следвани от адресите от последното ниво на T'' отляво надясно, представляват последователността

$$\underbrace{00\ldots0}_{\text{дължина } h},\ \underbrace{00\ldots01}_{\text{дължина } h},\ldots,\underbrace{011\ldots1}_{\text{дължина } h},\ \underbrace{10\ldots0}_{\text{дължина } h},\ \underbrace{10\ldots01}_{\text{дължина } h},\ldots,\underbrace{\mathsf{bin}(n)}_{\text{дължина } h}$$

Да разгледаме случай 4.

- На ниво 0, коренът r наистина получава адрес ϵ.
- Да разгледаме произволно ниво d в T, където $1 \leqslant d \leqslant h-1$. 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 $0 \leqslant d-1 \leqslant h-2$, 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 предложение. Ако $d-1 < h-2$, адресите на 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 предложение.

  Да разгледаме случая $d-1 = h-2$ по отношение на T''. Сега част (б') на индуктивното предположение е приложима. Съгласно нея, адресите на на върховете от ниво $d-1$ в T'' са булевите стрингове от 0^{h-2} до $\mathsf{bin}_{h-2}(q)$ 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 $\mathsf{bin}_{h-2}(q) = 1^{d-1}$.

  Заключаваме, че за $1 \leqslant d \leqslant h-1$, 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 на върховете от ниво $h-1$ в T' са булевите стрингове от 0^{h-1} до $\mathsf{bin}_{h-1}(p)$, 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 $0\mathsf{bin}_{h-1}(p)$, 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 $\mathsf{bin}(q + 1) = 10^{h-1}$. От друга страна, $p \geqslant q + 1$, така че $\mathsf{bin}(p) = 1\sigma$ за някой булев стринг σ с дължина $h - 1$. Тогава $\mathsf{bin}(p + q + 1) = 10\sigma$. С други думи, $\mathsf{bin}(n) = 10\sigma$. Но тогава $\mathsf{bin}_h(n) = 0\sigma = 0\mathsf{bin}_{h-1}(p)$.

Доказахме, че наистина адресите на върховете от последното ниво на T са лексикографски сортираните стрингове от 0^h до $\mathsf{bin}_h(n)$. □

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.

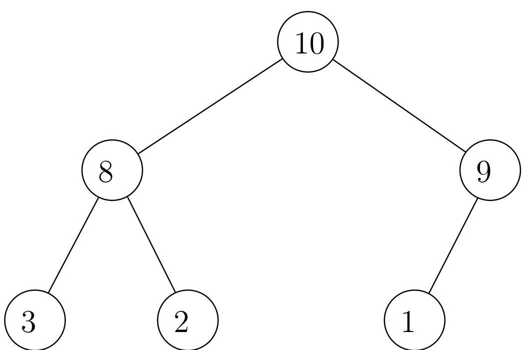
Определение 51: Двоична пирамида (binary heap)

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).

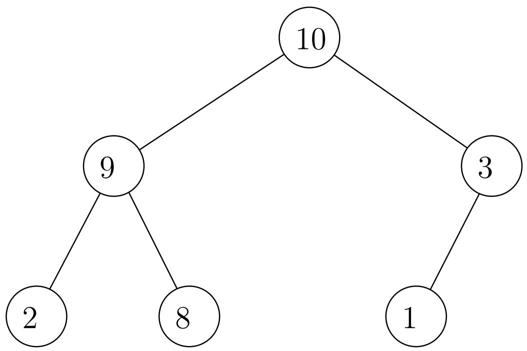
Тук ще разглеждаме максимални пирамиди и казвайки "пирамида" ще имаме предвид "максимална пирамида". Но изложението може да бъде променено лесно, така че да се отнася за минимални пирамиди. Също така, ще изпускаме "двоични", понеже не разглеждаме пирамиди, които не са двоични.

Ето пример за пирамида:



Очевидно фòрмата 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. Примерно, това също е пирамида със тези ключове:

†В Допълнение 30 дискутираме броя на различните пирамиди с n върха.



Наблюдение 30

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.

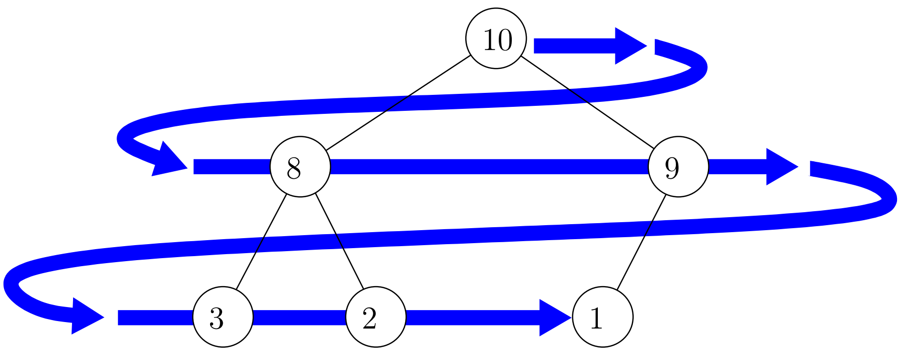
5.1.4 Реализиране на пирамида чрез масив

Определение 52

Нека $T = (V, E)$ е попълнено двоично дърво с n върха. *Линеаризация* на T се нарича този масив $A[1 \mathinner{..} n]$, в който за всяко i, $1 \leqslant i \leqslant n$, $A[i]$ 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 32, съществува биекция между множеството от адресите и множеството $\{1, 2, \ldots, n\}$, така че определението е смислено. И така, пирамидите може да се реализират много ефикасно чрез масиви. Линеаризацията на последния 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 32 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 допълителна памет, като при това обхождането е много ефикасно като сложност по

време. Последното твърдение се основава на факта (виж Наблюдение 31), 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 31.

Наблюдение 31

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 $A\left[\left\lfloor\frac{i}{2}\right\rfloor\right]$, 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. □

Наблюдение 31 ни дава основание да дефинираме следните изчислителни примитиви[†], чиито имена са достатъчно информативни:

Parent(i)
  return (**if** $i \geqslant 2$ **then** $\left\lfloor\frac{i}{2}\right\rfloor$ **else** i)

Left(i)
  return (**if** $2i \leqslant n$ **then** $2i$ **else** Undefined)

Right(i)
  return (**if** $2i+1 \leqslant n$ **then** $2i+1$ **else** Undefined)

Level(i)
  return $\lfloor \log_2 i \rfloor$

IsLeaf(i)
  return (**if** $i > \left\lfloor\frac{n}{2}\right\rfloor$ **then** Yes **else** No)

IsInternalVertex(i)
  return **not** IsLeaf(i)

IsRoot(i)
  return (**if** $i = 1$ **then** Yes **else** No)

[†]"Примитив" е, неформално казано, лесна за възприемане функция с малка сложност, която се използва често от алгоритмите, които ни интересуват.

Наблюдение 32

Забележете, че $\textsc{Parent}(1) = 1$. С други думи, коренът е родител на себе си съгласно примитива $\textsc{Parent}$. Това не следва по никакъв начин от Наблюдение 31! Можеше да възприемем алтернативния подход, а именно, че коренът няма родител, но изложението е по-кратко и елегантно, ако всеки елемент да има родител, включително и коренът.

Освен това, за $t \geqslant 1$, дефинираме, че

$$\textsc{Parent}^t(i) = \begin{cases} i, & \text{ако } t = 0 \\ \textsc{Parent}\big(\textsc{Parent}^{t-1}(i)\big), & \text{ако } t > 0 \end{cases}$$

и

$$\textsc{Ancestors}(i) = \{\textsc{Parent}^t(i) \,|\, t \geqslant 1\}$$

Казваме, че $A[j]$ е *предшественик* на $A[i]$, ако $j \in \textsc{Ancestors}(i)$. По това определение никой елемент на линеаризацията не е предшественик на себе си, освен $A[1]$. Предвид последното, ясно е, че $\textsc{Ancestors}(i)$ е крайно множество, въпреки че "$t \geqslant 1$" означава "$t \in \mathbb{N}^+$".

Определение 53: 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 инверсия. *Директна пирамидална инверсия*, или накратко *директна инверсия*, ако от контекста е ясно, че става дума за пирамидални инверсии, е всяка пирамидална инверсия $\langle i, j\rangle$, такава че $i = \textsc{Parent}(j)$. Пирамидални инверсии, които не са директни, се казват *индиректни*.

Например, на Фигура 5.10 е показано попълнено дърво с ключове (неговата линеаризация 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.

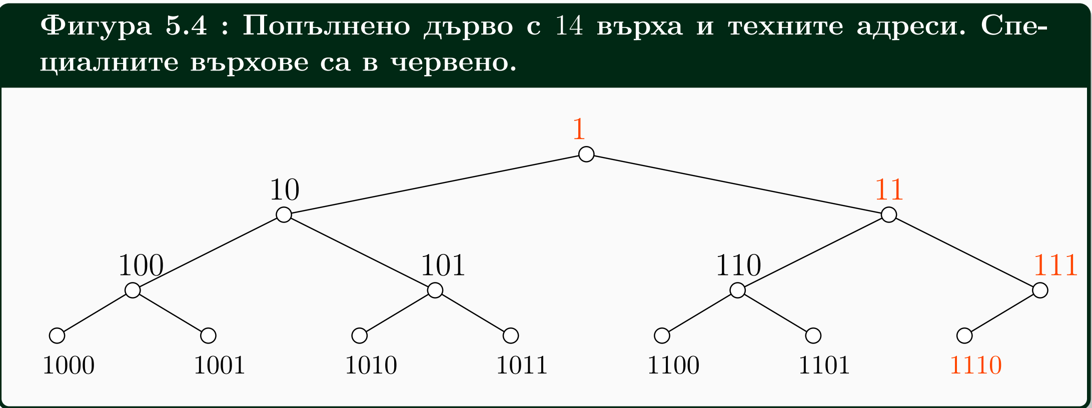
Наблюдение 33

Масивът от ключове A е пирамида тогава и само тогава, когато няма нито една пирамидална инверсия. □

Допълнение 30: 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, линеаризации на пирамиди. Изложението следва изложението в [86, 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 $\mathsf{bin}(n)$). Тези върхове са наречени в [86], *специалните върхове*, а дърветата, вкоренени в тях, *специалните поддървета*. В [86, стр. 157, упр. 20 и 21] е показано, че ако $\mathsf{bin}(n) = b_k b_{k-1} \ldots b_1 b_0$, където $k = \lfloor \lg n \rfloor$ и $b_k = 1$, то размерите[a] на специалните поддървета, записани в двоична позиционна бройна система, са

$$1b_{k-1} \ldots b_1 b_0, \qquad 1b_{k-2} \ldots b_1 b_0, \qquad \ldots \qquad 1b_1 b_0, \qquad 1b_0, \qquad 1$$

а размерите на неспециалните поддървета числа от вида $2^m - 1$, защото те са съвършени двоични дървета, и броевете на неспециалните поддървета от всяка срещаща се големина са съответно

$$\left\lfloor \frac{n-1}{2} \right\rfloor \text{ числа } 1,$$
$$\left\lfloor \frac{n-2}{4} \right\rfloor \text{ числа } 3,$$
$$\left\lfloor \frac{n-4}{8} \right\rfloor \text{ числа } 7,$$
$$\ldots$$
$$\left\lfloor \frac{n-2^{k-1}}{2^k} \right\rfloor \text{ числа } 2^k - 1$$

Знаейки размерите на поддърветата, можем да изчислим H_n: броя на начините да разположим ключовете $\{1, 2, \ldots, n\}$ в пирамида, като използваме резултата от [86, стр.

67, упр. 20]:

$$H_n = \frac{n!}{s_1 \times s_2 \times \cdots \times s_n} \tag{5.1}$$

където $\{s_1, s_2, \ldots, s_n\}_M$ е мултимножеството от всички размери на поддървета (вж. Определение 38). Примерно, ако елементите са 14, каквото е дървото на Фигура 5.4, начините да сложим ключовете $\{1, 2, \ldots, 14\}$, така че дървото да е пирамида, са на брой

$$H_{14} = \frac{14!}{14 \times 6 \times 2 \times 1 \times 1^6 \times 3^3 \times 7^1} = 2\,745\,600$$

Сравнете H_{14} с $14! = 87\,178\,291\,200$.

Редицата H_n за $n \geqslant 1$ е редица A056971 в онлайн енциклопедията на целочислените редици. Точният израз (5.1) не дава представа за асимптотиката на H_n. Асимптотиката на H_n се разглежда в [73]. 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 константна c.

[a] Това са общо $k = \lfloor \lg n \rfloor$ числа, колкото е височината на попълненото двоично дърво с n върха съгласно Лема 27.

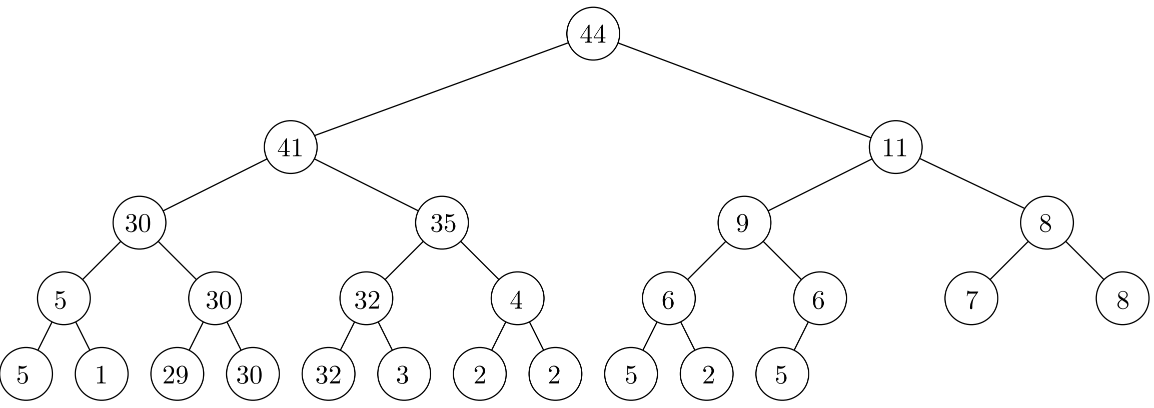
5.1.5 Подпирамида

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 преставя 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).

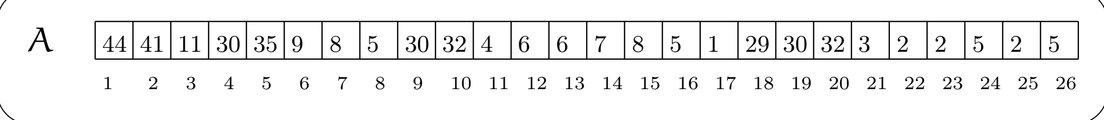
Наблюдение 34

Ако $A[1\,..\,n]$ е пирамида, то за всяко $k \in \{1, 2, \ldots, n\}$, $A[\![k]\!]$ е пирамида. □

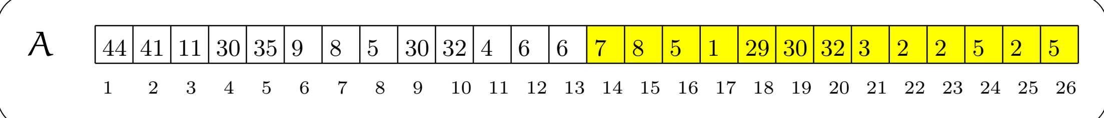
Като пример за пирамида и нейната линеаризация, да разгледаме пирамидата, показана на Фигура 5.5. Фигура 5.6 показва нейната линеаризация. Фигура 5.7 показва листата на пирамидата. На Фигура 5.8 е показана подпирамидата $A[\![3]\!]$. Връх 3 е третият връх в масива (дясното дете на корена, ако разсъждаваме в термините на дървото). Фигура 5.9 показва листата на $A[\![3]\!]$ в червено.



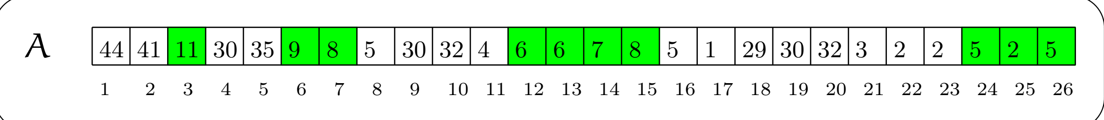
Фигура 5.5: Пирамида с 26 върха.



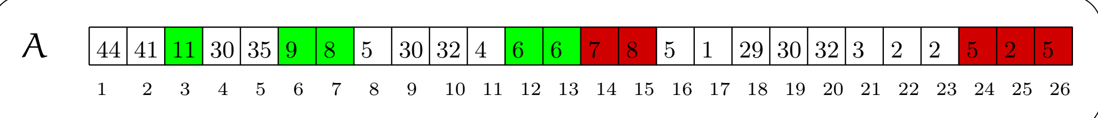
Фигура 5.6: Линеаризацията на пирамида от Фигура 5.5.



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 $\mathsf{T}[\![u]\!]$. "Обхожда" означава "извежда ключовете на $\mathsf{T}[\![u]\!]$ в *preorder*[†]" (а не по нива).

Traverse Binary SubHeap($A[1..n]$: пирамида, i: число от $\{1, 2, \ldots, n\}$)
1 **print** i
2 *left* ← Left(i)
3 *right* ← Right(i)
4 **if** *left* $\leqslant n$
5   Traverse Binary SubHeap(A, *left*)
6 **if** *right* $\leqslant n$
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 в произволен порядък. 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 30.

5.2.1 Наивно построяване на пирамида

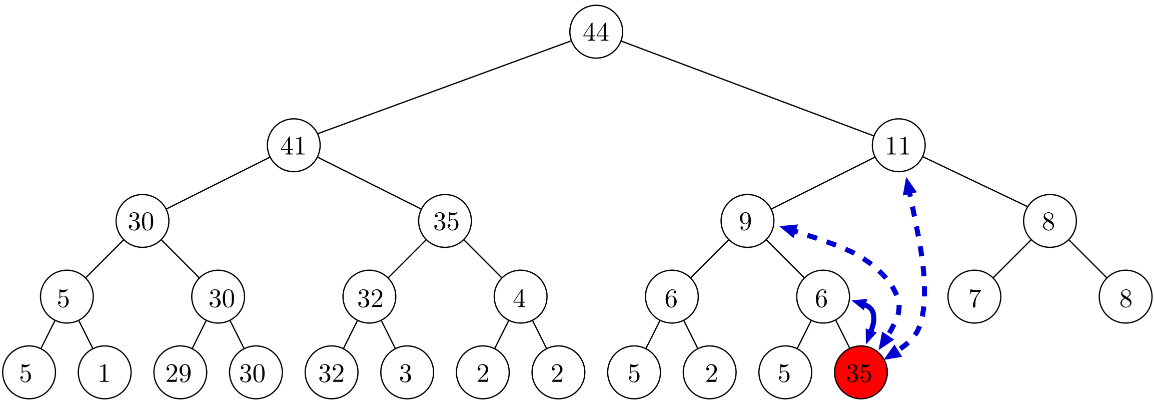
Най-естественият начин да бъде превърнат произволен масив от числа в пирамида чрез разместване е чрез последователно добавяне на елементи в частично построена пирамида.

Naive Build Heap, outline($A[1..n]$: масив от цели числа)
1 **for** $i \leftarrow 2$ **to** n
2   добави $A[i]$ към досега построената пирамида $A[1..i-1]$,
    така че $A[1..i]$ да стане пирамида

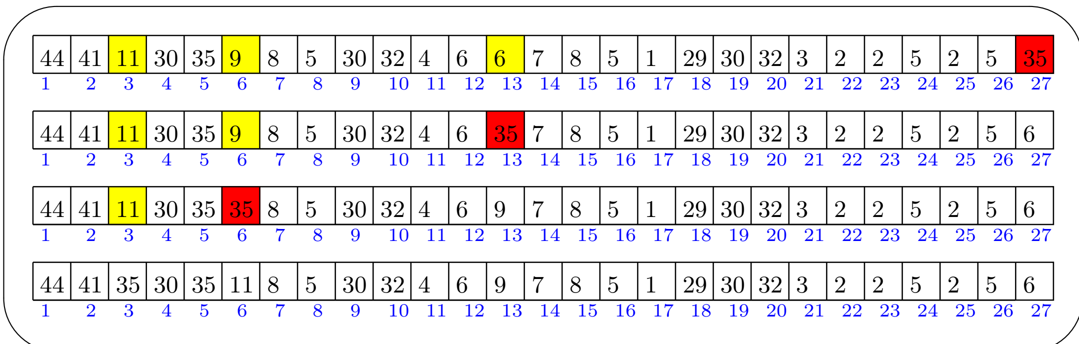
Това описание е прекалено общо, но идеята е ясна: "добави към $A[1..i-1]$" не е просто увеличаване на индекса i. Дори $A[1..i-1]$ да е пирамида, $A[1..i]$ не е непременно пирамида, защото може да има пирамидални инверсии $\langle \text{Parent}^t(i), i \rangle$. Това е илюстрирано на Фигура 5.10, на която 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 непрекъсната последователност). 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 [131].



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 непрекъсната 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.

NAIVE BUILD HEAP, DETAILED($A[1..n]$: масив от ест. числа)

1 **for** $i \leftarrow 2$ **to** n
2 $\quad k \leftarrow i$
3 $\quad$ **while** $k > 1$ **and** $A[\text{PARENT}(k)] < A[k]$ **do**
4 $\quad\quad \text{swap}(A[k], A[\text{PARENT}(k)])$
5 $\quad\quad k \leftarrow \text{PARENT}(k)$

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 54: 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-горния 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 35: 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 $i \in \textsc{Ancestors}(n)$. Нещо повече, ако $\langle i, n \rangle$ е инверсия, то за всяко k, такова че $i < k < n$ и $k \in \textsc{Ancestors}(n)$, е вярно, че $\langle k, n \rangle$ е инверсия.

Лема 33: 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 допускаме, 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 $A[\textsc{Parent}(k)] \geqslant A[k]$, 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 допускаме, че в момента t има инверсии. Съгласно Наблюдение 35, всички инверсии са от вида $\langle x, i \rangle$, където $x \in \textsc{Ancestors}(i)$. Дефинираме a като минималното x, такова че $\langle x, i \rangle$ е инверсия. Следното твърдение е инвариант за **while** цикъла.

Инвариант 10: Вътрешният цикъл на NAIVE BUILD HEAP

При всяко достигане на ред 3, множеството от пирамидалните инверсии в текущия $A[1..i]$ е $\{\langle b, k \rangle \,|\, b \in \textsc{Ancestors}(k) \text{ и } b \geqslant a\}$.

База. Когато изпълнението достигне до ред 3 за първи път, $A'[1..i]$ реализира почти-пирамида по допускане от условието на Лема 33. Тъй като $k = i$ заради присвояването на ред 2, твърдението на инварианта става “множеството от пирамидалните инверсии в текущия $A[1..i]$ е $\{\langle b, i \rangle \,|\, b \in \textsc{Ancestors}(i) \text{ и } b \geqslant a\}$”, което следва веднага от Наблюдение 35 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 $A[\textsc{Parent}(k)]$, с което множеството от

пирамидалните инверсии в текущия $A[1..i]$ става $\{\langle b, k\rangle \mid b \in \textsc{Ancestors}(\textsc{Parent}(k))$ и $b \geqslant a\}$. Но след изпълнението на ред 5, спрямо новото k, отново е вярно, че множеството от пирамидалните инверсии в текущия $A[1..i]$ е $\{\langle b, k\rangle \mid b \in \textsc{Ancestors}(k)$ и $b \geqslant a\}$.

Терминация. При последното достигане на ред 3 е вярно поне едно от двете: $k \leqslant 1$ или $A[\textsc{Parent}(k)] \geqslant A[k]$.

- Ако $k \leqslant 1$, то $\textsc{Ancestors}(k) = \varnothing$, така че множеството от пирамидалните инверсии в текущия $A[1..i]$ е празното множество. С други думи, инверсии няма.
- Нека $A[\textsc{Parent}(k)] \geqslant A[k]$. От това и транзитивността на релацията на наредба на ключовете следва, че за всяко $b \in \textsc{Ancestors}(k)$, $\langle b, k\rangle$ не е инверсия. Но съгласно инварианта, множеството от пирамидалните инверсии в текущия $A[1..i]$ е

  $$\{\langle b, k\rangle \mid b \in \textsc{Ancestors}(k) \text{ и } b \geqslant a\}$$

  Заключаваме, че това множество е празното множество[†]. С други думи, пирамидални инверсии няма. □

Теорема 45: 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 11: Цикълът на Naive Build Heap

Всеки път, когато изпълнението е на ред 1, $A[1..i]$ реализира почти-пирамида.

База. При първото достигане на ред 1 е вярно, че $i = 2$. Съгласно Определение 54, всеки двуелементен масив реализира почти-пирамида, така че инвариантът е верен. ✓

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 33. Съгласно лемата, при допускането, че при влизането в **while**-а е изпълнено текущият $A[1..i]$ да реализира почти-пирамида и k да е равно на i, вярно е, че при излизането от **while**-а, $A[1..i]$ реализира пирамида. Тогава при следващото достигане на ред 1, променливата i се инкрементира и спрямо новото i е вярно, че $A[1..i]$ реализира почти-пирамида.

Терминация. При последното достигане на ред 1 е вярно, че $i = n + 1$. Твърдението "$A[1..n+1]$ реализира почти-пирамида" е формално некоректно, защото говори за несъществуващия елемент $A[n+1]$[‡]. Ако обаче игнорираме тази несъществена подробност, това твърдение ни дава желания резултат, защото казва, че в $A[1..n]$ няма пирамидални инверсии. Тоест, $A[1..n]$ реализира пирамида. □

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, $\lfloor \lg i \rfloor$ пъти за всяко i, понеже височината на дървото $A[1..i]$ е

[†] А за да бъде то празно, трябва да е вярно, че за всяко $b \in \textsc{Ancestors}(k)$, $b < a$.

[‡] Тази формална непрецизност можеше да бъде избягната, ако поначало бяхме присъединили към входния $A[1..n]$ още един елемент $A[n+1]$ със стойност $-\infty$. Този елемент би имал роля, аналогична на т. нар. *sentinel* в Mergesort, който там е $+\infty$. Вижте например ред 7 на Merge.

$\lfloor \lg i \rfloor$. Тогава сложността се определя от сумата

$$\sum_{i=2}^{n} \lfloor \lg i \rfloor = \Theta\left(\sum_{i=2}^{n} \lg i\right) = \Theta\left(\lg\left(2 \times 3 \times \cdots \times (n-1) \times n\right)\right) = \Theta\left(\lg n!\right) \tag{5.2}$$

От Теорема 21 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 [44]. Самият алгоритъм е показан на стр. 297. 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 $u.\text{key} \geqslant v.\text{key}$ и $u.\text{key} \geqslant w.\text{key}$, то T е пирамида. Да допуснем, че $u.\text{key} < v.\text{key}$ или $u.\text{key} < w.\text{key}$. Без ограничение на общността, нека $u.\text{key} < v.\text{key}$ и $u.\text{key} < w.\text{key}$. 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 $v.\text{key} < w.\text{key}$. 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 инверсии—ако изобщо има—само в поддървото, чийто корен сега е u. Грубо казано, броят на инверсиите ще намалее наполовина или повече.[‡]

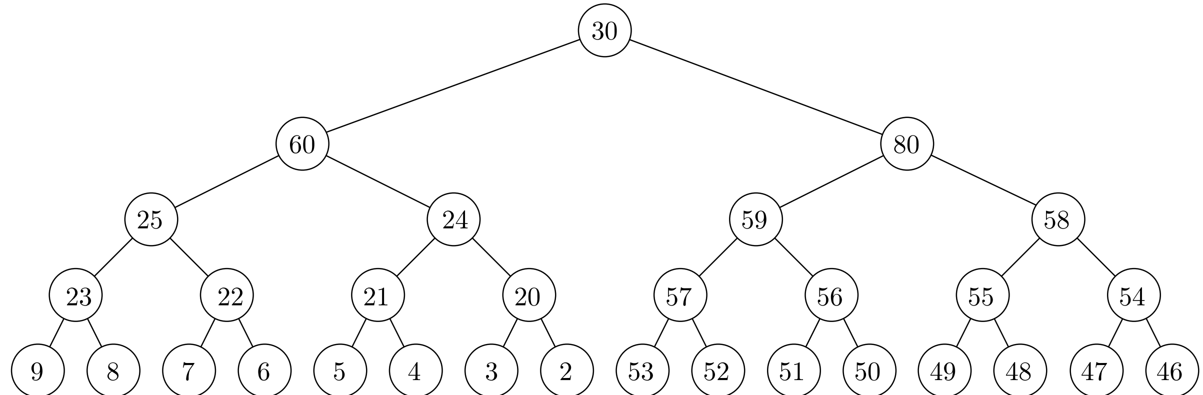
Забележете, че има примери, в които е по-изгодно да разменяме корена не с детето с по-големия ключ, а с детето с по-малкия ключ. Такова дърво е показано на Фигура 5.12. Първо да видим какво ще стане, ако върху това дърво прилагаме размяна на връх с това от двете деца, което има по-голям ключ. Фигура 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

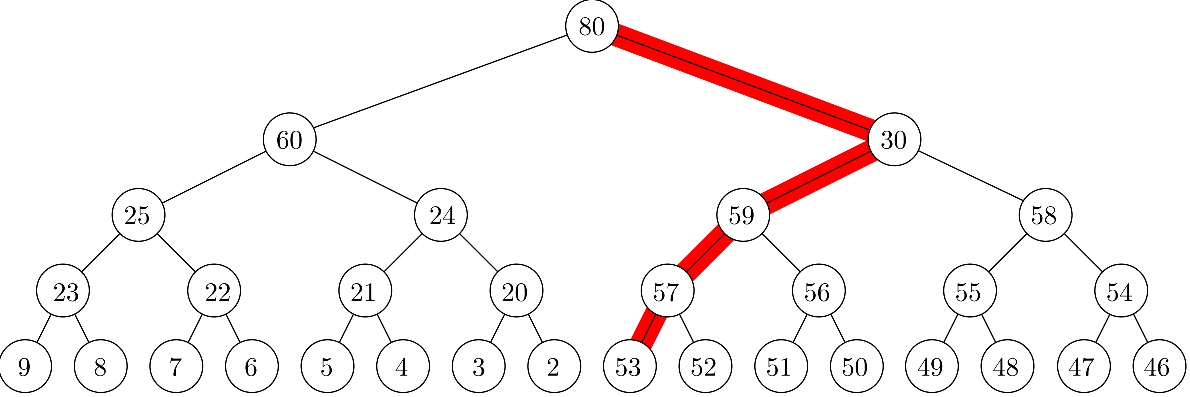
[†]Ако v и w имат еднакви ключове, ще разменим u с кой да е от тях.

[‡]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.

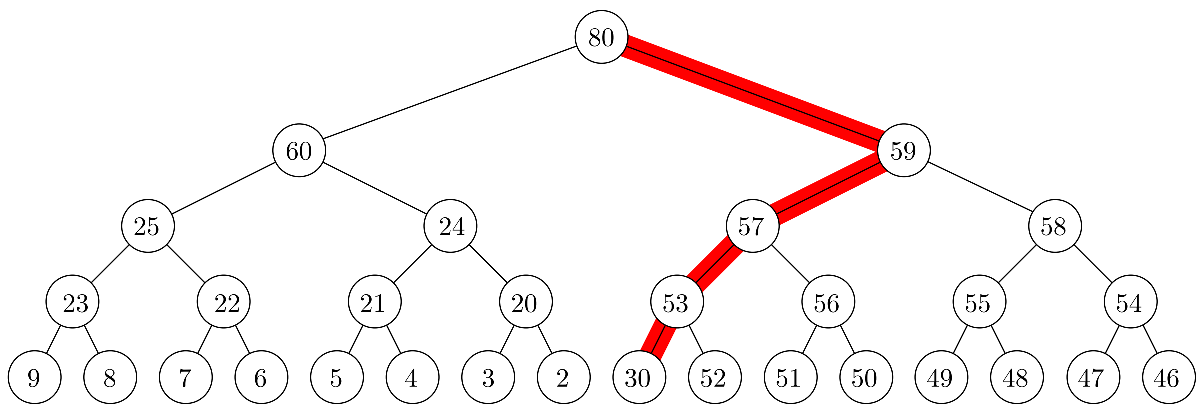
ключ може да доведе до $\Omega(n)$ размени, преди да получим пирамида, докато разменянето с детето с по-голям ключ води до $O(\lg n)$ размени **винаги**. Пример за дърво, върху което разменянето с детето с по-малък ключ води до $\Omega(n)$ размени (примерът се мащабира за безброй много n) е показан на Фигура 5.16. На Фигура 5.17 е показана пирамидата, която се получава от дървото на Фигура 5.16 след $\Omega(n)$ размени.



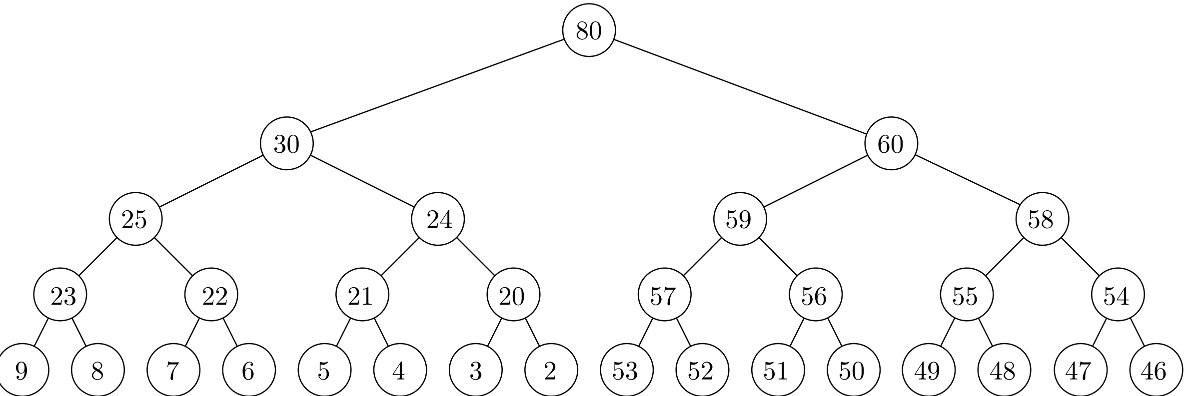
Фигура 5.12: Попълнено дърво с ключове, в което двете деца на корена са корени на поддървета-пирамиди, а ключът на корена е по-малък от ключовете на двете му деца. По-изгодно като минимален брой размени е коренът да бъде разменен с детето с по-малкия ключ.



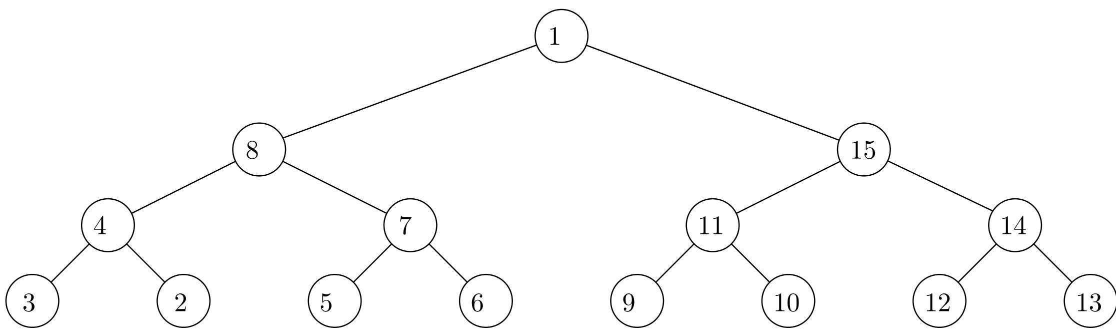
Фигура 5.13: По отношение на дървото от Фигура 5.12, ако разменим корена с неговото дете с по-голям ключ и продължим размените по същия начин—с детето с по-голям ключ—ще направим 4 размени, докато върхът с ключ 30 не стане листо. Тук е показано дървото след първата размяна и е очертан пътят, който трябва да измине върхът с ключ 30. Тази фигура не показва окончателната позиция на върха с ключ 30, а само началото на “пътуването” му.



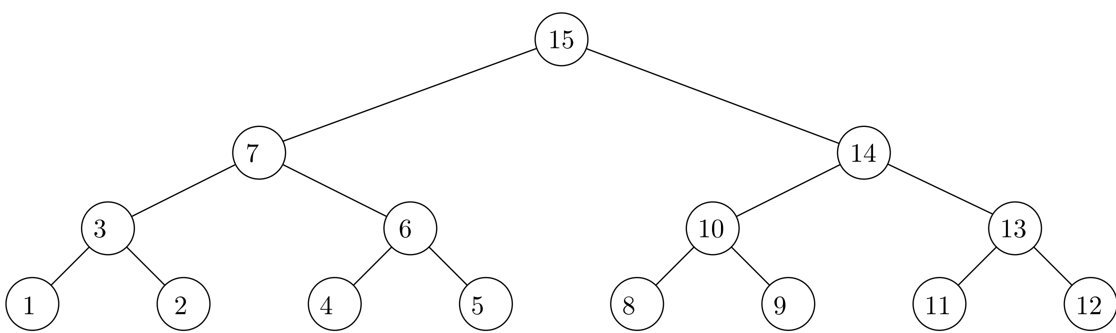
Фигура 5.14: По отношение на дървото от Фигура 5.13 е показано дървото след всички размени, където размените са от вида “разменяме корен с това дете, което има по-голям ключ”. Извършени са 4 размени и дървото е пирамида. Очертан е пътят, който е “изминал” върхът с ключ 30.



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 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 $i > \left\lfloor \frac{n}{2} \right\rfloor$, то $A[i]$ отговаря на листо и няма какво да правим, понеже $A[\![i]\!]$ е тривиална пирамида.

Нека $i \leqslant \left\lfloor \frac{n}{2} \right\rfloor$; с други думи, нека $A[i]$ отговаря на вътрешен връх. БОО, нека $2i+1 \leqslant n$. 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

```
Iterative Heapify(A[1..n]: масив от цели числа, i: индекс в A)
 1  (* A[[Left(i)]] и A[[Right(i)]], ако съществуват, са пирамиди *)
 2  j ← i
 3  while j ⩽ ⌊n/2⌋ do
 4      left ← Left(j)
 5      right ← Right(j)
 6      if A[left] > A[j]
 7          (* Няма смисъл да проверяваме дали left ⩽ n, понеже j ⩽ ⌊n/2⌋ *)
 8          largest ← left
 9      else
10          largest ← j
11      if right ⩽ n and A[right] > A[largest]
12          largest ← right
13      if largest ≠ j
14          swap(A[j], A[largest])
15          j ← largest
16      else
17          break
```

Лема 34

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 Iterative Heapify е, че $A[\![i]\!]$ е пирамида при неговото приключване.

Доказателство: Следното твърдение е инвариант за **while**-цикъла (редове 2–17).

Инвариант 12: Цикълът на Iterative Heapify

Всеки път, когато изпълнението е на ред 3, единствените възможни директни инверсии в $A[\![i]\!]$ са $\langle j, \text{Left}(j)\rangle$ и $\langle j, \text{Right}(j)\rangle$[a], ако $A[\text{Left}(j)]$ и $A[\text{Right}(j)]$ съществуват.

[a]По друг начин казано, ако съществуват пирамидални инверсии в $A[\![i]\!]$, то те са в $A[\![j]\!]$.

База. $j = i$. Съгласно допусканията, всеки от $A[\![\text{Left}(i)]\!]$ и $A[\![\text{Right}(i)]\!]$, ако съществува, е пирамида, така че твърдението е в сила. ✓

Поддръжка. Допускаме, че твърдението е в сила в някой момент, в който изпълнението е на ред 3 и изпълнението ще бъде на ред 3 поне още веднъж.[§] И така, $j \leqslant \lfloor \frac{n}{2} \rfloor$ и поне едно от $\text{Left}(j) \leqslant n$ и $\text{Right}(j) \leqslant n$ е изпълнено, а именно $\text{Left}(j) \leqslant n$. Тогава поне $A[\text{Left}(j)]$ е дефинирано. Без ограничение на общността ще допуснем, че и $\text{Left}(j) \leqslant n$, и $\text{Right}(j) \leqslant n$, за да избегнем ненужни подслучаи.

Случай I: $A[\mathit{left}] > A[j]$ и $A[\mathit{right}] > A[j]$ в началото на изпълнението на цикъла, и освен това $A[\mathit{right}] > A[\mathit{left}]$. Условието на ред 6 е Истина и присвояването на ред 8 се случва. Условието на ред 11 също е Истина, така че и присвояването на ред 12 се случва. Когато изпълнението е на ред 13, $\mathit{largest}$ е равно на right. Условието на ред 13 е Истина и на ред 14, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle \mathit{right}, \text{Left}(\mathit{right})\rangle$ и $\langle \mathit{right}, \text{Right}(\mathit{right})\rangle$:

- нито един елемент от $A[\![i]\!]$, който е извън $A[\![j]\!]$, не е променян;
- $A[j] > A[\mathit{left}]$, така че $\langle j, \text{Left}(j)\rangle$ не може да е пирамидална инверсия;
- $A[j] > A[\mathit{right}]$, така че $\langle j, \text{Right}(j)\rangle$ не може да е пирамидална инверсия;
- $A[\![\mathit{left}]\!]$ не е бил модифициран;
- нито един от $A[\![\text{Left}(\mathit{right})]\!]$ и $A[\![\text{Right}(\mathit{right})]\!]$ не е бил модифициран.

Но на ред 15, j става равен на right. Следователно, инвариантът е в сила при следващото достигане на ред 3.

Случай II: $A[\mathit{left}] > A[j]$ и $A[\mathit{right}] > A[j]$ в началото на изпълнението на цикъла, но $A[\mathit{right}] \not> A[\mathit{left}]$. Условието на ред 6 е Истина и присвояването на ред 8 се случва. Условието на ред 11 е Лъжа, следователно присвояването на ред 12 не се случва и когато изпълнението е на ред 13, $\mathit{largest}$ е равно на left. Условието на ред 13 е Истина и на ред 14, $A[j]$ и $A[\mathit{largest}]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle \mathit{left}, \text{Left}(\mathit{left})\rangle$ и $\langle \mathit{left}, \text{Right}(\mathit{left})\rangle$:

- нито един елемент от $A[\![i]\!]$, който е извън $A[\![j]\!]$, не е променян;
- $A[j] > A[\mathit{left}]$, така че $\langle j, \text{Left}(j)\rangle$ не може да е пирамидална инверсия;
- $A[j] \geqslant A[\mathit{right}]$, така че $\langle j, \text{Right}(j)\rangle$ не може да е пирамидална инверсия;
- $A[\![\mathit{right}]\!]$ не е бил модифициран;
- нито един от $A[\![\text{Left}(\mathit{left})]\!]$ и $A[\![\text{Right}(\mathit{left})]\!]$ не е бил модифициран.

[§]Последното имплицира, че ред 17 няма да бъде изпълнен.

Но на ред 15, j става равен на *left*. Следователно, инвариантът е в сила при следващото достигане на ред 3.

Случай iii: $A[left] \not> A[j]$ и $A[right] > A[j]$ в началото на изпълнението на цикъла. Условието на ред 6 е ЛЪЖА пристояването на ред 10 се случва. Условието на ред 11 е TRUE, така че присвояването на ред 12 се случва и когато изпълнението е на ред 13, *largest* е равно на *right*. Условието на ред 13 е ИСТИНА и на ред 14, $A[j]$ и $A[largest]$ биват разменени. Твърдим, че единствените възможни пирамидални инверсии в $A[\![i]\!]$ след размяната са $\langle right, \text{LEFT}(right)\rangle$ и $\langle right, \text{RIGHT}(right)\rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай I**. Но на ред 15, j става равен на *largest*. Следователно, инвариантът е в сила при следващото достигане на ред 3.

Случай IV: $A[left] > A[j]$ и $A[right] \not> A[j]$ в началото на изпълнението на цикъла. Условието на ред 6 е TRUE, така че присвояването на ред 8 се случва. Условието на ред 11 е FALSE, така че присвояването на ред 12 не се случва и когато изпълнението е на ред 13, *largest* е равно на *left*. Условието на ред 13 е TRUE и на ред 14, $A[j]$ и $A[largest]$ биват разменени. Твърдим, че единствените в $A[\![i]\!]$ след размяната са $\langle left, \text{LEFT}(left)\rangle$ и $\langle left, \text{RIGHT}(left)\rangle$. Доказателството е точно като доказателството на аналогичното твърдение в **Случай II**. Но на ред 15, j става равен на *largest*. Следователно, инвариантът е в сила при следващото достигане на ред 3.

Случай V: $A[left] \not> A[j]$ и $A[right] \not> A[j]$ в началото на изпълнението на цикъла. Но това е невъзможно при текущите допускания, защото очевидно тогава ред 17 би бил достигнат и изпълнението не би се върнало повече на ред 3.

Терминация. Цикълът може да бъде напуснат по два начина: през ред 3, когато $j > \left\lfloor \frac{n}{2} \right\rfloor$, и през ред 17. Да разгледаме първата възможност. Тогава и $\text{LEFT}(j)$, и $\text{RIGHT}(j)$ са индекси извън $A[1..n]$. В този случай инвариантът казва, 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 17. Очевидно, за да бъде достигнат ред 17, трябва да бъде вярно, че $largest = j$ на ред 13. А за да е вярно това, и $A[left] \leqslant A[j]$, и $A[right] \leqslant A[j]$ трябва да са в сила в началото на изпълнението на цикъла, понеже това е единственият начин ред 10 да бъде достигнат и ред 12 да не бъде достигнат. Но ако $A[left] \leqslant A[j]$ и $A[right] \leqslant A[j]$, пирамидални инверсии в $A[\![i]\!]$ няма, така че $A[\![i]\!]$ е пирамида. □

5.2.2.4 HEAPIFY в рекурсивен вариант

```
RECURSIVE HEAPIFY(A[1..n]: масив от цели числа, i: индекс в A)
 1  (* A[[LEFT(i)]] и A[[RIGHT(i)]], ако съществуват, са пирамиди *)
 2  left ← LEFT(i)
 3  right ← RIGHT(i)
 4  if left ⩽ n and A[left] > A[i]
 5      largest ← left
 6  else
 7      largest ← i
 8  if right ⩽ n and A[right] > A[largest]
 9      largest ← right
10  if largest ≠ i
11      swap(A[i], A[largest])
12      RECURSIVE HEAPIFY(A, largest)
```

Лема 35

При допускането, че $A[1..n]$ и i са такива, че всеки от $A[\![\textsc{Left}(i)]\!]$ и $A[\![\textsc{Right}(i)]\!]$, 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 $A[\![\textsc{Left}(i)]\!]$ *и* $A[\![\textsc{Right}(i)]\!]$*—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 $\textsc{Left}(i)$, и $\textsc{Right}(i)$ 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 $\leqslant h-1$ с корен някой j, такъв че $A[j]$ е в $A[\![i]\!]$, е изпълнено, че Recursive Heapify(A, j) превръща $A[\![j]\!]$ чрез разместивания в пирамида.

Индуктивна стъпка. Да разгледаме изпълнението на Recursive Heapify(A, i). Без ограничение на общността, да допуснем, че $\textsc{Left}(i) \leqslant n$ и $\textsc{Right}(i) \leqslant n$, така че редове 4 и 8 са съответно

if $A[\mathit{left}] > A[i]$

и

if $A[\mathit{right}] > A[\mathit{largest}]$

Случай I: $A[\mathit{left}] > A[i]$, $A[\mathit{right}] > A[i]$ и $A[\mathit{right}] > A[\mathit{left}]$. 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, $\mathit{largest}$ е равно на right. Условието на ред 10 е Истина и на ред 11, $A[i]$ и $A[\mathit{largest}]$ 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 $A[\![\mathit{right}]\!]$. От друга страна, $A[\![\mathit{left}]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\mathit{right}]$, е:

- по-голям от текущия $A[\mathit{left}]$ по допускане;
- не по-малък от текущия $A[\mathit{right}]$, по следните причини. Първоначално, $A[\![\mathit{right}]\!]$ е пирамида, така че бившият $A[\mathit{right}]$ е не по-малък от всеки друг елемент от $A[\![\mathit{right}]\!]$ в онзи момент (в началото). В края, елементите на $A[\![\mathit{right}]\!]$ се състоят от началните елементи без началния $A[\mathit{right}]$, плюс началния $A[i]$. Тъй като началният $A[\mathit{right}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![\mathit{right}]\!]$, текущият $A[i]$ не е по-малък от текущия $A[\mathit{right}]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай II: $A[\mathit{left}] > A[i]$, $A[\mathit{right}] > A[i]$ и $A[\mathit{right}] \not> A[\mathit{left}]$. 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 $A[\mathit{largest}]$ 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 $A[\![\mathit{left}]\!]$. От друга страна, $A[\![\mathit{right}]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\mathit{left}]$, е:

- не по-малък от текущия $A[\mathit{right}]$ по допускане
- не по-малък от текущия $A[\mathit{left}]$, по следните причини. Първоначално, $A[\![\mathit{left}]\!]$ е пирамида, така че бившият $A[\mathit{left}]$ е не по-малък от всеки друг елемент на $A[\![\mathit{left}]\!]$ в онзи момент (в началото). В края, елементите на $A[\![\mathit{left}]\!]$ се състоят от началните елементи без началния $A[\mathit{left}]$, плюс началния $A[i]$. Тъй като началният $A[\mathit{left}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![\mathit{left}]\!]$, текущият $A[i]$ не е по-малък от текущия $A[\mathit{left}]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай III: $A[\mathit{left}] \not> A[i]$ и $A[\mathit{right}] > A[i]$. 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 $A[\mathit{largest}]$ 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 $A[\![\mathit{right}]\!]$. От друга страна, $A[\![\mathit{left}]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\mathit{right}]$, е:

- по-голям от текущия $A[\mathit{left}]$ заради допусканията и транзитивността на неравенствата;
- не по-малък от текущия $A[\mathit{right}]$ по следните причини. Първоначално, $A[\![\mathit{right}]\!]$ е пирамида, така че бившият $A[\mathit{right}]$ е не по-малък от всеки друг елемент на $A[\![\mathit{right}]\!]$ в онзи момент (в началото). В края, елементите на $A[\![\mathit{right}]\!]$ се състоят от началните елементи без началния $A[\mathit{right}]$, плюс началния $A[i]$. Тъй като началният $A[\mathit{right}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![\mathit{right}]\!]$, текущият $A[i]$ не е по-малък от текущия $A[\mathit{right}]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай IV: $A[\mathit{left}] > A[i]$ и $A[\mathit{right}] \not> A[i]$. 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 *left*. Условието на ред 10 е Истина и на ред 11, $A[i]$ и $A[\mathit{largest}]$ 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 $A[\![\mathit{left}]\!]$. От друга страна, $A[\![\mathit{right}]\!]$ е пирамида, защото не е бил променян по никакъв начин. От трета страна, текущият $A[i]$, тоест първоначалният $A[\mathit{left}]$, е:

- по-голям от текущия $A[\mathit{right}]$ заради допусканията и транзитивността на неравенствата;

- не по-малък от текущия $A[\mathit{left}]$ поради следните причини. Първоначално, $A[\![\mathit{left}]\!]$ е пирамида, така че бившият $A[\mathit{left}]$ е не по-малък от всеки друг елемент от $A[\![\mathit{left}]\!]$ в онзи момент (в началото). В края, елементите на $A[\![\mathit{left}]\!]$ се състоят от началните елементи без началния $A[\mathit{left}]$, плюс началния $A[i]$. Тъй като началният $A[\mathit{left}]$ е по-голям от началния $A[i]$ и не по-малък от всеки друг елемент от $A[\![\mathit{left}]\!]$, текущият $A[i]$ не е по-малък от текущия $A[\mathit{left}]$.

Тогава, $A[\![i]\!]$ е пирамида.

Случай V: $A[\mathit{left}] \not> A[i]$ и $A[\mathit{right}] \not> A[i]$. 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, $\mathit{largest}$ е равно на i. Условието на ред 10 е Лъжа и изпълнението терминира, оставяйки $A[\![i]\!]$ непроменен. Съгласно допусканията, $A[\![\mathit{left}]\!]$ и $A[\![\mathit{right}]\!]$ са пирамиди и $A[\mathit{left}] \not> A[i]$ и $A[\mathit{right}] \not> A[i]$. Тогава, $A[\![i]\!]$ е пирамида. □

5.2.2.5 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 Recursive Heapify на стр. 294, или Iterative Heapify на стр. 292.

```
Build Heap(A[1..n]: array of integers)
1   for i ← ⌊n/2⌋ downto 1
2       Heapify(A, i)
```

Теорема 46: 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 13: Цикълът на Build Heap

Всеки път, когато изпълнението е на ред 1, $A[\![i+1]\!]$, $A[\![i+2]\!]$, $\ldots$, $A[\![n]\!]$ са пирамиди.

База. При първото достигане на ред 1, i е равно на $\lfloor \frac{n}{2} \rfloor$. Тогава $A[\![\lfloor \frac{n}{2} \rfloor + 1]\!]$, $A[\![\lfloor \frac{n}{2} \rfloor + 2]\!]$, $\ldots$, $A[\![n]\!]$ 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 47: 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 44, броят на върховете с височина k е $\left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil$. И така, ако $T(n)$ е функцията на сложността, то в най-лошия случай,

$$T(n) = \sum_{k=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{\lfloor \frac{n}{2^k} \rfloor}{2} \right\rceil \Theta(k) = \sum_{k=1}^{\lfloor \lg n \rfloor} \Theta\left(\frac{n}{2^k} k\right)$$

Да разгледаме $\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k$. В сила е

$$\sum_{k=1}^{\lfloor \lg n \rfloor} \frac{n}{2^k} k \leqslant \sum_{k=1}^{\infty} \frac{n}{2^k} k = n \sum_{k=1}^{\infty} \frac{k}{2^k}$$

Тривиално се доказва, че редът $\sum_{k=1}^{\infty} \frac{k}{2^k}$ е сходящ, примерно с критерия на d'Alembert:

$$\lim_{k \to \infty} \frac{\frac{k+1}{2^{k+1}}}{\frac{k}{2^k}} = \frac{1}{2} < 1$$

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 $T(n) \asymp n$. □

5.3 Сортиращ алгоритъм HEAPSORT

Следният сортиращ алгоритъм е предложен от Williams [146]. Той е първият от бързите сортиращи алгоритми, които ще разгледаме. "Бърз" наричаме алгоритъм, чиято сложност по време е $O(n \lg n)$.

Нека $A.size$ относно масива $A[1 \,..\, n]$ е число, такова че $1 \leqslant A.size \leqslant n$ и HEAPIFY работи върху подмасива $A[1 \,..\, A.size]$, а **не** върху $A[1 \,..\, n]$.

HEAPSORT($A[1 \,..\, n]$)

```
1  BUILD HEAP(A)
2  A.size ← n
3  for i ← n downto 2
4      swap(A[1], A[i])
5      A.size ← A.size − 1
6      HEAPIFY(A, 1)
```

Лема 36

HEAPSORT е сортиращ алгоритъм.

Доказателство: Нека $A'[1..n]$ означава първоначалния масив. Следното твърдение е инвариант на цикъла за **for**-цикъла (редове 3–6).

Инвариант 14: Цикълът на HEAPSORT

Всеки път, когато изпълнението на HEAPSORT е на ред 3:

- Текущият подмасив $A[i+1..n]$ се състои от $n-i$ на брой най-големи елементи на $A'[1..n]$ в сортиран вид.
- Освен това, текущият $A[1..i]$ е пирамида.

База. При първото достигане на ред 3, $i = n$. Подмасивът $A[i+1..n]$ е празен, следователно, в празния смисъл (*vacuously*), той се състои от нула на брой най-големи елементи от $A'[1..n]$, в сортиран вид, следователно първата част на инварианта е в сила. $A[1..n]$ е пирамида съгласно Теорема 46, 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 3, което не е последното. Нека да наричаме масива A в този момент, A''. Съгласно първата част на индуктивното предположение, $A''[i+1..n]$ се състои от $n-i$ на брой елементи от A' в сортиран вид. Съгласно втората част на индуктивното предположение, $A''[1]$ е максимален елемент от $A''[1..i]$. След размяната на ред 4, $A''[i..n]$ съдържа $n-i+1$ на брой най-големи елемента на A' в сортиран вид. Спрямо новата стойност на i при следващото достигане на ред 3, първата част на инварианта е в сила.

Ще докажем, че втората част на инварианта е в сила. Да приложим Лема 34 или Лема 35 в зависимост от това дали ползваме рекурсивна или итеративна HEAPIFY функция на ред 6. Трябва да се има предвид, че HEAPIFY работи върху $A[1..i-1]$, защото i е равно на $A.size$ в момента , в който изпълнението е на ред 6, а на ред 5, $A.size$ е получил стойност $i-1$. Заради това, на ред 6 текущият $A[i]$ е "извън обхвата" на HEAPIFY.

Терминация. Да разгледаме момента, в който изпълнението е на ред 3 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 $A'[1..n]$ в сортиран вид". Но тогава $A[1]$ е минимален елемент на $A'[1..n]$. С това доказателството за коректност на HEAPSORT приключва. □

Да разгледаме сложността по време на HEAPSORT. Сложността на BUILD HEAP е $\Theta(n)$, което доказахме в Теорема 47. Сложността на **for**-цикъла (редове 3–6) $\Theta(n \lg n)$, 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. (5.2)) 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. 294, 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 4, 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 31: HEAPSORT има сложност $\Theta(n \lg n)$ в най-добрия случай

Както отбелязахме на стр. 76, ние не разглеждаме сложността по време в най-добрия случай, понеже с тази мярка лесно може 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 $A[1] \leqslant \cdots \leqslant A[n]$, прекрати работата", които свалят сложността по време в най-добрия случай на $\Theta(n)$. Разглеждаме алгоритъма, както е описан на предишната страница.

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 $\Omega(n)$ върхове е вярно, че всеки от тях участва в $\Omega(\lg n)$ размени.

Нека $A[1 \,..\, n]$ е входният масив. Неко S е множеството от входните елементи, като $|S| = n$, понеже във входа няма повторения. Нека $X, Y \subseteq S$, като $|X| = |Y| = \frac{n}{2}$ [a] и $\forall x \in X \, \forall y \in Y : x < y$.
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 в общия случай индуцират множество дървета.
Най-много $\frac{n}{4}$ елемента от Y може да са листа на Π, понеже в двоично дърво листата не може да са повече от половината от върховете. А D е двоично дърво с $\frac{n}{2}$ върхове и листата на D са подмножество на листата на Π.
Нека L_i са върховете в Π с височина i, които са от Y. Току-що показахме, че $|L_0| \leqslant \frac{n}{4}$. Нека h е височината на Π. Принадлежността към $L_0, \ldots, L_h$ генерира разбиване на Y, като

$$|L_0| + |L_1| + \cdots + |L_h| = \frac{n}{2}$$

Предвид това, че $|L_0| \leqslant \frac{n}{4}$, имаме

$$|L_1| + |L_2| + \cdots + |L_h| \geqslant \frac{n}{4}$$

Ако допуснем, че $|L_1| < \frac{n}{8}$, следва, че

$$|L_2| + \cdots + |L_h| > \frac{n}{8}$$

което е очевидно невъзможно, защото $L_2, \ldots, L_h$ са пълни ниво́ в Π. Доказахме, че $|L_1| \geqslant \frac{n}{8}$. Конкретната стойност не е важна, 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 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. Ключовото наблюдение следното.

- Всеки елементи от L_0 **може** да бъде сложен на окончателното си място само с две размествания: първо "скача" във върха на пирамидата чрез swap$(A[1], A[i])$ на ред 4, после HEAPIFY$(A, 1)$ не размества нищо, понеже този елемент е достатъчно голям, и следващият swap$(A[1], A[i])$ на ред 4 ще го "изхвърли" извън текущия $A[1 \,..\, A.\text{size}]$ на окончателното му място в $A[1 \,..\, n]$.

- За разлика от елементите от L_0, тези от L_1 **трябва** да "пропътуват" цялото разстояние до корена, преди да бъдат "изхвърлени" извън текущия $A[1 \,..\, A.\text{size}]$ на окончателните си места в $A[1 \,..\, n]$ чрез swap$(A[1], A[i])$ на ред 4.

  За да се убедим, че е така, достатъчно е да забележим, че $L_1 \subset Y$ и елементите от L_1 трябва да бъдат сложени по окончателните си места още докато i е по-голяма или равна на $\frac{n}{2}$. Но единственият начин това да стане е чрез swap$(A[1], A[i])$ на ред 4. Тъй като i е прекалено голяма, за да бъде индекса на елемент от L_1, той 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** ниво́. Ерго, всеки такъв връх е бил участник в $\Omega(\lg n)$ размени.

Щом всеки от $\Omega(n)$ върха участва в $\Omega(\lg n)$ размени, алгоритъмът работи в $\Omega(n \lg n)$ време винаги.

[a] Когато пишем "$\frac{n}{2}$", имаме предвид $\lfloor \frac{n}{2} \rfloor$ или $\lceil \frac{n}{2} \rceil$. Това е без значение за асимптотиката.

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 [131, 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 [131, 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

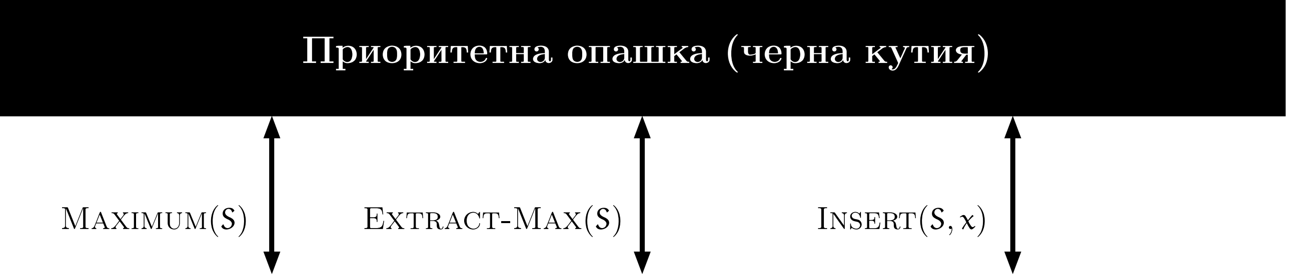
Приоритетните опашки имат елементи, всеки от които има две стойности: ключ и данни, свързани с този ключ. Ключовете обикновено са цели числа, макар че всеки тип данни, който може да се използва в контекста на сортирането, може да е типът данни на ключовете. Приоритетните опашки, подобно на пирамидите, 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 и др. [31, стр. 162]. Авторите дават 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 след малко, не е част от стандартния интерфейс на приоритетните опашки. Sedgewick и Wayne [131, 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[]`



Фигура 5.18: АТД приоритетна опашка като черна кутия. Конкретната имплементация е скрита. Отвън се "виждат" само трите функции на интерфейса.

- `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 и др., като INCREASE-KEY не е част от него, освен ако това не е казано изрично. Фигура 5.18 илюстрира приоритетна опашка като черна кутия плюс интерфейс.

Приложенията на приоритетните опашки са много. Всяка дейност, в която трябва да бъдат обработвани някакви неща, пристигащи асинхронно, като в даден момент може да се обработва само едно нещо, може да се моделира с приоритетна опашка. Нещата може да са, print jobs в операционна система, кораби за разтоварване в пристанище, самолети, кацащи на летище, спешно отделение в болница, и така нататък. С приоритетни опашки може и да се сортира: HEAPSORT можем да разглеждаме като сортиращ алгоритъм с последователно избиране на максимален елемент (какъвто е SELECTION SORT), 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 MAXIMUM(S) и EXTRACT-MAX(S) при тази реализация имат сложност $\Theta(1)$, докато 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, MAXIMUM(S) и EXTRACT-MAX(S) имат сложност $\Theta(n)$, докато 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 [31, 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) [86, 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. $S.size$ е размерът на пирамидата, тоест, само $S[1..S.size]$ е пирамида. Допускаме, че винаги $S.size < n$, така че няма да се притесняваме за препълване на масива в Heap-Insert. Ако пирамидата е празна, както е в самото начало при създаването ѝ, $S.size = 0$.

```
Heap-Maximum(S[1..n])
1  (* Допускаме, че S.size ⩾ 1 *)
2  return S[1]
```

```
Heap-Extract-Max(S[1..n])
1  (* Допускаме, че S.size ⩾ 1 *)
2  max ← S[1]
3  S[1] ← S[S.size]
4  S.size ← S.size − 1
5  Heapify(S, 1)
6  return max
```

```
Heap-Insert(S[1..n], x)
1  S.size ← S.size + 1
2  i ← S.size
3  S[i] ← x
4  while i > 1 and S[Parent(i)] < S[i] do
5      swap(S[Parent(i)], S[i])
6      i ← Parent(i)
```

Коректността на тези функции е абсолютно очевидна след изложението в Секция 5.2. Сложността на Heap-Maximum е $\Theta(1)$, а на Heap-Extract-Max и Heap-Insert е $\Theta(\lg(S.size))$. Ако не ползваме "n" за големината на масива, а за големината на самата пирамида, и изразяваме сложността чрез символа "n", то двете функции 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. В някои случаи (реализиране на приоритетна опашка), оптималният подход (пирамида) е **компромис** (на английски, *tradeoff*) между два екстремални подхода (сортиран масив и несортиран масив). Относно две различни изисквания

(бързо изваждане на максимален елемент и бързо вмъкване на нов елемен), при единият екстремален подход имаме оптимален резултат ($\Theta(1)$) по първото изискване, но изключително лош резултат ($\Theta(n)$) по второто; а при другия подход, оптимален резултат по второто изискване, но изключително лош резултат по първото. Компромисният подход (отказ от константно време за коя да е от операциите и постигане на задоволително време и за двете операции), направен интелигентно, се оказва печеливш.

5.4.4 Функцията Increase-Key

Според [31], функцията от интерфейса Increase-Key може да се реализира с двоична пирамида по следния начин.

```
Heap-Increase-Key(S[1 .. n], i, k)
1  if k < S[i]
2      error "опит да бъде намален ключът"
3  S[i] ← k
4  while i > 1 and S[Parent(i)] < S[i] do
5      swap(S[Parent(i)], S[i])
6      i ← 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 било Increase-Key(S, x, k), където x е елемент от опашката. Но ако реализираме функцията с пирамида и дадем параметър x, който е елемент (с други думи, ако x е старата стойност на ключа, която трябва да стане k), този елемент трябва да бъде първо открит, което е операция с линейна сложност в пирамидите[†], дори ако допуснем уникални ключове. Така че, за да постигнем логаритмична сложност и на тази функция от интерфейса, се налага да "счупим" изискването за абстрактен тип данни и да заложим в нейната спецификация, че приоритетната опашка е реализирана именно чрез масив и че някак знаем индекса на елемента, чийто ключ искаме да повишим.

[†]Примерно, най-малкият ключ може да е във всяко листо

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 алгоритмичната 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-Младей: 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

Въвеждаме понятието *алгоритмична схема*. Това означава означава общ 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 се споменават бегло на стр. 718.

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 по-малък от някаква предварително зададена големина, задачата върху него се решава по някакъв тривиален начин, примерно с "брутална сила", и се преминава директно към стъпка **Комбинирай**.

[†] 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.

- **Владей** – пускаме алгоритъма върху всяка от по-малките части и получаваме резултатите от всяко пускане.
- **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. Разделянето се случва ако и само ако размерът на входът надхвърля еди-колко; в противен случай, генерираме резултат по друг начин, например с пълно изчерпване на възможностите, който метод наричаме "с брутална сила"[†].

Както ще видим в многобройни примери, за някои изчислителни задачи има ефикасни алгоритми, които първо разделят входа на части, стига той да е достатъчно голям, получават чрез рекурсивни викания изход за всяка от частите, и накрая комбинират тези изходи, за да конструират окончателния изход. За съжаление, този подход съвсем не е универсален. За много задачи—по правило, за всички много тежки изчислителни задачи—схемата **Разделяй-и-Владей** не е приложима[‡].

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 на стр. 242. 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 премитане, 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 [16].

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

$$|a-c| + |b-d|$$

[†]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—който се е справял с даден силен противник, като първо е създавал вътрешни разделения в него, а после 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 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.

[§]L_m разстоянието между (a, b) и (c, d) е $(|a-c|^m + |b-d|^m)^{\frac{1}{m}}$.

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 в $\Theta(1)$.

Изч. Задача 19: 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 $1 \leqslant i \leqslant n$; $n \geqslant 2$
решение: $\min\{\text{dist}(p_i, p_j) : 1 \leqslant i < j \leqslant n\}$

Изч. Задача 20: 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 $1 \leqslant i \leqslant n$; $n \geqslant 2$
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.

Ние ще конструираме алгоритъм за задачата в олекотения вариант. Има очевиден алгоритъм с брутална сила, опитващ всички ненаредени двойки точки, но то работи във време $\Theta(n^2)$. Това е прекалено бавно. Ще построим алгоритъм, работещ в $\Theta(n \lg n)$, изграден по схемата Разделяй-и-Владей. Изложението тук повтаря в основни линии изложението в [31, стр. 1039].

Ще направим пример с точките, показани на Фигура 6.1.

Фигура 6.1 : Точките, които ще ползваме за пример.

За удобство да въведем една 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 началната сортировка.

   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 28 в Допълнение 23). 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:

$$A = [p_5, p_2, p_{10}, p_{11}, p_1, p_7, p_8, p_4, p_{12}, p_6, p_9, p_3]$$
$$B = [p_7, p_9, p_{10}, p_4, p_1, p_5, p_3, p_{12}, p_{11}, p_8, p_2, p_6]$$

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:

$$B = [(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1), (p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)] \quad (6.1)$$

Смисълът от това ще стане ясен нататък.

Допълнение 32: 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):

$$P = [(69, 33), (29, 54), (96, 44), (75, 46), (45, 34), (62, 52), (56, 10), (21, 36), (36, 20), (41, 47), (78, 63), (92, 15)]$$

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

$$B = [((56, 10), 7), ((92, 15), 12), ((36, 20), 9), ((69, 33), 1), ((45, 34), 5), ((21, 36), 8), ((96, 44), 3), ((75, 46), 4), ((41, 47), 10), ((62, 52), 6), ((29, 54), 2), ((78, 63), 11)]$$

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

$$P = [((69, 33), ?), ((29, 54), ?), ((96, 44), ?), ((75, 46), ?), ((45, 34), ?), ((62, 52), ?), ((56, 10), ?), ((21, 36), ?), ((36, 20), ?), ((41, 47), ?), ((78, 63), ?), ((92, 15), ?)]$$

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 $\Theta(n \lg n)$.

[a] Дори 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 $n \geqslant 2$), така че базата не трябва да е $n = 1$.

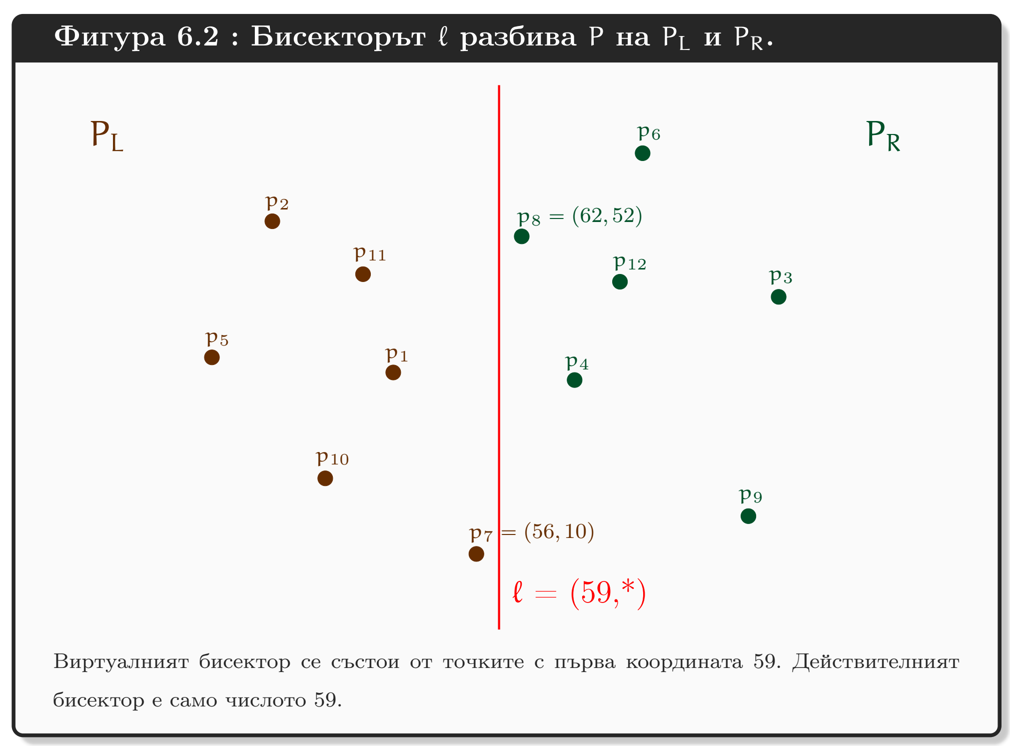
Да допуснем, че $n > 3$.

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 $A\left[1..\left\lceil\frac{n}{2}\right\rceil\right]$ се казва A_L, а подмасивът $A\left[\left\lceil\frac{n}{2}\right\rceil+1..n\right]$ 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 $\frac{1}{2}\left(A\left[\left\lceil\frac{n}{2}\right\rceil\right].x + A\left[\left\lceil\frac{n}{2}\right\rceil+1\right].x\right)$. Не е съществено важно бисекторът да е именно средното аритметично! Ако $A\left[\left\lceil\frac{n}{2}\right\rceil\right].x$ и $A\left[\left\lceil\frac{n}{2}\right\rceil+1\right].x$ са различни, всяко число от отворения интервал $\left(A\left[\left\lceil\frac{n}{2}\right\rceil\right].x, A\left[\left\lceil\frac{n}{2}\right\rceil+1\right].x\right)$ 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 $\left\lceil\frac{n}{2}\right\rceil = 6$ и $\left\lceil\frac{n}{2}\right\rceil+1 = 7$, така че $A\left[\left\lceil\frac{n}{2}\right\rceil\right].x = 56$ и $A\left[\left\lceil\frac{n}{2}\right\rceil+1\right].x = 62$, така че $\frac{1}{2}\left(A\left[\left\lceil\frac{n}{2}\right\rceil\right].x + A\left[\left\lceil\frac{n}{2}\right\rceil+1\right].x\right) = \frac{56+62}{2} = 59$, което е и стойността на бисектора.

Фигура 6.2 : Бисекторът ℓ разбива P на P_L и P_R.



Виртуалният бисектор се състои от точките с първа координата 59. Действителният бисектор е само числото 59.

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.

Фигура 6.3 : Дегенеративни случаи. ℓ си остава бисектор.

Точките имат една и съща първа координата. Визуално, бисекцията $P_L = \{q_1, q_2, q_5, q_7, q_8, q_9\}$ и $P_R = \{q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}\}$ е произволна.

Всички точки съвпадат. Бисекцията е произволна и не е показана.

Както вече казахме, алгоритмичната бисекцията 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 въде подчертано отново.

Наблюдение 36

В текущия контекст, в общия случай не е вярно, че $B_L = B\left[1 \mathinner{..} \left\lceil\frac{n}{2}\right\rceil\right]$ и не е вярно, че $B_R = B\left[\left\lceil\frac{n}{2}\right\rceil + 1 \mathinner{..} n\right]$. 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

$$\begin{aligned} B = [&(p_7, 6), (p_9, 11), (p_{10}, 3), (p_4, 8), (p_1, 5), (p_5, 1),\\ &(p_3, 12), (p_{12}, 9), (p_{11}, 4), (p_8, 7), (p_2, 2), (p_6, 10)] \end{aligned}$$

Слагаме в B_L точно тези елементи на B, чиито индекси в A са в множеството $\left\{1, 2, \ldots, \left\lceil\frac{n}{2}\right\rceil\right\}$, а в B_R слагаме точно тези елементи на B, чиито индекси в A са в множеството $\left\{\left\lceil\frac{n}{2}\right\rceil + 1, \left\lceil\frac{n}{2}\right\rceil + 2, \ldots, n\right\}$. В нашия пример, в B_L слагаме елементите на B, чиято втора компонента е в множеството $\{1, 2, \ldots, 6\}$, а в B_R слагаме тези, чиято втора компонента е в множеството $\{7, 8, \ldots, 12\}$. 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

$$\begin{aligned} B_L &= [p_7, p_{10}, p_1, p_5, p_{11}, p_2]\\ B_R &= [p_9, p_4, p_3, p_{12}, p_8, p_6] \end{aligned}$$

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

$$d_M = \min\{\text{dist}(x, y) \mid x \in P_L, y \in P_R\} \tag{6.2}$$

  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 неефикасен. 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 37

Ако някакви прекосяващи отсечки не са къси, то няма смисъл да изчисляваме дължините им при изчисляването на d_M.

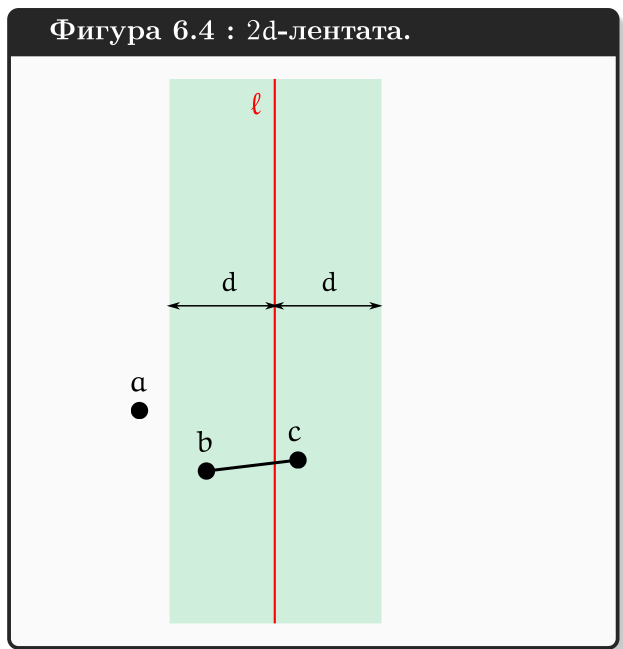
Да си представим точките от P заедно с правата-бисектор ℓ. Да дефинираме $2d$-*лентата* като областта от равнината, чиито точки са на разстояние не по-голямо от d от двете страни на ℓ. Иначе казано, ако поне единият край на някоя прекосяваща отсечка е извън $2d$-лентата, то нейната дължина е по-голяма от d и не ни интересува.

Наблюдение 38

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 38 и Наблюдение 37, 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:

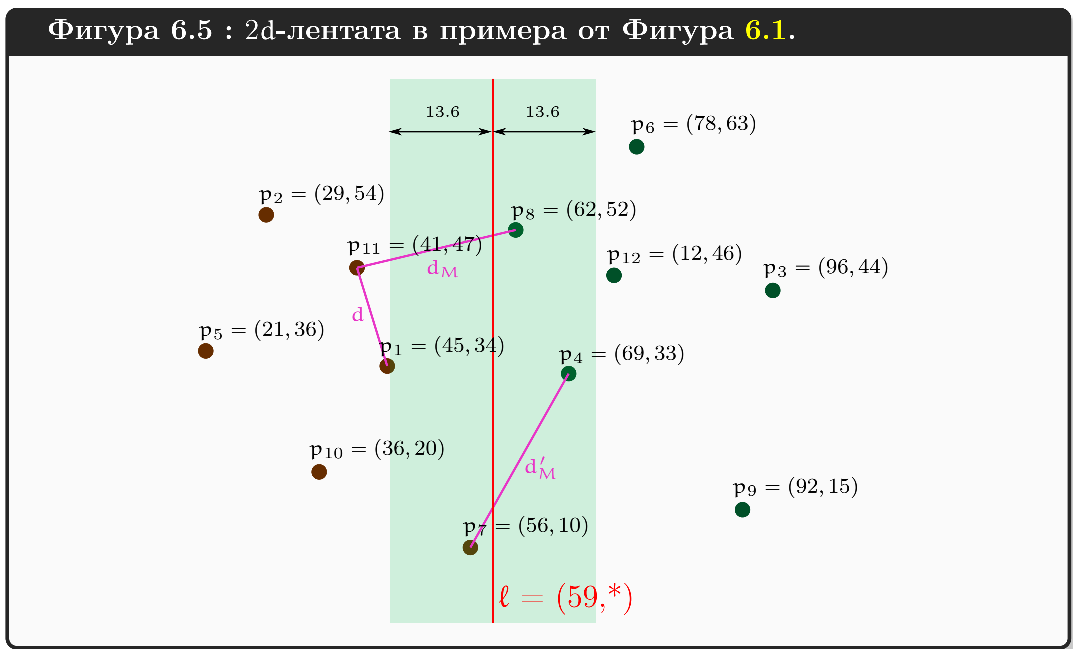
$$d_M = \min\{\text{dist}(x, y) \mid x \in P_L, y \in P_R\} \tag{6.3}$$
$$d'_M = \min\{\text{dist}(x, y) \mid x \in P_L \text{ и } x \text{ е в } 2d\text{-лентата}, y \in P_R \text{ и } y \text{ е в } 2d\text{-лентата}\} \tag{6.4}$$

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 $d_M > d$[†]; 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 38, не е къса, от което следва, че $d_M > d$.

Фигура 6.5 : 2d-лентата в примера от Фигура 6.1.



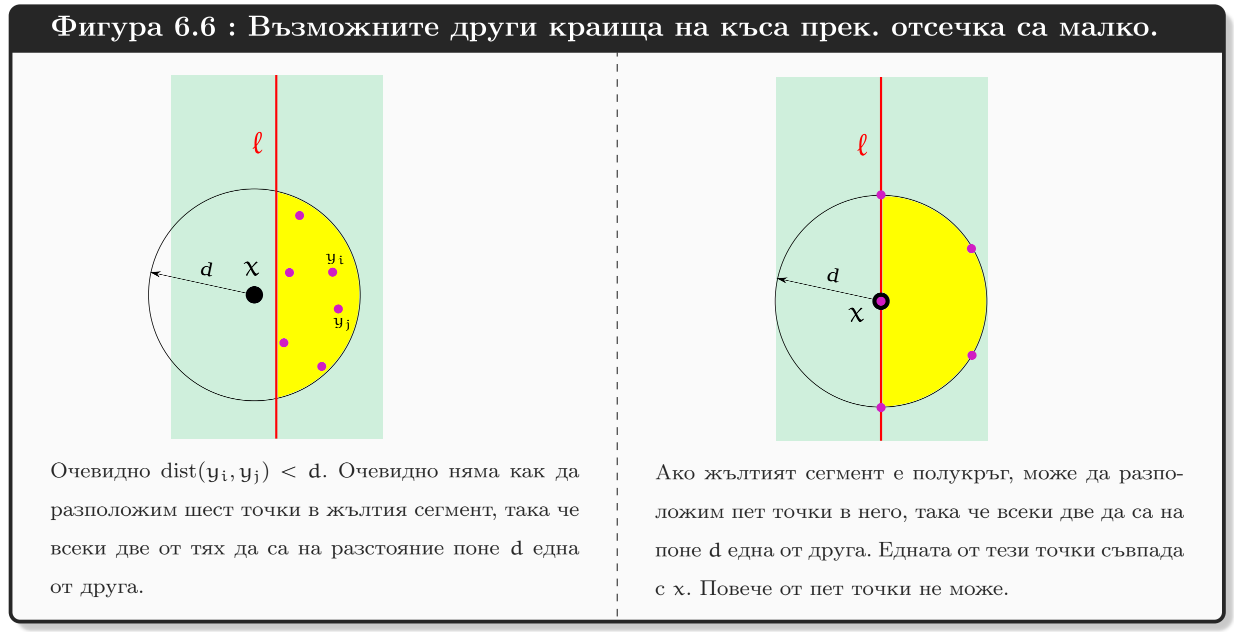
Само три точки от P попадат в $2d$-лентата: p_7, p_4 и p_8 (p_1 е извън $2d$-лентата). Тогава $d'_M = \min\{\text{dist}(x, y) \,|\, x \in \{p_7\}, y \in \{p_4, p_8\}\} = \text{dist}(p_4, p_7) = \sqrt{(56-69)^2 + (10-33)^2} \approx 26.41968963$. Виждаме, че в нашия пример е изпълнено $d_M < d'_M$, понеже 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 подобрение изобщо не е подобрение. Ако всички точки от P са в $2d$-лентата и търсим d_M изчерпателно, то сложността пак би била квадратична.

Второ подобрение. 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 в $y_1, y_2, \ldots, y_{100}$ 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 $y_1, \ldots, y_{100}$ 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 близо една до друга, че разстоянието d_R не може да е повече от $\text{dist}(y_i, y_j)$, следователно, би трябвало $d \leqslant \text{dist}(y_i, y_j)$, 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.

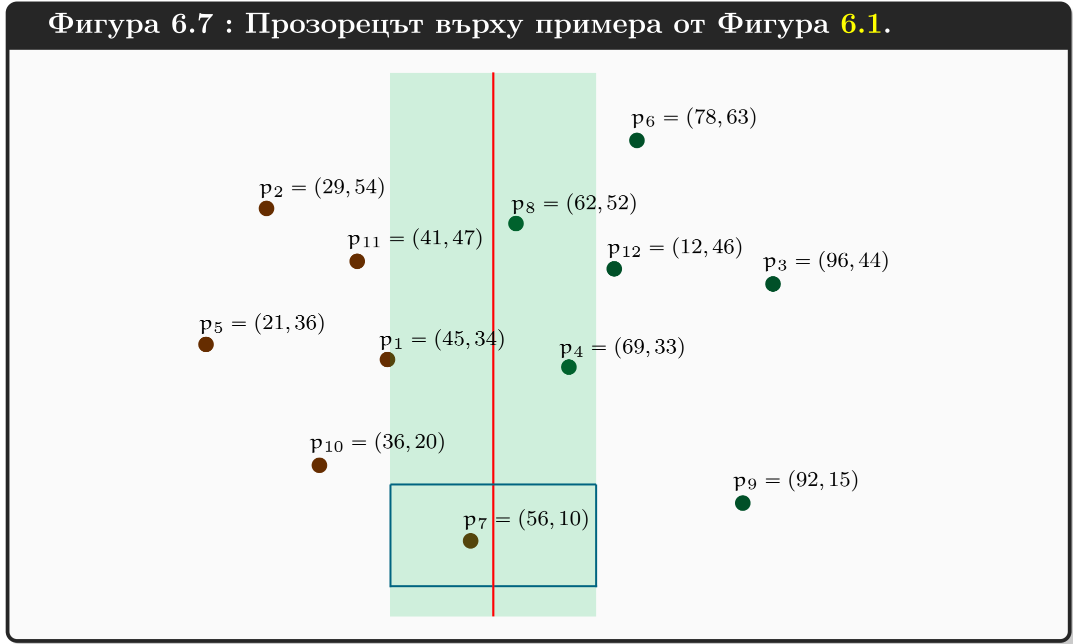
Очевидно $\mathrm{dist}(y_i, y_j) < d$. Очевидно няма как да разположим шест точки в жълтия сегмент, така че всеки две от тях да са на разстояние поне d една от друга.

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 [31, 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.

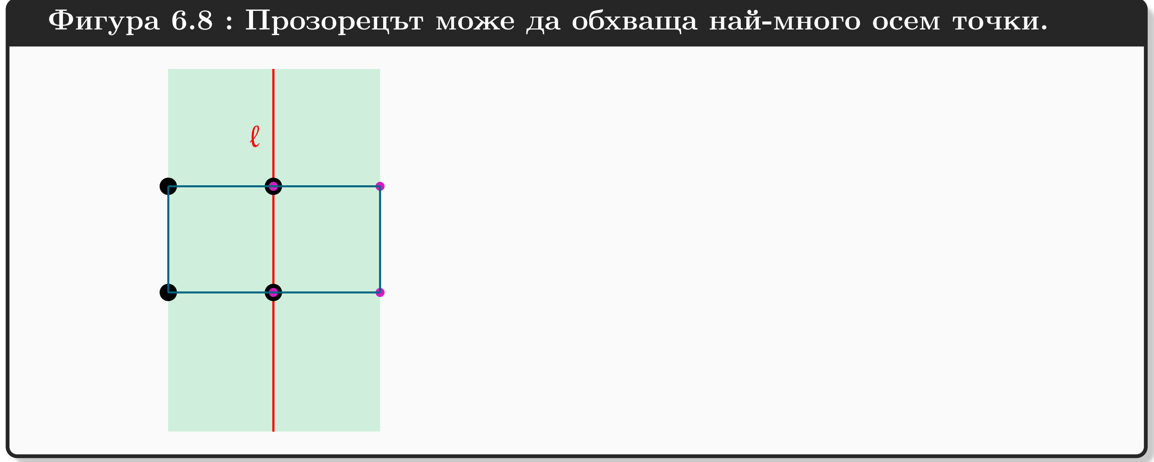
Представяме си *прозорец* – правоъгълник с размери $2d \times d$, със страни, успоредни на координатните оси, като голямата страна е успоредна на абсцисата, а центърът на прозореца лежи върху бисектора. Идеята е да плъзнем прозореца от най-долу (най-малката втора координата) до най-горе (най-голямата втора координата), като центърът му се плъзга по бисектора, а страните му остават успоредни на координатните оси. Фигура 6.7 показва прозореца, нарисуван 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 точкти 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 кой 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 $|p_i.x - \beta| \leqslant d$.

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 $k \geqslant 8$. 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 $D[i \mathinner{..} j]$, 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 $D[i \mathinner{..} j]$ е $\binom{8}{2} = 28$.
  - i++, j++.

След края на това, полученото m е точно търсеното d'_M.

> *Забележете, че масивът $D[i \mathinner{..} j]$ не реализира точно прозореца $2d \times d$, за който стана дума във второто подобрение. Ако D е достатъчно голям, нашата алгоритмична реализация работи с $D[i \mathinner{..} j]$, 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 $D[i \mathinner{..} j]$ 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 опитва 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 [22], [23]. Изложението тук е по учебника на Cormen, Leiserson, Rivest и Stein [31, стр. 220–222].

Изч. Задача 21: Избор на елемент по големина

екземпляр: крайно непразно множество $S \subset \mathbb{Z}$; $k \in \mathbb{N}$, такова че $0 \leqslant k \leqslant |S| - 1$
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 [31] задачата е дефинирана точно така.

Тази задача може да се реши тривиално във време $\Theta(n \lg n)$ чрез бързо сортиране на S, например чрез Heapsort, последвано от връщане на $(k+1)$-вия елемент. Тук ще разгледаме по-бързо решение: линеен алгоритъм, наречен PICK в [23] и Select в [31]. Алгоритъмът вика процедура Partition, която, както е практически същата като Partition–Lomuto от алгоритъма Quicksort (вижте Подсекция 6.4.2), с две незначителни разлики:

- първо, елементът *pivot*, спрямо който се разместват елементите на масива, не е непременно най-десният елемент поначало, а е произволен елемент, който се среща; в началото той се открива и се разменя с най-десния, след което се изпълнява Partition–Lomuto;
- второ, върната стойност е *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.

Ефектът от работата на PARTITION е следният: елементите на $A[l .. h]$ биват разместени по такъв начин, че:

- подмасивът $A[l .. pp-1]$ се състои точно от елементите на $A[l .. h]$, по-малки от $pivot$;
- $A[pp]$ съдържа $pivot$;
- подмасивът $A[pp+1 .. h]$ се състои точно от елементите на $A[l .. h]$, по-големи от $pivot$;

Тук допускаме, че елементите на масива са два по два различни; това допускане не е в сила при QUICKSORT (Секция 6.4), но в контекста на алгоритъма за избор на елемент по големина допускаме, че няма елементи с еднакви ключове.

Подчертаваме, че алгоритъмът PARTITION **не сортира**: в края му, нито "малките елементи" в $A[l .. pp-1]$ са непременно сортирани, нито "големите елементи" в $A[pp+1 .. h]$ са непременно сортирани. Следва псевдокод на PARTITION. Функцията find-index връща индекса на елемента-първи аргумент в масива-втори аргумент, при допускането, че този елемент се среща точно веднъж в този масив.

```
PARTITION(A[l .. h]: int; pivot: element from A[l .. h])
1  k ← find-index(pivot, A[l .. h])
2  swap(A[k], A[h])
3  pp ← l
4  for i ← l to h − 1
5      if A[i] < pivot
6          swap(A[i], A[pp])
7          pp ← pp + 1
8  swap(A[pp], A[h])
9  return pp − 1
```

Следва псевдокод на SELECT. Да си припомним дефиницията на ИЗБОР НА ЕЛЕМЕНТ ПО ГОЛЕМИНА на предната страница. Множеството S е реализирано чрез масив $A[1..n]$, чиито елементи са два по два различни. Вторият аргумент k е броят на елементите, по-малки от този, който връща алгоритъмът. Описанието на алгоритъма, взето от [31], е на по-високо ниво от обичайния псевдокод.

```
SELECT(A[1..n]: int; k ∈ {0, 1, ..., n − 1})
1   (∗ A[x] ≠ A[y] за 1 ⩽ x < y ⩽ n ∗)
2   if n = 1
3       return A[1]
4   q ← ⌈n/5⌉, r ← n mod 5
5   if r = 0
6       разбий A на подмасиви B_1, ..., B_q, всеки с 5 елемента
7   else
8       разбий A на подмасиви B_1, ..., B_{q−1}, всеки с 5 елемента, и B_q с r елемента
9   за i ∈ {1, 2, ..., q}, сортирай B_i и намери съответната медиана m_i
10  направи масив [m_1 .. m_q] от медианите и го наречи C
11  m ← SELECT(C, ⌊(q−1)/2⌋)
12  j ← PARTITION(A, m)
13  if k = j
```

```
14         return m
15     else
16         if k < j
17             return Select(A[1 .. j], k)
18         else
19             return Select(A[j + 2 .. n], k − j − 1)
```

Да разгледаме малък пример. Нека $n = 25$ и елементите на A са 1, 2, ..., 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 6) 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 9), 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 10). На ред 11, $\left\lfloor \frac{5-1}{2} \right\rfloor = 2$, 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 11, m получава стойност 11.

На ред 12 викаме PARTITON върху A с втори аргумент 11. На ред 12 процедурата връща $j = 10$ (което е напълно очаквано, ако елементите на A са 1, ..., 25 и $pivot$ е 11). Самият A изглежда така след разместванията, направени от PARTITION:

$$A = [6, 8, 5, 1, 7, 4, 3, 10, 2, 9, 11, 20, 16, 24, 22, 25, 12, 18, 15, 14, 13, 19, 23, 21, 17]$$

Тъй като $k = 17$ и $j = 10$, условието на ред 13 е лъжа, условието на ред 16 също е лъжа и изпълнението отива на ред 19. $A[j + 2 .. n]$ е $A[12 .. 25]$. В случая:

$$A[12 .. 25] = [20, 16, 24, 22, 25, 12, 18, 15, 14, 13, 19, 23, 21, 17] \tag{6.5}$$

$k - j - 1$ е 6, така че викането на ред 19 е SELECT($A[12 .. 25], 6$). Очевидно в този пример това е коректно, понеже осемнадесетия ($17 + 1 = 18$) по големина елемент на входния A е същият като седмия ($6+1 = 7$) по големина елемент на "редуцирания" A от (6.5). Забележете, че в конкретния пример седмият по големина елемент в масива в (6.5) се намира на седма позиция (става дума за числото 18), но това е случайно; ако винаги беше така, нямаше да има нужда от рекурсивното викане на ред 19.

С това приключва разглеждането на примера.

Коректност. Сега ще докажем коректността на SELECT. Твърдим, че ако е даден масив $A[1..n]$ от две по две различни цели числа и $k \in \{0, 1, \dots, n-1\}$, то SELECT$(A, k)$ връща $(k+1)$-вия по големина елемент на A. Ще покажем коректността на алгоритъма със силна индукция по n.

Спирачката на рекурсията е $n = 1$ (ред 2). Ако $n = 1$, единствената допустима стойност за k е 0, понеже $k \in \{0, 1, \dots, n-1\}$. Наистина, на ред 3 алгоритъмът връща $A[1]$, който наистина е $(k+1)$-вия по големина елемент при $k = 0$. ✓

Да допуснем, че $n \geqslant 2$. Разглеждаме направо ред 12. За коректността няма значение точно каква е стойността на m. От значение е само това, че m е елемент от масива и PARTITION(A, m) на ред 12 размества A по такъв начин и връща такава стойност j, че:

- всички елементи, по-малки от m, са в подмасива $A[1..j]$;
- $A[j+1]$ съдържа m;
- всички елементи, по-големи от m, са в подмасива $A[j+2..n]$.

Разглеждаме три случая.

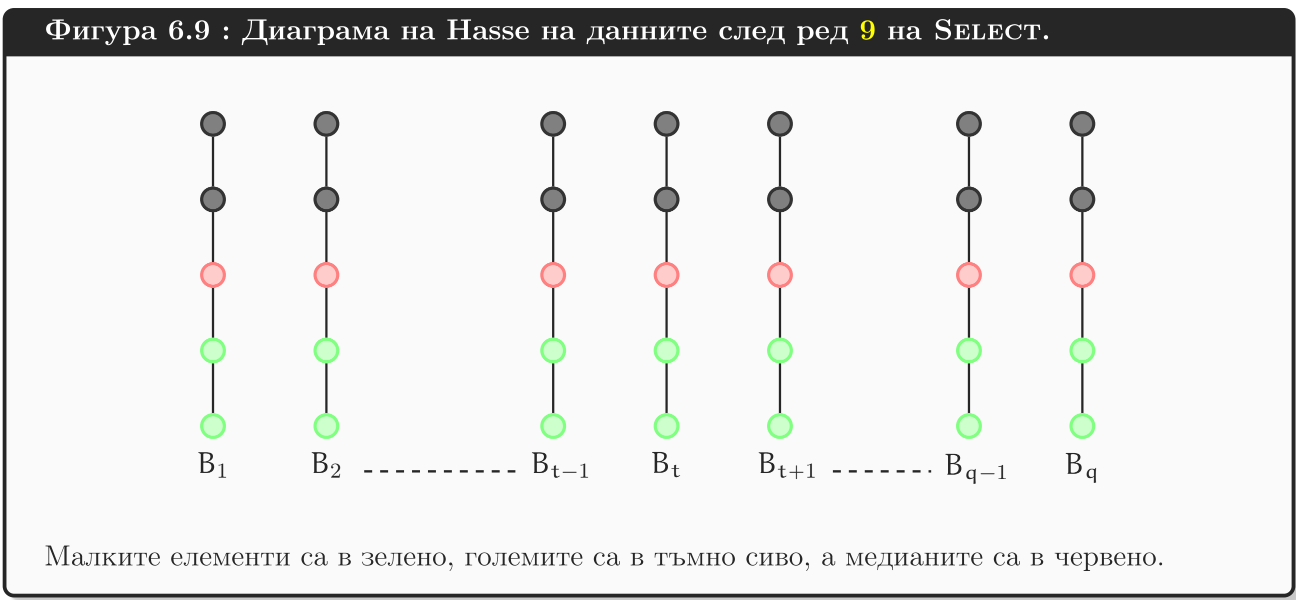
Случай 1. Ако $k = j$ (ред 13), то m е $(k+1)$-вият по големина елемент и алгоритъмът коректно връща m на ред 14.

Случай 2. Ако $k < j$ (ред 16), 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 $A[1..j]$. Викането на процедурата SELECT$(A[1..j], k)$ на ред 17 връща $(k+1)$-вия по големина елемент на $A[1..j]$ съгласно индукционното предположение.

Случай 3. Ако $k > j$ (ред 19), 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 $A[j+1]$. Следователно, викането на SELECT$(A[j+2..n], k-j-1)$ на ред 19 връща желания елемент.

Сложност по време. 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 9 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 9 на Select.



Малките елементи са в зелено, големите са в тъмно сиво, а медианите са в червено.

На ред 11 викаме Select върху масива от медианите $[m_1 \mathinner{..} m_q]$ с втори аргумент $\left\lfloor \frac{q-1}{2} \right\rfloor$. Съгласно индуктивното предположение (да си припомним, че правим доказателството със силна индукция), на ред 11, Select връща този елемент на масива $[m_1 \mathinner{..} m_q]$, който по големина е номер $\left\lfloor \frac{q-1}{2} \right\rfloor + 1$. Но $\left\lfloor \frac{q-1}{2} \right\rfloor + 1 = \left\lfloor \frac{q-1+2}{2} \right\rfloor = \left\lfloor \frac{q+1}{2} \right\rfloor$, а по определение елементът, който по големина е номер $\left\lfloor \frac{q+1}{2} \right\rfloor$, е медианата. Следователно, на ред 11, 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 34 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(A, m) на ред 12 слага m на позиция $\left\lfloor \frac{q+1}{2} \right\rfloor$ в A и връща j, равно на $\left\lfloor \frac{q-1}{2} \right\rfloor$. В най-лошия случай k не е равно на j (проверката на ред 13), защото тогава алгоритъмът би терминирал с **return** m на ред 14. В най-лошия случай се стига до рекурсивно викане или на ред 17, или на ред 19. Кое точно от тези рекурсивни викания се случва, зависи от k при фиксирано j. Съществено 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 11 получи стойността на медианата на A, масивите $A[1 \mathinner{..} j]$ (ред 17) и $A[j+2 \mathinner{..} n]$ (ред 19) 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.

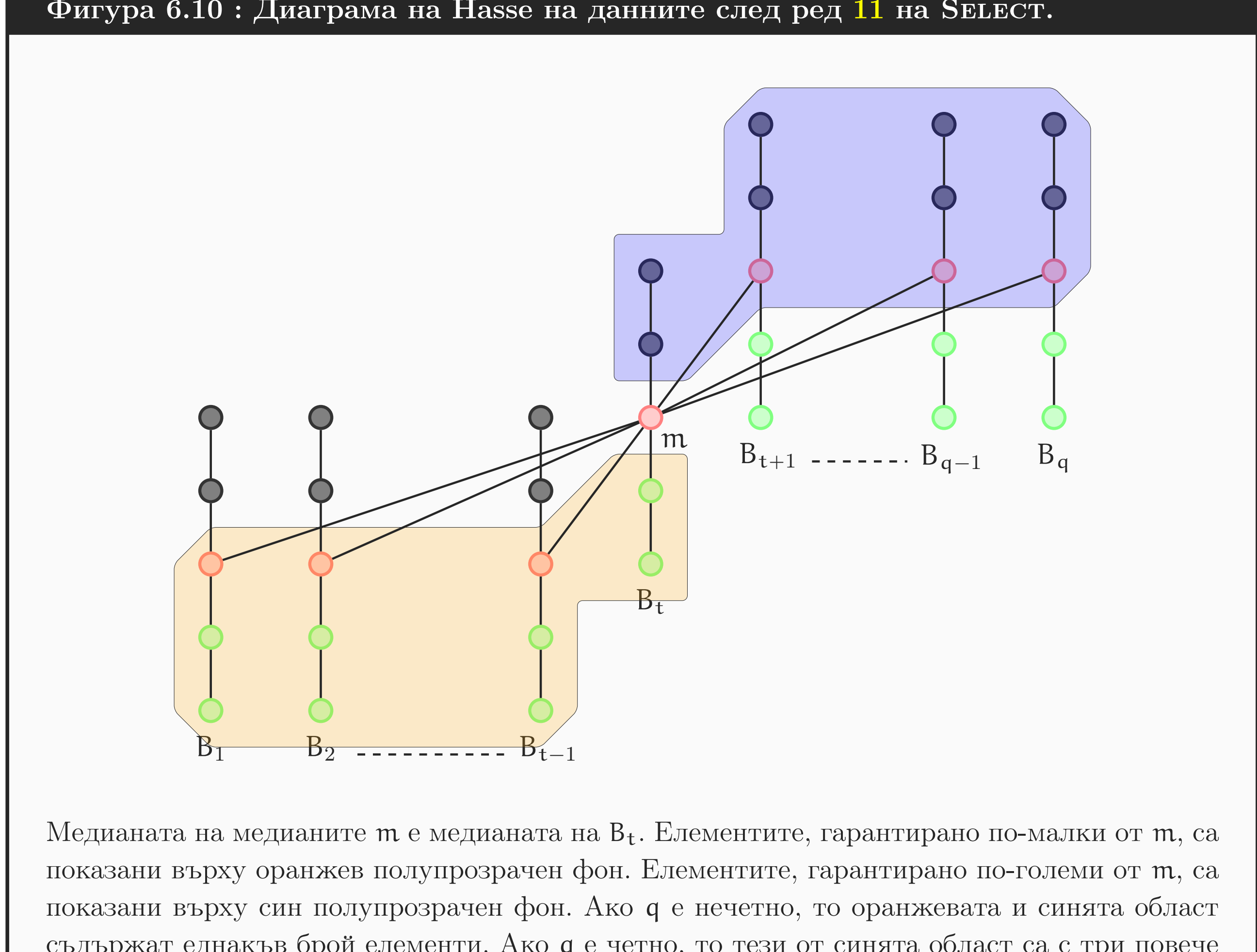
Както видяхме обаче, m може да не е медианата на A. Колкото по-отдалечена по стойност е m от медианата на A, толкова повече $A[1 \mathinner{..} j]$ и $A[j + 2 \mathinner{..} n]$ се различават по големина, а в най-лошия случай рекурсивното викане (ред 17 или ред 19) ще се случи върху по-големия от тях.

БОО, нека

- $B_{\left\lfloor \frac{q+1}{2} \right\rfloor}$ е масивът, от който идва m; нека $t = \left\lfloor \frac{q+1}{2} \right\rfloor$,
- медианите на $B_1, \ldots, B_{t-1}$ са по-малки от m,
- а медианите на $B_{t+1}, \ldots, B_q$ са по-големи от нея.

При това положение, Фигура 6.10 показва диаграмата на Hasse за елементите на A след изпълнението на ред 11.

Фигура 6.10 : Диаграма на Hasse на данните след ред 11 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 повече, понеже вземаме за медиана долния кандидат.

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 три елемента от всеки от масивите $B_1, \ldots, B_{t-1}$, плюс още два елемента от B_t. Може да има и други елементи, по-малки от m—някои от зелените елементи на $B_{t+1}, \ldots, B_q$—но само за тези от оранжевата област е гарантирано, че са по-малки от m. Броят на елементите от оранжевата област е точно

$$3\left\lfloor \frac{q-1}{2} \right\rfloor + 2 = 3\left\lfloor \frac{\frac{n}{5}-1}{2} \right\rfloor + 2 \approx \frac{3n}{10}$$

Апроксимираме сложния израз с $\frac{3n}{10}$, игнорирайки адитивната константа $+2$, нотацията $\lfloor\,\rfloor$ и -1 в числителя. Лесно се вижда, че това не променя асимптотиката на алгоритъма, който конструираме.

Напълно аналогично, апроксимираме броя на елементите, гарантирано по-големи от m, с $\frac{3n}{10}$. Това е долна граница за броя на елементите, по-големи от m.

Ключовото наблюдение е, че максималният размер на масива, върху който се прави рекурсивно викане (ред 17 или ред 19), е приблизително n минус получената долна граница $\frac{3n}{10}$.

Това е минималният брой елементи, които **гарантирано няма** да бъдат част от множеството, върху което се извършва рекурсивно викане. С други думи, на ред 17 или ред 19 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 12 е върху вход с големина приблизително $\frac{n}{5}$, а извън рекурсивните викания, SELECT извършва линейна по време работа, изразът е:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \tag{6.6}$$

Ще решим (6.6) чрез теоремата на Akra-Bazzi (Теорема 29). Имаме $a_1 = a_2 = 1$, $b_1 = \frac{1}{5}$, $b_2 = \frac{7}{10}$ 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 29,

$$\begin{aligned}
T(n) &\asymp n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\, du\right) = n^p \left(1 + \int_1^n \frac{du}{u^p}\right) \\
&= n^p \left(1 + \frac{u^{1-p}}{(1-p)}\Big|_1^n\right) = n^p \left(1 + \frac{n^{1-p}}{(1-p)} - \underbrace{\frac{1^{1-p}}{(1-p)}}_{\text{константа}}\right) \\
&\asymp n^p \left(1 + \frac{n^{1-p}}{(1-p)}\right) \asymp n^p + n^1 \asymp n^1
\end{aligned}$$

И така, SELECT е линеен алгоритъм.

Допълнение 33: 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 $B_1, \ldots, B_{q-1}$ (ред 9). 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($A[1\,..\,n]$: int; $k \in \{0, \ldots, n-1\}$)

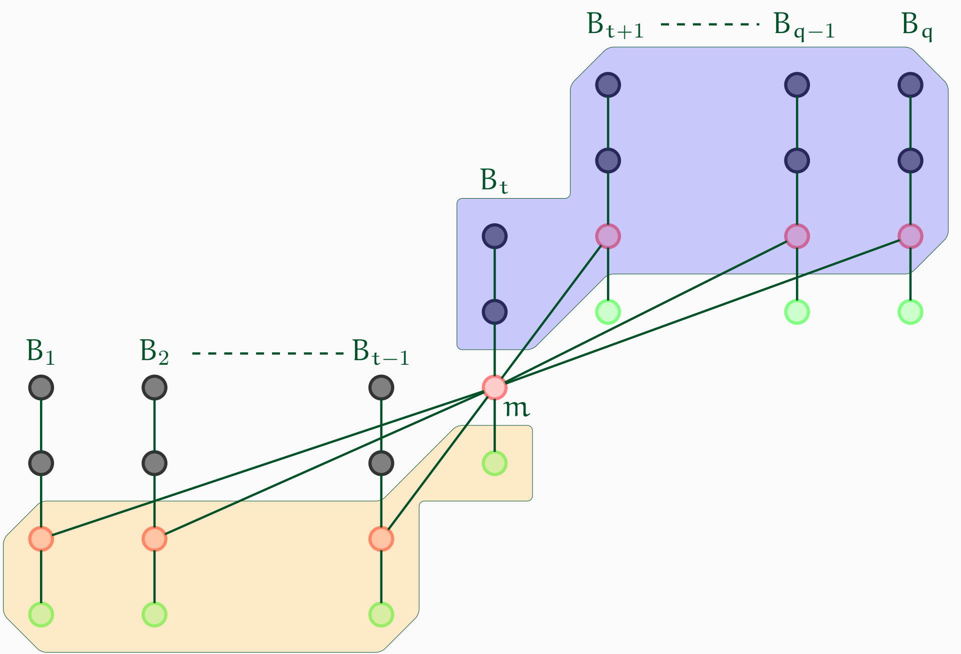
```
1  (* A[x] ≠ A[y] за 1 ≤ x < y ≤ n *)
2  if n = 1
3      return A[1]
4  q ← ⌈n/4⌉, r ← n mod 4
5  if r = 0
6      разбий A на подмасиви B_1, ..., B_q, всеки с 4 елемента
7  else
8      разбий A на подмасиви B_1, ..., B_{q-1}, всеки с 4 елемента, и B_q с r елемента
```

Нататък алгоритъмът е същият. Да анализираме сложността по време. БОО, нека n е кратно 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 трябва да дефинираме медианата като средното аритметично на втория и третия елемент. Медианата трябва да е **елемент, който се среща**. БОО, дефинираме, че $B_i[3]$ е медианата на B_i. Тогава за всяко $i \in \{1, 2, \ldots, q\}$, 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 11, а най-лошият случай се получава, когато $k \neq j$ и правим едното от рекурсивните викания $\text{SELECT}(A[1..j], k)$ или $\text{SELECT}([j+2..n], k-j-1)$. Най-лошия случай и в този алгоритъм се получава, когато масивът, върху който се прави рекурсивното викане, е с максимален резмер.
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 $B_1, \ldots, B_{t-1}$ са по-малки от m,
- а медианите на $B_{t+1}, \ldots, B_q$ 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 $B_1, \ldots, B_{t-1}$, плюс още един елемент от B_t. Може да има и други елементи, по-малки от m—някои от зелените елементи на $B_{t+1}, \ldots, B_q$—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 $\lfloor\,\rfloor$ 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 $\lfloor\,\rfloor$ 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 Akra-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

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + n$$

Ще го решим чрез теоремата на Akra-Bazzi (Теорема 29). Имаме $a_1 = a_2 = 1$, $b_1 = \frac{1}{7}$, $b_2 = \frac{5}{7}$ и $g(n) = n$. Числото p е уникалното число, такова че

$$a_1 \cdot b_1^p + a_2 \cdot b_2^p = 1 \leftrightarrow \left(\frac{1}{7}\right)^p + \left(\frac{5}{7}\right)^p = 1$$

С Maple(TM) получаваме $p \approx 0.7632025267$. Важното е, че $0 < p < 1$.

Съгласно Теорема 29,

$$T(n) \asymp n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\,du\right) = n^p \left(1 + \int_1^n \frac{du}{u^p}\right)$$

$$= n^p \left(1 + \frac{u^{1-p}}{(1-p)}\Big|_1^n\right) = n^p \left(1 + \frac{n^{1-p}}{(1-p)} - \underbrace{\frac{1^{1-p}}{(1-p)}}_{\text{константа}}\right)$$

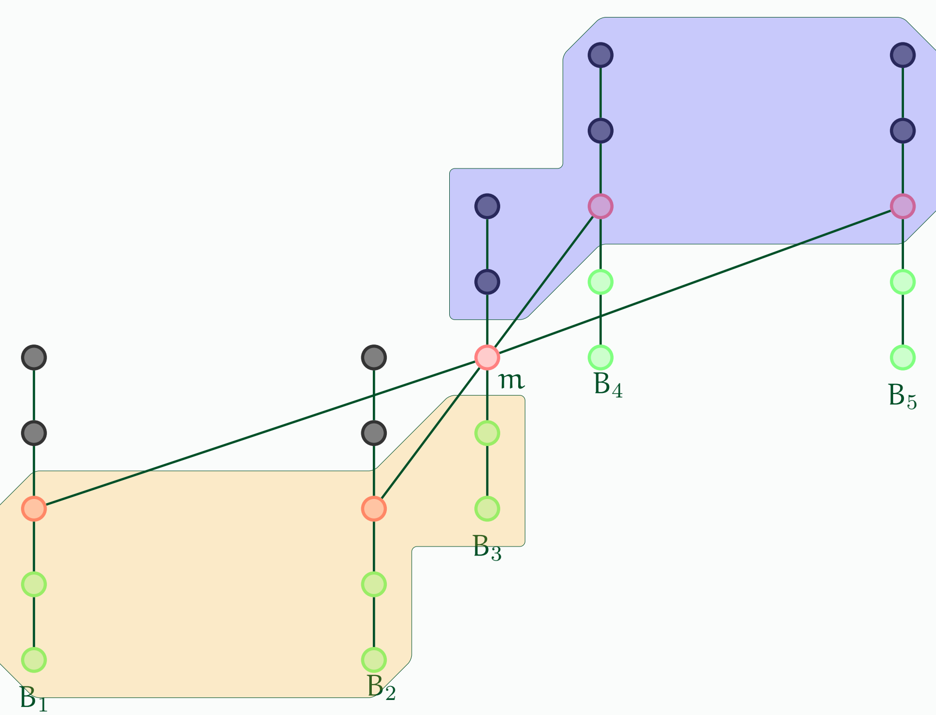
$$\asymp n^p \left(1 + \frac{n^{1-p}}{(1-p)}\right) \asymp n^p + n^1 \asymp n^1$$

И така, Select остава линеен алгоритъм, ако константата, спрямо която е конструиран, е 7.

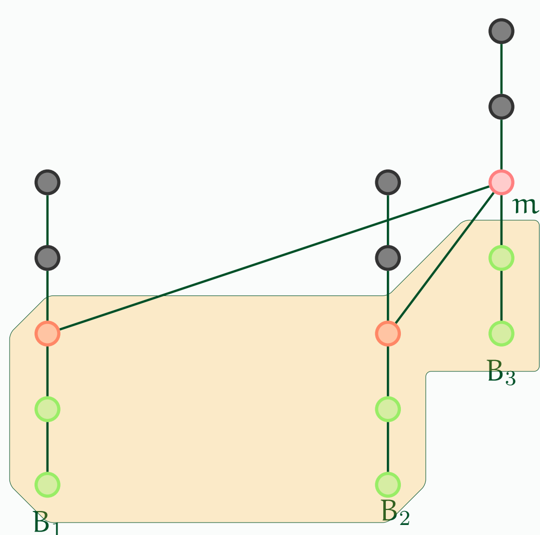
Допълнение 34: Най-лош случай за първото викане в Select

Разглеждаме примера за вход на Select на стр. 323. 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 11, 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:



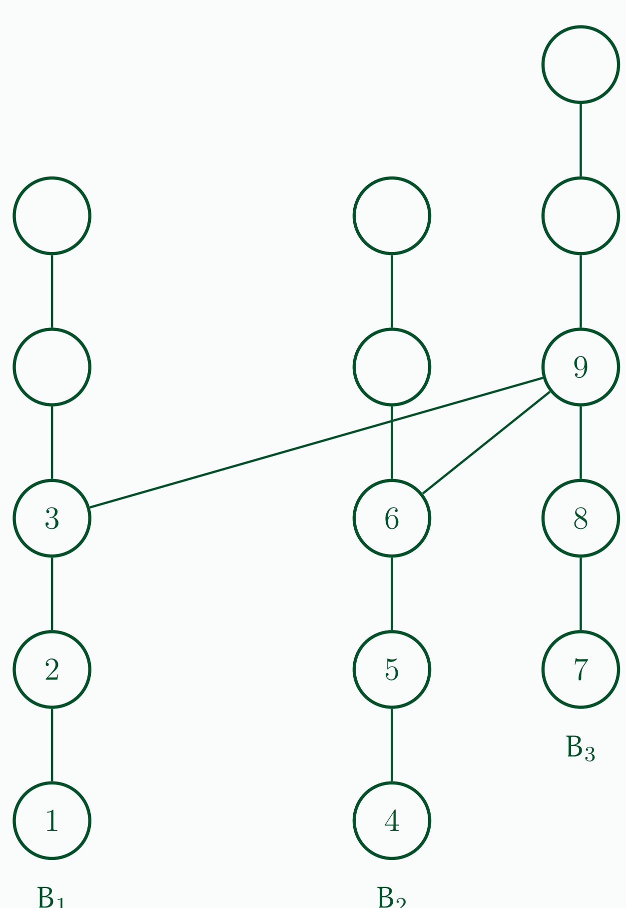
БОО, да се фокусираме върху елементите, по-малки от m:



$\mathfrak{m}$ 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 $\mathfrak{m}$. Така че точна долна граница за броят на елементите, по-малки от $\mathfrak{m}$, 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 $\mathfrak{m} = 9$. Въпросните осем елемента са числата $1, \ldots,$ 8. Ето едно тяхно възможно разположение:



За да се окажат по този начин числата $1, \ldots, 9$ след сортирането на $B_1, \ldots, B_5$ 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 желаната "разпределение" на $1, \ldots, 9$ 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 изобретен от John von Neumann през 1945 г. [86]. Той е типичен пример за алгоритъм, изграден по схемата Разделяй-и-Владей. 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 последователност. Естествено, всичко това става, когато входът е достатъчно голям. Ако входът е с размер единица или нула, алгоритъмът не прави нищо (защото празният масив и едноелементният масив са сортирани) – това е спирачката на рекурсията. Акцентът е върху третата фаза, където се извършва истинската работа по сортирането.

Първо ще дадем пример за сортиране с MERGESORT. Примерът нарочно използва вход с големина, която е точна степен на двойката, но кодът, който даваме 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

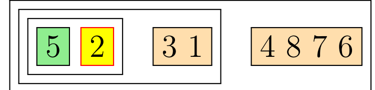
Отново, с жълт фон означаваме само тази част от началния масив, която е вход на текущото рекурсивно викане. Понеже големината на този вход е четири, което е по-голямо от едно, пак делим входа на две и правим две рекурсивни викания, първото върху частта вляво:

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 правим две рекурсивни викания, първото върху частта вляво:

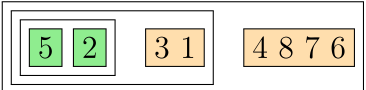
5 | 2 | 3 1 | 4 8 7 6

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 на свой ред вика алгоритъма върху другия подмасив с големина едно:



†В действителност разделянето е на една част с размер $\left\lfloor \frac{n}{2} \right\rfloor$ и друга част с размер $\left\lceil \frac{n}{2} \right\rceil$.

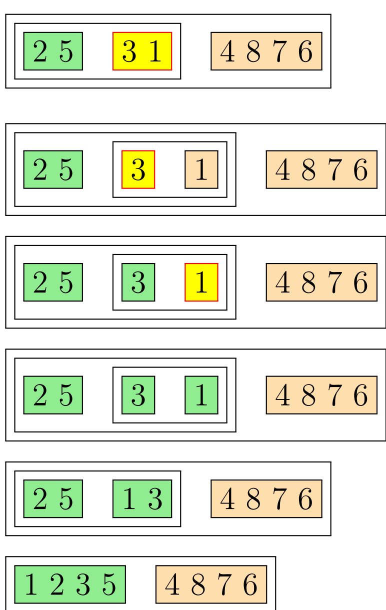
Светлозеленият фон означава, че върху този подмасив вече е достигнато дъното на рекурсията. Отново големината на текущия вход (жълт фон) е едно и алгоритъмът директно връща управлението на горното ниво, бидейки привършил и с двете викания



Сега алгоритъмът слива двата сортирани подмасива с големини единици в един сортиран подмасив с големина две:

2 5 3 1 4 8 7 6

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:

1 2 3 5 4 6 7 8

Сливането на двата сортирани подмасива води до

1 2 3 4 5 6 7 8

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. Символът ∞ означава число, по-голямо от всяко друго число, което може да се срещне. Този символ се нарича *пазач* (на английски, *sentinel*, по терминологията на [31]). Използването му не е задължително, но с него кодът става по-лесен за четене и верификация (Определение 7).

Merge($A[1..n]$: int; l, mid, h: цели числа, такива че $1 \leqslant l < mid < h \leqslant n$)

1 (* подмасивите $A[l, \ldots, mid]$ и $A[mid+1, \ldots, h]$ са сортирани *)

2 $n_1 \leftarrow mid - l + 1$

```
3   n₂ ← h − mid
4   създай L[1,...,n₁ + 1] и R[1,...,n₂ + 1]
5   L[1 .. n₁] ← A[l,...,mid]
6   R[1 .. n₂] ← A[mid + 1,...,h]
7   L[n₁ + 1] ← ∞
8   R[n₂ + 1] ← ∞
9   i ← 1
10  j ← 1
11  for k ← l to h
12      if L[i] ≤ R[j]
13          A[k] ← L[i]
14          i++
15      else
16          A[k] ← R[j]
17          j++
```

Лема 37

Да допуснем, че подмасивите $A[l,\ldots,mid]$ и $A[mid+1,\ldots,h]$ във входа на MERGE са сортирани. След терминирането на MERGE, масивът $A[l,\ldots,h]$ се състои точно от елементите на входните $A[l,\ldots,mid]$ и $A[mid+1,\ldots,h]$, но в сортиран вид.

Доказателство: Приемаме за очевидно, че при първото достигане на ред 11, масивите L и R са точни копия на съответно $A[l,\ldots,mid]$ и $A[mid+1,\ldots,h]$, плюс един sentinel накрая.

Инвариант 15: MERGE

част i Всеки път, когато изпълнението на MERGE е на ред 11, $A[l,\ldots,k-1]$ 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 $A[l,\ldots,k-1]$ всъщност е $A[l,\ldots,l-1]$, 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 38. 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 38

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 елементи 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 $n_1 + n_2 = mid - l + 1 + h - mid = h - l + 1 > h - l$. ↯

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 $L[i] \leqslant R[j]$. Очевидно, $L[i] < \infty$. Съгласно **част ii** от индуктивното предположение и допускането, че $L[i] \leqslant R[j]$, 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 $A[l, \ldots, k\text{-}1]$. Изпълнението отива на ред 13. Забелязваме, че $A[k]$ не е по-малък от нито един елемент на $A[l, \ldots, k\text{-}1]$ и заключаваме, че $A[l, \ldots, k]$ 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 $A[l, \ldots, k\text{-}1]$ 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 $L[i] \nleqslant R[j]$, 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 $A[l, \ldots, h]$ съдържа $h - l + 1$ най-малки елемента на L и R, в сортиран вид". Имайки предвид факта, че L и R съдържат точно елементите на входния $A[l, \ldots, h]$, заключаваме, че текущият $A[l, \ldots, h]$ съдържа точно елементите на входния $A[l, \ldots, h]$, но в сортиран вид. □

```
MERGESORT(A[1..n]: int; l, h: индекси в A)
1  if l < h
2      mid ← ⌊(l+h)/2⌋
3      MERGESORT(A, l, mid)
4      MERGESORT(A, mid + 1, h)
5      MERGE(A, l, mid, h)
```

Лема 39

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 $A[l, \ldots, h]$ 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 $A[l, \ldots, mid]$ и $A[mid + 1, \ldots, h]$ (редове 3 и 4) при всички рекурсивни викания, такива че $h > l$. От Ле-

[†]Забележете, че това не е доказателство с инвариант на цикъла, понеже MERGESORT не е итеративен, а е рекурсивен, алгоритъм.

ма 6.3 следва, че в края на работата на текущото рекурсивно извикване, целият $A[l, \ldots, h]$ 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 извикване е MERGESORT($A[1, \ldots, n]$). С MERGESORT, тази стъпка е тривиална: просто забелязваме, че алгоритъмът приключи работата си, $A[1, \ldots, n]$ е сортиран. □

Сложността по време на MERGESORT се определя с рекурентното уравнение:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{6.7}$$

Решението, съгласно втория случай на Мастър теоремата, е $T(n) \asymp n \lg n$. Сложността по памет е $\Theta(n)$, ако алгоритъмът се имплементира грамотно (имплементация, която буквално следва псевдокода горе би имала линейна сложност по време, но на практика би била твърде бавна заради честото алокиране и деалокиране на памет). 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 окаже вляво от този от R в окончателната подредба.

Допълнение 35: Бързо намиране на броя на инверсиите в масив.

Определение 55: 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 $1 \leqslant i < j \leqslant n$ 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 55 и Определение 53. Посоката на неравенството в Определение 53 е обратната: там се иска $A[i] < A[j]$, но това е, защото Определение 53 е в контекста на максимални пирамиди (вж. Определение 51). Ако ставаше дума за минимални пирамиди, неравенството в Определение 53 щеше да е $A[i] > A[j]$, също както в Определение 55. Изключвайки несъществената подробност за посоката на неравенството, забелязваме, че в някакъв смисъл Определение 53 е частен случай на Определение 55, тъй като в Определение 53 се иска $A[i]$ да е предшественик на $A[j]$, което е частен случай на изискването $i < j$ в Определение 55. И така, Наблюдение 39 е аналог на Наблюдение 33.

Наблюдение 39

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 големини. 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 ← ⌊(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 $i \in \{l, \ldots, mid\}$ и $j \in \{mid + 1, \ldots, h\}$. Merge-modified е малка модификация на Merge на стр. 336.

```
Merge-modified(A[1 .. n]: int; l, mid, h: 1 ⩽ l < mid < h ⩽ n)
 1  (* подмасивите A[l, ..., mid] и A[mid + 1, ..., h] са сортирани *)
 2  n₁ ← mid − l + 1
 3  n₂ ← h − mid
 4  създай L[1, ..., n₁ + 1] и R[1, ..., n₂ + 1]
 5  L[1 .. n₁] ← A[l, ..., mid]
 6  R[1 .. n₂] ← A[mid + 1, ..., h]
 7  L[n₁ + 1] ← ∞
 8  R[n₂ + 1] ← ∞
 9  i ← 1
10  j ← 1
```

```
11  c ← 0
12  for k ← l to h
13      if L[i] ⩽ R[j]
14          A[k] ← L[i]
15          i++
16      else
17          A[k] ← R[j]
18          c ← c + n₁ − 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 $L[i] \leqslant R[j]$, то елементите $L[i]$ и $R[j]$ не задават инверсия (очевидно). Очевидно инверсиите между елементи от $L[1..i-1]$ и елементи от $R[1..j]$ са толкова, колкото са инверсиите между елементи от $L[1..i]$ и елементи от $R[1..j]]$. Това, че инкрементираме i на ред 15, без да променяме c, е правилно.

- Ако $L[i] > R[j]$, то елементните $L[i]$ и $R[j]$ задават инверсия (очевидно). Освен това, всеки елемент от $L[i+1..n_1]$ задава инверсия с $R[j]$, понеже L е сортиран, а релацията $\leqslant$ е транзитивна. Тогава всеки елемент от $L[i..n_1]$ задава инверсия с $R[j]$, а това са $n_1 - i + 1$ елемента, което означава и $n_1 - i + 1$ инверсии. Ерго, инверсиите между елементи от $L[1..i]$ и $R[1..j]$ са с $n_1 - i + 1$ повече от инверсиите между елементи от $L[1..i-1]$ и $R[1..j]$. Това, че инкрементираме 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 [68] в началото на 1960-те (вж. [66], [67], [68]). 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 различен 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 **Ком-**

бинирай. 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 36: 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 да 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–HOARE

Фазата **Разделяй** на QUICKSORT се реализира чрез функция, която е широко известна под името PARTITION. Ние вече видяхме една имплементация на тази функция на стр. 322, но за пълнота на изложението тук допускаме, 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 Hoare. Ето как изглежда изглежда функцията PARTITION в

[32, 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 "$A[j] \leqslant \texttt{pivot}$" от [32, стр. 154] е транслирано в "$A[j] > \texttt{pivot}$" тук.

```
Partition–Hoare(A[1, 2, ..., n]: цели числа; l, h: индекси в A, такива че l < h)
1   pivot ← A[l]
2   i ← l − 1
3   j ← h + 1
4   while True do
5       do
6           j ← j − 1
7       while A[j] > pivot
8       do
9           i ← i + 1
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 избрания pivot в смисъл, че $X \geqslant$ pivot и $Y \leqslant$ 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 предложения псевдокод, ходовете на j и 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 $\Theta(n \lg n)$.

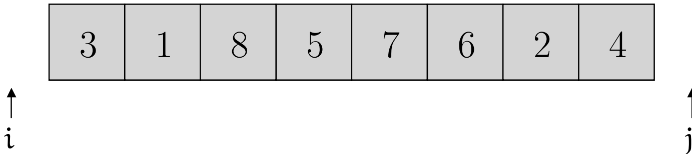
По-прецизно казано, по време на работата на алгоритъма, масивът е разбит на три зони (някои, но не всички, от които може да са празни):

- **Зелена зона** $A[l, \ldots, i]$. Там са елементи, по-малки или равни на `pivot`.
- **Сива зона** $A[i+1, \ldots, j-1]$. Това е неизследваната част от масива.
- **Жълта зона** $A[j, \ldots, h]$. Там са елементи, по-големи или равни на `pivot`.

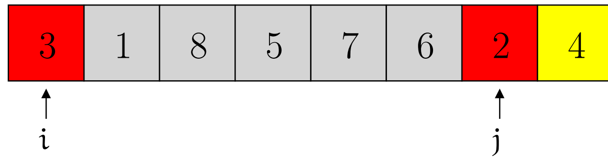
Изборът на имена за зоните е произволен. Освен това, оцветяваме в червено двете клетки на масива, върху които "се спират" индексите j и i, вървейки съответно отгоре надолу и отдолу нагоре. Тези червени клетки са в някакъв смисъл "нарушители": клетката, върху която спира j, съдържа елемент, който е по-малък от pivot, а клетката, върху която спира i, съдържа елемент, който е по-голям от pivot. След извършването на swap-а на ред 12, червената клетка, която е била съседна на жълтата зона, преминава в зелената зона и вече я оцветяваме в зелено, и обратното, червената клетка, която е била съседна на зелената зона, преминава в жълтата зона и вече я оцветяваме в жълто.

Когато и двата **do ... while** цикъла приключат, и $A[i] \geqslant$ `pivot`, и $A[j] \leqslant$ `pivot`. В случай, че $i < j$, изпълнението отива на ред 12; след swap-а, както казахме, зелената и жълтата зона нарастват с по една клетка, и отново индексите j и i тръгват, съответно надолу и нагоре. Ако $i \nless j$, няма какво повече да се прави.

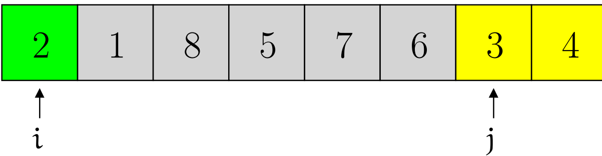
Ще демонстрираме работата на Partition–Hoare с пример. Нека



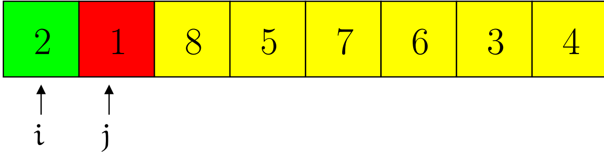
като масивът е индексиран от 1, с други думи, l е 1 и h е 8. В началото `pivot` е $A[l] = 3$. После i и j "застават" извън границите на масива, така че зелената и жълтата зона са празни:



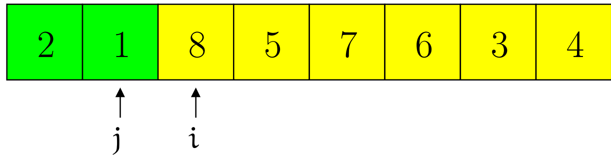
После j "слиза" надолу, първо става 8, но $A[8] \nleqslant$ `pivot`, така че j става 7 и изпълнението на първия **do ... while** цикъл (редове 5-6) спира, понеже $A[7] \leqslant$ `pivot`. Започва изпълнението на втория **do ... while** цикъл (редове 8-9). Индексът i се "качва" нагоре и става 1, но $A[1] = 3$, така че $A[1] \geqslant$ `pivot`, така че вторият **do ... while** цикъл спира. Ситуацията е следната:



Изпълнението достига ред 11. Понеже $i < j$, изпълняваме размяната, която в случая е размяна на $A[1]$ с $A[7]$ получаваме:



После j "слиза" надолу и става 2, спирайки върху тази стойност:



После i се "качва" нагоре и става 3, спирайки върху тази стойност:

2	1	8	5	7	6	3	4

j → позиция 2, i → позиция 3

Вече не оцветяваме в червено клетките $A[j]$ и $A[i]$, защото това са елементи, чиято принадлежност към съответно жълтата и зелената зона вече сме установили.

Изпълнението отново е на ред 11. Сега вече $i \not< j$, така че изпълнението преминава към ред 14, като алгоритъмът връща стойност 2. И наистина, $A[1, \ldots, 2]$ се състои от елементи, които са не по-големи от pivot.

Формално доказателство за коректността на Partition–Hoare няма да даваме, но следният инвариант може да се докаже по познатия ни начин по индукция и оттам следва коректността. Няма нужда да се посочва, че елементите на A във всеки момент са точно същите като елементите на входния масив, защото единственото място в алгоритъма, на което се променя масива, е ред 12, а там променянето става чрез размяна на елементи.

Инвариант 16: Partition–Hoare

При всяко достигане на ред 4 на Partition–Hoare:

$$\forall x \in A[l \mathinner{..} i] : x \leqslant \texttt{pivot}$$
$$\forall x \in A[j \mathinner{..} h] : x \geqslant \texttt{pivot}$$

Освен това, ако $i \leqslant j$:

$$\exists x \in A[l \mathinner{..} j-1] : x \leqslant \texttt{pivot}$$
$$\exists x \in A[i+1 \mathinner{..} h] : x \geqslant \texttt{pivot}$$

Известни са няколко вариации на Partition–Hoare, една от които е следната. Тя е описана в [144, стр. 16]. Тази функция ползва sentinel, който е $-\infty$ и се намира в $A[0]$. Числата, които ще се сортират, са в $A[1 \mathinner{..} n]$. В оригинала вътрешните цикли са **repeat . . . until**. За удобство при четене, тук те са сменени с **do . . . while** цикли.

Partition–Hoare–another($A[0, 1, 2, \ldots, n]$: масив; l, h: индекси в A, такива че $l < h$)

```
1   (* Данните за сортиране са A[1 .. n]. A[0] е −∞. *)
2   pivot ← A[h]
3   i ← l − 1
4   j ← h
5   while TRUE do
6       do
7           j ← j − 1
8       while A[j] > pivot
9       do
10          i ← i + 1
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 и може да правим

следващите викания върху подмасиви, в които той не присъства.

6.4.2 Имплементация на фазата Разделяй чрез Partition–Lomuto

Сега само ще покажем алтернативен псевдокод на функция, която пренарежда масива по желания начин. Идеята за този код е на Nico Lomuto, около 1984 г., и е публикувана в статията *Programming Pearls: Little Languages* в списанието Communications of the ACM [14].

Partition–Lomuto($A[1, 2, \ldots, n]$: цели числа; l, h: индекси в A, такива че $l < h$)

```
1  pivot ← A[h]
2  pp ← l
3  for i ← l to h − 1
4      if A[i] < pivot
5          swap(A[i], A[pp])
6          pp ← pp + 1
7  swap(A[pp], A[h])
8  return pp
```

Името на променливата pp идва от "pivot position".

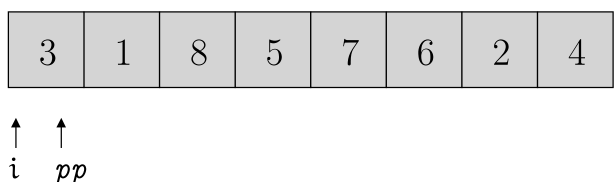
И при Partition–Lomuto можем да дефинираме зелен район (малките елементи), жълт район (големите елементи) и сив район, само че взаимното им разположение е различно, сега те са зелен, жълт и сив, отляво надясно. Partition–Lomuto "търкаля" жълтия район надясно, разменяйки най-левия му елемент с текущия $A[i]$, но само в случай, че текущият $A[i]$ е по-голям или равен на pp.

Ето пример за работата на Partition–Lomuto. Нека, както и в предишния пример, A е следният масив:

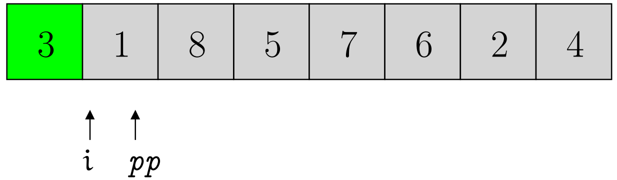
$A =$

3	1	8	5	7	6	2	4

В началото pivot е 4. Индексите i и pp "застават" под първия елемент.

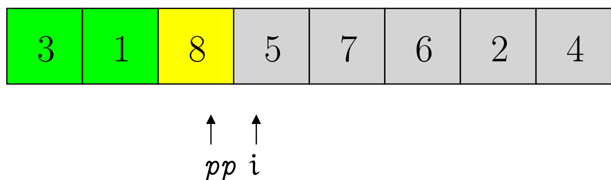


Условието на ред 4 е истина, елементът 3 се разменя със себе си, и после i и pp се оказват под втория елемент.

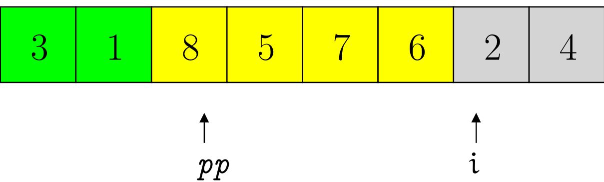


Елементът 1 се разменя със себе си и после i и pp се оказват под третия елемент.

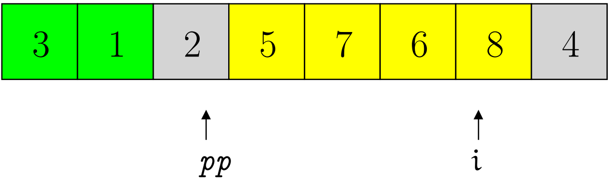
Сега вече A[i] $\nless$ pivot и тялото на цикъла не се изпълнява. Индексът i вече изпреварва *pp* и между тях се оформя районът от елементи, по-големи или равни на pivot.



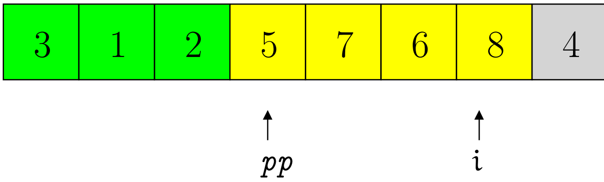
После се срещат няколко елемента, по-големи от pivot, които алгоритъмът "прескача", и се стига до тази ситуация:



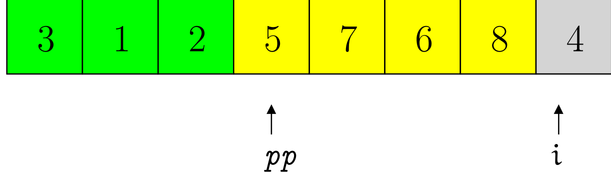
Сега условието на ред 4 е истина. Следва размяна на елементите 8 и 2:



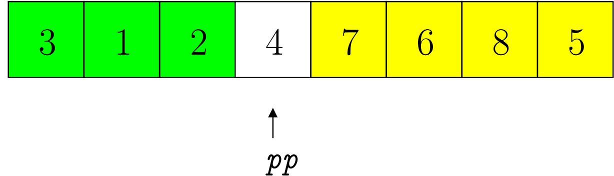
После *pp* пак се инкрементира:



Индексът i се инкрементира за последен път, жълтият район нараства, след което цикълът не се изпълнява повече:



После се прави размяната на ред 7:



На ред 8 алгоритъмът връща индексът на четвъртия елемент, който е `pivot`-ът (четворката). Този елемент си е на мястото. Вляво от него са само по-малки елементи. Вдясно, само по-големи или равни.

Ще докажем формално коректността на PARTITION–LOMUTO.

Лема 40

Ефектът от работата на PARTITION–LOMUTO е следният. В края на алгоритъма, спрямо някаква стойност pp, входният масив е пренареден така, че:

$$\forall x \in A[l \mathinner{..} pp - 1] : x < A[pp]$$
$$\forall x \in A[pp + 1 \mathinner{..} h] : x \geqslant A[pp]$$

Освен това, алгоритъмът връща именно pp.

Доказателство:

Следното твърдение е инвариант на **for**-цикъла (редове 3–6).

Инвариант 17: PARTITION–LOMUTO

При всяко достигане на ред 3 на PARTITION–LOMUTO:

$$\forall x \in A[l \mathinner{..} pp - 1] : x < pivot$$
$$\forall x \in A[pp \mathinner{..} i - 1] : x \geqslant pivot$$

База. Когато изпълнението е на ред 3 за първи път, $i = l$ и $pp = l$. Инвариантът е:

$$\forall x \in A[l \mathinner{..} l - 1] : x < pivot$$
$$\forall x \in A[l \mathinner{..} l - 1] : x \geqslant pivot$$

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:

$$\forall x \in A[l \mathinner{..} pp - 1] : x < pivot \wedge A[i] < pivot \tag{6.8}$$

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, (6.8) става:

$$\forall x \in A[l \mathinner{..} pp] : x < pivot \tag{6.9}$$

След инкрементирането на pp на 6, (6.9) става:

$$\forall x \in A[l \mathinner{..} pp\text{-}1] : x < pivot \tag{6.10}$$

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

$$\forall x \in A[pp \mathinner{..} i-1] : x \geqslant pivot \tag{6.11}$$

След размяната на $A[i]$ и $A[pp]$, която става на ред 5, в сила е:

$$\forall x \in A[pp+1 \mathinner{..} i] : x \geqslant pivot \tag{6.12}$$

След инкрементиранията на pp на 6 и неявното инкрементиране на i при следващото достигане на ред 3, спрямо новите стойности на тези две променливи, (6.12) става:

$$\forall x \in A[pp \mathinner{..} i-1] : x \geqslant pivot \tag{6.13}$$

Случай II $A[i] \geqslant pivot$. От индуктивното предположение знаем, че:

$$\forall x \in A[l \mathinner{..} pp-1] : x < pivot$$
$$\forall x \in A[pp \mathinner{..} i-1] : x \geqslant pivot$$

Тогава:

$$\forall x \in A[l \mathinner{..} pp-1] : x < pivot$$
$$\forall x \in A[pp \mathinner{..} i] : x \geqslant pivot$$

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:

$$\forall x \in A[l \mathinner{..} pp-1] : x < pivot$$
$$\forall x \in A[pp \mathinner{..} i-1] : x \geqslant pivot$$

Терминация. При последното достигане на ред 3 е вярно, че $i = h$. Тогава

$$\forall x \in A[l \mathinner{..} pp-1] : x < pivot$$
$$\forall x \in A[pp \mathinner{..} h-1] : x \geqslant pivot$$

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 $pivot$. В сила вече е

$$\forall x \in A[l \mathinner{..} pp-1] : x < A[pp]$$
$$\forall x \in A[pp+1 \mathinner{..} h] : x \geqslant A[pp]$$

После алгоритъмът връща pp. □

6.4.3 Самият Quicksort

Псевдокодът на самия Quicksort е съвсем кратък. Процедурата Partition е или Partition–Lomuto, или Partition–Hoare–another.

Quicksort($A[1, 2, \ldots, n]$: цели числа; l, h: индекси в A)
1 **if** $l < h$
2     $mid \leftarrow$ 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. 342, 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($A[1, 2, \ldots, n]$: цели числа; l, h: индекси в A)
1 (* На стр. 154 в [32] *)
2 **if** $l < h$
3     $mid \leftarrow$ 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 са 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:

$$T(n) = \frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + T(n-k)\right) + (n-1) \tag{6.14}$$

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 рекурентно уравнение се чете така: работата на QUICKSORT е сумата от

- сумата по всички възможности за pivot ($k \in \{1, \ldots, n\}$) от сумата от работата върху получени спрямо него двата 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 (6.14).

$$\begin{aligned}
T(n) &= \frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + T(n-k)\right) + (n-1) \\
&= \frac{1}{n}\left(\sum_{k=1}^{n} T(k-1) + \sum_{k=1}^{n} T(n-k)\right) + (n-1) \\
&= \frac{1}{n}\left(\underbrace{T(0) + T(1) + \ldots + T(n-1)}_{\sum_{k=1}^{n} T(k-1)} + \underbrace{T(n-1) + T(n-2) + \ldots + T(0)}_{\sum_{k=1}^{n} T(n-k)}\right) + (n-1)
\end{aligned} \tag{6.15}$$

Очевидно двете подчертани суми в (6.15) са равни, така че

$$T(n) = \frac{2}{n}\big(T(0) + T(1) + \ldots + T(n-1)\big) + (n-1) \tag{6.16}$$

Умножаваме двете страни на (6.16) по n и получаваме

$$nT(n) = 2\big(T(0) + T(1) + \ldots + T(n-1)\big) + n(n-1) \tag{6.17}$$

Но тогава, ако n е достатъчно голямо,

$$(n-1)T(n-1) = 2\big(T(0) + T(1) + \ldots + T(n-2)\big) + (n-1)(n-2) \tag{6.18}$$

Изваждаме (6.18) от (6.17) и получаваме

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\ &= 2T(n-1) + 2(n-1) \quad \leftrightarrow \\ nT(n) &= (n+1)T(n-1) + 2(n-1) \end{aligned} \tag{6.19}$$

Делим двете страни на (6.19) на $n(n+1)$ и получаваме

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{6.20}$$

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 (6.20), получаваме следното рекурентно уравнение:

$$S(n) = S(n-1) + \frac{2(n-1)}{n(n+1)} \tag{6.21}$$

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.

$$\begin{aligned} S(n) &= S(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= S(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(n-3) + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\ldots \\ &= S(1) + \frac{2\cdot 1}{2\cdot 3} + \frac{2\cdot 2}{3\cdot 4} + \frac{2\cdot 3}{4\cdot 5} + \ldots + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= S(1) + 2\sum_{k=2}^{n} \frac{k-1}{k(k+1)} \end{aligned} \tag{6.22}$$

Да разгледаме (6.22). $S(1)$ 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 24, $\sum_{k=1}^{n} \frac{1}{k} \asymp \lg n$, 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 сложност 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: Демонстираме 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 видяхме в предната лекция, сортирането на прозволни елементи, чиито индивидуални стойности не можем четем, а само можем да сравняваме елементите 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 възможните стойности на елементите. Съвсем прост пример е сортирането на булев масив – очевидно е достатъчно да преброим колко са елементите от всеки вид и после да презапишем входа със съответния брой нули и единици, като нулите са преди единиците, което може да стане в $\Theta(n)$ време.

Алгоритъмът, който ще разгледаме първо, се основава на идеята за преброяване на елементите от всяка големина. Както ще стане ясно обаче, той 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 просто презаписване (на числа). Дори когато казваме "входът е масив от числа", имаме предвид, че входът е масив от записи с ключове-числа.

7.2 Counting Sort

Входът е масив от числа $A[1\,..\,n]$. Известно е, че за някакво $k \in \mathbb{N}^+$, за всяко $A[i]$ е изпълнено

$$A[i] \in \{1, 2, \ldots, k\}$$

Алгоритъмът ползва работен масив $C[0\,..\,k]$ и още един масив $B[1\,..\,n]$, в който се записва резултата от сортирането.

```
Counting Sort(A[1..n]: positive integers; k: a positive integer)
 1  (* k е такова, че 1 ⩽ A[i] ⩽ k за всички i *)
 2  създай B[1..n] и C[0..k]
 3  for i ← 0 to k
 4      C[i] ← 0
 5  for i ← 1 to n
 6      C[A[i]] ← C[A[i]] + 1
 7  for i ← 1 to k
 8      C[i] ← C[i] + C[i − 1]
 9  for i ← n downto 1
10      B[C[A[i]]] ← A[i]
11      C[A[i]] ← C[A[i]] − 1
12  return B
```

Първо ще покажем работата на алгоритъма с пример и после ще дадем формално доказателство за коректност. Нека $n = 8$, $k = 6$ и масивът A е:

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

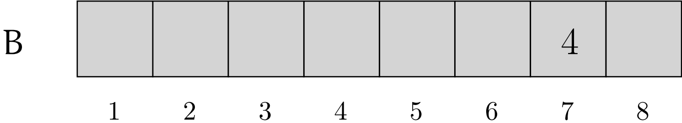
Очевидно първият **for**-цикъл (редове 3–4) просто нулира масива C. Вторият **for**-цикъл (редове 5–6) преброява по колко елемента от всяко $i \in \{1, \ldots, k\}$ има в A и записва това в C:

	0	1	2	3	4	5	6
C	0	2	0	2	3	0	1

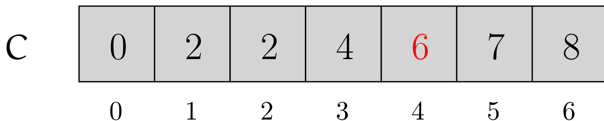
Очевидно, $\sum_{i=0}^{k} C[i] = n$. Третият **for**-цикъл (редове 7–8) присвоява на всяко $C[i]$ броя на всички елементи на A, по-малки или равни на i:

	0	1	2	3	4	5	6
C	0	2	2	4	7	7	8

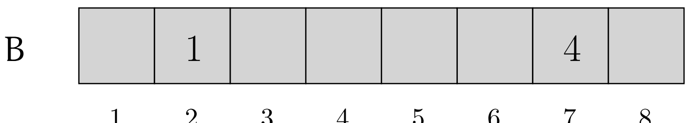
Очевидно, сега $C[k] = n$. Същината на алгоритъма е в четвъртия **for**-цикъл (редове 9–11). В началото $i = 8$, $A[8] = 4$, $C[4] = 7$. $B[7]$ става 4:



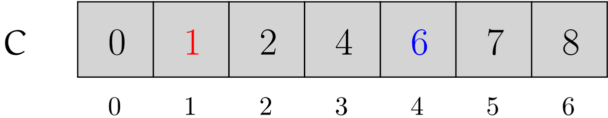
а C става:



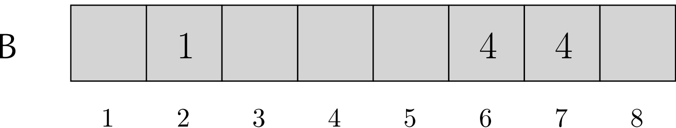
В червено е току-що намаленият елемент (заради ред 11) на C. После $i = 7$, $A[7] = 1$, $C[1] = 2$. $B[2]$ става 1:



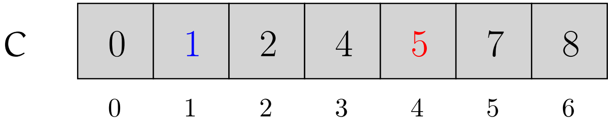
а C става:



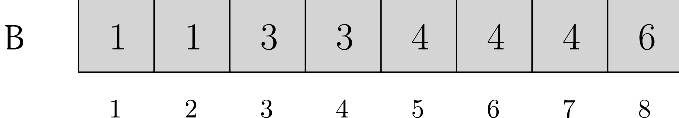
После $i = 6$, $A[6] = 4$, $C[4] = 6$. $B[6]$ става 4:



а C става:



И така нататък. В края на алгоритъма B е:



а C е:

	0	1	2	3	4	5	6
C	0	0	2	2	4	7	7

В Определение 38 въведохме нотацията "$\#(a, A)$" за кратността на a в мултимножеството A. Тук ще използваме нотацията "$\#(x, Z[1, \ldots, j])$", за да означаваме кратността на x в мултимножеството от елементите на $Z[1..j]$, където Z е масив, j е индекс, сочещ в него, а x е елемент, чийто тип позволява да е елемент на Z.

Сега ща докажем коректността на COUNTING SORT.

Лема 41

След приключването на втория **for**-цикъл (редове 5–6), за всяко j, такова че $1 \leqslant j \leqslant k$, $C[j]$ съдържа броя на елементите в масива A, които са равни на j.

Доказателство:

Инвариант 18: Вторият for-цикъл на COUNTING SORT

Всеки път, когато изпълнението е на ред 5, за всеки елемент $C[j]$, където $1 \leqslant j \leqslant k$, е изпълнено $C[j] = \#(j, A[1, \ldots, i-1])$.

База. При първото достигане на ред 5, всички елементи на C са нули. От друга страна, i е 1, и така $A[1, \ldots, i-1]$ е празен. Твърдението е вярно.

Поддръжка. Да допуснем, че твърдението е в сила при дадено достигане на ред 5, което не е последното. Нека стойността на $C[A[i]]$ е y в момента, в който изпълнението е на ред 6. По индуктивното предположение, $y = \#(A[i], A[1, \ldots, i-1])$. Очевидно е, че $\#(A[i], A[1, \ldots, i-1]) + 1 = \#(A[i], A[1, \ldots, i])$. След изпълнението на ред 6, $C[A[i]]$ се увеличава с единица, така че $C[A[i]]$ става равно на $\#(A[i], A[1, \ldots, i])$. Тъй като всички останали елементи на C (освен $C[A[i]]$) остават непроменени от текущото изпълнение на **for**-цикъла, е вярно, че:

- за всеки елемент $C[j]$ освен $C[A[i]]$, $C[j] = \#(j, A[1, \ldots, i-1])$, а също така $C[j] = \#(j, A[1, \ldots, i])$
- $C[A[i]] = \#(A[i], A[1, \ldots, i])$.

Като цяло, за всеки елемент $C[j]$ е в сила $C[j] = \#(j, A[1, \ldots, i])$. Това е преди инкрементирането на i. След инкрементиравнето на i, $C[j] = \#(j, A[1, \ldots, i-1])$ за всяко j, такова че $1 \leqslant j \leqslant k$.

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 "за всеки елемент $C[j]$, където $1 \leqslant j \leqslant k$, е в сила $C[j] = \#(j, A[1, \ldots, n])$." □

Лема 42

След приключването на третия **for**-цикъл (редове 7–8), за всяко j, такова че $1 \leqslant j \leqslant k$, $C[j]$ съдържа броя на елементите в масива A, които са по-малки или равни на j.

Доказателство:

Инвариант 19: Третият for-цикъл на Counting Sort

Всеки път, когато изпълнението е на ред 7, за всяко j, такова че $0 \leqslant j \leqslant i-1$, $C[j] = \sum_{t=1}^{j} \#(t, A[1, \ldots, n])$.

База. При първото достигане на ред 7, i е 1, така че твърдението е $C[0] = \sum_{t=1}^{0} \#(t, A[1, \ldots, n]) = 0$. Но $C[0]$ е наистина 0, защото то се инициализира с 0 от първия **for**-цикъл и вторият **for**-цикъл не му присвоява нищо (понеже $A[i]$ не може да е 0). ✓

Поддръжка. Да допуснем, че твърдението е в сила при някое достигане на ред 7, което не е последното. Елементът $C[i]$ не е променян (засега) от изпълнението на третия **for**-цикъл, така че съгласно Лема 41, $C[i] = \#(i, A[1, \ldots, n])$. По индуктивното предположение, $C[i-1] = \sum_{t=1}^{i-1} \#(t, A[1, \ldots, n])$. След изпълнението на ред 8 имаме $C[i] = \sum_{t=1}^{i} \#(t, A[1, \ldots, n])$. По отношение на новата стойност на i, за всяко j, такова че $0 \leqslant j \leqslant i-1$, $C[j] = \sum_{t=1}^{j} \#(t, A[1, \ldots, n])$.

Терминация. Да разгледаме момента, в който изпълнението е на ред 7 за последен път. Очевидно $i = n+1$. Заместваме i с $n+1$ в инварианта и получаваме "за всяко j, такова че $0 \leqslant j \leqslant n$, $C[j] = \sum_{t=1}^{j} \#(t, A[1, \ldots, n])$." □

Определение 56

За всяко $j \in \{1, \ldots, k\}$, j е *съществено*, ако съществува елемент на A със стойност j.

Съгласно Лема 41, ненулевите елементи на C след втория **for**-цикъл са точно елементите, чиито индекси в C са съществени.

Определение 57

За всеки x от A, *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 позицията му в масива след изпълнението на стабилен сортиращ алгоритъм.

Теорема 48: Counting Sort е стабилен сортиращ алгоритъм.

Counting Sort е стабилен сортиращ алгоритъм.

Доказателство:

Инвариант 20: Четвъртият for-цикъл (редове 9–11)

Всеки път, когато изпълнението е на ред 9, за всяко j, такова че $1 \leqslant j \leqslant k$ и j е съществено, $C[j]$ съдържа правилното място на най-десния елемент от подмасива $A[1,\ldots,i]$, който има стойност j. Нещо повече, всички елементи от $A[i+1,\ldots,n]$ са на своите правилни места в B.

База. При първото изпълнение на ред 7, i е n. Първата част от инварианта гласи "за всяко j, такова че $1 \leqslant j \leqslant k$ и j е съществено, $C[j]$ е правилното място на най-десния елемент от подмасива $A[1,\ldots,n]$, който има стойност j". Забележете, че C все още не е променян от четвъртия цикъл, така че Лема 42 е в сила. За всяка стойност j, която се появява в A, правилното място на най-дясното j в A е равно на сумата от броевете на елементите със стойност $\leqslant j$. Според Лема 42, $C[j]$ е равно точно на тази сума.

Да разгледаме втората част от инварианта. Подмасивът $A[i+1,\ldots,n] = A[n+1,\ldots,n]$ е празен, така че твърдението е в сила. ✓

Поддръжка. Да допуснем, че твърдението е в сила при някое достигане на ред 9, което не е последното. Стойността на $A[i]$ е някое съществено цяло число j между 1 и k. Нещо повече, то е <u>**най-десният**</u> елемент в $A[1,\ldots,i]$ със стойност j. Съгласно индуктивното предположение, неговото правилно място е $C[A[i]]$ и алгоритъмът го копира точно там (ред 10).

Да допуснем, че има и други елементи със стойност j в $A[1,\ldots,i]$. На ред 11, $C[A[i]]$ бива декрементирано, така че то вече съдържа правилното място на j в $A[1,\ldots,i-1]$. След като i бъде декрементирано е вярно, че елементът на C, който току-що беше декрементиран съдържа правилното място на най-дясното j в $A[1,\ldots,i]$. Тъй като всички останали елементи на C са непроменени, първата част от инварианта е в сила.

Сега да допуснем, че в $A[1,\ldots,i]$ няма други елементи със стойност j. Тогава j не е съществен по отношение на останалата част на A (която предстои да бъде сканирана от четвъртия цикъл), така че стойността на $C[A[i]]$ е без значение за алгоритъма оттук нататък. Наистина, след декрементирането на ред 11, $C[A[i]]$ сочи клетка в B, която е мястото на друг елемент (а не на j). Но, както казахме, стойността на това $C[A[i]]$ няма да бъде използвана до края на алгоритъма, понеже тази стойност на $A[i]$ няма да се среща повече. Първата част от инварианта се запазва и в този случай.

Сега ще докажем втората част от инварианта. Да разгледаме момента, когато изпълнението е на ред 9 в началото на текущата итерация. Съгласно индуктивното предположение, всички елементи от $A[i+1,\ldots,n]$ са на своите правилни места в B. Току-що доказахме, че

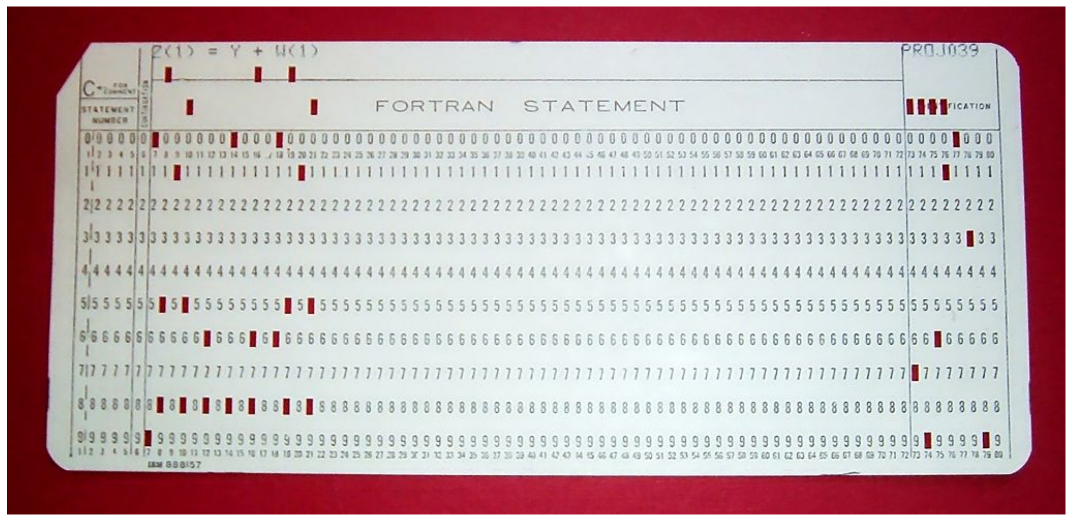
по време на това изпълнение на цикъла, елементът $A[i]$ бива копиран на правилното място. Тогава всички елементи от $A[i, \ldots, n]$ са на своите правилни места в B. По отношение на новата стойност на i, вярно е, че всички елементи от $A[i+1, \ldots, n]$ са на своите правилни места в B.

Терминация. При завършването на цикъла, $i = 0$. Заместваме i с 0 във втората част на инварианта и получаваме "всички елементи от $A[1, \ldots, n]$ са 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 $k = O(n)$, та е $\Theta(n)$. Следователно, алгоритъмът не е *in-place*.

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 реални компютри, и използван до 70те и дори 80те години на 20 век в компютрите. 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 с лиценз CC2.5; върху картата е написана в кодиран вид част от програма на Fortran:



Стандартната Холерит перфокарта има 80 колони, а във всяка колона може да бъде пробита дупка на точно едно измежду дванадесет възможни места. Всяка колона представлява символ. Ако символът е (десетична) цифра, дупката е на една от десетте позиции, маркирани с цифрите. Другите две позиции са за не-цифрова информация. Ако картата записва число, то може да има най-много 80 десетични разряда, колкото са колоните.

Задачата е, при дадено множество карти, които записват числа, картите да се сортират от електро-механично устройство, така че записаните числа да са в ненамаляващ ред. Електромеханичното 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.

Първата идея за сортиращ алгоритъм, който да управлява това електро-механично устройство, е да сортираме картите по най-старшия разряд, после по следващия, и така нататък, до най-младшия разряд. Недостатъкът на този подход е, че след сортирането по най-старши разряд трябва да държим картите в десет различни купчини[†] до края, след сортирането по втори най-старши разряд купчините ще станат сто[‡] и така нататък, което не е практично. Правилният подход е разрядите (колоните) да се тълкуват като ключове: старшият разряд е първичният ключ, вторият най-старши е вторичният, и така нататък, най-младшият разряд е 80-ичния ключ. Както знаем от лекцията по сортиране, ако сортираме цялата купчина (без да я разбиваме на подкупчини и подподкупчини и т. н.) първо по 80-ичния ключ, после от 79-ичния ключ, и така нататък, и най-накрая по първичния ключ, ще имаме правилно сортирани карти. Но: **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.

Ще дадем пример за работата на Radix Sort. Ще сортираме числата 7444, 1320, 1692, 8185, 4007, 6281, 5139 и 7921. Представяме 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.

7 4 4 **4**	→	1 3 **2** 0	→	4 **0** 0 7	→	**4** 0 0 7	→	1 3 2 0
1 3 2 **0**		6 2 **8** 1		1 **3** 2 0		**8** 1 8 5		1 6 9 2
1 6 9 **2**		7 9 **2** 1		7 **9** 2 1		**5** 1 3 9		4 0 0 7
8 1 8 **5**		1 6 **9** 2		5 **1** 3 9		**6** 2 8 1		5 1 3 9
4 0 0 **7**		7 4 **4** 4		7 **4** 4 4		**1** 3 2 0		6 2 8 1
6 2 8 **1**		8 1 **8** 5		6 **2** 8 1		**7** 4 4 4		7 4 4 4
5 1 3 **9**		4 0 **0** 7		8 **1** 8 5		**1** 6 9 2		7 9 2 1
7 9 2 **1**		5 1 **3** 9		1 **6** 9 2		**7** 9 2 1		8 1 8 5

Псевдокодът на Radix Sort е съвсем прост (съгласно [31]):

Radix Sort($A[1,\ldots,n]$: положителни числа, записани с един и същи брой цифри; d: броят на цифрите)

```
1  (* най-младшата цифра е номер 1, най-старшата цифра е номер d *)
2  for i ← 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, примерно 10, то сложността по време е $\Theta(dn)$.

Използвайки Radix Sort, можем да решим следната задача: да се сортира в линейно време масив от n числа, всяко от което принадлежи на множеството $\{1, 2, \ldots, n^2\}$. Ако

[†] Купчините биха били десет, ако и десетте цифри се появяват като най-старша цифра на някоя карта.

[‡] Ще станат сто, ако и десетте цифри се появяват като и като най-старша цифра, и като втора най-старша цифра, на някоя карта. Корекция на Емилиян Рогачев: Купчините ще станат 100, ако всяка наредена двойка цифри се среща като начални две цифри.

опитаме директно да я решим 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 $\lfloor \lg n^2 \rfloor + 1 = \lfloor 2 \lg n^2 \rfloor + 1 \approx 2 \lg n$ 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 има най-много $\Theta(n)$ различни стойности.

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 58: 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 58 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 59: 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 58 и Определение 59 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 60: 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 $f_G : E \to V \times V$.

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: $\mathsf{N}(u)$ и $\mathsf{N}(U)$: съседите на u и U

Нека $G = (V, E)$ е обикновен граф, $u \in V$ и $U \subseteq V$.

$$\mathsf{N}(u) = \{v \in V \,|\, u \text{ и } v \text{ са съседи}\}$$
$$\mathsf{N}(U) = \bigcup_{x \in U} \mathsf{N}(x) \setminus U$$

Буквата 'N' идва от **n**eighbourhood. В алгоритмите върху графи обаче пишем "**for** $u \in \text{adj}[u]$", а не "**for** $u \in \mathsf{N}(u)$". Причината е, че нотацията "$\mathsf{N}(u)$" се ползва в контекста на теорията на графите, а "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: $\mathsf{SC}(G)$ и $\mathsf{SCC}(G)$: релацията на силна свързаност и нейните класове на екв.

Нека G е ориентиран граф. "$\mathsf{SC}(G)$" означава релацията на силна свързаност над $V(G)$, а "$\mathsf{SCC}(G)$" означава множеството от нейните класове на еквивалентност.

Определение 61: 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 не знае общоприет български термин за "ориентиран граф без цикли/ориентиран ацикличен граф", нито харесва "огбц/оаг", така че ще ползва "даг".

Определение 62: 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 8: 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 40: Ограничения за m при дадено n

Ако разглеждаме обикновен неориентиран граф без примки, то $0 \leqslant m \leqslant \binom{n}{2}$, като границите са точни. Ако разглеждаме обикновен ориентиран граф без примки, то $0 \leqslant m \leqslant n(n-1)$, като границите са точни. Ако разглеждаме обикновен ориентиран граф, който може да има примки, $0 \leqslant m \leqslant n^2$, като границите са точни. Ако разглеждаме мултиграф, няма горна граница за m, така че $0 \leqslant m$ 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 $G = (V, E)$, $V = \{u_1, \ldots, u_n\}$ и $E = \{e_1, \ldots, e_m\}$.

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 41

Ако неориентиран граф, който е мултиграф или не, 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 е $m + n$, ако е представен чрез списъци. Това е в сила и за мултиграфи, и за графи, които не са мултиграфи. От друга страна, ако е представен чрез матрица, размерът е n^2.

Конвенция 9

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 7: “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 63: 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 $|E(G)| \leqslant c|V(G)|$.

Забележете, че въпросът дали даден граф е разреден граф е **безсмислен въпрос**, говорейки формално. Съгласно Определение 63, въпросът е смислен само за безкрайни множества от графи, или *класове от графи*. Важни класове графи, които са разредени, са:

дърветата. Знаем, че за всяко дърво $m = n - 1$.

планарните графи. Знаем, че за всеки планарен граф $m \leqslant 3n - 6$.

графите, в които максималната степен на връх е ограничена от константа. Знаем, че $\sum_{u \in V} d(u) = 2m$, което влече $\Delta(G)n \geqslant 2m$, а оттук $m \leqslant \frac{\Delta(G)}{2}n$. Да си припомним, че "$\Delta(G)$" е максималната степен на връх в графа G.

[†]Това няма **нищо общо** с динамичното програмиране, което ще разгледаме в Лекция 12

Наблюдение 42: 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** неориентирани графи няма – всички графи са само ориентирани, а понякога на тези със "сдвоените" ребра казваме "неориентирани 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 неориентирани графи няма. Ето някои от проблемите, до които би довело решението да отречем съществуването на неориентирани графи.

- Веднага ще следва, че няма нетривиални ациклични неориентирани 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 класифицират ребрата на графа по някакъв начин, ще класифицират 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 и то да бъде класифицирано в един клас.

- Някои алгоритмични задачи, които по принцип са върху ориентирани графи, може да бъдат разгледани и върху неориентирани графи в някои случаи. Например, Обхождане на граф или Най-къс път в граф[†]. Но задачите върху неориентирани графи като Минимално покриващо дърво или Върхово покриване или Доминиращо множество 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. Например, обобщението на Минимално покриващо дърво върху ориентирани графи е **NP**-пълната задача Minimum Equivalent Digraph[‡] [51, стр. 65].

[†]Както ще видим в Лекция 11, Най-къс път в граф върху неориентирани графи е безсмислена при наличие на отрицателни тегла, докато върху ориентирани графи има смисъл дори при отрицателни тегла, стига да няма отрицателни цикли.

[‡]MED е дефинирана за ориентирани графи, които не са тегловни. Ясно е, че ако добавим и тегла, задачата няма да стане по-лесна.

8.2 Обхождания на графи

Обхождането на графи е базова изчислителна задача. 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 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 "зациклял".

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 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 екстремен случай, да я обходим цялата,
- в друг екстремен 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 64: достижим подграф

Нека G е ориентиран мултиграф с възможни примки и u е връх в него. *Подграфът на* G*, достижим от* u е подграфът на G, индуциран от $\{v \in V(G) \mid$ съществува път от u 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: необходен и обходен. Белите върхове са необходените, 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 зацикли или да пропусне. Нашето обхождане не зациклят, 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. $\lightning$

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.

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.

[†] 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 интейрфейс. Аналогично, стек е АТД с интерфейс push и pop.

Приемете, че върховете са числата 1, 2, ..., 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 color$[1..n]$, където color$[i]$ е текущият цвят на връх i; 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 62) 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 $T_1, \ldots, T_k$, 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 $T_1, \ldots, T_k$. Казваме, че колекцията $\{T_1, \ldots, T_k\}$ 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 65: 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 $\delta(u, v) \leqslant \delta_H(u, v)$.

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. 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 65 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 терминирането 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 $\boxed{\text{enqueue}(Q, y)}$. Алгоритъмът спира, когато Q стане празна. Очевидно интерфейсът 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 Псевдокод на BFS от един стартов връх

BFS($G = (V, E)$: ориентиран мултиграф с възможни примки, като $V = \{1, \ldots, n\}$; $s \in V$)

```
1   for i ← 1 to n
2       color[i] ← white
3       d[i] ← ∞
4       π[i] ← Nil
5   color[s] ← gray
6   d[s] ← 0
7   създай опашка Q
8   enqueue(Q, s)
9   while not isempty(Q) do
10      x ← dequeue(Q)
11      for y ∈ adj[x]
12          if color[y] = white
13              color[y] ← gray
14              d[y] ← d[x] + 1
15              π[y] ← x
16              enqueue(Q, y)
17      color[x] ← black
```

Коректността на BFS. Това, че BFS терминира върху всеки вход, е очевидно: 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 освен това графът е краен, рано или късно тя опашката ще стане празна и BFS ще спре.

В Теорема 49 и помощните леми, нека $G = (V, E)$ е произволен ориентиран мултиграф и s е произволен връх в него. Нека G' е подграфът на G, достижим от s.

Лема 43

При терминирането на BFS(G, s) няма сиви върхове.

Доказателство: Следното твърдение е инвариант за **while** цикъла (редове 9–17).

Инвариант 21: Цикълът на BFS (1)

Всеки път, когато изпълнението е на ред 9, множеството от сивите върхове е точно множеството от върховете в опашката Q.

База. При първото достигане на ред 9 твърдението е вярно, понеже на ред 7 е създадена празна опашка, а на ред 8 в нея е сложен (само) връх s. От друга страна, в този момент s е единственият сив връх.

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. По допускане, всички сиви върхове са в Q и в Q има само сиви върхове. Един сив връх бива изваден от Q на ред 10, но този връх става черен на ред 17. По време на изпълнение на тялото на цикъла, всяко бяло дете на този връх става сиво (ред 13), но след това то влиза в опашката на ред 16. Очевидно е, че при следващото достигане на ред 9 твърдението е вярно.

Терминация. При последното достигане на ред 9 опашката Q е празна. Това означава, че няма сиви върхове. □

Доказателството на Лема 44 е практически същото като доказателството за коректност на идеята на обхождането на стр. 369.

Лема 44

При терминирането на $\text{BFS}(G, s)$:

- за всеки $v \in V(G')$ е вярно, че $\text{color}[v] = \text{black}$ и $d[v] < \infty$;
- за всеки $v \in V \backslash V(G')$ е вярно, че $\text{color}[v] = \text{white}$ и $d[v] = \infty$.

Доказателство: Разглеждаме момента, в който BFS терминира. От Лема 43 знаем, че при терминирането на BFS сиви върхове няма. Очевидно е, че белите върхове са точно тези, чиято d-стойност е ∞, а черните са тези, чиято d-стойност не е ∞. Очевидно е също така, че всички върхове от $V \backslash V(G')$ са бели. Остава да докажем, че всички върхове от $V(G')$ са черни.

Да допуснем противното. Тогава съществува връх $u \in V(G')$, който е бял в разглеждания момент. Щом u е връх в G', съществува път p от s до u. Очевидно $u \neq s$, тъй като в този момент s е черен. Щом началото на p е черен връх, а краят му е бял връх, съществува ребро (a, b) в p, такова че a е черен и b е бял. Но щом a е черен, той е станал черен на ред 17 в някакъв момент t, в който променливата x е имала съдържание a. Преди a да стане черен, всяко негово дете y е било "прегледано" на ред 11. Но b е дете на a, щом има ребро (a, b), така че в някой момент $\hat{t} < t$ е било вярно, че $y = b$. Тъй като цветът на $y = b$ е бил бял и в момента $\hat{t}$, булевото условие на ред 12 е било TRUE и връх b трябва да е станал сив на ред 13. Веднъж престанал да бъде бял, той няма как да се окаже бял при терминиране на BFS. ↯

Лема 45

При терминирането на $\text{BFS}(G, s)$, за всеки $v \in V$ е вярно, че $d[v] \geq \delta(s, v)$.

Доказателство: Ще докажем твърдението само за върховете от $V(G')$, тъй като онези от $V \backslash V(G')$ не са достижими от s, така че $\forall V \backslash V(G') : d[v] = \infty$ в края на алгоритъма, а $\delta(s, v) = \infty$ по дефиниция. Следното твърдение е инвариант за **while** цикъла (редове 9–17).

Инвариант 22: Цикълът на BFS (2)

Всеки път, когато изпълнението е на ред 9, за всеки връх $v \in V(G')$ е вярно, че $d[v] \geq \delta(s, v)$.

База. При първото достигане на ред 9 разглеждаме поотделно s и останалите върхове.

- Да разгледаме s. Тъй като $d[s] = 0$ (ред 6) и $\delta(s, s) = 0$ (от теорията на графите), неравенството $d[s] \geqslant \delta(s, s)$ 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 $d[v] \geqslant \delta(s, v)$ 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

  $$d[x] \geqslant \delta(s, x)$$

  Оттук

  $$d[x] + 1 \geqslant \delta(s, x) + 1$$

  което е същото като

  $$d[y] \geqslant \delta(s, x) + 1$$

  Но $\delta(s, x) + 1 \geqslant \delta(s, y)$, понеже, щом има път p от s до x, има и път от s до y с дължина $|p| + 1$[†]. Тогава

  $$d[y] \geqslant \delta(s, y)$$

  Доказахме, че за всеки връх y, който е бил "обработен" при изпълнение на текущата итерация, $d[y] \geqslant \delta(s, y)$.

- 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.

  Неравенството очевидно остава в сила за върховете на G', които не са деца на x; те не биват "обработени" изобщо в текущата итерация.

Терминация. При последното достигане на ред 9 за всеки $v \in V(G')$ е вярно, че $d[v] \geqslant \delta(s, v)$, което и трябваше да покажем. □

Лема 46 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. С други думи, опашката "обработва" върховете по d-стойности. Което, както ще видим в Теорема 49, е същото като да "обработва" върховете по ориентираните разстояния от s.

[†] В [31] това очевидно твърдение е Lemma 22.1 на стр. 598.

Лема 46: (Lemma 22.3 в [31, стр. 599])

Нека в произволен момент t от работата на $\text{BFS}(G, s)$, опашката Q съдържа следната пермутация на върхове: $\langle u_1\, u_2 \cdots u_k \rangle$, като u_1 е най-отдавна сложеният връх, а u_k е най-скоро сложеният връх. Тогава

$$d[u_1] \leqslant d[u_2] \leqslant \cdots \leqslant d[u_k] \leqslant d[u_1] + 1$$

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.

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 момент t от изпълнението на BFS. Щом това достигане на ред 9 не е последното, опашката Q не е празна в момент t. Да кажем, че Q съдържа пермутацията на върхове $\langle u_1\, u_2 \cdots u_k \rangle$. На ред 10 връх u_1 бива изваден от Q и сложен в променливата x. Вече опашката съдържа пермутацията $\langle u_2\, u_3 \cdots u_k \rangle$, така че неравенствата по отношение на върховете в опашката стават

$$d[u_2] \leqslant \cdots \leqslant d[u_k] \leqslant d[u_1] + 1$$

Имайки предвид това, че $d[u_1] \leqslant d[u_2]$, имаме $d[u_1] + 1 \leqslant d[u_2] + 1$, откъдето следва, че

$$d[u_2] \leqslant \cdots \leqslant d[u_k] \leqslant d[u_2] + 1$$

Ерго, твърдението остава в сила веднага след изваждането на връх от опашката.

Да се убедим, че твърдението остава в сила след вкарването на връх в опашката във вътрешния цикъл (редове 11–16). Разглеждаме произволен връх u', който е дете на текущия x (ред 11) и е бял. Помним, че $x = u_1$. Рано или късно, променливата y получава съдържание u' и този връх бива сложен в опашката на ред 16 в някакъв момент $\hat{t} > t$. Преди да бъде сложен в опашката, връх u' получава d-стойност на ред 14. А именно, $d[u'] \leftarrow d[u_1] + 1$. Текущият първи връх в опашката е u_2, а вече видяхме, че $d[u_1] + 1 \leqslant d[u_2] + 1$. Оттук следва, че $d[u'] \leqslant d[u_2] + 1$. Но съдържанието на опашката в момент $\hat{t}$ е $\langle u_2\, u_3 \cdots u' \rangle$, така че твърдението остава в сила.

Очевидно твърдението остава в сила дори когато не се добавят нови върхове в Q, а също така и когато Q е празна; ако Q е празна, твърдението е вярно в празния смисъл (vacuously). □

Теорема 49: Коректността на BFS (Theorem 22.5 в [31, стр. 599])

След терминирането на $\text{BFS}(G, s)$:

1. Обходеният подграф на G е точно G'.
2. За всеки връх $v \in V(G')$, $d[v] = \delta(s, v)$.
3. Дървото на обхождането, реализирано чрез $\pi[1 \mathinner{..} n]$, е дърво на най-къси пътища за G' с корен s.

Доказателство: Твърдение 1 следва директно от Лема 44.

Ще докажем 2. Да допуснем, че съществува връх $u \in V(G')$, такъв че $d[u] \neq \delta(s,u)$ след терминирането на BFS(G,s). Нека v е такъв връх, като освен това стойността $\delta(s,v)$ е минимална. Очевидно $v \neq s$. Съгласно Лема 45, $d[v] \geqslant \delta(s,v)$. От това и допускането, че $d[v] \neq \delta(s,v)$ следва, че $d[v] > \delta(s,v)$.

Щом $v \in V(G')$ 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 родителя 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 62). 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 **for**-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 върха в променливата x е бил някой $a \in V(G')$, 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 Лема 46, че $d[a] \leqslant d[w]$, което е същото като $d[a] + 1 \leqslant d[w] + 1$. Тогава $d[v] \leqslant d[w] + 1$, 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 Лема 46, $d[v] \leqslant d[w]$, в противоречие с (8.1). ↯

Ще докажем 3. Нека дървото на обхождането, реализирано чрез масива на родители $\pi[1 \,..\, n]$, се казва T. Твърди се, че за всеки връх $v \in V(G')$, $\delta(s,v) = \delta_T(s,v)$. Тривиално е да докажем това по индукция по реда на влизане на върховете в опашката.

Базата е за $v = s$ и твърдението очевидно е вярно. Нека $v \neq s$. v получава своите d- и π-стойности на някоя итерация на на **while**-цикъла, променливата y има стойност v, а променливата x има някаква стойност u, където u е родител на v в G' и е родител**ят** на v в T. А именно, $d[v]$ става $d[u] + 1$ и $\pi[v]$ става u. Тъй като u влиза в опашката преди v, по индукционното предположение, пътят от s до u в T е най-къс път от s до u в G. Имайки предвид това и факта, че $\delta(s,v) = \delta(s,u) + 1$, виждаме, че след присвояването $\pi[v] \leftarrow u$, масивът π реализира най-къс от s до v, така че $\delta(s,v) = \delta_T(s,v)$. □

Сложност по време. Реализацията на BFS, която разгледахме, работи във 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 (допускаме, че $G' = G$):

> **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 това разсъждение следва асимптотична горна граница $O(nm)$, но, в общия случай, тя не е точна. Да, има безкрайни класове графи, за които това разсъждение е приложимо. Примерно, клас графи, в който всеки граф има един и същи брой ребра; тоест, $m = \Theta(1)$. Тогава наистина сметката е такава. Но в общия случай това не е така.

Прецизното разсъждение, което ни дава точна асимптотична оценка винаги, е да се игнорира **while**-цикълът и да се фокусираме върху вътрешния цикъл, тоест, върху **for**-цикъла. Но не върху всички изпълнения на **for**-цикъла в рамките на едно изпълнение на **while**-цикъла, а върху всички изпълнения на **for**-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, може да разсъждаваме така:

> **while**-цикълът се изпълнява n пъти, по веднъж за всеки връх, а всяко изпълнение отнема време, в най-лошия случай, $\Theta(n)$, така че сложността е $\Theta(n^2)$.

Това разсъждение вече не е грешно. Наистина, най-лошият случай, в който $m = \Theta(n)$, е точно такъв. Но $\Theta(n + m)$ остава по-прецизна оценка, тъй като функцията в израза е на две променливи и това е по-информативно за общия случай.

И така, сложността по време на BFS е $\Theta(n + m)$.

8.4.2 Псевдокод на 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, ..., 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 ∈ 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      for 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 замисимост 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 амортизирания 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 съседи са зелени, техните съседи без s са червени, и така нататък,
- и обратно, s е зелен, неговите съседи са червени, техните съседи 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, грубо казано, такава: има стаи, в които сме били, но с които не сме приключили; ние първо преглеждаме всички коридори, излизащи от тях, преди да продължим с новите стаи, като новите стаи са тези, които сме открили чрез въпросните коридори.

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 връщаме колкото е възможно по-малко назад до стая, в която сме били, и опитваме същото

[†]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 използва по време на работата си!

нещо, само че излизайки от нея през врата, която не сме използвали. На английски тази идея се нарича *backtracking*[†].

Допълнение 37: Разликата между BFS и DFS, илюстрирана с шах

Ще илюстрираме разликата между двете идеи—BFS и DFS—с шаха. Нека е дадена задача да открием мат в три хода в дадена позиция за белите[a]. 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.

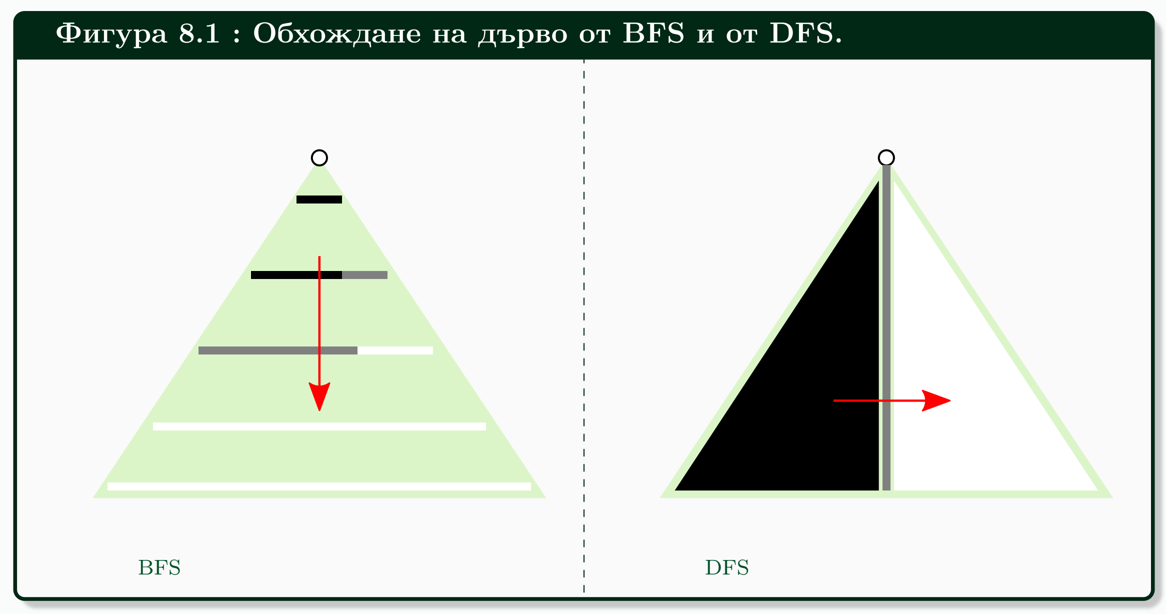
В тази редица от полуходове се редуват квантори $\exists$ и $\forall$, като $\exists$ са на белите полуходове, а $\forall$, на черните. Тъй като възможните полуходове от дадена позиция може да са десетки, дори при малко фигури, говорим за дърво с разклоненост от порядъка на десетки. Височината му в полуходове е шест. Това е дървото на BFS-а, който току-що си представихме. При BFS идеята генерираме нивàта 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, 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.

[†]Backtracking е алгоритмична схема, също както разделяй-и-владей е алгоритмична схема. В този курс няма да разглеждаме алгоритми, използващи backtracking, освен DFS.

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 нивá. DFS генерира елементите на нивата, но последователно, а не всички наведнъж.
Ако си представим търсенето на мат като дърво с корен началната позиция, BFS генерира цялото дърво[б], вървейки вертикално, отгоре надолу, ако коренът е горе. Докато DFS поддържа само един вертикален път от корена до листата, като този път се движи хоризонтално, образно казано. Във всеки момент DFS съхранява в явен вид само позициите от този път, а следващите позиции се пресмятат от текущите. Фигура 8.1 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 8.1 : Обхождане на дърво от BFS и от DFS.



[а]Ето сайт с такива *шахматни задачи*.
[б]Почти цялото; може би без върхове от най-долното ниво.

Ако заменим в кода на BFS опашката със стек (съответно и функциите от интерфейса с push и pop), ще получим **нещо като** DFS, но не точно DFS. Нека читателят пробва да замени опшаката 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 [31, 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 нивá, които се пресмятат като d-стойностите в BFS, но при DFS нивáта 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 $f[1..n]$ съхра-

няват тази информация за върховете.

Разглеждаме вариант на DFS, който задължително обхожда целия граф, подобно на модификацията на BFS на стр. 376. Тривиално лесно е да направим от него DFS, който получава стартов връх като аргумент и обхожда само подграфа, който е достижим от този стартов връх.

```
DFS(G = (V, E), като V = {1, ..., n})
1   time е глобална
2   for i ← 1 to n
3       color[i] ← white
4       π[i] ← Nil
5   time ← 0
6   for i ← 1 to n
7       if color[i] = white
8           DFS-VISIT(G, i)
```

```
DFS-VISIT(G = (V, E); x ∈ V)
1   color[x] ← gray
2   time++
3   d[x] ← time
4   foreach y ∈ adj[x]
5       if color[y] = white
6           π[y] ← x
7           DFS-VISIT(G, y)
8   color[x] ← black
9   time++
10  f[x] ← time
```

Много подробен пример за работата на DFS има в [31, стр. 605]. Забележете, че наредените двойки $(d[v], f[v])$, по всички $v \in V$, напълно определят работата на DFS върху даден граф. Конвенцията на [31] е върховете да се рисуват като елипси, а в елипсата, съответстваща на връх v, наредената двойка $(d[v], f[v])$ да се записва като "$d[v]/f[v]$". Например, в рисунката, показваща края на алгоритъма, връх u е маркиран с "1/8", което означава, че връх u е бил открит в момент 1 и е бил финализиран в момент 8.

Формално доказателство за коректността на DFS няма да правим тук. Аргументацията на стр. 369 за коректността на общата схема е достатъчна. Сложността по време на DFS е $\Theta(n+m)$. Това се извежда със същите разсъждения, с които се извежда линейната сложност на BFS.

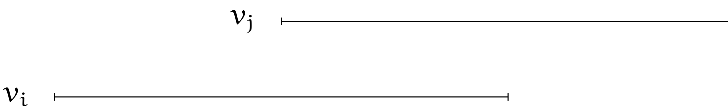
8.5.3 Свойства на DFS

Скобова структура на интервалите, генерирани от DFS. За целите на това изложение, нека върховете са $v_1, \ldots, v_n$. Очевидно е, че променливата time, която се инициализира с 0 на ред 5, приема последователно стойностите 1, 2, ..., $2n$, тъй като бива инкрементирана два пъти за всеки връх x в DFS-VISIT. Да си представим числовата ос с тези $2n$ стойности на t. Ако групираме стойностите по двойки за отделните върхове, получаваме n интервала $[d[v_1], f[v_1]]$, $[d[v_2], f[v_2]]$, ..., $[d[v_n], f[v_n]]$. 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 22.5 в [31, стр. 607] илюстрира добре интервалите на DFS.

Теорема 50: Скобова структура на интервалите при DFS

В текущия контекст, ако два различни интервала имат непразно сечение, то единият се съдържа строго в другия: .

Доказателство: Твърди се, че е невъзможно да има такива интервали: . Да допуснем противното. Нека има върхове v_i и v_j, такива че $d[v_i] < d[v_j] < f[v_i] < f[v_j]$:



DSF първо открива v_i, а после v_j, като v_i не е финализиран в момента на откриването на v_j. Това означава, рекурсивното викане R_i на DFS-VISIT, в което $x = v_i$, не е приключило в момента, в който в някое следващо рекурсивно викане R_j е вярно, че $x = v_j$.

Същото нещо, по-подробно казано, звучи така.

- В рекурсивното викане R_i е вярно, че $x = v_i$ и се изпълнява цикълът на редове 4–7. $d[v_i]$ получава своята стойност на ред 3. За някое дете u' на v_i е вярно, че когато y (в R_i) стане u' се прави рекурсивно викане R' на DFS-VISIT.
- В рекурсивното викане R' е вярно, че $x = u'$ и се изпълнява цикълът на редове 4–7. За някое дете u'' на u' е вярно, че когато y (в R') стане u'' се прави рекурсивно викане R'' на DFS-VISIT.
- И така нататък ...
- В рекурсивното викане R_j е вярно, че $x = v_j$.

Ключовото наблюдение е, че в момента, в който $d[v_j]$ получава своята стойност на ред 3—това става в рекурсивното викане R_j—викането R_i още не е приключило и в R_i се изпълнява цикълът на редове 4–7, така че ред 10 в R_i не е достигнат и $f[v_i]$ все още не е получила стойност. Със сигурност $f[v_i]$ ще получи стойност, но това ще е за стойност на time, по-голяма от стойността на time в момента, в който $d[v_j]$ получава своята стойност. ↯ □

Класифициране на ребрата на ориентиран граф от DFS. Нека $G = (V, E)$ е ориентиран мултиграф с възможни примки. Нека гората на обхождането, реализирана чрез масива π, е A. Да си спомним, че A е множество от дървета. В края на DFS(G) ребрата от E се разбиват[†] на следните четири категории:

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.

[†]Някои дялове-категории на това разбиване може да са празни, така че, формално, правилно е да се каже “се разбиват на най-много четири категории.

В [31, 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 [31, стр. 605] и на Фигура 22.5 в [31, стр. 607], като на Фигура 22.5.c 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. Приемаме това за очевидно.
- 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. Ерго, само от черния цвят на y не можем да класифицираме точно (x, y).

Сега да разгледаме възможностите за взамните 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 $d[y] < f[y]$, има $3 + 2 + 1 = 6$ възможности от общи комбинаторни съображения. С по-подробен анализ, възможностите стават само три.

1. x ⊢—⊣ ⊢—⊣ y Това е невъзможно. Щом (x, y) е ребро, няма как x да бъде финализиран на ред 10 преди бялото му дете y да бъде открито на ред 5.
2. x y Това е невъзможно съгласно Теорема 50.
3. y x y е предшественик на x. В такъв случай y е сив и (x, y) е ребро назад. Всички примки са такива.
4. y x x е родителят на y. В такъв случай y е бял и (x, y) е дървесно ребро или y е черен и (x, y) е ребро напред.
5. y x Това е невъзможно съгласно Теорема 50.
6. y ⊢—⊣ ⊢—⊣ x Това е възможно. Щом y е финализиран преди откриването на x, нито x е предшественик на y, нито y е предшественик на x. В този случай y е черен и (x, y) е ребро настрани.

Сега е ясно как да различаваме дали (x, y), при черен връх y, е ребро напред или ребро настрани. Ако $d[x] < f[y]$, то (x, y) е ребро напред, ако $f[y] < d[x]$, то (x, y) е ребро настрани.

Следната лема обобщава казаното дотук. Доказателството ѝ са току-що извършените разсъждения.

Лема 47: 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.

Класифициране на ребрата на неориентиран граф от DFS. Нека $G = (V, E)$ е неориентиран мултиграф с възможни примки. Нека гората на обхождането, реализирана чрез масива π, е A. В края на DFS(G) ребрата от E се разбиват на следните две категории:

дървесни ребра. Това са точно ребрата на A.

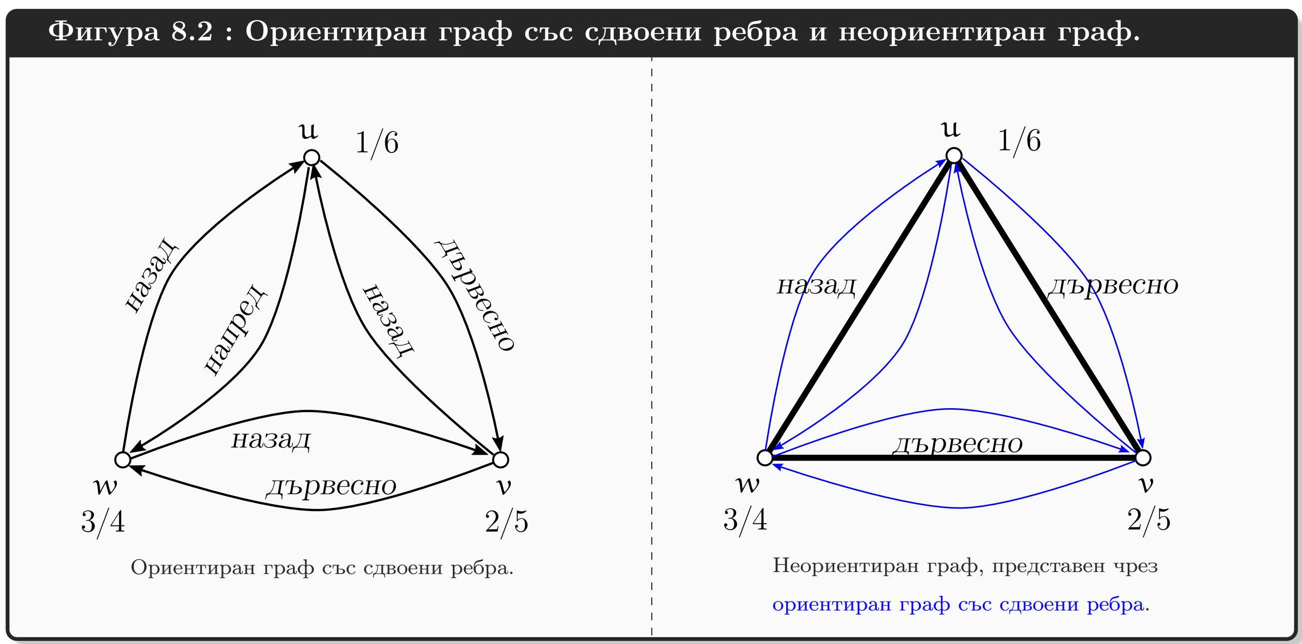
ребра назад. (u, v) е ребро назад, ако u и v са върхове от едно и също дърво $T \in A$, като v е предшественик на u в T. Примките в G задължително са в тази категория.

С други думи, в неориентираните графи няма ребра напред или ребра настрани. Общото за ребрата напред или 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 y на ред 5 е черен—е вярно, че неориентираното ребро (x, y) **вече** е било обходено от DFS и **вече** е било класифицирано 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. 366) 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 класифицираме едно и също неориентирано ребро по два начина, съответно за двете ориентирани ребра, които го представят. И класифицирането на реброто никога не става като ребро напред или настрани.

Разгледайте Фигура 8.2. 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, докато реброто (v, u) е ребро назад, тоест, ребрата от двойката биват класифицирани по различен начин, докато в неориентирания граф на тях отговаря само едно ребро и то бива класифицирано като дървесно, защото първият път, в който прекосяваме едното от двете отговарящи му ориентирани ребра от представянето (в синьо), откриваме бял връх (v).

Фигура 8.2 : Ориентиран граф със сдвоени ребра и неориентиран граф.



Ориентиран граф със сдвоени ребра.

Неориентиран граф, представен чрез ориентиран граф със сдвоени ребра.

Фигура 8.2 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 класифицирано като ребро напред само ако другото ребро от двойката вече е било прекосено и класифицирано като ребро назад. Ако графът е неориентиран и това са само ориентираните ребра от преставянето, очевидно е, че реброто като един обект ще бъде класифицирано съгласно първото прекосяване, а именно като ребро назад. С аналогични разсъждения може да изведем, че в неориентирани графи няма ребра настрани.

Следната теорема дава формално доказателство, че при неориентираните графи ребрата са само дървесни и ребра назад.

Теорема 51: Theorem 22.10 в [31, стр. 610]

При всяка работа на DFS върху неориентиран граф G, всяко обходено ребро е дървесно или ребро назад.

Доказателство: Разглеждаме произволно ребро $(u, v) \in E(G)$. БОО, нека $d[u] < d[v]$. Тогава DFS и открива, и финализира v преди да финализира u; това е очевидно.

Да разгледаме двете възможности за обхождане на реброто (u, v):

1. Ако то бъде обходено в посока от u към v, то чрез него ще открием бял връх (v) и ще го класифицираме като дървесно.

2. Ако то бъде обходено в посока от v към u, то чрез него ще открием сив връх (u) и ще го класифицираме като ребро назад. □

8.5.4 Приложения на DFS

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 52: 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 DFS(G) открива поне едно ребро назад.

Доказателство: В едната посока, да допуснем, че $e = (u, v)$ е ребро назад. Ако G е неориентиран, нека обхождането става в посока от u към v. Както вече видяхме, това означава, че в момента на обхождането, u и v са сиви, като 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 $u = v$.

В другата 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 съседен връх u в c.

1. v има поне два съседни върха u, w, които са от цикъла (ако става дума за мултиграф, може $u = w$; ако c е примка, може дори $v = u = w$; всичко това е без значение за доказателството), ако G е неориентиран;

2. v има поне един съседен връх u, който е от цикъла (възможно е $u = v$, ако реброто е примка; това е без значение за доказателството), ако G е ориентиран.

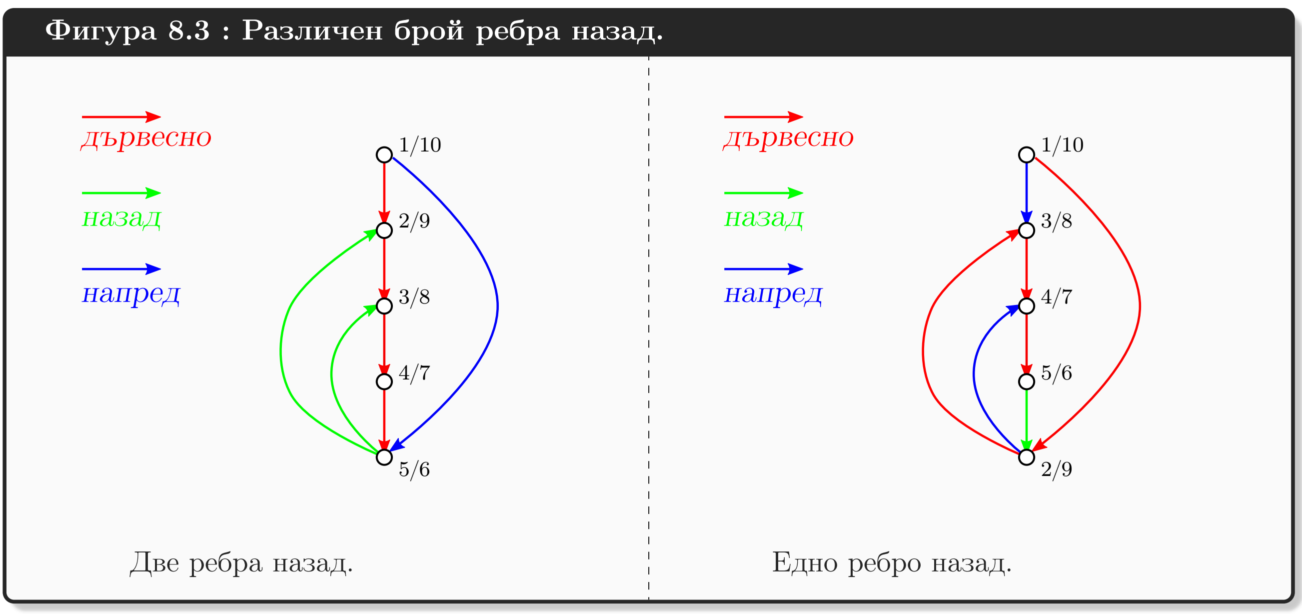
Когато DFS-VISIT минава през списъка на съседство на v, открива u като сив връх и реброто (v, u) бива класифицирано като ребро назад. □

Наблюдение 43

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 8.3 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. □

Фигура 8.3 : Различен брой ребра назад.



Две ребра назад.

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 даго-ве: алгоритъмът на Kahn и алгоритъмът на Tarjan. Разглеждаме линеен 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 66: Топологическо сортиране

Топологическо сортиране на G, или накратко *топо-сортиране на* G, е всяка биекция $h : V \to \{1, \ldots, n\}$, 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 конструиране 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 53: НДУ за съществуване на топологическо сортиране

Съществува топологическо сортиране на G тогава и само тогава, когато G е даг.

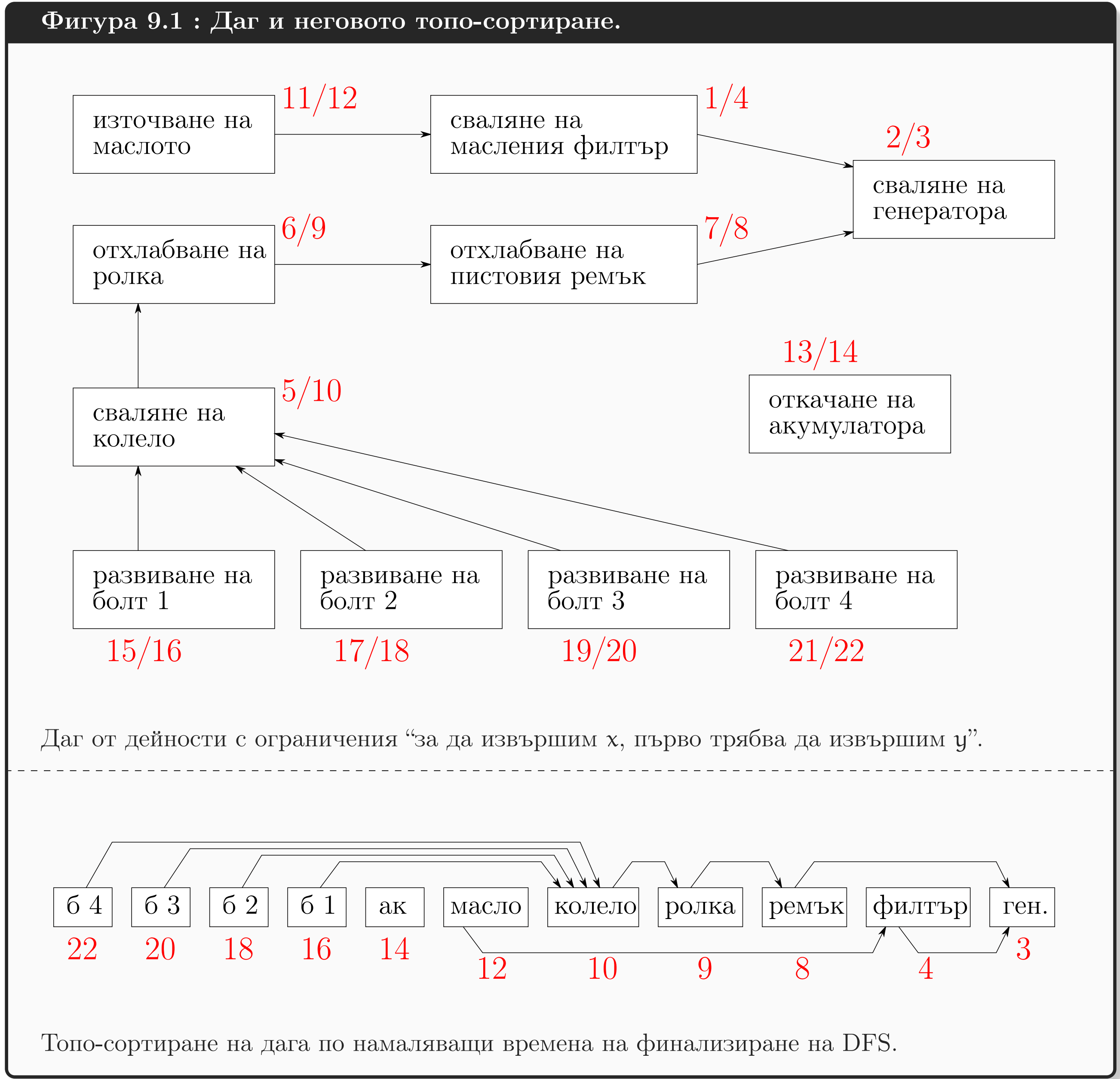
Източник в G, на английски *source*, е всеки връх в G, чиято полустепен на входа е нула. *Сифон в* G, на английски *sink*, е всеки връх в G, чиято полустепен на изхода е нула.

Теорема 54: Поне един източник и поне един сифон в даг

Във всеки даг има поне един източник и поне един сифон.

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 предшествия 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 38: 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 Makholm 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. [79], 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 39: 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 изолирани върхове; затова дейността сваляне на акумулатор не присъства.

```
$ 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 [138]. 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 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. Топологическата 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.

```
TSORT(Даг G = (V, E), като V = {1, ..., n})
1   T[1..n], time и ℓ са глобални
2   for i ← 1 to n
3       color[i] ← white
4       π[i] ← NIL
5   time ← 0,  ℓ ← n
6   for i ← 1 to n
7       if color[i] = white
8           TSORT-REC(G, i)
9   return T[1..n]
```

```
TSORT-REC(G = (V, E); x ∈ V)
1   color[x] ← gray
2   time++
3   d[x] ← time
4   foreach y ∈ adj[x]
5       if color[y] = white
6           π[y] ← x
7           TSORT-REC(G, y)
8   color[x] ← black
9   time++
10  f[x] ← time
11  T[ℓ] ← x
12  ℓ--
```

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 54—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 55: Коректността на топологическото сортиране на Tarjan

Ако G е даг, пермутацията $\pi = \langle u_1 u_2 \cdots u_n \rangle$ на върховете на G, такава че $f[u_1] > f[u_2] > \cdots > f[u_n]$ след терминирането на DFS(G), реализира топологическо сортиране.

Доказателство: Нека (u, v) е произволно ребро на графа. Съгласно Лема 47, 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: $\pi = \cdots u \cdots v \cdots$, така че 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 52 това би означавало, 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. □

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 46: 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. Анализирайте 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 една единствено топо-сортиране, която е точно същото като подредбата на върховете в p.

- В другата посока, ако топо-сортирането е единствено, то за всеки два върха u и v, които са съседи 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 p има ребро от левия към десния, то топо-сортировката 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 "преброждане" 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 "преброждането" на всички списъци, което, както знаем, става във време $\Theta(n + m)$.

Дори да се наложи да се генерира второ топо-сортиране, сложността по време остава $\Theta(n + m)$. □

[†] Това е най-лошият случай за броя на топо-сортиранията – G няма ребра изобщо, така че всяка пермутация на върховете реализира топо-сортиране.

9.3 Алгоритъм на Kahn за топологическо сортиране

Този алгоритъм е описан още през 1962 г. [79]. Той работи, в някакъв смисъл, обратно на алгоритъма на Tarjan. Докато алгоритъмът на Tarjan върви отдясно наляво, 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(Даг G = (V, E), като V = {1, ..., n})
 1  S ← ∅
 2  for i ← 1 to n
 3      A[i] ← 0
 4  for x ← 1 to n
 5      foreach y ∈ adj[x]
 6          A[y]++
 7  for i ← 1 to n
 8      if A[i] = 0
 9          S ← S ∪ {i}
10          color[i] ← gray
11      else
12          color[i] ← 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 $A_{\overline{B}}$ е множеството от останалите клетки на A.

Да си припомним, че "$G - B$" означава графа, получен от G с изтриване на върховете от B.

Приемаме за очевиден ефектът от работата на редове 1–13, поради което го обобщаваме в следното наблюдение без доказателство.

Наблюдение 44: 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 56: 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 23: while-цикълът на алгоритъма на Kahn

Всеки път, когато изпълнението е на ред 14,

① S е множеството от източниците на $G - B$ и се състои точно от сивите върхове.

② B е множеството $\{T[i] \mid 1 \leqslant i \leqslant k\}$.

③ Нито един връх от B не е дете на връх от $G - B$.

④ $A_{\overline{B}}$ съдържа полустепените на входа на $G - B$.

⑤ $T[1..k]$ съдържа топологическа сортировка на подграфа на G, индуциран от B.

База. Разглеждаме първото достигане на ред 14.

Ще покажем ①. Според Наблюдение 44, в този момент $B = \emptyset$, така че $G - B = G$ и ① става "S е множеството от източниците на G и се състои точно от сивите върхове", което е част от Наблюдение 44.

Ще покажем ②. От една страна, в този момент $B = \emptyset$. От друга страна, в този момент $k = 0$ заради ред 13, така че $T[1..k]$ е празен.

Ще покажем ③. Тъй като $B = \emptyset$, твърдението е вярно в празния смисъл.

Ще покажем ④. Щом $B = \emptyset$, то $A_{\overline{B}} = A$ и ④ става "A съдържа полустепените на входа на върховете на G", което е част от Наблюдение 44.

Ще покажем ⑤. Щом $k = 0$, то $T[1..k]$ е празен. От друга страна, подграфът на G, индуциран от B, 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 B. Тогава в момента t', x вече хем не е сив, хем не е връх на $G - B$.

Множеството от източниците в $G - B - x$ се състои точно от тези върхове, които в $G - B$ 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 $G - B - x$ от момент t е $G - B$ 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 $G - B$, понеже $x \in S$ в този момент. Това означава, че x не е дете на нито един връх от $G - B$ в момент t. След това към B се добавя един единствен връх, а именно x (ред 24). Очевидно в момент t', ③ е в сила за текущото B.

Ще покажем ④. Изпълнението на тялото на **while**-цикъла засяга $A_{\overline{B}}$ по два начина. Първо, връх x, който в момент t не е в B, "влиза" в B на ред 24; ерго, в момент t, връх x е в $A_{\overline{B}}$, 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 полустепента на входа на всички деца на x. Твърдим, че в момент t, $\forall y \in adj[x] : A[y] > 0$. 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 полустепента 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 $T[j]$, $1 \leqslant j \leqslant k - 1$. 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 топологическа сортировка на подграфа на G, индуциран от B. Но от ② знаем, че в този момент $B = \{T[i] \,|\, 1 \leqslant i \leqslant k\}$, 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 топологическа сортировка на G. □

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 мминаваме през всички списъци на съседства с контантна 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 решание на задачата 2SAT от съждителната логика (вижте Допълнение 70).

Нека навсякъде в тази секция $G = (V, E)$ е ориентиран граф. За по-голяма яснота на изложението тук няма да именуваме върховете с цели положителни числа, а с малки латински букви.

Знаем, че релацията на силна свързаност $\mathsf{SC}(G)$ е релация на еквивалентност. Множеството от нейните класове на еквивалентност означаваме с “$\mathsf{SCC}(G)$” (Нотация 6). Подграфите на G, индуцирани от елементите на $\mathsf{SCC}(G)$, са силно свързаните компоненти на G.

Забележете приликата между фактор-релация (Определение 32) и фактор-граф (Определение 67).

Определение 67: Фактор-граф

Нека $R \subseteq V(G) \times V(G)$ е релация на еквивалентност. Нека $\mathfrak{X}$ е множеството от класовете на еквивалентност на R. *Фактор-графът на* G *спрямо* R[a] е ориентираният граф G/R с множество от върхове $\mathfrak{X}$, като за всеки два различни класа $P, Q \in \mathfrak{X}$, 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 $(a, b) \in E(G)$.

[a]На 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 $\mathsf{SC}(G)$”.

Нотация 8: $G/\mathsf{SC}(G)$ е фактор-графът на G

Фактор-графът на G се означава с “$G/\mathsf{SC}(G)$”.

Очевидно $\mathsf{SCC}(G)$ е множеството от върховете на $G/\mathsf{SC}(G)$.

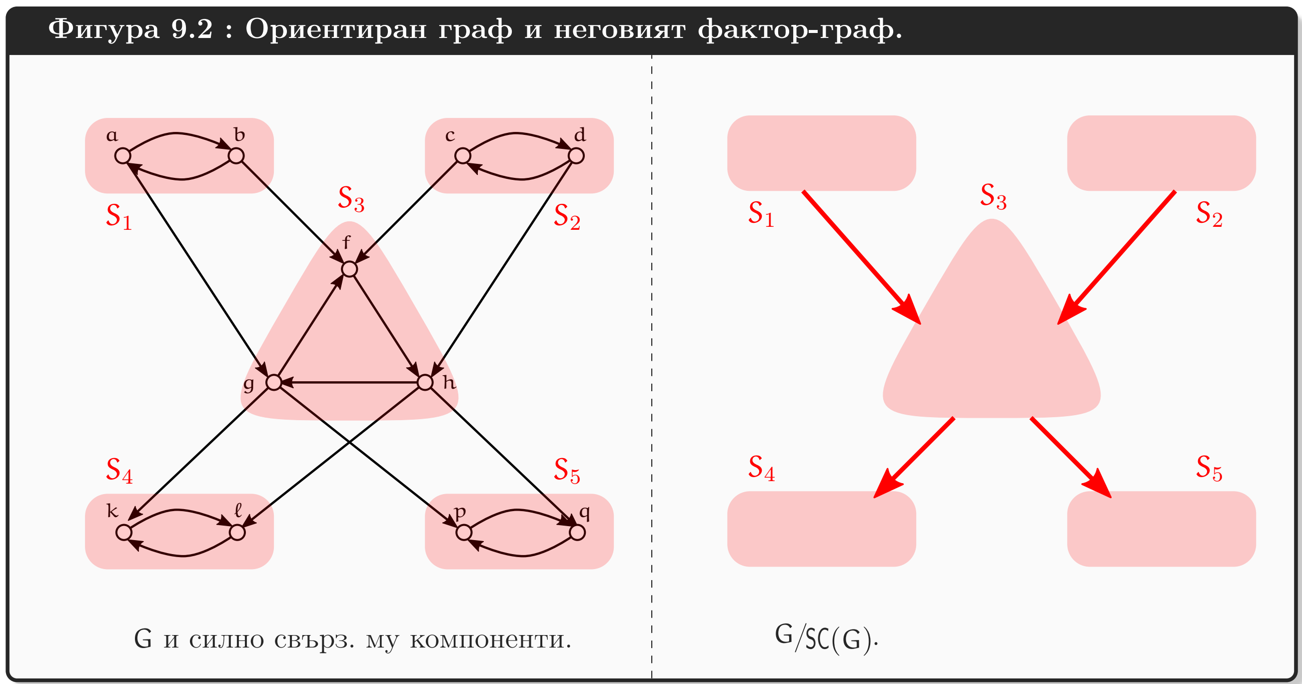
Наблюдение 45: $G/_{SCC(G)}$ е даг.

$G/_{SCC(G)}$ е даг. Ако в $G/_{SCC(G)}$ има цикъл, за всеки два различни върха X и Y от цикъла—тези върхове са различни елементи на $SCC(G)$—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, според което X и Y са различни елементи на $SCC(G)$.

Това, което нарекохме "фактор-граф", на английски се нарича *the condensed graph* или *the component graph* [31, стр. 617].

Фигура 9.2 показва ориентиран граф и неговият фактор-граф.

Фигура 9.2 : Ориентиран граф и неговият фактор-граф.



G и силно свърз. му компоненти. $G/_{SCC(G)}$.

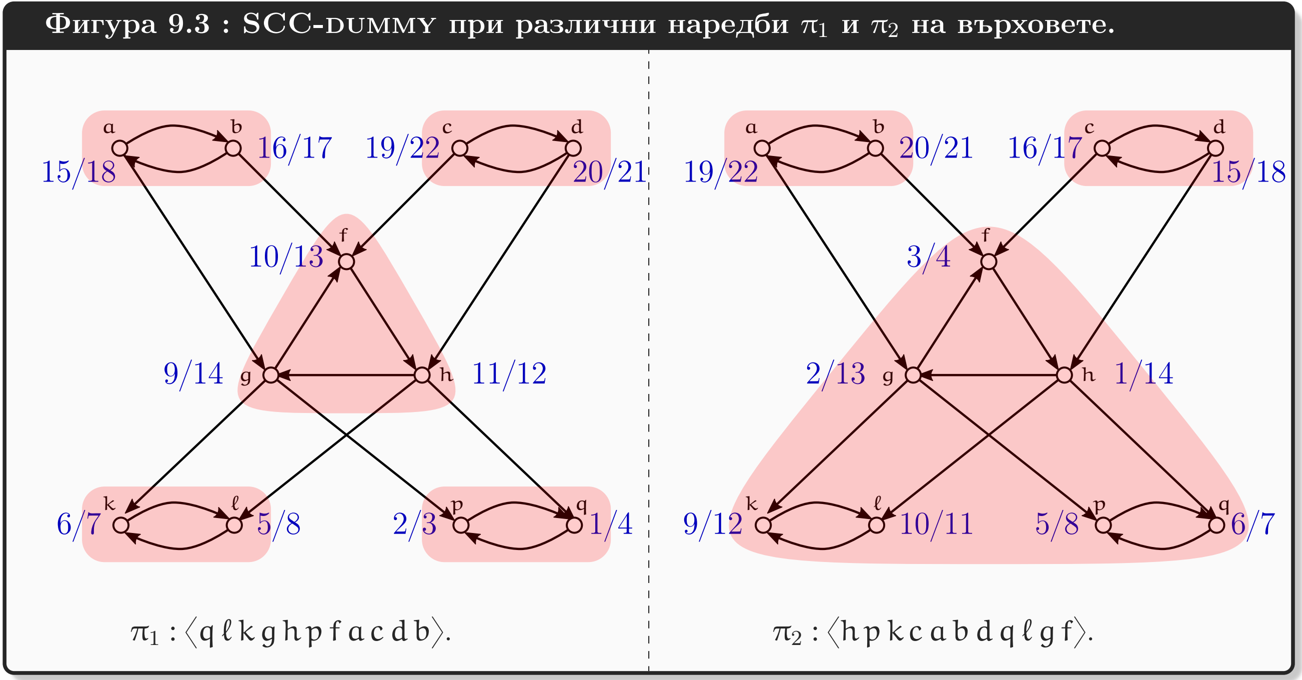
Да се опитаме да решим задачата за намиране на силно свързаните компоненти на G чрез следната малка модификация на DFS, наречена SCC-DUMMY. Както знаем, силно свързаните компоненти не са само множества от върхове, а са истински графи с върхове и ребра, но за простота генерирането на силно свързаните компоненти ще се състои само в генерирането на техните множества от върхове. Лесно се вижда, че върнатото T е точно $SCC(G)$.

```
SCC-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           SCC-DUMMY-REC(G, v)
10          T ← T ∪ {S}
11  return T
```

```
SCC-DUMMY-REC(G = (V, E); x ∈ V)
1   color[x] ← gray
2   time++
3   d[x] ← time
4   S ← S ∪ {x}
5   foreach y ∈ adj[x]
6       if color[y] = white
7           SCC-DUMMY-REC(G, y)
8   color[x] ← black
9   time++
10  f[x] ← time
```

Да си представим работата на SCC-DUMMY върху графа G от Фигура 9.2. Ключово наблюдение е, че разбиването T 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 SCC-DUMMY при две различни наредби π_1 и π_2 на върховете. π_1 и π_2 водят до различни разбивания: ако наредбата е π_1, получаваме точно $\mathsf{SCC}(G)$, но ако наредбата е π_2, получаваме разбиване на само три дяла.

Фигура 9.3 : SCC-DUMMY при различни наредби π_1 и π_2 на върховете.



$\pi_1 : \langle q\, \ell\, k\, g\, h\, p\, f\, a\, c\, d\, b \rangle$. $\pi_2 : \langle h\, p\, k\, c\, a\, b\, d\, q\, \ell\, g\, f \rangle$.

Лесно се вижда защо π_2 "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 $\{k, \ell\}$, във време 9, преди да финишира върху h във време 14. Така с едно викане на SCC-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 да съдържа върховете на точно една силно свързана компонента.

Наблюдение 46

Ако искаме след излизането от първото викане на SCC-DUMMY-REC на ред 9 множеството S на ред 10 да съдържа върховете на **само една** силно свързана компонента G_1, първата стойност на v на ред 6 трябва да е връх от G_1 и G_1 да е такава, че обхождането да не може да "излезе" от нея. π_1 от Фигура 9.3 е пример за такава наредба. Първият връх в нея, а именно q, е от силно свързана компонента, от която обхождането не може да излезе.

Определение 68: Благоприятна силно свързана компонента

Силно свързана компонента, от която обхождането не може да излезе, ще наричаме *благоприятна силно свързана компонента*.

Наблюдение 47

Благоприятните силно свързани компоненти на G са сифоните на фактор-графа $G/_{SC(G)}$. Тъй като $G/_{SC(G)}$ е даг, той има поне един сифон съгласно Теорема 54, така че G има поне една благоприятна силно свързана компонента.

За да намерим силно свързаните компоненти чрез DFS, не е достатъчно първият връх в наредбата на ред 6 да е от благоприятна силно свързана компонента G_1. След излизането от викането на ред 9 трябва следващият бял връх, за който условието на ред 7 е истина, 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 **for**-цикъла на ред 6.

Ребрата, които не принадлежат на силно свързани компоненти, са от особена важност за задачата, която решаваме. Те са тези ребра, които позволяват на обхождането да се "прехвърля" от една силно свързана компонента в друга.

Определение 69: 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 (a, g), (b, f), (c, f), (d, h), (g, k), (h, ℓ), (g, p) и (h, q).

Наблюдение 48

За да намерим силно свързаните компоненти на G чрез SCC-DUMMY, достатъчно е наредбата на върховете по отношение на ред 6 да е такава, че за всяко важно ребро (x, y), краят y да е вляво от началото x.

Става дума за наредба, която, по отношение на важните ребра, е обратна на топологическа сортировка: в топо-сортирането се иска всички ребра да са "вчесани" отляво надясно, а сега се иска важните ребра да са "вчесани" отдясно наляво.

И така достигаме до идеята, че ако алгоритъмът получи като вход "правилна" наредба на върховете, тоест, масив от върховете $L[1..n]$, подреден според Наблюдение 48, той ще работи коректно.

```
SCC(G = (V, E); L[1..n]: масив от върховете)
1   S и time са глобални
2   T ← ∅
3   for i ← 1 to n
4       color[i] ← white
5   time ← 0
6   for i ← 1 to n
7       if color[L[i]] = white
8           S ← ∅
9           SCC-REC(G, L[i])
10          T ← T ∪ {S}
11  T ← T ∪ {S}
```

```
SCC-REC(G = (V, E); x ∈ V)
1   color[x] ← gray
2   time++
3   d[x] ← time
4   S ← S ∪ {x}
5   foreach y ∈ adj[x]
6       if color[y] = white
7           SCC-REC(G, y)
8   color[x] ← black
9   time++
10  f[x] ← time
```

Като пример да вземем пак Фигура 9.3. Ако $L = [q, \ell, k, g, h, p, f, a, c, d, b]$ (подфигурата вляво), алгоритъмът ще намери силно свързаните компоненти.

Ефикасен алгоритъм за намиране на силно свързаните компоненти на ориентиран граф. Следният алгоритъм решава ефикасно задачата за намиране на силно свързаните компоненти. Открит е от Rao Kosaraju през 1978 [2]. Необходимо ни е следното определение.

Определение 70: Транспониран граф

Транспонираният граф на G е графът $G^T = (V(G), \{(u, v) \mid (v, u) \in E(G)\})$.

На английски терминът е *the transpose of* G ([31, 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 алгоритъмът за Tarjan на стр. 391 за топологическо сортиране. 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 подрежда върховете по намаляващи стойности на време на финализиране, което може да бъде направено за всеки ориентиран граф, върху който е пуснат DFS. Разбира се, ако графът не е даг, то изходът няма смисъл на топологическо сортиране на графа, защото такова не съществува, но това е друго нещо.

```
KOSARAJU(Ориентиран граф G = (V, E))
1   L[1..n] ← TSORT(G)
2   намери G^T
3   SCC(G^T, L)
```

Коректност. Аргументацията за коректност ползва тези факти, които са прекалено очевидни, за да бъдат наречени “теорема”.

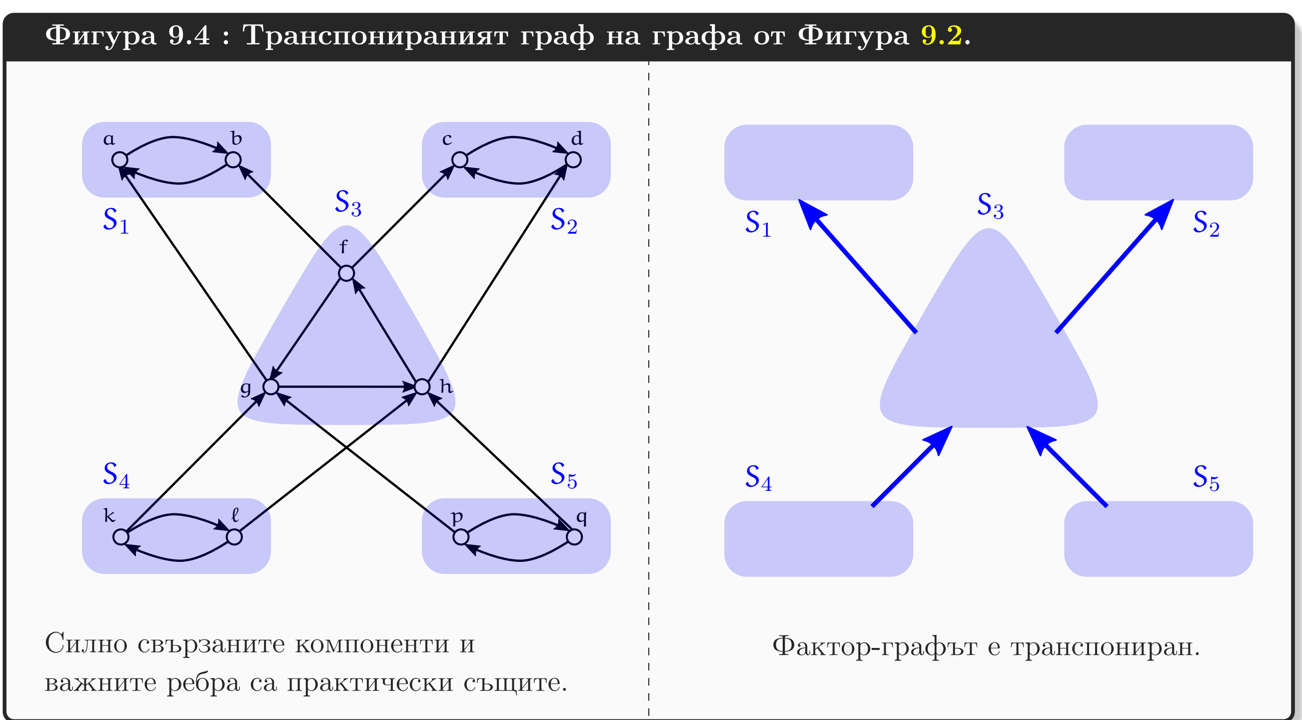
Наблюдение 49

$\mathsf{SCC}(G)$ и $\mathsf{SCC}(G^T)$ съвпадат. Нещо повече: $\left(G/\mathsf{SC}(G)\right)^T = G^T/\mathsf{SC}(G^T)$. Оттук следва, че

- източниците в $G/\mathsf{SC}(G)$ са точно сифоните в $G^T/\mathsf{SC}(G^T)$ и обратното, сифоните в $G/\mathsf{SC}(G)$ са точно източниците в $G^T/\mathsf{SC}(G^T)$.
- важните ребра в G са същите като в G^T, но с обратна ориентация.

Фигура 9.4 илюстрира това наблюдение. 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 важните ребра в G и в G^T са същите, но с обратни посоки; примерно, (a, g) е важно ребро в G, а (g, a) е важно ребро в G^T, и така нататък.

Фигура 9.4 : Транспонираният граф на графа от Фигура 9.2.



Силно свързаните компоненти и важните ребра са практически същите.

Фактор-графът е транспониран.

Определение 71: argmin и argmax

Да разгледаме произволна функция $f : X \to \mathbb{R}$, където X е произволен домейн. Нека f има минимум върху X. Тогава

$$\underset{x \in X}{\operatorname{argmin}} f(x) \stackrel{\text{def}}{=} \{x \in X \,|\, (\forall y \in X : f(x) \leqslant f(y))\} \tag{9.1}$$

"argmax" се дефинира аналогично.

Имената идват от "*argument minimum*" и "*argument maximum*". Кратко обяснение за смисъла на тези понятия. Изразът

$$\min \{f(x) \,|\, x \in X\} \tag{9.2}$$

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 мининуми върху X, има смисъл да говорим за множеството от елементите на X, върху които f има минимум. Ако това множество е едноелементно, то "argmin" означава неговия (единствен) елемент; аналогичното е в сила и за "argmax".

Определение 72: Индуцирана наредба на силно свързаните компоненти

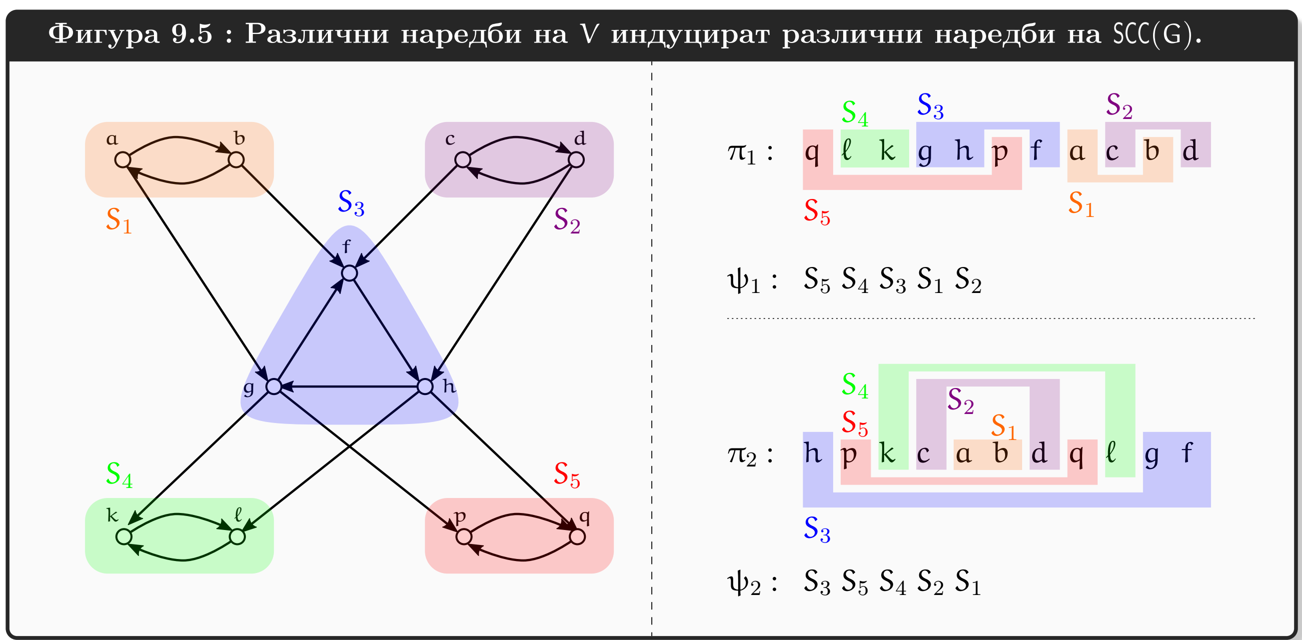
Нека е дадена наредба π на върховете на G. Говорим за биекция $\pi : V \rightarrow \{1, \ldots, n\}$. Нека $SCC(G) = \{S_1, \ldots, S_t\}$. Нека $\phi : \{1, \ldots, t\} \rightarrow \{1, \ldots, n\}$ е функцията, която връща минималния индекс в π на връх от S_i. Формално,

$$\forall i \in \{1, \ldots, t\} : \phi(S_i) = \operatorname*{argmin}_{w \in S_i} \pi(w)$$

Нека $\psi = \langle S_{j_1}, \ldots, S_{j_t} \rangle$ е уникалната наредба на $S_1, \ldots, S_t$, за която $\phi(S_{j_1}) < \phi(S_{j_2}) < \cdots < \phi(S_{j_t})$. Казваме, че ψ *е наредбата на силно свързаните компоненти, индуцирана от* π. Или просто ψ *е наредбата, индуцирана от* π, ако е ясно, че става дума за силно свързаните компоненти.

Като пример да вземем графа G от Фигура 9.2. Той има единадесет върха и пет силно свързани компоненти. Фигура 9.5 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 ψ_1 и ψ_2, на силно свързаните му компоненти.

Фигура 9.5 : Различни наредби на V индуцират различни наредби на $SCC(G)$.



Лема 48: Алгоритъмът на Tarjan и важните ребра

Нека $G = (V, E)$ е произволен ориентиран граф. Нека TSORT е алгоритъмът на Tarjan за топологическо сортиране. Нека TSORT(G) връща $L[1..n]$. Да мислим за L като за биекция с домейн $\{1, \ldots, n\}$ и кодомейн V. За всяко важно ребро (x, y) на G е вярно, че $L^{-1}(x) < L^{-1}(y)$.

Доказателство Лемата казва нещо просто: важните ребра са "вчесани" отляво надясно, точно както е при топологическо сортиране. **Не се твърди**, че всички ребра на графа са

така! Може G да е цикличен и тогава няма как всички негови ребра да са отляво надясно по отношение на L. Твърдение е само за важните ребра.

Да допуснем противното: съществува важно ребро $(x, y) \in E$, такова че $L^{-1}(x) > L^{-1}(y)$:

Както знаем от Секция 9.2, TSORT връща масив от върховете, сортирани по намаляващи времена на финализиране. Тогава $f[x] < f[y]$. Да си припомним Лема 47: съществуват точно три възможности за $d[x]$, $d[y]$, $f[x]$ и $f[y]$, и само в една от тях $f[x]$ е по-малко от $f[y]$. А именно,

$$d[y] < d[x] < f[x] < f[y]$$

Но това се случва само ако (x, y) е ребро назад, така че в момента, в който връх x е открит, връх y вече е бил открит. Следователно в G съществува път от y до x.

Щом има път от y до x и има ребро (x, y), върховете x и y са в една и съща силно свързана компонента. Тогава (x, y) не е важно ребро. □

Следствие 21

TSORT(G) връща масив, който индуцира наредба на силно свързаните компоненти на G, която е топо-сортировка на $\mathsf{SCC}(G)$.

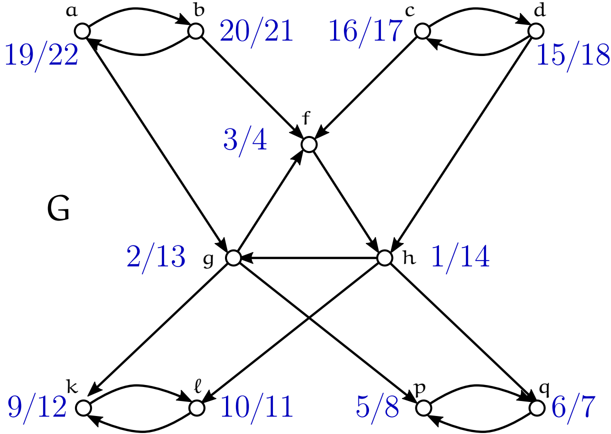
В някакъв смисъл може да кажем, че TSORT(G) топо-сортира фактор-графа на G. В частния случай, когато G е даг, TSORT(G) топо-сортира G, тъй като G практически съвпада с фактор-графа си. Ерго, Следствие 21 влече коректността на алгоритъма на Tarjan.

Теорема 57: Коректността на алгоритъма на Kosaraju

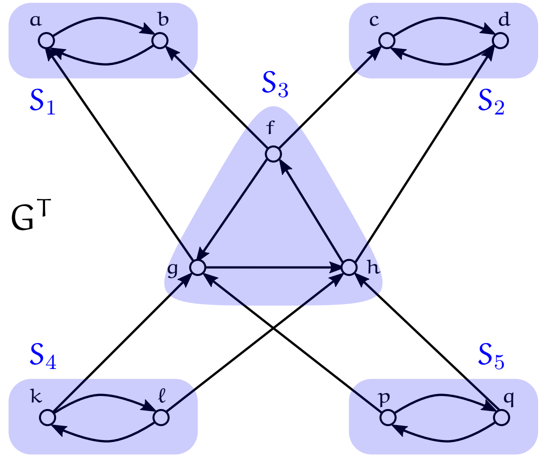
Алгоритъмът на Kosaraju намира свързаните компоненти на входния граф G.

Доказателство Съгласно Лема 48, на ред 1 върнатият масив L е такъв, че всички важни ребра на G са отляво надясно по отношение на наредбата на върховете, която той реализира. Тогава всички важни ребра на G^T са отдясно наляво по отношение на наредбата на върховете, която L реализира. Прилагаме Наблюдение 48 и заключаваме, че викането SCC(G^T, L) на ред 3 генерира силно свързаните компоненти на G. □

Ето пример за работата на алгоритъма на Kosaraju. Пак разглеждаме графа от Фигура 9.2. Алгоритъмът на Kosaraju първо изпълнява алгоритъма на Tarjan, който по същество е DFS. Да кажем, че този DFS работи така:



Тогава алгоритъмът на Tarjan връща $L = [a, b, d, c, h, g, k, \ell, p, q, f]$. Транспонираният G^T е



Очевидно L индуцира наредбата на силно свърваните компоненти $\psi_1 = \langle S_1, S_2, S_3, S_4, S_5 \rangle$. Изпълнението на $\text{SCC}(G^T, L)$ връща множествата от техните върхове в този ред.

Може да възникне следният въпрос: наместо на ред 1 да генерираме пермутацията на върховете в намаляващ ред по времената на финализиране и после да генерираме G^T на ред 2, не може ли на ред 1 да генерираме пермутацията на върховете в **нарастващ ред** по времената на финализиране и после да викнем SCC върху **оригиналния** граф (а не върху неговия транспониран) с въпросната пермутация по нарастване на времената на финализиране? Това не би подобрило асимптотиката на алгоритъма, но определено би го опростило.

Отговорът е, че не може. Наистина има примери за ориентирани графи, върху които можем да намерим силно свързаните компоненти, спестявайки си усилието по генериране на транспонирания граф по описания начин. Но примерът, който току-що видяхме, е контрапример за тази идея. Да видим защо. Пермутацията-инверсия на L е $\overline{L} = [f, q, p, \ell, k, g, h, c, d, b, a]$. Тази пермутация индуцира наредбата на силно свързаните компоненти $\psi_2 = \langle S_3, S_5, S_4, S_2, S_1 \rangle$. Забележете, че ψ_2 **не е** инверсията на ψ_1! ψ_2 започва със S_3, която не е сифон на фактор-графа на G. Ерго, $\text{SCC}(G, \overline{L})$, 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. Което е некоректно. И така, в общия случай не може да минем без генерирането на G^T.

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 лиейна 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^T (това има смисъл на слагане на ребро (v, u) в G^T). И накрая, работата на SCC 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 73: 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.

Определение 74: Срязващ връх и мост, алтернативно определение

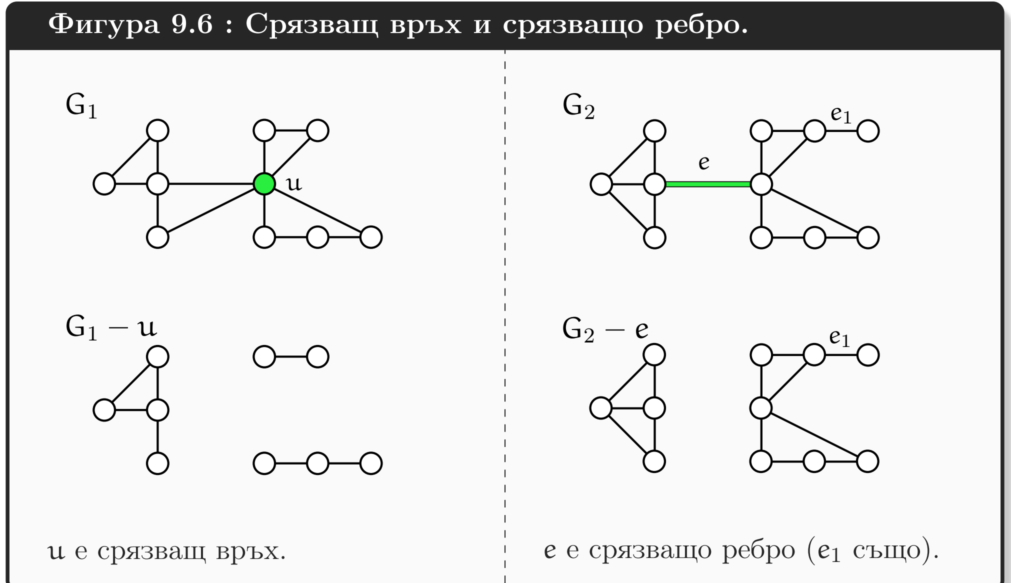
Срязващ връх в G е всеки $u \in V$, такъв че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа u като вътрешен връх. Мост е всяко $e \in E$, такова че за някои различни върхове $v, w \in V$ е вярно, че всеки път между v и w съдържа e.

Лема 49

Определения 73 и 74 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, или с други думи, 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 73 и 74. В графа G_1, връх u е срязващ. В графа G_2, и 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.

e е срязващо ребро (e_1 също).

Приложение на срязващи върхове и ребра. 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 си "говорят". В разпаднатата мрежа има компютри, които не могат да комуникират, защото между тях няма маршрут. А срязващо ребро отговаря на жична връзка, чиято повреда би довела до разпадане на мрежата на две подмрежи, които не си "говорят".

И така, срязващите върхове и мостовете отговарят на някакъв 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.

Определение 75: Разцепване на срязващи върхове

Нека $u \in V$ е срязващ връх. Нека $G - u$ има k свързани компоненти, наречени $G_1, \ldots, G_k$. Нека $z_1, \ldots, z_k$ са k нови върхове – такива, които не са от V. Нека, за $1 \leqslant i \leqslant k$,

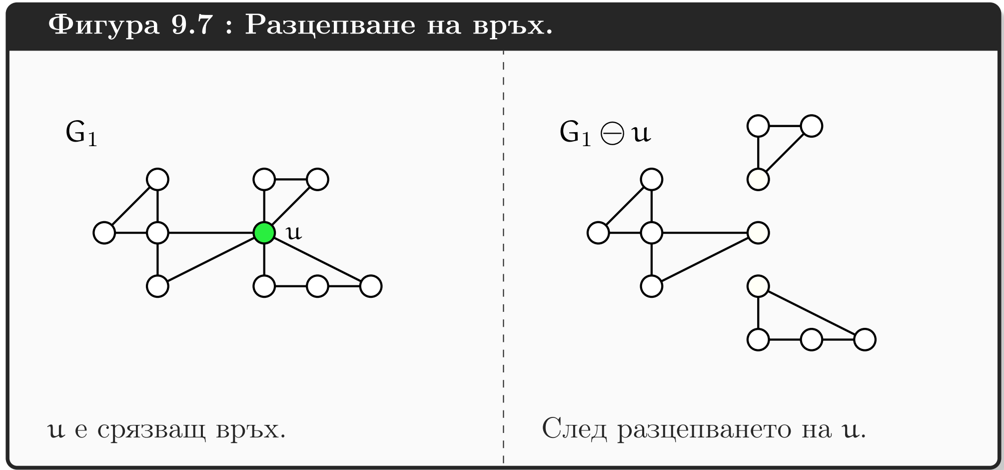
$$H_i = (V(G_i) \cup \{z_i\}, E(G_i) \cup \{(z_i, x) \mid x \in V(G_i) \cap N(u)\})$$

Да *разцепим* u означава да преобразуваме G в колекцията $H_1, \ldots, H_k$. Графът–резултат означаваме с $G \ominus u$.

Очевидно $H_1, \ldots, H_k$ 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 $u_1, \ldots, u_t$ са различни срязващи върхове в G, пишем $G \ominus u_1 \ominus \cdots \ominus u_t$. Резултатът от разцепването на всички срязващи върхове е предмет на следващата дефиниция.

Определение 76: 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 77: Блокове в граф, алтернативна дефиниция

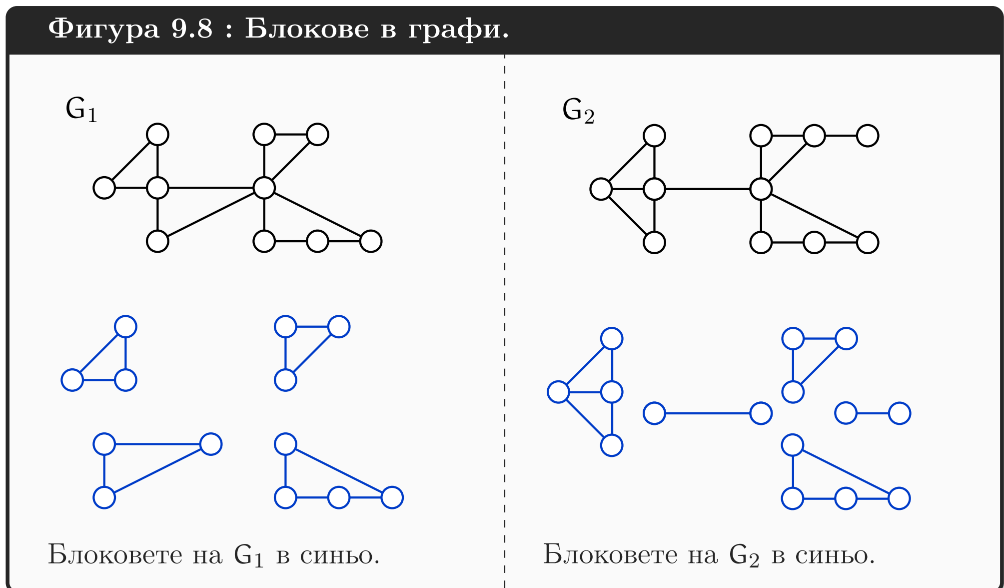
Нека $G = (V, E)$ е граф. *Блоковете на* G са тези свързани компоненти, които се получават от разцепването на всички срязващи върхове.

Наблюдение 50

Определения 76 и 77 са еквивалентни.

Фигура 9.8 илюстрира блоковете на два графа G_1 и G_2. Тя илюстрира и "разцепване на срязващи върхове".

Фигура 9.8 : Блокове в графи.



Блоковете на G_1 в синьо. Блоковете на G_2 в синьо.

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.

Това е друго важно 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 73, 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 неефикасен алгоритъм, дори да можем да трием и връщаме върхове в $\Theta(1)$: сложността му в най-лошия случай би била $\Theta(n(n+m))$.

Ефикасният алгоритъм, който ще разгледаме сега, е базиран на [133, Section 5.9.2, pp. 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 $V(T) = V$ and $E(T) = \{e \in E \,|\, e$ е класифицирано като дървесно ребро от DFS$\}$. Разглеждаме три категории върхове на **графа** спрямо **дървото** и необходимите и достатъчни условия за това, връх от дадена категория да е срязващ.

Лема 50: Листата на дървото

В текущия контекст, нито едно листо не може да е срязващ връх. Ако v е листо на T, очевидно $G - v$ има точно една свързана компонента.

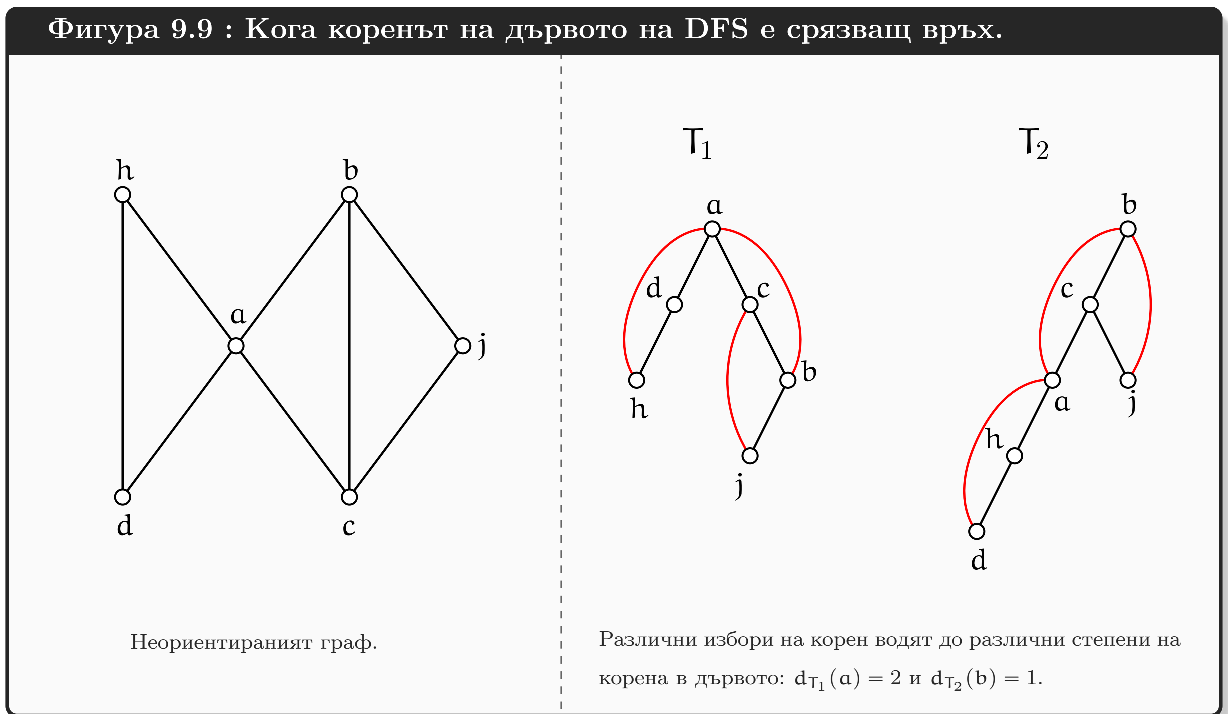
Лема 51: Коренът на дървото

В текущия контекст, коренът е срязващ връх в G тстк степента му **в дървото** е поне 2.

Доказателство: Твърдението е напълно очевидно. Фигура 9.9 е илюстрация. 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 2, което точно съответства на факта, че в графа връх a е срязващ. Ако b е коренът, то той има степен в дървото 1, 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 $d_{T_2}(b) = 1$.

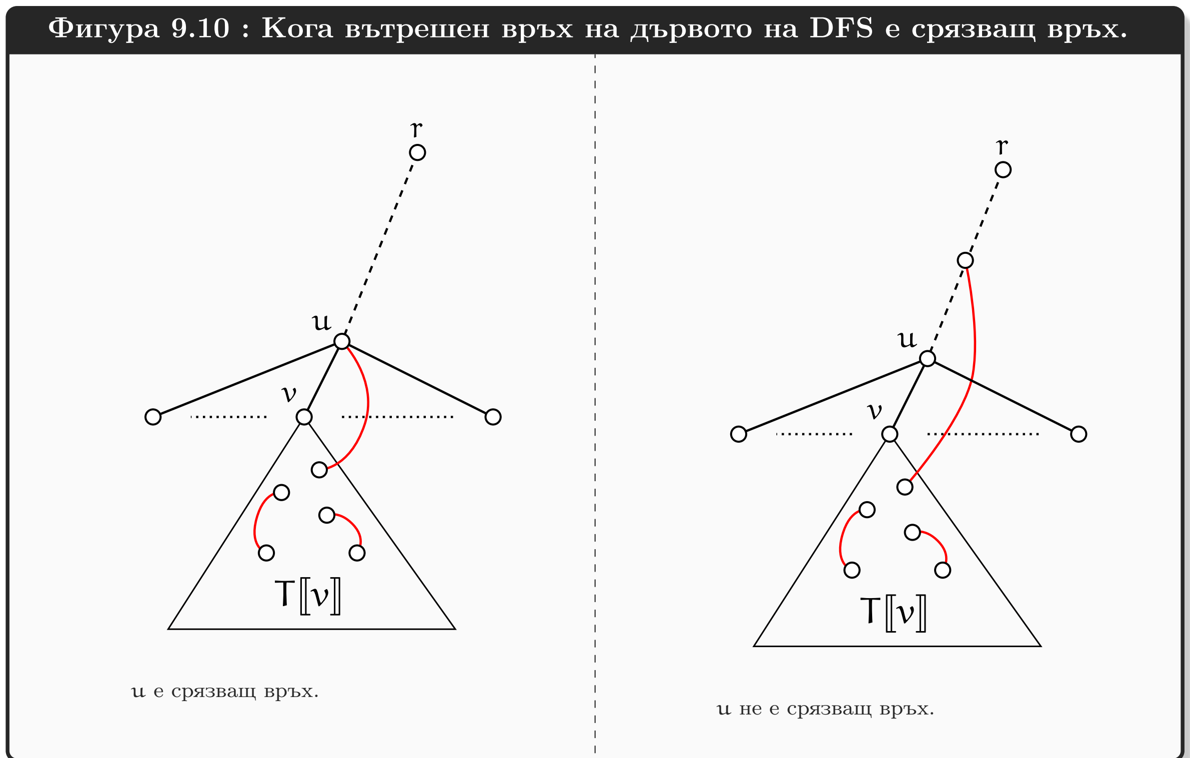
Лема 52: Останалите върхове на дървото

В текущия контекст, всеки връх u, който не е листо или коренът, е срязващ връх тстк има дете v на u в T, такова че няма нито едно ребро назад между връх на $T[\![v]\!]$ и предшественик на u в дървото.

Доказателство: Да допуснем, че има дете v на u (в дървото), такова че няма ребро назад между никой връх на $T[\![v]\!]$ и 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 дървото) има 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. □

Фигура 9.10 илюстрира Лема 52.



Фигура 9.10 : Кога вътрешен връх на дървото на DFS е срязващ връх.

Следният алгоритъм 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 дървото и в графа не са непременно същите, като за всеки $v \in V$ е вярно, че $\text{dist}_G(u,v) \leqslant \text{dist}_T(u,v)$. Това е така, защото T е подграф на G.

Алгоритъмът поддържа масив от нива́та $\text{level}[1..n]$, 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 46). Пак подчертаваме, че **не става дума за разстояния в** G; знаем, че DFS в общия случай не изчислява разстояния в графа. Очевидно, $\text{level}[u] = 0$.

```
Find Cut Vertices(G = (V, E): неориентиран свързан граф)
1   level[1 .. n] е глобален
2   foreach a ∈ V
3       color[a] ← white
4   нека a е произволен връх от V
5   FCV-rec(G, a, 0);
```

```
FCV-rec(G = (V, E), x ∈ V, ℓ ∈ ℕ)
1   color[x] ← gray
2   level[x] ← ℓ
3   minback ← level[x]
4   if level[x] = 0
5       isroot ← True
6   else
7       isroot ← False
8   count ← 0
```

```
 9   iscutvertex ← FALSE
10   foreach y ∈ adj[x]
11       if color[y] = white {
12           count++
13           b ← FCV-REC(G, y, ℓ + 1)
14           if  b ⩾ level[x]  and ( not isroot )
15               iscutvertex ←  TRUE
16           minback ←  min{minback, b} }
17       if color[y] = gray {
18           if level[y] < minback and level[y] ≠ level[x] − 1
19               minback ←  level[y] }
20   if  iscutvertex  or ( isroot and count ⩾ 2 )
21       print x
22   color[x] ←  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 $V(T[\![x]\!])$.

Лема 50, Лема 51 и Лема 52 са достатъчна обосновка за коректността, тъй като рекурсивната функция FCV-REC имплементира техните необходими и достатъчно условия. Функцията FCV-REC прави следното.

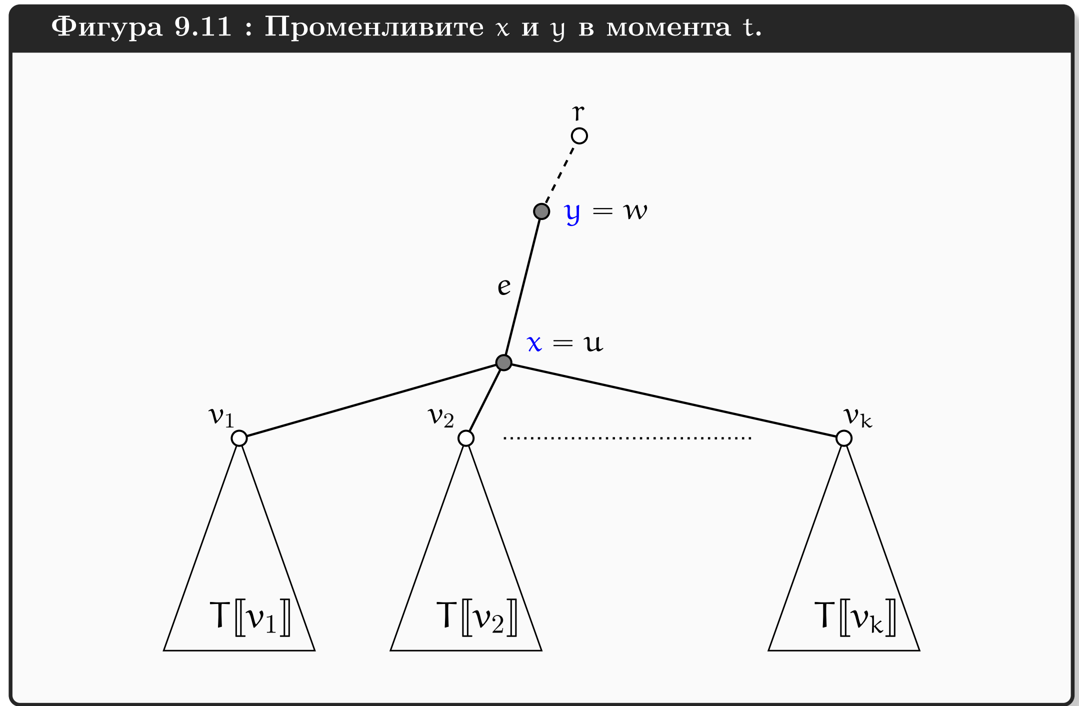
- Очевидно не категоризира никое листо на T като срязващ връх.
- Категоризира върхът-корен на T като срязващ връх тстк той има поне две деца в T. Това също е очевидно от редове 4, 5, 8 и 12 и второто условие в дизюнкцията на ред 20.
- За всеки друг вид връх x, FCV-REC сравнява на ред 14 следните числа
  - 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 level$[x]$, то връх x е срязващ съгласно Лема 52.

  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, Лема 52 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 $v_1, \ldots, v_k$ в T, за някое $k \geqslant 1$. 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 : Променливите x и y в момента t.



И $u = x$, и $w = y$ са нарисувани като сиви върхове, а $v_1, \ldots, v_k$ 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 $v_1, \ldots, v_k$ 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 $y = v$; в този предишен момент w е бил сив, а v е бил бял, така че за DFS това ребро е дървесно. Второто прекосяване не може да характеризира реброто по друг начин.

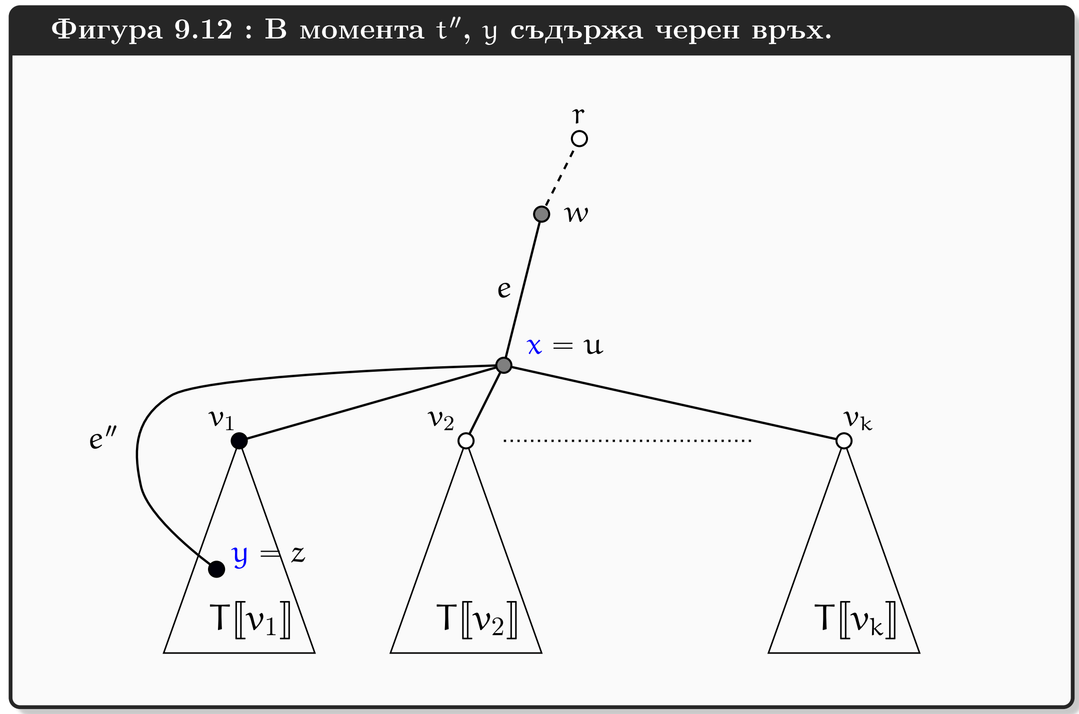
Това си заслужава да се повтори: реброто $e = (w, v)$ е неориентирано и бива представено в списъците на съседство както чрез появата на v в списъка на w, така и чрез появата на w в списъка на v. Но e е едно единствено ребро, понеже графът е неориентиран. Това, че в списъците му съответстват два елемента, е особеност на списъчното представяне на неориентираните графи. При работата на DFS, която разглеждаме, e бива прекосено първоначално от w (родителят на v в дървото) към v. 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 $\text{level}[y] \neq \text{level}[x] - 1$ 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 графи 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.

Фигура 9.12 : 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 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.

[†]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 класифицира e'' по друг начин. Всичко това съответства на казаното на на стр. 383.

Лема 53

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. □

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 ∈ V
3       color[u] ← white
4   u е произволен връх от V
5   FBR-REC(G, u, 0)
```

```
FBR-REC(G = (V, E), x ∈ V, ℓ ∈ ℕ)
1   color[x] ← gray
2   level[x] ← ℓ
3   minback ← level[x]
4   foreach y ∈ adj[x]
5       if color[y] = white {
6           b ← FBR-REC(G, y, ℓ + 1)
7           if b > level[x]
8               print (x, y)
9           else
10              minback ← min {minback, b}
11      if color[y] = gray {
12          if level[y] < minback and level[y] ≠ level[x] − 1
13              minback ← level[y] }
14  color[x] ← BLACK
15  return minback
```

Коректност и сложност. Доказателството за коректност почива на Лема 53. 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 Лема 53:

- 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 съседи 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 78: Тегловен граф

Тегловен граф е наредена двойка (G, w) от граф $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ и *тегловна функция* $w : \mathsf{E} \to \mathbb{R}$.

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 $w : \mathsf{E} \to \{1, 2\}$. 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: $\circ\!\xrightarrow{\;2\;}\!\circ$ с последователност от две ребра с тегла единици: $\circ\!\xrightarrow{\;1\;}\!\circ\!\xrightarrow{\;1\;}\!\circ$. Но този подход за свеждане на задачи върху тегловни графи към задачи върху не-тегловни графи не се мащабира. Ако ограничението за теглата е $w : \mathsf{E} \to \{1, \ldots, 1\,000\,000\}$, 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 79: 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 51

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 именувани дървета е най-много n^{n-2} и тази граница е точна.

Определение 80: 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 $w(T') \leqslant w(T'')$.

Формално, задачата МПД, на английски MST, се дефинира така.

Изч. Задача 22: 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 само теглото на минимално покриващо дърво без самото дърво.

Следното определение е буквален превод на "safe edge" от [31, стр. 626].

Определение 81: Сигурно ребро.

Нека G е тегловен граф и $A \subseteq E$ е такова, че съществува МПД T, такова че $A \subseteq E(T)$. 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 идея на алгоритмите за конструиране на МПД се съдържа в следния псевдокод ([31, стр. 626]). МПД се определя напълно от своите ребра, така че да конструираме МПД е същото като да конструираме множеството от ребрата му.

Общ вид на МПД(G: граф, w: тегловна функция)
1 $A \leftarrow \varnothing$
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.

Цялата трудност тогава е, как да намираме сигурно ребро по отношение на множество от ребра. Теорема 58 дава достатъчна обосновка за намирането на сигурни ребра от алгоритмите, които ще разгледаме. Първо ни трябват няколко определения.

Определение 82: 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.

Теорема 58: Theorem 23.1 от [31, стр. 627] (МПД теоремата)

Нека $G = (V, E)$ е свързан тегловен граф с тегловна функция $w : E \to \mathbb{R}$. Нека $A \subseteq E$ е такова, 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: Ако $e \in E(T)$, то ние сме готови с доказателството. Да допуснем, че $e \notin E(T)$. Ще покажем, че има МПД T', такова че $A \cup \{e\} \subseteq E(T')$.

Добавяме e към T и получаваме уницикличен граф U. Нещо повече, със сигурност e е ребро от цикъла c на U. За реброто e знаем, че прекосява среза. Със сигурност съществува ребро e' в c, различно от e, което прекосява среза. 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 $w(e) \leqslant w(e')$, следователно $w(e) - w(e') \leqslant 0$. Тогава $w(T') \leqslant w(T)$. Но T е МПД, така че $w(T) \leqslant w(T')$ по дефиниция. Тогава $w(T) = w(T')$ и T' също е МПД. Което съдържа както ребрата от A, така и e. □

10.2 Алчни алгоритми

Предложената обща идея за намиране на МПД на предишната страница заедно с Теорема 58 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 58 гарантира, че **редицата от локално оптимални избори** (на леки ребра) води до **глобално оптимално решение** (МПД).

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 МПД, локалната алчност е печеливша стратегия. Има обаче задачи, в които проявата на локална алчност води до решение, което не е оптимално. Всички **NP**-трудни задачи са такива. Като пример, да разгледаме задачата KNAPSACK, или задачата за раницата (Подсекция 12.6.3). Не-оптималността на очевидното алчно решение за нея има на стр. 539.

Подробно теоретично изложение на фундамента на алчните алгоритми има в Секция 16.2 на [31, стр. 423] и Секция 16.4 на [31, стр. 437], както и [115, 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.

Допълнение 40: Една транспортна задача с алчно решение

Нека C_1, C_2, ..., C_n са n града, 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 $1 \leqslant i < n$.

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 $\max\{\ell - d_i, 0\}$.

Всеки C_i има нужда от $m_i \in \mathbb{Z}^+$ литра вода. Казваме, че C_i е *удовлетворен*, ако $m_i \leqslant x_i$; 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 23: Транспортиране със загуби

екземпляр: $d_1 > 0, \ldots, d_{n-1} > 0$: разстояния, $x_1, \ldots, x_n$: начални наличности вода, $m_1 > 0, \ldots, m_n > 0$: 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?

За всяко $i \in \{1, , \ldots, , n\}$, нека $\mathcal{A}_i$ бъде редицата от градове $\langle C_1, C_2, \ldots, C_i \rangle$. Ще казваме, че редицата $\mathcal{A}_i$ 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*.

Нека $q \in \mathbb{Z}^+$. За всяко $i \in \{1, \ldots, n\}$, 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.
- $\mathcal{A}_i$ е *изолирана*, ако $\mathcal{A}_i$ е добра, но ако количеството вода в C_i бъде намалено с $\geqslant d_i$ литра, то $\mathcal{A}_i$ става искаща.

Забележете следните неща.

1. Всяка $\mathcal{A}_i$ е или q-излишна за точно едно $q \in \mathbb{Z}^+$, или критична, или q-дефицитна за точно едно $q \in \mathbb{Z}^+$.
2. Ако $i < n$, от изолирана $\mathcal{A}_i$ 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 примера.

Следният алчен алгоритъм решава ТРАНСПОРТИРАНЕ СЪС ЗАГУБИ.

```
TRANSPORT(d_1, ..., d_{n-1}, x_1, ..., x_n, m_1, ..., m_n)
 1  d_n ← 0
 2  s ← 0
 3  for i ← 1 to n
 4      Δ ← x_i − m_i
 5      if s + Δ ⩾ 0
 6          s ← max {s + Δ − d_i, 0}
 7      else
 8          s ← s + Δ − d_i
 9  if s ⩾ 0
10      return TRUE
11  else
12      return FALSE
```

Теорема 59: Коректността на TRANSPORT

Алгоритъм TRANSPORT връща TRUE тстк $\mathcal{A}_n$ е добра.

Доказателство: Следното твърдение е инвариант за **for**-цикъла (редове 3–8).

Инвариант 24: 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 приосвояването 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 пъта, 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: $s_{old} + \Delta \geqslant 0$. 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. $s_{old} + \Delta \geqslant 0$.

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: Да допуснем, че $s_{old} + \Delta - d_i \leqslant 0$. Тогава $s_{new} = 0$. 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 $s_{old} + \Delta \geqslant 0$. Тоест, $\Delta \geqslant 0$. 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 $\Delta - d_i \leqslant 0$. Тогава $s_{new} = 0$. Ясно е, че $\mathcal{A}_i$ е добра, щом $\mathcal{A}_{i-1}$ е добра и $\Delta \geqslant 0$, но нито един литър не може да се транспортира от C_i към C_{i+1}, без 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 $\mathcal{A}_i$ is $|\Delta|$-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, $|s_{old}| > d_{i-1}$.
Доказахме, 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 $s_{old} + \Delta \geqslant 0$. 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 $s_{old} + \Delta - d_i \leqslant 0$. Тогава $s_{new} = 0$.
Ще покажем, че $\mathcal{A}_i$ е изолирана. Количеството вода в C_i е Δ. Но $\Delta \leqslant -s_{old} + d_i$, 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 $\Delta = x_i - m_i$. Заключаваме, 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

$$\forall x, y \in \mathbb{R}, \text{ такива че } x < 0 \text{ и } x + y < 0 : y - |x| + |x + y| = 0$$

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 е 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. □

Сложността по време на TRANSPORT е очевидно $\Theta(n)$.

10.3 Алгоритъм на Prim

Алгоритъмът на Prim за намиране на МПД далечно прилича на обхождане. Той започва от един стартов връх s, който е част от входа, намира най-леко ребро, свързващо s с друг връх, слага го в дървото, после намира най-леко ребро, свързващо някой от тези два върха с друг връх, слага го в дървото, и така нататък. В терминологията на ОБЩ ВИД НА МПД на стр. 418, в алгоритъма на Prim, ребрата от A във всеки момент образуват едно (свързано) 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 58. 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,

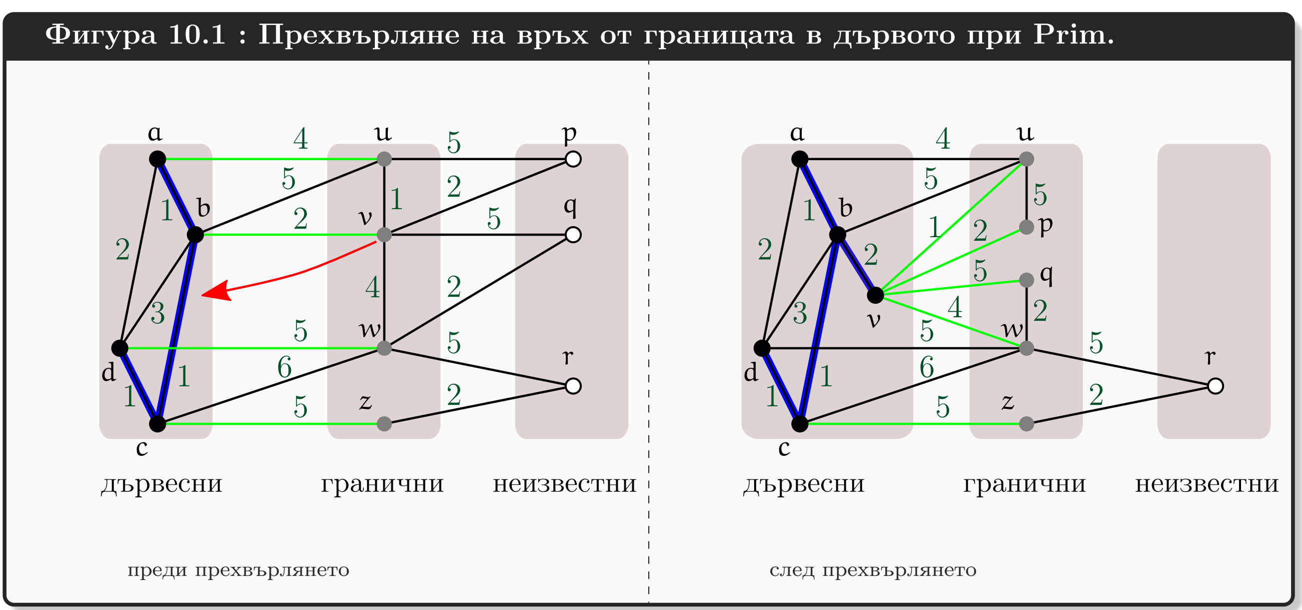
- върховете от $V(T)$ са *дървесните върхове* (на английски, *tree vertices*),
- върховете от $N(V(T))$ са *граничните върхове* (на английски, *fringe vertices*),
- върховете от $V \setminus N[V(T)]$ са *неизвестните върхове* (на английски, *unseen vertices*).

Тази класификация на практика дублира класификацията от обхождането на графи на върховете на бели, сиви и черни на стр. 367, 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 [31], а на tree, fringe и unseen е например от [133, 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 y. 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 $\Theta(n^2)$.

Сега се фокусираме върху това, как да работим във време $\Theta(1)$ за всеки 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 *гранично 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 инцидентен с дървесни, гранични и неизвестни върхове. След избора на ред 4 прехвърляме връх от границата в дървото, като обаче това може да наложи промени в някои кандидат-ребра и гранични тегла, така че на следващата итерация, стойностите на всички кандидат-ребра и гранични тегла да са коректни и отново да можем да направим избора, извършвайки $\Theta(1)$ работа за всеки граничен връх. 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 $\text{status}[1..n]$, където $\text{status}[i] \in \{\text{дървесен}, \text{граничен}, \text{неизвестен}\}$. Този масив е аналог на масива color от обхожданията.

МПД PRIM1($G = (V, E)$: неор. свързан граф, w: тегл. ф-я върху G, s: стартов връх)

1 $V(T) \leftarrow \{s\}$
2 $A \leftarrow \varnothing$
3 **foreach** $u \in V$
4 $\quad$ $\text{status}[u] \leftarrow$ неизвестен

```
5   status[s] ← дървесен
6   foreach y ∈ adj[s]
7       status[y] ← граничен
8   x ← s
9   while V\V(T) ≠ ∅ do
10      (* x е последният сложен в дървото връх *)
11      foreach y ∈ adj[x]
12          if status[y] = граничен
13              ако w((x, y)) е по-малко от граничното тегло на y,
                    смени кандидат-реброто на y с (x, y)
14          if status[y] = неизвестен
15              status[y] ← граничен
16              кандидат-реброто на y става (x, y)
17      намери кандидат-ребро e с минимално тегло
18      x ← граничният връх на e
19      status[x] ← дървесен
20      V(T) ← V(T) ∪ {x}
21      A ← A ∪ {e}
22  return A
```

Коректност и сложност. Подробно доказателство за коректност няма да правим. Инвариантът е, при всяко достигане на управлението на ред 9, ребрата от A задават МПД на подграфа на G, индуциран от $V(T)$. Използвайки Теорема 58, лесно показваме, че реброто e на ред 17 е сигурно за A.

Сложността по време в най-лошия случай е $\Theta(n^2)$.

- Първо, всички намирания на кандидат-ребра, по време на цялата работа на алгоритъма, става във време $\Theta(n^2)$. Ето защо. Най-лошият случай е графът да е пълен. Тогава unseen 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 на дървесни и гранични. Тогава на всяка итерация на **while**-а се разглеждат $|V \backslash V(T)|$ (гранични) върхове, за да се намери кандидат-ребро с минимално тегло на ред 17. Тогава следната сума дава асимптотиката на общата (по цялото изпълнение на алгоритъма) работа за намирането на кандидат-ребрата:

  $$\Theta(n-1) + \Theta(n-2) + \cdots + \Theta(2) + \Theta(1)$$

  Тривиално се пресмята, че $\Theta(n-1) + \Theta(n-2) + \cdots + \Theta(1) \asymp \Theta(n^2)$.

- 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 цялата работа 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 върхът е u, неговият ключ е $u.\mathrm{key}$. Самото МПД се реализира с масив на предшествията. За разлика от [31], този масив ще записваме като $\pi[1..n]$.

```
МПД Prim2(G = (V, E): неор. свързан граф, w: тегл. ф-я върху G, s: стартов връх)
 1  foreach u ∈ V
 2      u.key ← ∞
 3      π[u] ← Nil
 4  s.key ← 0
 5  създай празна приоритетна min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← Extract-Min(Q)
 9      foreach y ∈ adj[x]
10          if y ∈ Q and w((x, y)) < y.key
11              y.key ← w((x, y))
12              π[y] ← x
13  return π[1..n]
```

Този псевдокод следва плътно псевдокода от [31, 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.

Коректност.

Теорема 60: Коректността на МПД Prim2

Масивът $\pi[1..n]$, който МПД Prim2 връща, реализира МПД на G.

Доказателство. Първо едно помощно твърдение.

Инвариант 25: while-цикълът на МПД Prim2

При всяко достигане на ред 7:

① $Q \neq \emptyset \rightarrow (\exists u \in Q : u.\text{key} < \infty)$.

② $\forall u \in Q \setminus \{s\}$: ако $u.\text{key} < \infty$, то:

- ❶ $\pi[u] \neq \text{Nil}$
- ❷ $\pi[u] \notin Q$
- ❸ $u.\text{key}$ е теглото на най-леко ребро, инцидентно с u и прекосяващо среза $\{V \setminus Q, Q\}$.

③ $\forall u \in Q$: ако $u.\text{key} = \infty$, то: съществува $v \in Q$, такъв че $v.\text{key} < \infty$ и има път между u и v, чиито върхове са само от Q.

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 $V = Q$, $s.\text{key} < \infty$ 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 $z.\text{key} < \infty$, доказателството е готово. Да допуснем, че в момента t, за всеки $z \in Q$, $z \neq x$, е вярно, че $z.\text{key} = \infty$. Но част ③ от индуктивното предположение влече, че x поне един съсед z' в Q. Очевидно на ред 11 $z'.\text{key}$ 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, $Q_f = \{v \in Q \,|\, v.\text{key} < \infty\}$ и $Q_\infty = \{v \in Q \,|\, v.\text{key} = \infty\}$. 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 ❶, ❷ и ❸ остават в сила в момента t'.

Разглеждаме произволен $v \in Q_f \cup N(x)$. Ако в момента t, $w((x, v)) \geqslant v.\text{key}$, то булевото условие на ред 10 е лъжа, когато y стане v, и редове 11–12 не се изпълняват. Тогава в момента t', ❶, ❷ и ❸ остават в сила за v от индукционното предположение. Ако в момента t, $w((x, v)) < v.\text{key}$, 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 ❶ и ❷ са в сила за v. На ред 11, $v.\text{key}$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото, инцидентно с v, прекосяващо ребро, така че и ❸ е в сила.

Разглеждаме произволен $v \in Q_\infty \cup N(x)$. Когато y стане v (ред 9), булевото условие на ред 10 задължително е истина, понеже $v \in Q$ и $v.\text{key} = \infty$. После на ред 12, $\pi[v]$ става x, така че ❶ и ❷ са в сила за v. На ред 11, $v.\text{key}$ става $w((x, v))$. Очевидно сега (x, v) е най-лекото (и единствено), инцидентно с v, прекосяващо ребро, така че и ❸ 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 $v \in Q_\infty \backslash N(x)$, твърдението остава в сила заради свързаността на графа. □

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 Инвариантът 26 е в сила.

Инвариант 26: while-цикълът на МПД Prim2

При всяко достигане на ред 7, $\{(u,\pi[u]) \mid u \in V \backslash (\{s\} \cup Q)\} = E(T) \cap E(G_{\overline{Q}})$.

База. Разглеждаме първото достигане на ред 7. В този момент $\{s\} \cup Q = V$, така че $V \backslash (\{s\} \cup Q) = \emptyset$ и множеството в лявата страна е празно. Но множеството в дясната страна също е празно, понеже всички върхове са в Q и $E(G_{\overline{Q}})$ е празно. Със сигурност такова 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 Инвариант 25 е вярно, че $\exists u \in Q : u.\text{key} < \infty$.

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 най-леко ребро, прекосяващо среза $\{V \backslash Q, Q\}$. Но това следва директно от Инвариант 25. Съгласно Теорема 58, 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 среза $\{V \backslash Q, Q\}$.

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. Заключаваме, че след изваждането на този връх от Q, множеството $\{(u,\pi[u]) \mid u \in V \backslash (\{s\} \cup Q)\}$ 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, $V \backslash (\{s\} \cup Q) = V \backslash \{s\}$; заключаваме, че $\{(u,\pi[u]) \mid u \in V \backslash \{s\}\} = E(T)$. С други думи, МПД Prim2 е построил МПД чрез $\pi[1..n]$. □

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**-цикъла (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**-цикъла 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 Descrease-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 DESCREASE-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 $\Theta(m + n \lg n)$. Събираемото $n \lg n$ 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 наместо 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 [31, 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 ненамаляващия 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

[†] А не в $\Theta(n + m)$ – защо?

[‡] За теглата на ребрата няма ограничения, така че не може да ползваме COUNTING SORT.

$o(\lg n)$, 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, 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 [133]. 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.

МПД Kruskal($G = (\{1, \ldots, n\}, E)$: неориентиран свързан граф, w: тегловна ф-я върху G)

```
1   A ← ∅
2   направи разбиването {{1}, {2}, ..., {n}}
3   сортирай E по тегла
4   count ← 0
5   while count < n − 1 do
6       нека e = (u, v) е следващото ребро в сортираната редица
7       if component(u) ≠ component(v)
8           A ← A ∪ {e}
9           identify(component(u), component(v))
10          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 58, реброто e е сигурно за A.

Сложност по време. За да бъде ефикасен МПД Kruskal, трябва проверката дали $\boxed{\text{component}(u) \neq \text{component}(v)}$ (ред 7) и операцията $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ (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 $\boxed{\text{component}(u) \neq \text{component}(v)}$ така:

```
component(u) ≠ 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 $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ така:

```
identify(component(u), 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.

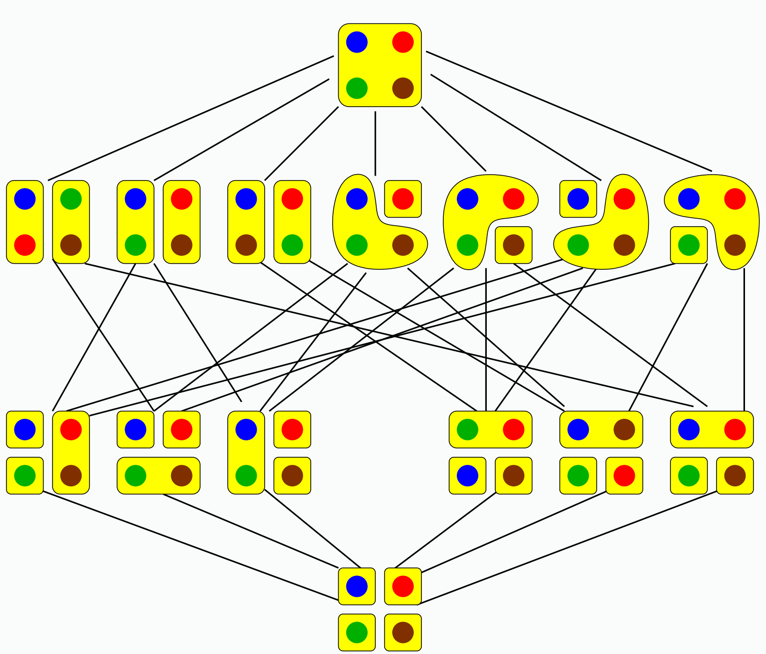
Допълнение 41: Частична наредба $\sqsubseteq$ върху разбиванията

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 [19] и [118] 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 нивá 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 $\boxed{\text{component}(u) \neq \text{component}(v)}$ става във време $\Theta(1)$. Но операцията $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ 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

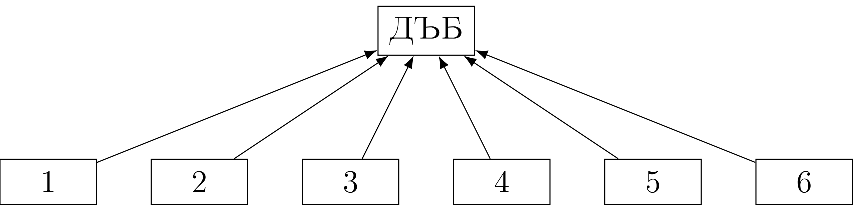
Очевидно при този начин на съхранение на имената, сливането на дялове става в линейно, а не в логаритмично време.

- 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 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 опитаме 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 62). 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.

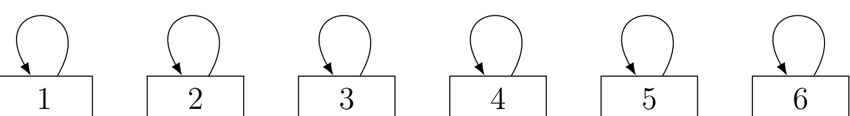
Забележете това: "върховете са еднотипни". При компонентите ДЪБ и БУК от миналата подсекция, имената на компонентите бяха съвсем различни от имената на елементите в тях. Математическият подход към описанието на дадено разбиване е точно такъв – ако е дадено множество $Z = \{z_1, \dots, z_n\}$ и разглеждаме 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 разбиванено (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 Find(u) връща лидера на компонентата, в която се намира u, а функцията Union(i, j) работи само върху лидери на компоненти.

Ето псевдокодът на Find. $\pi[u]$ е родителят на u.

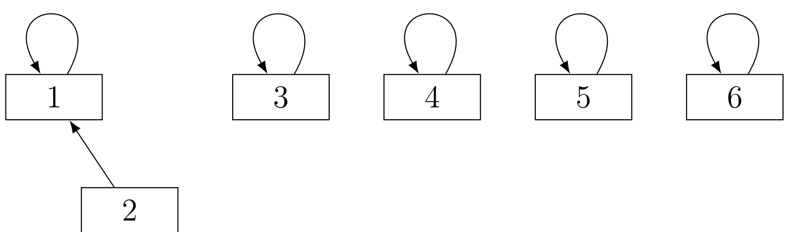
```
Find(u)
1  if u ≠ π[u]
2      return Find(π[u])
3  else
4      return u
```

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:

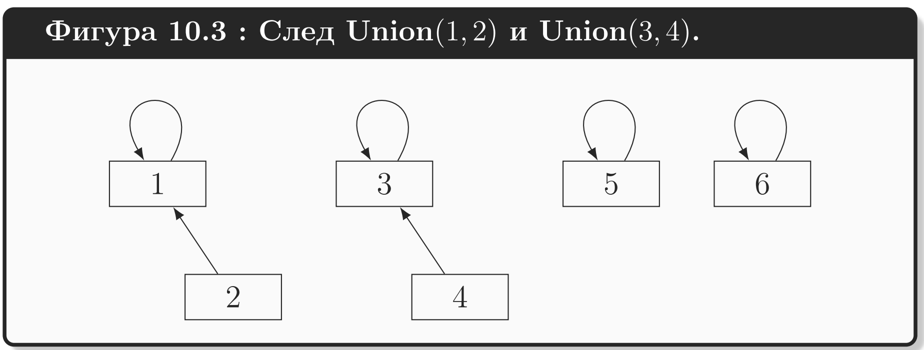


В този момент има шест компоненти, всяка с по един елемент, който е и лидерът. Find върху който и да е елемент ще върне самия него. Да кажем, че се извърши Union$(1, 2)$. След това вече има само пет дяла, като единият е от два елемента, а именно 1 и 2. Да кажем, че лидерът на двуелементния дял е 1. Тъй като имплементираме с ориентирани дървета, трябва указателят на 2 вече да сочи към 1:

[†] С други думи, минималният елемент на решетката от Допълнение 41.

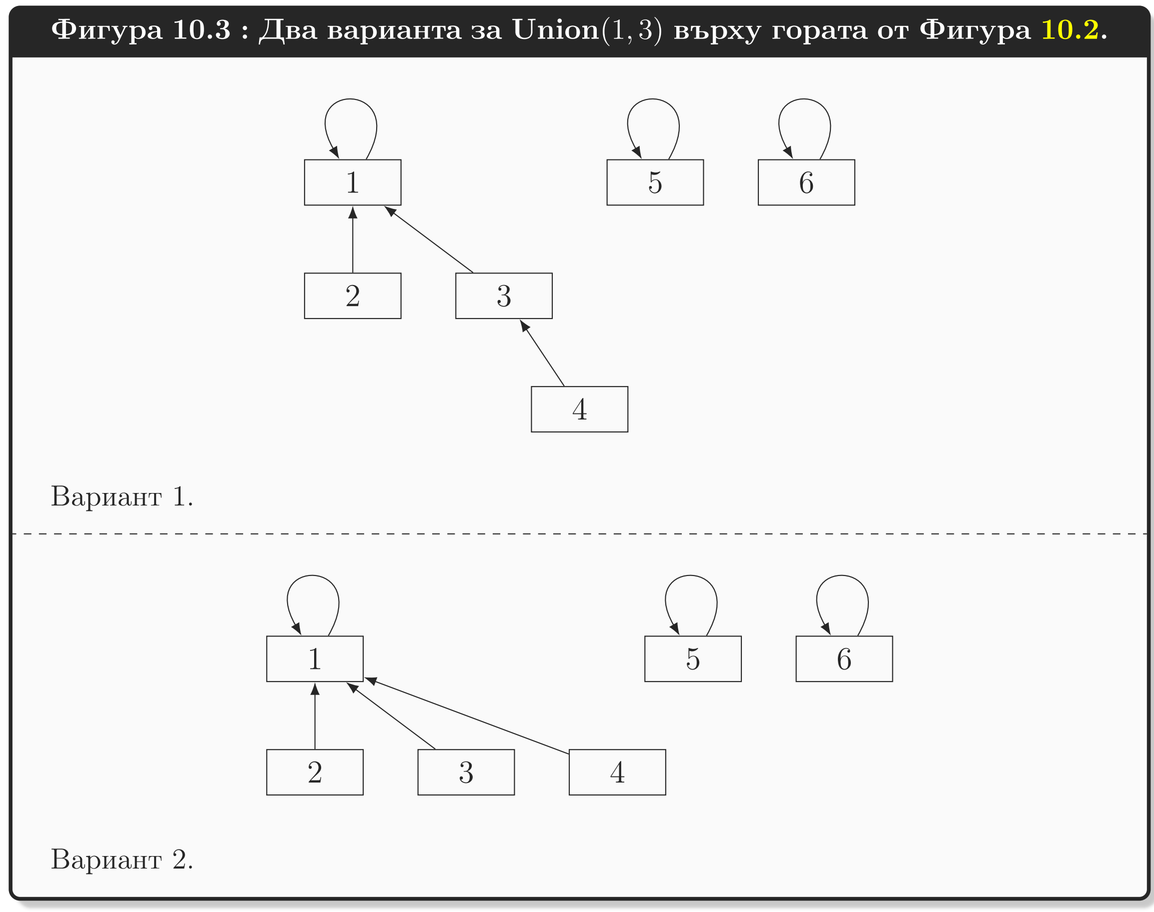


Да кажем, че следващото сливане е Union(3, 4). Резултатът е показан на Фигура 10.2.



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.



Фигура 10.3 : Два варианта за Union(1, 3) върху гората от Фигура 10.2.

[†]Викане Union(1, 4) би било некоректно, защото 4 не е лидер.

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 кой 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.

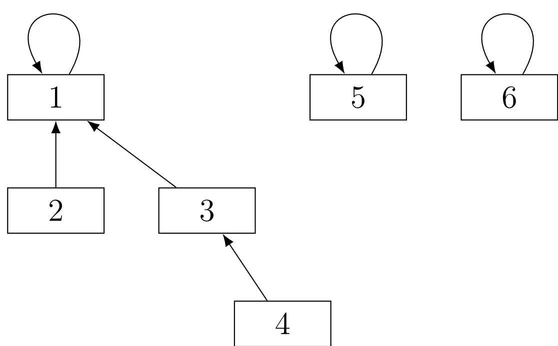
Обаче ние предпочитаме Вариант 1. Фигура 10.3 показва малки дървета, но става ясно, че ако сливаме две произволни дървета, всяко с височина едно, и резултатът трябва да е с височина едно, в най-лошия случай Union ще работи в линейно време в броя на върховете на тези две дървета, който, на свой ред, може 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 52

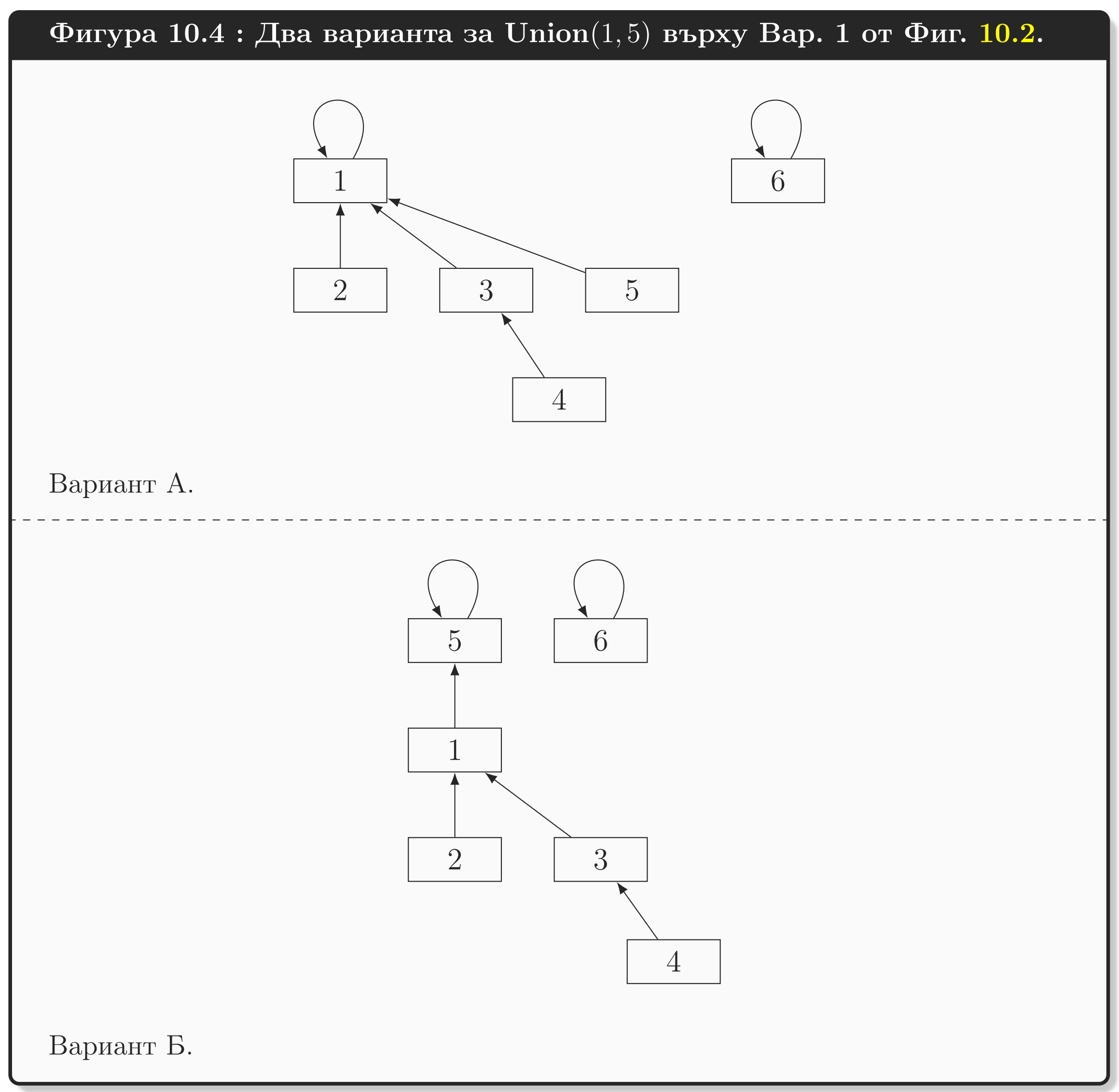
При реализацията с ориентирана гора, сложността по време на Find(u) в най-лошия случай е $\Theta(h)$, където h е височината на дървото, чийто връх е u.

Като пример, да разгледаме пак дървото от Вариант 1 на Фигура 10.3:



Нека извършим 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 Union$(1, 5)$ върху Вар. 1 от Фиг. 10.2.



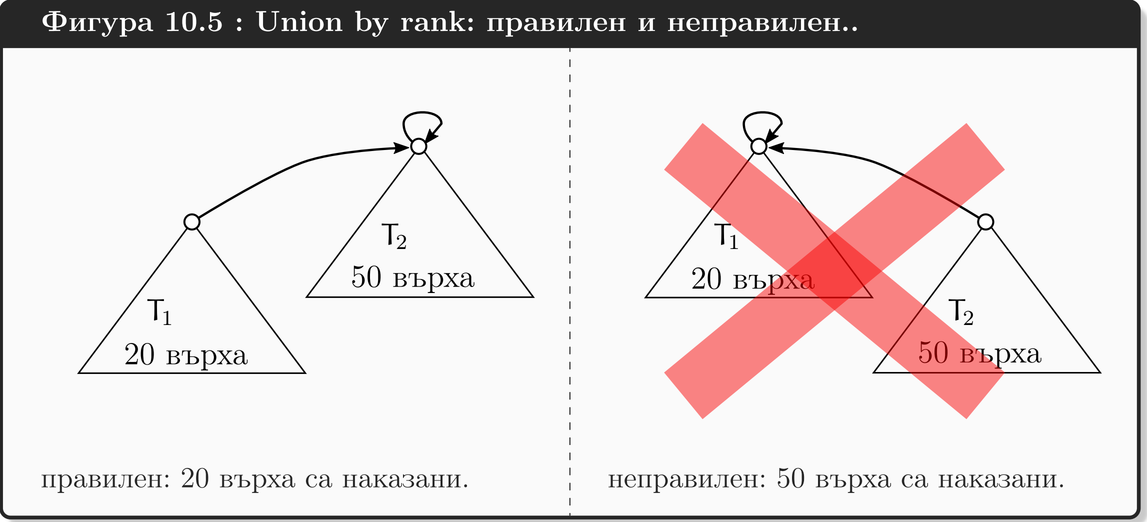
Ясно е, че изборът на 5 за нов лидер (Вариант Б) беше неудачен. Нещо повече. Серия от такива неудачни избори може да доведе до дървета с линейна височина, откъдето в най-лошия случай Find ще работи в линейно време. За да избегнем това, прилагаме две *евристики*[†].

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 указателя на корена на дървото с по-малък ранг (ако са с еднакви рангове, решението се взема произволно).

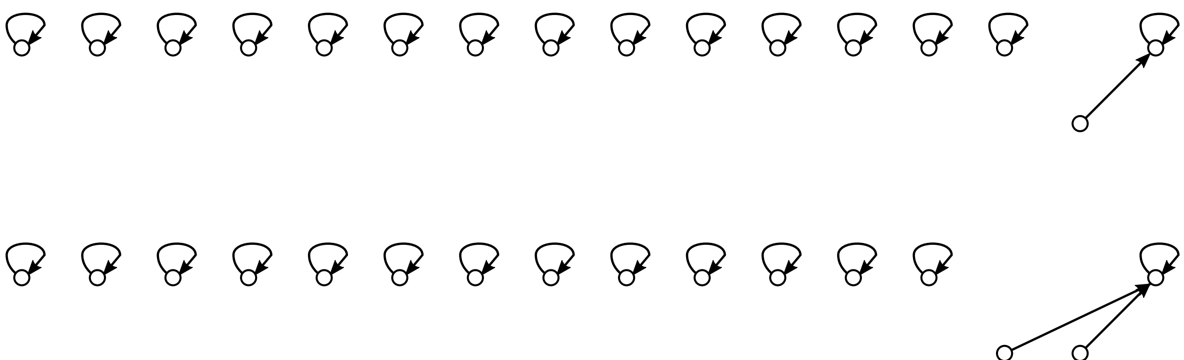
[†]На английски терминът е *heuristic*. Свободно казано, означава добра идея за решаване на някаква задача. Евристиките не гарантират оптималност на решението и дори не дават гаранция за това, колко далече от оптималното е тяхното решение.

Фигура 10.5 : Union by rank: правилен и неправилен..

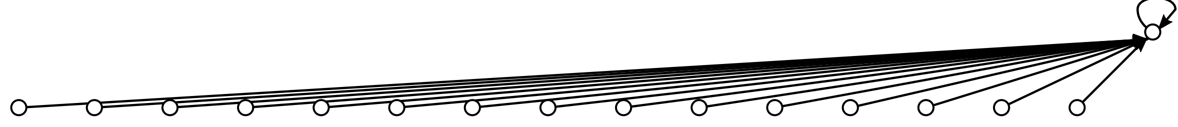


Да разгледаме примери за създаване на дървета (разбивания) чрез Union by rank. Започваме с шестнадесет върха.

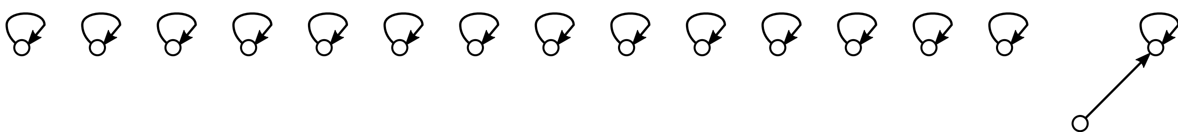
Ако сме късметлии, може сливанията да стават така:



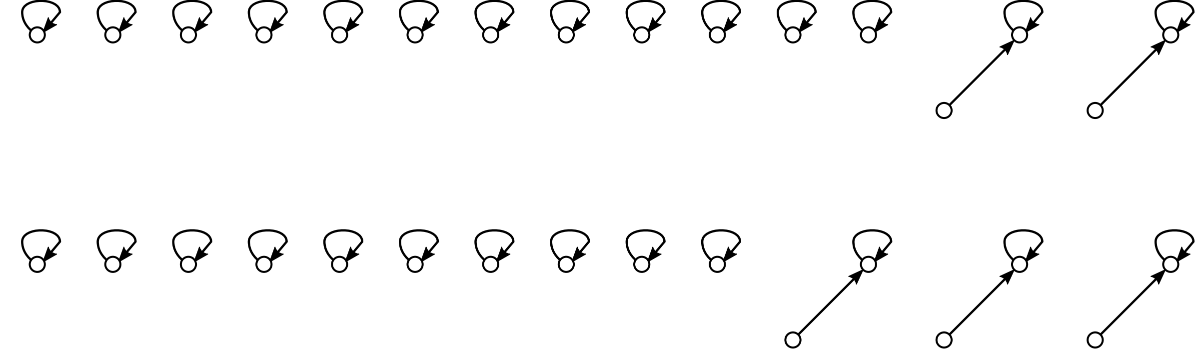
и така нататък, докато получим дърво с височина едно:



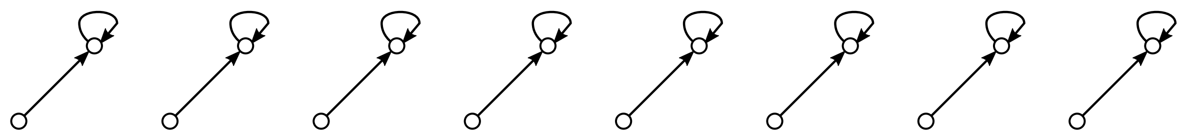
Ако не сме късметлии, може сливанията да стават по следния начин. Първото сливане отново е:



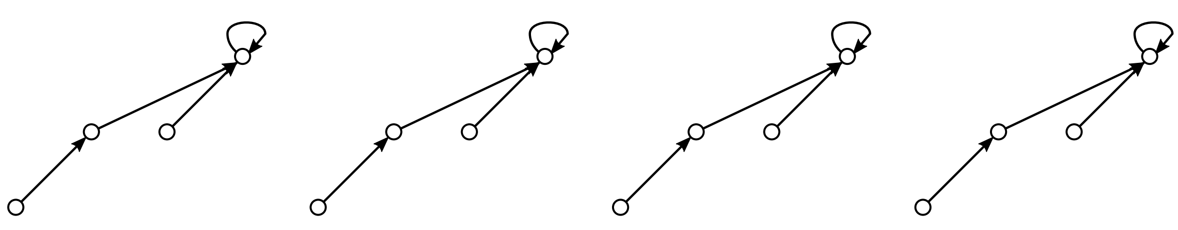
После обаче продължават така:



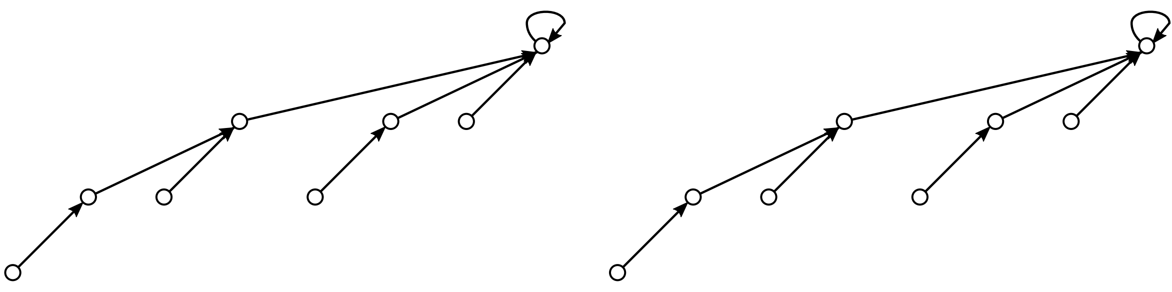
и така нататък до:



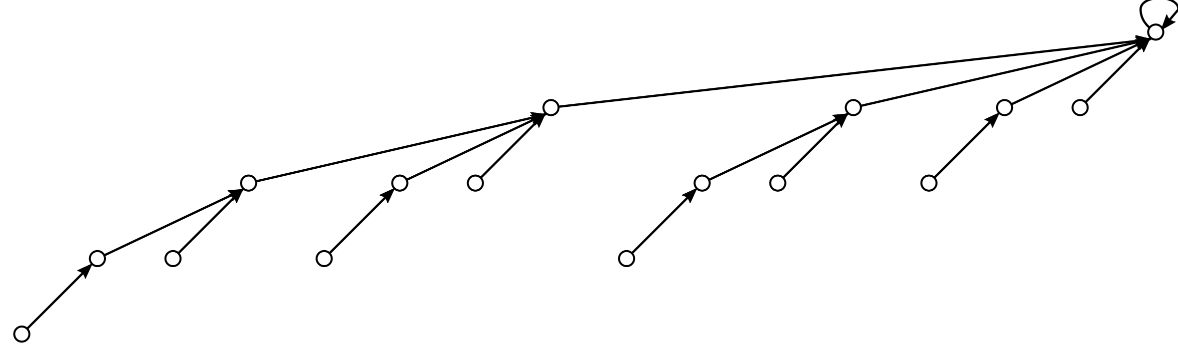
После имаме:



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 61: 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 $h \leqslant c \lg n$. 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 $0 \leqslant c \cdot \lg 1$. ✓

Индукционно предположение Допускаме, че съществува константа $c > 0$, така че за някое $h \geqslant 0$, всяко дърво, генерирано от Union by rank, $h \leqslant c \lg n$.

Индукционна 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

$$h \leqslant c \lg n_1$$
$$h \leqslant c \lg n_2$$

БОО, нека $n_1 \geqslant n_2$. Тогава

$$c \lg(n_1 + n_2) \geqslant c \lg(n_2 + n_2) = c \lg 2n_2 = c \lg n_2 + c \geqslant \quad \text{// съгласно инд. предположение}$$
$$h + c \geqslant h + 1$$

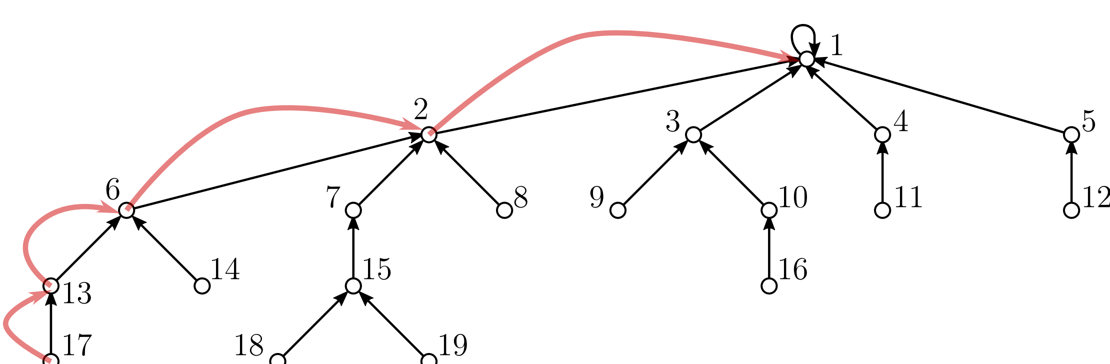
Има такава константа, да кажем $c = 1$. □

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 $\boxed{\text{component}(u) \neq \text{component}(v)}$ (ред 7) и операцията $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ (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 $\Omega(m \lg n)$.

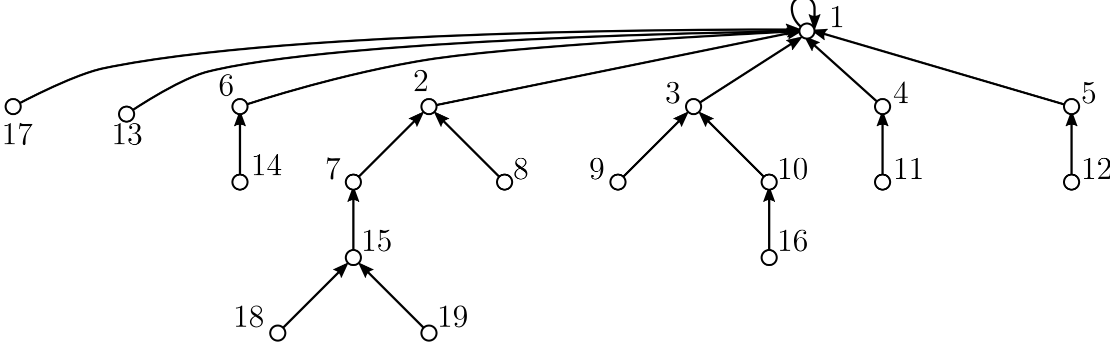
Path compression. Да си припомним текущия псевдокод на Find.

```
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 "катерене нагоре" в дървото за достигане до корена, върховете, през които минаваме, да ги наречем $u_1, \ldots, u_k$, 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 $u_1, \ldots, u_k$, 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 $\quad \pi[u] \leftarrow$ 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 [31, 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 неогратичено нарастваща функция на n, но от практическа гледна точка, $\alpha(n) \leqslant 4$ 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.

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 $w : E \to \mathbb{R}$.

Конвенция 10

В тази лекция, казвайки "път", имаме предвид път, който не е непременно прост.

Нотация 9: $u \rightsquigarrow v$, $u \overset{p}{\rightsquigarrow} v$, $\mathsf{Paths}(u, v)$

Нека $u, v \in V$. "$u \rightsquigarrow v$" е кратък запис за "ориентиран път от u до v". "$u \overset{p}{\rightsquigarrow} v$" е кратък запис за "$p$ е ориентиран път от u до v". "$\mathsf{Paths}(u, v)$" означава $\{p \mid u \overset{p}{\rightsquigarrow} v\}$.

Да си припомним, че пътищата не са непременно прости. По тази причина, в ориентиран граф с цикли, $\mathsf{Paths}(u, v)$ (може да) е безкрайно множество. В неориентиран граф, всяко ребро задава безкрайна колекция от пътища, така че $\mathsf{Paths}(u, v)$ е безкрайно множество, освен особените случаи на липса на пътища между u и v, както и на $u = v$ като изолиран връх.

Определение 83: Тегло на път

Теглото на пътя p е $w(p) = \sum_{e \in E(p)} w(e)$.

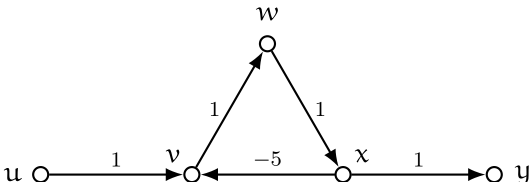
Определение 84: Отрицателен цикъл

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 53: 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 $\mathsf{Paths}(u, v)$ 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 54: 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 85: Теглото на най-къс път

Теглото на най-къс път от u *до* v е

$$\delta(u,v) = \begin{cases} \infty, \text{ ако } \mathsf{Paths}(u,v) = \varnothing \\ -\infty, \text{ ако } \mathsf{Paths}(u,v) \neq \varnothing \text{ и } \exists p \in \mathsf{Paths}(u,v) : p \text{ съдържа отрицателен цикъл} \\ \min\{w(p) \mid p \in \mathsf{Paths}(u,v)\}, \text{ ако } \mathsf{Paths}(u,v) \neq \varnothing \text{ и } \neg\exists p \in \mathsf{Paths}(u,v) : p \text{ съд. отр. цик.} \end{cases}$$

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 53. 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 телго, 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 11

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 12

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 54.

11.1.2 Варианти на задачата

Известни са следните варианти на задачата за най-къс път в тегловен граф.

Изч. Задача 24: 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 25: 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 26: 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 27: 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.

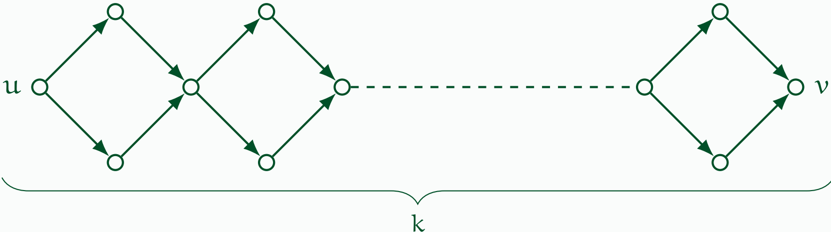
Ние ще разгледаме подробно Single Source Shortest Paths, като за краткост ще я наричаме SSShP. Ние ще решим SSShP директно в тежкия вариант.

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 подборно. 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, Секция 12.3. 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 42: 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 [31, стр. 645].

Теорема 62: 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:

$$p = \underbrace{s \rightsquigarrow u \overset{q}{\rightsquigarrow} v \rightsquigarrow t}_{p}$$

Тогава q е най-къс път от u до v.

[a]Не 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.

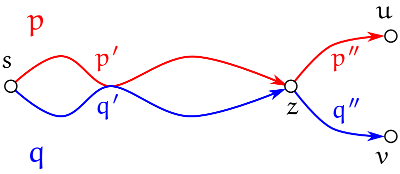
Доказателство: Ако допуснем, че има път q', такъв че $u \overset{q}{\rightsquigarrow} v$ и $w(q') < w(q)$, веднага заключаваме, че p не е най-къс път от s до t, понеже замяната на q с q' в p намалява $w(p)$ с $w(q) - w(q')$. □

В доказателството на Теорема 62 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 при отрицателни тегла и по-специално при наличието на отрицателни цикли, това "изрязване" може да увеличи теглото на пътя, което е проблем за доказателството.

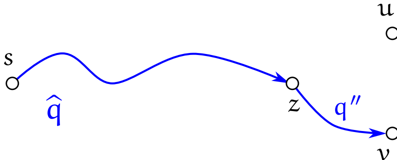
11.1.4 Дърво на най-късите пътища

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 $\pi[1 \mathrel{..} n]$, 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 който 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':

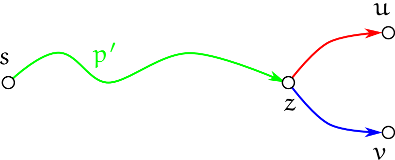


Твърдим, че $w(p') = w(q')$. Да допуснем противното. БОО, нека $w(p') < w(q')$. Тогава в q подменяме q' с p':



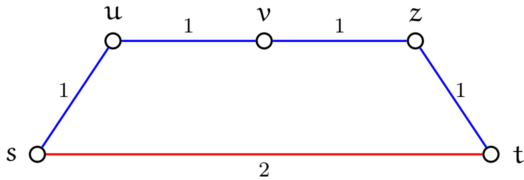
и получаваме път $\widehat{q}$ от s до v, такъв че $w(\widehat{q}) < w(q)$, в противоречие с допускането, че q е най-къс път от s до v.

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:



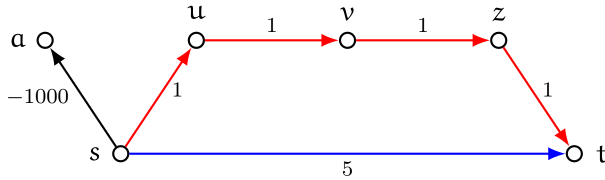
Най-късият път между s и t е реброто (s, t) с тегло 2 (в червено). Но МПД-то (то е само едно) се състои от четирите сини ребра, всяко с тегло 1. Ако си представим работата на алгоритъма на Kruskal върху този граф, сортирайки ребрата, той ще постави в началото четирите ребра с тегло 1 и след това реброто 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 55

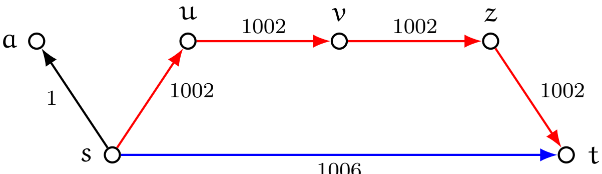
Задачите за намиране на МПД и намиране на дърво на най-късите пътища върху неориентирани тегловни графи са принципно различни. Всяко МПД, за всеки два върха s и t, дава (уникален) път p между s и t, но в общия случай p не е най-къс път в графа между s и t.

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 58 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 56

Нека G тегловен неориентиран граф с тегловна функция $w : E \to \mathbb{R}$. Нека $w' : E \to \mathbb{R}$ 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 $w : E \to \mathbb{R}$ и $w' : E \to \mathbb{R}$ 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 алгоритмите 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 54; 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 12.12 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 12.12, 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 51) при обръщане на знака на теглата. Иначе казано, ако G е ориентиран тегловен граф, върху който са дефинирани тегловни функции $w, w' : E \to \mathbb{R}$, 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.

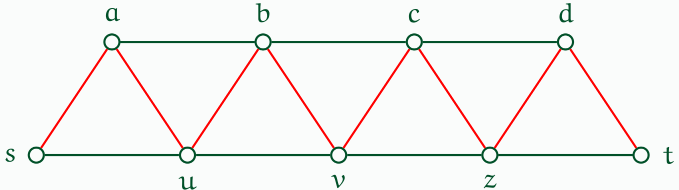
Допълнение 43: За структурата на най-дългите пътища

За задачата за най-дългите пътища не е вярно, че оптималната структура се състои от оптимални подструктури (сравнете с Теорема 62). 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.

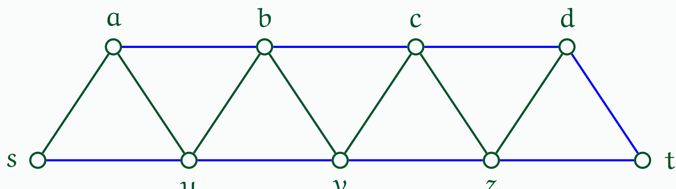
Изч. Задача 28: Longest Path между два върха, оптимизационна версия

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 нарисуван 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 44: 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 едновременното 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:

$$\text{EUR} \rightarrow \text{BGN} \rightarrow \text{MNT} \rightarrow \text{CZK} \rightarrow \text{HUF} \rightarrow \text{EUR}$$

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:

$$\text{EUR} \rightarrow \text{BGN} \rightarrow \text{MNT} \rightarrow \text{CZK} \rightarrow \text{HUF} \rightarrow \text{EUR}$$

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 $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$, такава че

$$M[i_1, i_2] \times M[i_2, i_3] \times \cdots \times M[i_{k-1}, i_k] \times M[i_k, i_1] > 1 \quad (11.1)$$

Това е Problem 24.3 в [31, 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:

$$\lg(M[i_1, i_2]) + \lg(M[i_2, i_3]) + \cdots + \lg(M[i_{k-1}, i_k]) + \lg(M[i_k, i_1]) > 0 \quad (11.2)$$

което е същото като

$$-\lg(M[i_1, i_2]) - \lg(M[i_2, i_3]) - \cdots - \lg(M[i_{k-1}, i_k]) - \lg(M[i_k, i_1]) < 0 \quad (11.3)$$

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.

[a] Курсовете 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 [31, 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 пó 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 $d[1..n]$ 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 $\pi[1..n]$, 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 ∈ V(G)
2      v.d ← ∞
3      v.π ← Nil
4  s.d ← 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 ← x.d + w((x, y))
3      y.π ← x
```

Смисълът е напълно ясен: опитваме да намерим път с по-малко тегло $s \leadsto y$ от този, който е оптимален до момента. Стойността $y.d$ има смисъл на дължината на най-къс път $s \leadsto y$ според информацията до момента. Опитваме да намерим по-къс път до y, ползвайки x като предпоследен връх. Ако се окаже, че $y.d \leqslant x.d + w((x, y))$, 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 отпускаме. Според [31, стр. 647], произходът на името "релаксация" е следният. Резултатът от Relax((x, y)) може да бъде разглеждан като релаксиране на ограничението (constraint) $y.d \leqslant x.d + w((x, y))$; 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 $y.d \leqslant x.d + w((x, y))$, 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 [31, стр. 650]

Лема 54: Неравенство на триъгълника

За всяко ребро $(x, y) \in E(G)$ е изпълнено

$$\delta(s, y) \leqslant \delta(s, x) + w((x, y)) \tag{11.4}$$

Доказателство: Ако няма път $s \leadsto x$, то $\delta(s, x) = \infty$ и дясната страна на (11.4) е ∞. Неравенството остава в сила независимо от това, дали няма път $s \leadsto y$, което влече, че лявата страна е ∞, или има път $s \leadsto y$, което влече, че лявата страна е или число, или $-\infty$.

Нека има път $s \leadsto x$. Ако съществува поне един $s \leadsto x$, който съдържа отрицателен цикъл, то $\delta(s, x) = -\infty$. Но тогава очевидно съществува $s \leadsto y$, който съдържа отрицателен цикъл, така че и $\delta(s, y) = -\infty$ и (11.4) е в сила.

Нека нито един $s \leadsto x$ не съдържа отрицателен цикъл. Тогава $-\infty < \delta(s, x) < \infty$. Нека p е най-къс път $s \overset{p}{\leadsto} x$. Тогава $w(p) = \delta(s, x)$. Ако съществува поне един $s \leadsto y$, 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 $s \overset{p'}{\rightsquigarrow} y$. Нещо повече: $w(p') = w(p) + w((x, y)) = \delta(s, x) + w((x, y))$. Очевидно $\delta(s, y) \leqslant w(p')$. Тогава $\delta(s, y) \leqslant \delta(s, x) + w((x, y))$. □

Лема 55: Горна граница и точна горна граница

Нека G е инициализиран с ISS(G, s). Тогава, за всеки връх $v \in V(G)$, $v.d \geqslant \delta(s, v)$, 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 55 с Лема 45.

Доказателство: Ще докажем, че $v.d \geqslant \delta(s, v)$ 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 $s.d \geqslant \delta(s, s)$, 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 $s.d \geqslant \delta(s, s)$. Нека $v \neq s$. Вярно е, че $v.d \geqslant \delta(s, v)$, 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:

$$\begin{aligned} v.d &= u.d + w((u, v)) && \text{// защото Relax работи така} \\ &\geqslant \delta(s, u) + w((u, v)) && \text{// от индуктивното предположение за } u \\ &\geqslant \delta(s, v) && \text{// от Лема 54} \end{aligned}$$

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 доказания факт, че $v.d \geqslant \delta(s, v)$ 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. □

Лема 56: 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 55, $v.d \geqslant \delta(s, v)$, след произволна редица от релаксации на ребрата. Но $\delta(s, v) = \infty$ съгласно Определение 85. □

Лема 57: 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 55, ако веднъж се постигне равенство $u.d = \delta(s, u)$, то се запазва нататък. В частност, след релаксирането на (u, v), в сила е

$$\begin{aligned} v.d &\leqslant u.d + w((u, v)) \quad // \text{ очевидно от кода на Relax} \\ &= \delta(s, u) + w((u, v)) \quad // \text{ както вече казахме} \\ &= \delta(s, v) \quad // \text{ Теорема 62} \end{aligned}$$

Щом $v.d \leqslant u.d + w((u, v))$ и $v.d \geqslant u.d + w((u, v))$ (Лема 55), то $v.d = u.d + w((u, v))$ и това равенство остава в сила до края (пак Лема 55). □

11.2 Алгоритъмът на Dijkstra в два варианта

Ще разгледаме един от най-популярните алгоритми за намиране на най-къс нът 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 който алгоритъмът на Dijkstra не работи коректно.

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. 426 и МПД Prim2 на стр. 428—с подобна разлика между тях.

```
SSShP Dijkstra1(G: ориентиран тегловен граф, w: тегл. ф-я върху G, s: стартов връх)
1   ISS(G, s)
2   S ← ∅
3   while ∃u ∈ V\S : u.d < ∞ do
4       избери x ∈ V\S, такъв че x.d е минимална
5       S ← S ∪ {x}
6       foreach y ∈ adj[x]
7           Relax((x, y))
```

```
SSShP Dijkstra2(G: ориентиран тегловен граф, w: тегл. ф-я върху G, s: стартов връх)
1   ISS(G, s)
2   S ← ∅
3   създай празна приоритетна min опашка Q
4   Q ← V
5   while not isempty(Q) do
6       x ← Extract-Min(Q)
7       S ← S ∪ {x}
8       foreach y ∈ adj[x]
9           Relax((x, y))
```

Допълнение 45: За подобието между МПД PRIM2 и SSSHP 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. 428 и на предната страница.

- За удобство тук описваме Dijkstra подборно, без ISS и RELAX.
- Множеството S 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. Тук няма S.
- 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 ∈ V(G)
 2      v.key ← ∞
 3      π[v] ← Nil
 4  s.key ← 0
 5  празна приор. min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← EXTRACT-MIN(Q)
 9      foreach y ∈ adj[x]
10          if y.key > w((x, y))
11              y.key ← w((x, y))
12              π[y] ← x
```

```
SSSHP DIJKSTRA2(G, w, s)
 1  foreach v ∈ V(G)
 2      v.d ← ∞
 3      v.π ← Nil
 4  s.d ← 0
 5  празна приор. min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← EXTRACT-MIN(Q)
 9      foreach y ∈ adj[x]
10          if y.d > x.d+w((x, y))
11              y.d ← x.d+w((x, y))
12              y.π ← x
```

Ясно се вижда, че единствената формална разлика между двата псевдокода 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 предварително 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 (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 58. 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).

Допускаме БОО, че целият G е достижим от s. "SSShP Dijkstra" означава кой да е от двата варианта на алгоритъма.

Теорема 63: 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 терминирането на SSShP Dijkstra(G, w, s):

$$\forall v \in V(G) : v.d = \delta(s, v)$$

Доказателство: Приемаме за очевидно, че всеки връх на графа "влиза" в S на някоя итерация на **while**-а. Ще докажем, че за всеки връх $v \in V(G)$, в момента, в който v влиза в S е вярно, че $v.d = \delta(s, v)$. Това влече верността на теоремата заради Лема 55.

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 S. Нещо повече: нека v е **първият** връх, който "влиза" в S с d-стойност, различна от минималното тегло на път от s до него. Очевидно $v \neq s$, защото $s.d = 0$ и, при положителни тегла, $\delta(s, s) = 0$.

Разглеждаме тази итерация, на която в S "влиза" връх v. В началото на въпросната итерация, нека t е моментът точно преди v да "влезе" в S. Щом $v \neq s$, то $S \neq \emptyset$ при тази итерация. По допускане, съществува $s \rightsquigarrow v$. Разглеждаме най-къс $s \overset{p}{\rightsquigarrow} v$. Щом теглата са положителни, такъв има.

В момента t е вярно, че първият връх на p (това е s) е в S, а последният връх (това е v) не е в S. Дефинираме y като най-близкият до s връх в p, който в момента t не е в S. Възможно е $y = v$, възможно е и $y \neq v$. Важното е, че y е добре дефиниран. Нека върхът преди y в p се казва x (възможно е $x = s$). Забележете, че всички върхове в p от s до x включително в момента t са в S, и y е първият връх в p, в посока s към v, който не е в S:

$$p = \underbrace{s \cdots\cdots\cdots x}_{\text{в } S} \; \underbrace{y}_{\text{не е в } S} \; \cdots\cdots\cdots \; \underbrace{v}_{\text{не е в } S}$$

Ще докажем, че $y.d = \delta(s, y)$ в момента t. Знаем, че $x \in S$ в t и че v е първият връх, влизащ в S, за който d-стойността не е равна на претегленото разстояние от s. Връх x е бил сложен в S на някоя предишна итерация, в някакъв момент $t' < t$. Следва, че в момента t', $x.d = \delta(s, x)$, и съгласно Лема 55, във всеки момент след t' продължава да е вярно, че $x.d = \delta(s, x)$. Но на итерацията, в която x е сложен в S, реброто (x, y) е било релаксирано – вижте псевдокода. Прилагаме Лема 57 и заключаваме, че

$$y.d = \delta(s, y) \tag{11.5}$$

след това релаксиране, включително и в момента t.

Съгласно Теорема 62, $\delta(s, y) \leqslant \delta(s, v)$, тъй като теглата са положителни, а s, y и v са наредени в този ред върху най-къс път от s до v. Съгласно Лема 55, $\delta(s, v) \leqslant v.d$ по всяко време от работата на алгоритъма. Тогава

$$\delta(s, y) \leqslant \delta(s, v) \leqslant v.d \tag{11.6}$$

От (11.5) и (11.6) заключаваме, че в момента t:

$$y.d = \delta(s, y) \leqslant \delta(s, v) \leqslant v.d \tag{11.7}$$

Ключовото наблюдение е, че и y, и v са върхове извън S в момента t и причината алгоритъмът да избере v, а не y, като връх, който да влезе в S, е една единствена:

$$v.d \leqslant y.d \tag{11.8}$$

От (11.7) и (11.8) заключаваме, че

$$v.d = y.d \tag{11.9}$$

Но тогава (11.7) всъщност е верига от равенства:

$$y.d = \delta(s, y) = \delta(s, v) = v.d \tag{11.10}$$

и в частност $\delta(s, v) = v.d$. Тогава, съгласно Лема 55, това остава в сила до края на алгоритъма, в противоречие с допускането, че $\delta(s, v) \neq v.d$ в края. ↯ □

Ако не е вярно, че целият G е достижим от s, Лема 56 казва, че върховете, които не са достижими от s, са с d-стойности ∞ в края на алгоритъма. Имайки предвид Теорема 63, заключаваме, че върховете, които не са достижими от s са точно тези, които са с d-стойности ∞ в края на алгоритъма.

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.

11.3 Най-къси пътища в дагове

Ако G е 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** $y \in adj[x]$
5.   Relax((x, y))

Ще покажем коректността на този алгоритъм. С учебна цел ще направим две доказателства: първото е "традиционното" доказателство (примерно, [31, 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 64: 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 терминирането на DAG SSShP(G, w, s):

$$\forall v \in V(G) : v.d = \delta(s, v) \tag{11.11}$$

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 56. ✓

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 55). ✓

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 $\{s, \ldots, v'\}$. Очевидно е, че по отношение на връх v, RELAX реализира

$$v.d = \min_{u \in U} \{u.d + w((u, v))\} \tag{11.12}$$

От друга страна обаче, очевидно

$$\delta(s, v) = \min_{u \in U} \{\delta(s, u) + w((u, v))\} \tag{11.13}$$

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, че 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 $v V(G) \setminus \{s\}$. 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. Щом x е недостижим от s, то x не може да се явява предпоследен връх по път от s до никой връх $y \in adj[x]$. Тогава за всеки $y \in adj[x]$ 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 $y \in adj[x]$ 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 $y \in adj[x]$ 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 $adj[x]$, 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 $y \in adj[x]$. Тогава за всеки $y \in adj[x]$ 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 $y \in adj[x]$ 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 индуктивтото 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 $y \in adj[x]$ 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 $adj[x]$, 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.

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 кой 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. □

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 62): 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 топологична сортировка.

  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 S (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 ← 1 to n − 1
3       foreach (x, y) ∈ E(G)
4           RELAX((x, y))
5   foreach (x, y) ∈ 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 57 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 55. 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 изпълняваве 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. □

Сложността по време, очевидно, е $\Theta(n(n+m))$, 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: задачата на опашката пред касата и пресмятане на числа на Fibonacci. Разглеждаме основата на алгоритмичната схема **Динамично Програмиране**, след което разглеждаме множество примери за нейното успешно прилагане: задачи за ефикасно пресмятане на комбинаторни функции, 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 за **NP**-пълната задача за намиране на най-дълъг път в графи, построен по схемата **Динамично Програмиране**.

12.1 Въведение с примери

12.1.1 Задачата за опашката пред касата

Дадена е каса за билети. Пред касата има опашка от n човека $h_1, h_2, \ldots, h_n$, в този ред. Известно е, че за $1 \leqslant i \leqslant n$, h_i би се забавил(а) d_i минути на касата, пазарувайки билетите си. Известно е, че за $1 \leqslant i \leqslant n-1$, h_i и h_{i+1}, очевидно съседи 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 $d_1, d_2, \ldots, d_n$ и времена на комбинирано пазаруване $c_1, c_2, \ldots, c_{n-1}$, 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 "тежкият вариант" са понятия, които въведохме на стр. 13.

Ако нямаше възможност за комбиниране на съседи 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 комбина и минат заедно през касата, възможността h_2 да се комбинира с h_3 отпада; и обратно, ако h_2 и h_3 са комбина, 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 съседи 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 вектор не може да има съседни единици, тъй като всеки човек от опашката, който има избор от две двойки (това са всички хора без първия и последния), 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 Fibonacci.

Допълнение 46: Колко са булевите вектори без съседни единици

Нека F_m е m-тото число на Fibonacci. 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 $m \geqslant 1$. 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 $m \geqslant 1$.

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

$$
\begin{aligned}
s_1 &= \langle h_1 \rangle \\
s_2 &= \langle h_1, h_2 \rangle \\
s_3 &= \langle h_1, h_2, h_3 \rangle \\
&\ldots \\
s_n &= \langle h_1, h_2, \ldots, h_n \rangle
\end{aligned}
$$

и се опитваме да получим оптимума за s_k чрез оптимумите за s_{k-1}, s_{k-2}, …, 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 за задачата 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. Ясно е, че този подход не води до 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, $3 \leqslant i \leqslant n$, 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_j е $\text{opt}(s_j)$, то в сила е

$$\text{opt}(s_i) = \begin{cases} d_1, & \text{ако } i = 1 \\ \min\{d_1 + d_2, c_1\}, & \text{ако } i = 2 \\ \min\{d_i + \text{opt}(s_{i-1}), c_{i-1} + \text{opt}(s_{i-2})\}, & \text{ако } i \geqslant 3 \end{cases} \tag{12.1}$$

(12.1) може да се имплементира като ефикасен алгоритъм. Ако изчисляваме оптимумите за s_3, s_4, …, s_n в този ред и ги съхраняваме в масив, то, за $3 \leqslant i \leqslant n$, 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 opt[1..n]
2  opt[1] ← d[1]
3  opt[2] ← min (d[1] + d[2], c[1])
4  for i ← 3 to n
5      opt[i] ← min (d[i] + opt[i − 1], c[i − 1] + opt[i − 2])
6  return 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 (12.1), 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, $opt[i-1]$ и $opt[i-2]$ 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 $opt[2]$, $opt[3]$, ..., $opt[n]$ е изчислена като минимум от две суми на ред 3 или ред 5, като тази цена е равна на едната или на другата сума (или на двете, ако те са равни помежду си). За всеки $opt[i]$ записваме индекс на елемент от масива, който участва в сумата, на която $opt[i]$ е равен; този индекс е или $i-1$, или $i-2$. След получаването на $opt[n]$, проследяваме веригата от индекси и това дава едно оптимално решение. Такова проследяване на английски се нарича *traceback*. На български ще казваме *обратно проследяване*.

Ето подробностите. Нека $n \geqslant 2$. Да препишем алгоритъма по следния (еквивалентен) начин, въвеждайки фиктивен елемент $opt[0] = 0$, за да може всички клетки от $opt[2..n]$ да получат стойностите си по един и същи начин.

```
ALG QUEUE PROCESSING-1(d[1..n], c[1..n−1])
1  създай масив от числа opt[0..n]
2  opt[0] ← 0
3  opt[1] ← d[1]
4  for i ← 2 to n
5      opt[i] ← min (d[i] + opt[i − 1], c[i − 1] + opt[i − 2])
6  return opt[n]
```

Да разсъждаваме отзад напред. $opt[n]$ е равен на минимума на $d[n] + opt[n-1]$ и $c[n-1] + opt[n-2]$ (ред 5).

- Да допуснем, че $d[n] + opt[n-1] < c[n-1] + opt[n-2]$. Разговорно, това означава, че h_n трябва да мине сам(а). Формално, това означава, че непосредственото изчисляване на $opt[n]$ става чрез $opt[n-1]$.

- Да допуснем, че $d[n] + opt[n-1] > c[n-1] + opt[n-2]$. Разговорно, това означава, че h_n трябва да се комбинира с h_{n-1}. Формално, това означава, че непосредственото изчисляване на $opt[n]$ става чрез $opt[n-2]$.

- Да допуснем, че $d[n] + \mathrm{opt}[n-1] = c[n-1] + \mathrm{opt}[n-2]$. Разговорно, това означава, че е все едно дали h_n минава сам(а), или се комбинира с h_{n-1}. Формално, това означава, че непосредственото изчисляване на $\mathrm{opt}[n]$ става чрез кой да е от $\mathrm{opt}[n-1]$ и $\mathrm{opt}[n-2]$. В този случай избираме произволно **точно единия** от $\mathrm{opt}[n-1]$ и $\mathrm{opt}[n-2]$ като елементът, участващ в непосредственото изчисляване на $\mathrm{opt}[n]$. Забележете, че не избираме и двата, защото искаме да получим едно решение, а не всички решения.

Във всеки случай, $\mathrm{opt}[n-1]$ да участва непосредствено в изчисляването на $\mathrm{opt}[n]$ е същото като h_n да **не се** комбинира с h_{n-1}; съответно, $\mathrm{opt}[n-2]$ да участва непосредствено в изчисляването на $\mathrm{opt}[n]$ е същото като h_n да **се** комбинира с h_{n-1}.

Разполагайки с елемента, участващ в непосредственото изчисляване на $\mathrm{opt}[n]$, можем да приложим същите разсъждения за него (стига n да е достатъчно голямо). Установяваме кой от предните елементи участва в неговото непосредствено изчисляване. И така нататък, докато не стигнем до началото на масива opt. По този начин, тръгвайки от $\mathrm{opt}[n]$ 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

$$\mathrm{opt}[0] \leftarrow 0$$
$$\mathrm{opt}[1] \leftarrow d[1] = 5$$
$$\mathrm{opt}[2] \leftarrow \min\left(d[2] + \mathrm{opt}[1], c[1] + \mathrm{opt}[0]\right) = \min\left(6 + 5, 7 + 0\right) = 7$$
$$\mathrm{opt}[3] \leftarrow \min\left(d[3] + \mathrm{opt}[2], c[2] + \mathrm{opt}[1]\right) = \min\left(3 + 7, 8 + 5\right) = 10$$
$$\mathrm{opt}[4] \leftarrow \min\left(d[4] + \mathrm{opt}[3], c[3] + \mathrm{opt}[2]\right) = \min\left(5 + 10, 4 + 7\right) = 11$$
$$\mathrm{opt}[5] \leftarrow \min\left(d[5] + \mathrm{opt}[4], c[4] + \mathrm{opt}[3]\right) = \min\left(9 + 11, 10 + 10\right) = 20$$

В червено са сумите-минимуми. Те дават индексите на предните елементи във веригата. Виждаме, че $\mathrm{opt}[5]$ се получава от $\mathrm{opt}[4]$, на свой ред $\mathrm{opt}[4]$ се получава от $\mathrm{opt}[2]$, а на свой ред $\mathrm{opt}[2]$ се получава от $\mathrm{opt}[0]$. Веригата от индекси е $\langle 0, 2, 4, 5 \rangle$. Двете места, в които има увеличение с 2, отговарят на комбинирания, а увеличението с 1 отговаря на индивидуално минаване.

- 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 $\mathrm{opt}[5]$ взехме минимума на равни стойности. Можехме да вземем $c[4] + \mathrm{opt}[3]$ и това щеше да 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.

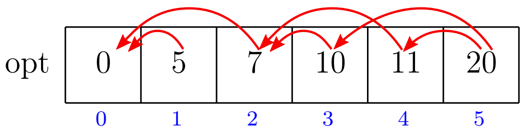
Наблюдение 57: Всички оптимални решения може да са прекалено много

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 ефикасно всички оптимални решения. В тази задача, такъв най-лош случай е $d[i] = 1$ за $1 \leqslant i \leqslant n$ и $c[i] = 2$ за $1 \leqslant i \leqslant n-1$. 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-2(d[1..n], c[1..n−1])
 1  създай масив от числа opt[0..n] и масив от индекси J[0..n]
 2  opt[0] ← 0, J[0] ← Nil
 3  opt[1] ← d[1], J[1] ← 0
 4  for i ← 2 to n
 5      if d[i] + opt[i−1] ⩽ c[i−1] + opt[i−2]
 6          opt[i] ← d[i] + opt[i−1]
 7          J[i] ← i−1
 8      else
 9          opt[i] ← c[i−1] + opt[i−2]
10          J[i] ← i−2
11  return (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 ефикасно изчислимо, но тук говорим концептуално, а не за генерирането му. Да си представим ориентиран граф G, в който върховете са клетките на $opt[0..n]$, а ребро $(opt[i], opt[k])$ има тстк стойността на $opt[i]$ се може да се изчисли чрез сума, в която участва $opt[k]$. Очевидно, $k \in \{i-1, i-2\}$, поради което G е даг. Също така, $opt[n]$ е източник, но може да има и други източници. За всяко $i \geqslant 1$, поне едно от ребрата $(opt[i], opt[i-1])$ и $(opt[i], opt[i-2])$ е налице, поради което връх $opt[0]$ е единственият сифон. Ключовото наблюдение е, че всеки максимален по включване път в G с начало $opt[n]$ (очевидно краят е $opt[0]$) 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 $opt[n]$ и край $opt[0]$ в G. Ето въпросният даг за примера, който видяхме преди малко:



И едно предупреждение. Следният алгоритъм за тежкия вариант на QUEUE PROCESSING е **НЕКОРЕКТЕН**. 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 $1 \leqslant i \leqslant n-1$. 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] ← d[1]
 3  if d[1] + d[2] < c[1]
 4      B[1] ← 0
 5      opt[2] ← d[1] + d[2]
 6  else
 7      B[1] ← 1
 8      opt[2] ← c[1]
 9  for i ← 3 to n
10      if d[i] + opt[i-1] < c[i-1] + opt[i-2]
11          B[i-1] ← 0
12          opt[i] ← d[i] + opt[i-1]
13      else
14          B[i-1] ← 1
15          opt[i] ← 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 съседи 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

$$F_n = \begin{cases} 0, & \text{ако } n = 0, \\ 1, & \text{ако } n = 1, \\ F_{n-1} + F_{n-2}, & \text{ако } n > 1 \end{cases}$$

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 1 836 311 903 пъти, а редът `fib(n-1) + fib(n-2);` се достига точно 1 836 311 902 пъти. С други думи, дървото на рекурсията има 1 836 311 903 листа и 1 836 311 902 вътрешни върхове. И всичко това е само за пресмятането на 46 различни стойности $F_0, \ldots, F_{45}$.

Следният фрагмент изчислява числа на Fibonacci, като помни последните досега изчислени стойности и не върши излишни изчисления отново и отново:

```
A[0] = 0;
A[1] = 1;
for (i = 2; i <= n;  i ++) {
  temp = A[0] + A[1];
  A[0] = A[1];
  A[1] = temp;
}
```

Изпълнението на този фрагмент за вход `47` е светкавично: 0.002 секунди.

12.2 Фундамент

И при пресмятането на минималното време за минаване на опашката през касата, и при пресмятането на числата на Fibonacci срещаме един и същи феномен: най-общо казано, задачата се свежда до *подзадачи*.

Конвенция 13: Задачи, подзадачи, подподзадачи

В Лекция 12 често се говори за някаква *задача* и нейните *подзадачи* и *подподзадачи*. Това са директни преводи на често срещаните английски термини *problem*, *subproblems* и *subsubproblems*, които се ползват често за описание на същността на схемата **Динамично Програмиране**.

Тази терминология обаче е формално несъвместима с Определение 1, според което "задача" е двуместна релация $\Pi \subseteq \mathcal{I} \times \mathcal{S}$. Формално, "подзачачи" би трябвало да са някакви подмножества на Π. А те не са.

Всъщност, когато говорим за *задача* Π, нейни *подзадачи* и техни *подподзадачи* в Лекция 12, имаме предвид съответно екземпляр x на Pi, други екземпляри $y_1, \ldots, y_k$, които са конституенти на x в някакъв смисъл (това са подзадачите) и за всеки y_i, екземпляри $z_{i,1}, \ldots, z_{i,t_i}$, които са конституенти на y_i в някакъв смисъл (това са подподзадачите).

Това обаче не е достатъчно, за да конструираме алгоритъм по схемата **Динамично Програмиране**. Само свеждането на задачата до подзадачи ни дава алгоритъм по схемата **Разделяй-и-Владей**. Както се убедихме, такъв алгоритъм за задачата с опашката за билети има сложност по време $\Theta(\phi^n)$.

Съществено за ефикасността е, че подзадачите **не са независими**, а имат общи подподзадачи. Примерно, в задачата с опашката за билети, решението за цялата опашка $h_1, \ldots, h_n$ се получава бързо от

- решение за подопашката $h_1, \ldots, h_{n-1}$, отговарящо на възможността h_n да не се комбинира с h_{n-1}, и
- решение за подопашката $h_1, \ldots, h_{n-2}$, отговарящо на възможността h_n да се комбинира с h_{n-1}.

Но подопашката $h_1, \ldots, h_{n-2}$ е конституент на подопашката $h_1, \ldots, h_{n-1}$, така че въпросните две решения не са независими. Решението за подопашката $h_1, \ldots, h_{n-1}$ ползва по удачен, ефикасен начин решението за подопашката $h_1, \ldots, h_{n-2}$.

Това е ключово! Въз основа на това наблюдение лесно може да измислим алгоритъм, който не е рекурсивен, а работи отдолу-нагоре и съхранява решенията за общите подподзадачи, при което отпада преизчисляването на едни и същи неща отново и отново и отново.

Същността на схемата **Динамично Програмиране** е ефикасно пресмятане на рекурсия. Всяка рекурсия има свое дърво на рекурсията. Да мислим за дървото на рекурсията като анти-арборесценция. Всеки връх на дървото е асоцииран със *състояние*, което е съответният вход на алгоритъма[†]. Ако идентифицираме върховете с еднакви състояния, получаваме даг, чиито източници са върховете с началните условия. Този даг има размер, който е драстично по-малък от размера на дървото. Или поне в примерите, които ще разгледаме, е точно така. Алгоритмичното решение за дадения вход се състои в топо-сортиране на дага и генериране на решение при обхождане на дага в реда на топо-сортировката. Разбира се, ние няма да строим дага експлицитно, нито ще го топо-сортираме експлицитно, но е много удобна абстракция да мислим за изчислението по този начин.

Допълнение 47: Не всеки Разделяй-и-Владей води до Дин. Прогр.

Има много примери, в които Разделяй-и-Владей не води до ефикасен алгоритъм, изграден по схемата Динамично Програмиране, защото подзадачите от фазата **Разделяй** нямат общи подподзадачи.

Последното е напълно вярно за MERGE SORT, защото сортиранията на двата подмасива нямат нищо общо, но MERGE SORT е ефикасен алгоритъм, така че този пример не е особено удачен. Удачен пример е задачата за местенията на Ханойските кули, ако бъде формулирана по подходящ начин: при дадени n диска да се генерира редицата от минимален брой ходове, която ги премества върху крайния 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 **редицата от ходовете**. Каква е сложността по време? Ако не отчитаме представянето на n, сложността по време не е дефинирана[a]. Ако n е представено в двоична позиционна бройна система, сложността е двойна експонента. Ако

[†]В примера с рекурсивното изчисление на числата на Fibonacci, ако пресмятаме F_{45}, състоянието на корена е 45, двете му деца имат състояния съответно 44 и 43, първото на свой ред имат две деца със състояния 43 и 42, а второто има две деца със състояния 42 и 41, и така нататък. Всяко листо има състояние или 1, или 0.

n е представено унарно, сложността е $\Theta(2^n)$, което е единична експонента. 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, 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. И наистина, алгоритъмът не е по схемата **Динамично Програмиране**, а е чист **Разделяй-и-Владей** алгоритъм с експоненциална сложност по време, при казаните допускания.

[a]Вижте Подсекция 2.2.5.

Заслужава да се прочете и мнението на Steve Skiena за схемата **Динамично Програмиране** [133, 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 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, примерно).

И още нещо. Както казва Skiena, трябва значителна практика, за да може дизайнерът на алгоритми да интернализира тази техника и да я прилага успешно.

Допълнение 48: За произхода на "Динамично Програмиране"

Интересен стрáничен 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 [11, 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", за разлика от съвременния смисъл.

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.

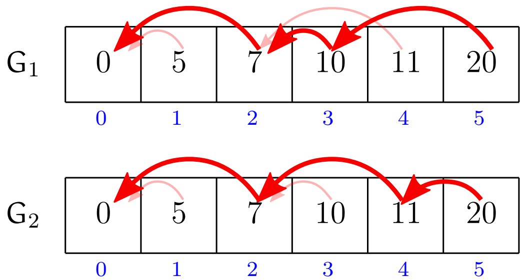
1. Алгоритъмът за олекотения вариант неизбежно ползва таблица за съхранение на междинни резултати.

[†]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.

2. Във всяка клетка c 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 зацикли. 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. 472 **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 част от оптиманлото 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 12.4, няма олекотен и тежък вариант и не се налага да генерираме оптимално решение с обратно проследяване.

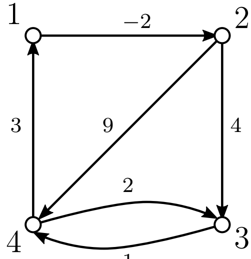
12.3 Най-къс път за всяка двойка върхове

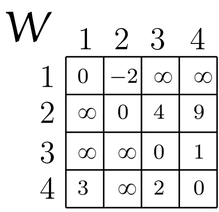
Разглеждаме задачата за най-късите пътища във варианта APShP: за всеки два върха u и v в тегловен ориентиран граф G, търсим най-къс път $u \rightsquigarrow v$. Това можем да постигнем, викайки от всеки връх алгоритъм за задачата във варианта SSShP (DIJKSTRA или BELLMAN-FORD). Но има интересен алгоритъм, специализиран 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. $G = (\{1, \ldots, n\}, E)$ 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 произволни 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 отиване от i в i е нула, а цената от i в j за $i \neq j$ е, ако няма ребро (i, j), безкрайност, иначе е цената на реброто. 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 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 тегло на най-къс път $i \rightsquigarrow j$.

12.3.1 Алгоритъм, реализиращ матрично умножение

Според [31, 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.

Говорейки подробно, дизайнът на всеки алгоритъм по схемата **Динамично Програмиране**, се състои от следните фази.

1. За всяко решение въвеждаме число, което наричаме *характеристика на решението*. Това ни позволява да говорим за решения с по-голяма или по-малка характеристика. В случая с Най-къс път искаме да характеризираме пътища. Задължително трябва да има решения с минимална характеристика. Те отговарят на дъното на рекурсията.

   Забележете, че характеристиката на решение е нещо съвършено различно от цената на решение, ако задачата е оптимизационна. Казано по друг начин, оптимално решение **не е непременно** решение с минимална характеристика. Примерно, ако характеристиката на решение на Най-къс път в тегловен граф е броят на ребрата, оптимално решение не е непременно път с едно ребро (става дума за различни върхове, така че не може да е с нула ребра).

2. По отношение на избраната характеристика, конструираме подходяща *рекурсивна декомпозиция* на оптималните решения: казваме кои са базовите оптимални решения и, най-важното, как да конструираме оптимални решения с по-голяма характеристика от оптимални решения с по-малка характеристика. В случая с Най-къс път, определяме кои са най-късите базови пътища (базовите са тези с минимална характеристика) и определяме как не-базов най-къс път с някаква характеристика се получава от друг(и) най-къс(и) път(ища) с по-малка характеристика.

   Забележете, че оптимизационната задача може да е минимизационна или максимизационна по отношение на цената. По отношение на характеристиката обаче винаги говорим

като за минимизационна задача: рекурсията строи окончателното решение от подрешения с по-малка характеристика, на свой ред тях от подподрешения с още по-малка характеристика, и така нататък до дъното на рекурсията.

Забележка 3

Рекурсията е по характеристиката! Ако задачата е оптимизационна, всяко решение така или иначе има цена (вижте Определение 4), която е число. Характеристиката на решението също е число, но в общия случай то е друго и **независимо от цената**.

3. След това конструираме ефикасен алгоритъм, който се базира на рекурсивната декомпозиция, но не работи рекурсивно, а отдолу-нагоре (bottom-up). С други думи, "изминава" само "тичането назад" (вижте Подсекция 1.1.7).

Първите две фази са математически. Третата фаза е алгоритмична. Ерго, самият алгоритъм се съставя в третата фаза, но първите две фази са решаващо важни. Ако не характеризираме смислено решенията на задачата и/или не намерим смислена рекурсивна декомпозиция на оптималните решения, няма да можем да конструираме алгоритъм.

Всичко това си заслужава да се повтори. Характеристика е нещо, присъщо на всяко решение на задачата, оптимално или не. Каква е точно характеристиката зависи от дизайнера на алгоритъма. Някои характеристики са по-удачни от други в смисъл, че водят до по-ефикасни алгоритми. Рекурсивната декомпозиция е по характеристиката, но е в сила само за оптималните решения: казваме как оптимални решения с по-голяма характеристика се строят от оптимални решения с по-малка характеристика. За решения, които не са оптимални, не мислим за рекурсивна декомпозиция изобщо. За да е ефикасна рекурсивната декомпозиция, трябва оптималните решения да се получават ефикасно от оптимални подрешения. Това е възможно за някои задачи и не е възможно за други задачи. Допълнение 43 показва, че при задачата за най-дългите пътища, ако характеристиката е броят на ребрата, то не е вярно, че оптимално решение (най-дълъг път) се състои от оптимални подрешения (най-дълги подпътища). За задачите, които са податливи на ефикасни алгоритмични решения по схемата **Динамично Програмиране**, е в сила следният принцип.

Определение 86: Принцип на оптималността

Дадена оптимизационна задача, по отношение на някаква характеристика, удовлетворява *принципа на оптималността*, ако подрешенията, по отношение на характеристиката, на всяко оптимално решение на свой ред са оптимални решения за съответните подекземпляри.

Най-къси пътища удовлетворява принципа на оптималността по отношение на характеристиката брой на ребрата, което доказахме в Теорема 62. Най-дълги пътища не удовлетворява принципа на оптималността по отношение на характеристиката брой на ребрата (Допълнение 43).

Заслужава да се прочете и мнението на Steve Skiena за принципа на оптималността [133, 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. Such would be the case with a form of edit distance where we are not allowed to use combinations of operations in certain particular orders. Properly formulated, however, many combinatorial problems respect the principle of optimality.*

Това, което Skiena нарича "state", е пълното описание на дадено решение, а не само характеристиката; за най-къс път, това е редицата от върховете на пътя. Задачата EDIT DISTANCE, която Skiena споменава, в тези лекционни записки разглеждаме в Подсекция 12.8.4.

Да се върнем на задачата APShP, която решаваме в момента. Да въведем следната характеристика на най-къс път $i \overset{p}{\leadsto} j$: p има някакъв брой ребра, но не повече от $n-1$. Това е числената характеристика, която разглеждаме в момента – броят на ребрата. Принципът на оптималността е в сила, така че можем да преминем от характеристиката към рекурсивната декомпозиция ето така.

Разглеждаме произволен най-къс път $i \leadsto j$. *Той има някакъв брой ребра. Да кажем, че има* t *ребра, където* $t \in \{0, \ldots, n-1\}$. *// това е тривиално ...*
Променяме леко формулировката: за някое t, *този най-къс път* $i \leadsto j$ *има не повече от* t *ребра.* *// очевидно – е, и?*
Продължаваме: всеки най-къс път $i \leadsto j$, *който има не повече от* t *ребра, където* $t \geqslant 1$, *се състои от най-къс път* $i \leadsto k$, *който има не повече от* $t-1$ *ребра, за някой връх* k, *плюс реброто* (k, j) *с тегло* $W[k, j]$. *// аха, това е!*

Ето подробната рекурсивна декомпозиция. Нека i и j са произволни, такива че $1 \leqslant i, j \leqslant n$.

- Дължината на най-къс път $i \leadsto j$ с не повече от 0 ребра е 0, ако $i = j$, или ∞, ако $i \neq j$. Това е базата на рекурсията.

- Дължината на най-къс път $i \leadsto j$ с не повече от t ребра е минимумът от тези две:
  1. дължината на най-къс път $i \leadsto j$ с не повече от $t-1$ ребра
  2. минимума от следните n на брой суми, по всички $k \in \{1, \ldots, n\}$:

     дължината на най-къс път $i \leadsto k$ с не повече от $t-1$ ребра плюс теглото $W[k, j]$ на реброто (k, j).

Ето схемата за изчисление, базирана на тази рекурсивна декомпозиция. За $t \in \{0, \ldots, n-1\}$, нека $D^{(t)}$ е $n \times n$ матрица от дължините на най-къси пътищата с не повече от t ребра. Тогава $D^{(t)}[i, j]$ дължината на най-къс път с не повече от t ребра от i до j. Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)}[i, j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases}$$

$$D^{(t)}[i, j] = \min\left\{D^{(t-1)}[i, j], \min\left\{D^{(t-1)}[i, k] + W[k, j] \mid 1 \leqslant k \leqslant n\right\}\right\}, \text{ за } t > 0$$

Това може да бъде опростено до

$$D^{(0)}[i,j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases} \tag{12.2}$$

$$D^{(t)}[i,j] = \min \{D^{(t-1)}[i,k] + W[k,j] \mid 1 \leqslant k \leqslant n\}, \text{ за } t > 0 \tag{12.3}$$

понеже $D^{(t-1)}[i,j]$ се среща в множеството $\{D^{(t-1)}[i,k] + W[k,j] \mid 1 \leqslant k \leqslant n\}$. За да се убедим в това, да видим, че при $k = j$, $\boxed{D^{(t-1)}[i,k] + W[k,j]}$ става $\boxed{D^{(t-1)}[i,j] + W[j,j]}$, което е $D^{(t-1)}[i,j]$, понеже $W[j,j] = 0$.

Решението е матрицата $D^{(n-1)}$, тъй като простите пътища не може да имат повече от $n-1$ ребра.

Изразите (12.2) и (12.3) са на практика **ефективен** рекурсивен алгоритъм: за всеки i и j, $D^{(n-1)}[i,j]$ се пресмята чрез рекурсивни викания на $D^{(n-2)}[i,k]$, за $1 \leqslant k \leqslant n$, а стойностите $W[k,j]$ 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 практически всички входове, които са интересни и възникват на практика, такъв алгоритъм е безполезен.

Но ние можем да направим друг алгоритъм, който работи съгласно (12.2) и (12.3), без ги следва буквално. Стойностите, които трябва да бъдат изчислени общо, не са необятно много. Общо всички матрици са $D^{(0)}$, $D^{(1)}$, ..., $D^{(n-1)}$, тоест, n матрици, всяка с n^2 клетки, така че, съгласно (12.2) и (12.3), трябва да сметнем n^3 стойности. 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 стойностите $D^{(298)}[1,2]$, ..., $D^{(298)}[1,300]$ ($D^{(298)}[1,1]$ е 0). На свой ред, пресмятането на всяка от $D^{(298)}[1,2]$, ..., $D^{(298)}[1,300]$ ползва $D^{(297)}[1,2]$, ..., $D^{(297)}[1,300]$. Това означава, че за да намерим $D^{(298)}[1,2]$, ние трябва да знаем $D^{(297)}[1,2]$, ..., $D^{(297)}[1,300]$. Дори да допуснем, че сме намерили $D^{(297)}[1,2]$, ..., $D^{(297)}[1,300]$ и после сме сметнали $D^{(298)}[1,2]$, когато започваме да пресмятаме $D^{(298)}[1,3]$, ще пресметнем всяко от $D^{(297)}[1,2]$, ..., $D^{(297)}[1,300]$ **отново**, ако следваме рекурсията в (12.3) буквално. Ако 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 от (12.2) и (12.3). Той вече 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.

Следният алгоритъм реализира тази идея. "MM" идва от "Matrix Multiplication".

Shortest Path MM(W: матрица $n \times n$, представяща тегловен ориентиран граф)

```
1  for i ← 1 to n
2      for j ← 1 to n
3          if i = j
4              D^(0)[i, j] ← 0
5          else
```

```
6              D^(0)[i,j] ← ∞
7   for t ← 1 to n − 1
8       for i ← 1 to n
9           for j ← 1 to n
10              D^(t)[i,j] ← ∞
11              for k ← 1 to n
12                  D^(t)[i,j] ← min {D^(t)[i,j], D^(t−1)[i,k] + W[k,j]}
13  return D^(n−1)
```

Коректността му е очевидна за читател, който/която е убеден/убедена в коректността на (12.2) и (12.3), 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 $i \in \{0, \ldots, n-1\}$, 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.

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 t ← 1 to n − 1
 for i ← 1 to n
  for j ← 1 to n
   D^(t)[i,j] ← ∞
   for k ← 1 to n
    D^(t)[i,j] ← min (D^(t)[i,j],
                      D^(t−1)[i,k] + W[k,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 t ← 1 to n − 1
 for i ← 1 to n
  for j ← 1 to n
   A^(t)[i,j] ← 0
   for k ← 1 to n
    A^(t)[i,j] ← + (A^(t)[i,j],
                    A^(t−1)[i,k] · W[k,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 полупръстен[†] $\mathbb{R} \cup \{\infty\}$ с операции min (абстрактно събиране) и събиране (абстрактно умножение), като абстрактната нула (идентитетът на абстрактното събиране) е ∞, а абстрактната единица (идентитетът на абстрактното умножение) е 0. Съответствията са подчертани чрез различни 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;

[†] 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.

3. абстрактното умножение е ляво и дясно дистрибутивно спрямо абстрактното събиране

$$a + \min(b, c) = \min((a + b), (a + c))$$
$$\min(a, b) + c = \min((a + c), (b + c))$$

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 [69, Глава 35].

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 правим така, използвайки $\Theta(m)$ матрични умножения:

$$\begin{aligned} A^2 &\leftarrow A \cdot A \\ A^3 &\leftarrow A^2 \cdot A \\ A^4 &\leftarrow A^3 \cdot A \\ &\dots \\ A^m &\leftarrow A^{m-1} \cdot A \end{aligned}$$

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 степенния 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 54) 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

$$W^{n-1} = W^{n} = W^{n+1} = W^{n+2} = \cdots$$

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 степенния 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

$$\lceil p \rceil \geqslant p, \quad \forall p \in \mathbb{R}$$
$$\lceil p \rceil - 1 < p, \forall p \in \mathbb{R}$$

имаме за $x > 0$

$$\lceil \lg x \rceil \geqslant \lg x \;\leftrightarrow\; 2^{\lceil \lg x \rceil} \geqslant 2^{\lg x} = x \;\leftrightarrow\; 2^{\lceil \lg x \rceil} \geqslant x$$
$$\lceil \lg x \rceil - 1 < \lg x \;\leftrightarrow\; 2^{\lceil \lg x \rceil - 1} < 2^{\lg x} = x \;\leftrightarrow\; 2^{\lceil \lg x \rceil - 1} < x$$

Заместваейки x с $n-1$, получаваме

$$2^{\lceil \lg (n-1) \rceil} \geqslant n-1$$
$$2^{\lceil \lg (n-1) \rceil - 1} < n-1$$

И така, търсената степен, на която е достатъчно да повдигнем W, за да получим желания 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. 482 започнахме от единичната матрица $D^{(0)}$ от съображения за елегантност. Тук започваме от W от съображения за ефикасност. Друга малка оптимизация, която може да се направи, е да не инициализираме $D^{(t)}[i,j]$ (която сега наричаме $\text{tmp}[i,j]$) с ∞, а директно с $W[i,1] + W[1,j]$, и да започнем най-вътрешния цикъл от $k = 2$; отново предпочитаме ефикасността пред теоретичната елегантност.

```
SHORTEST PATH MM, REPEATED SQUARING(W)
1  for t ← 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 k ← 2 to n
6                  tmp[i, j] ← min (tmp[i, j], W[i, k] + W[k, j])
7      W ← tmp
8  return W
```

Инвариант за най-външния цикъл е, всеки път, когато изпълнението е на ред 1, ако $\widehat{W}$

означава входната матрица, то $W = \widehat{W}^{2^{i-1}}$ за текущата W. Наистина, при първото достигане $i = 1$ и очевидно $W = \widehat{W}^{2^{1-1}} = \widehat{W}^{2^0} = \widehat{W}^1$ в този момент; при всяко следващото достигане степенният показател очевидно се идвоява; при последното достигане имаме $i = \lceil \lg(n-1) \rceil + 1$, така че $W = \widehat{W}^{2^{\lceil \lg(n-1) \rceil + 1 - 1}} = \widehat{W}^{2^{\lceil \lg(n-1) \rceil}}$ 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 $\Theta(n^2)$.

12.3.2 Алгоритъмът на Floyd-Warshall

Да характеризираме най-къс път $i \overset{p}{\rightsquigarrow} j$ по друг начин. Нека U е множеството от вътрешните върхове на p. Възможно е $U = \emptyset$. Върховете на графа са числата $1, \ldots, n$. Тогава p има максимален вътрешен връх. Примерно, ако $U = \{2, 20, 50\}$, максималният вътрешен връх е 50. А ако $U = \emptyset$, приемаме, че максималният вътрешен връх е 0. Това е характеристиката сега: максималният вътрешен връх в пътя.

Тази характеристика не е толкова тривиална, колкото характеристиката на стр. 481. Она-зи беше нещо очевидно, 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 нещо като субективизъм, 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.

Всеки най-къс път $i \overset{p}{\rightsquigarrow} j$ има множество от вътрешни върхове U, което може да е и празното множество. // това е тривиално ...
Нека $k = \max U$. Приемаме, че ако $U = \emptyset$, то $k = 0$ (а не $k = -\infty$). Очевидно е изпълнено $U \subseteq \{1, \ldots, k\}$. Тук k е точната горна граница на U. // е, и?
Променяме леко формулировката: за някое $k \in \{0, \ldots, n\}$ е изпълнено $U \subseteq \{1, \ldots, k\}$. С други думи, k става горна граница, не непременно точна, на U. // е, и?
Продължаваме: тогава p или съдържа k като междинен връх, или не го съдържа. В първия случай p се състои от най-къс път $i \overset{q}{\rightsquigarrow} k$, "слепен" в общия край k с най-къс път $k \overset{r}{\rightsquigarrow} j$, като q и r имат вътрешни върхове от множеството $\{1, \ldots, k-1\}$. Във втория случай множеството от вътрешните върхове на p е подмножество на $\{1, \ldots, k-1\}$. // аха, това е!

Ето подробната рекурсивна декомпозиция. Нека i и j са произволни, такива че $1 \leqslant i, j \leqslant n$.

- Дължината на най-къс път $i \rightsquigarrow j$, чието множество от вътрешни върхове е празното множество, е или 0, ако $i = j$, или $w((i, j))$, ако $i \neq j$ и $(i, j) \in E(G)$, или ∞, ако $i \neq j$ и $(i, j) \notin E(G)$. С други думи, ако няма вътрешни върхове, то дължината на най-къс път $i \rightsquigarrow j$ е точно $W[i, j]$.

[†]Под "различни номерации на един и същи граф" имаме предвид следното. Представете си, че поначало върховете на графа са с някакви абсолютни идентификатори, примерно $v_1, \ldots, v_n$, а после някаква биекция изобразява $\{v_1, \ldots, v_n\}$ в $\{1, \ldots, n\}$. 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".

- Нека $i \overset{p}{\rightsquigarrow} j$ е с минимално тегло и вътрешните върхове на p са от $\{1, \ldots, k\}$. Не се твърди, че всяко от тези числа се появява като (идентификатор на) вътрешен връх, а че това са разрешените стойности.
  - p може да съдържа връх $k \geqslant 1$ като вътрешен връх. Тогава $w(p)$ е сумата от дължината на най-къс път $i \rightsquigarrow k$, чиито вътрешни върхове са от $\{1, \ldots, k-1\}$, и дължината на най-къс път $k \rightsquigarrow j$, чиито вътрешни върхове са от $\{1, \ldots, k-1\}$. В този случай казваме, че k *помага.*
  - p може да не съдържа връх $k \geqslant 1$ като вътрешен връх. Тогава $w(p)$ е равна на дължината на най-къс път $i \rightsquigarrow j$, чиито вътрешни върхове са от $\{1, \ldots, k-1\}$. В този случай казваме, че k *не помага.*

Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)} \leftarrow W \tag{12.4}$$

$$D^{(k)}[i,j] \leftarrow \min\{\underbrace{D^{(k-1)}[i,j]}_{k \text{ не помага}}, \underbrace{D^{(k-1)}[i,k] + D^{(k-1)}[k,j]}_{k \text{ помага}}\}, \text{ за } k > 0 \tag{12.5}$$

Забележете алгоритмичното предимство на (12.4) и (12.5) пред (12.2) и (12.3)! В (12.4) и (12.5) се пресмята минимум на **2** числа, докато в (12.2) и (12.3) се пресмята минимум на $\mathbf{n}$ числа.

Естествено, ако имплементираме (12.4) и (12.5) директно, ще получим 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 предишния алгоритъм, ние няма да имплементираме (12.4) и (12.5) директно, а ще съобразим, че всъщност има само $\Theta(n^3)$ различни стойности, които се налага да изчислим, които са разпределени в матрици $D^{(0)}$, $D^{(1)}$, ..., $D^{(n)}$, всяка от които е $n \times n$ и се пресмята от предишната. Тези матрици ще запълваме отдолу-нагоре, точно както диктува схемата **Динамично Програмиране**.

Всичко това води до следния кубичен алгоритъм за задачата APShP, който е публикуван от Robert Floyd [45], но се базира на теорема на Stephen Warshall [143].

```
FLOYD-WARSHALL(W: матрица n × n, представяща тегловен ориентиран граф)
1  D^(0) ← W
2  for k ← 1 to n
3      for i ← 1 to n
4          for j ← 1 to n
5              D^(k)[i,j] ← 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 (12.4) и (12.5). 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 изполвайки 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 [31, стр. 699]. Да видим защо е така. Да кажем, че използваме само

една матрица D, която се инициализира с W. Същността на алгоритъма е

$$D[i,j] \leftarrow \min\left(D[i,j], D[i,k] + D[k,j]\right) \tag{12.6}$$

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 (12.6) става

$$D[i,j] \leftarrow \min\left(D[i,j], D[i,i] + D[i,j]\right) \tag{12.7}$$

Но $D[i,i]$ е задължително 0 при отсъствие на отрицателни цикли, така че (12.8) става

$$D[i,j] \leftarrow \min\left(D[i,j], D[i,j]\right) \tag{12.8}$$

което е същото като $D[i,j]$ да не се променя. Но това е коректно, защото дължината на най-къс път $i \leadsto j$, съдържащ вътрешен връх не по-голям от i (в момента сме допуснали, че $k = i$), е равна на дължината на най-къс път $i \leadsto j$, съдържащ вътрешен връх не по-голям от $i-1$, защото пътищата са задължително прости (Наблюдение 54) и връх i вече се появява като начало на пътя, ерго, не може да е вътрешен връх. Но $D[i,j]$ преди изпълнението на (12.7) е именно дължината на най-къс път $i \leadsto j$, съдържащ вътрешен връх не по-голям от $i-1$. С което решаваме задача 25.2-4 от [31, 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 [31, стр. 695], със следния смисъл на елементите: $\Pi[i,j]$ съдържа предшественика на връх j по най-къс път от връх i. Тъй като целият i-ти ред на Π описва най-къси пътища от връх i, можем да мислим за Π като за n масива на предшествие, каквито имахме в SSShP варианта, разположени в матрица. Иначе казано, i-ият ред на Π описва дърво на най-къси пътища с корен връх i.

Всъщност, алгоритъмът конструира редица от матрици $\Pi^{(0)}$, $\Pi^{(1)}$, ..., $\Pi^{(n)}$, 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 от връх i, който най-къс път има вътрешни върхове от $\{1,\ldots,k\}$. Ще опишем тези матрици рекурсивно.

$$\Pi^{(0)}[i,j] = \begin{cases} \text{Nil}, & \text{ако } i = j \text{ или } W[i,j] = \infty \\ i, & \text{ако } i \neq j \text{ и } W[i,j] < \infty \end{cases}$$

$$\Pi^{(k)}[i,j] = \begin{cases} \Pi^{(k-1)}[i,j], & \text{ако } D^{(k-1)}[i,j] \leqslant D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \quad // k \text{ не помага} \\ \Pi^{(k-1)}[k,j], & \text{ако } D^{(k-1)}[i,j] > D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \quad // k \text{ помага} \end{cases}, \text{ за } k > 0$$

Следва псевдокод на FLOYD-WARSHALL, който пресмята и пътища.

```
Floyd-Warshall paths(W: матрица n × n, представяща тегловен ориентиран граф)
 1  for i ← 1 to n
 2      for j ← 1 to n
 3          D^(0)[i,j] ← W[i,j]
 4          if i = j or W[i,j] = ∞
 5              Π^(0)[i,j] ← Nil
 6          else
 7              Π^(0)[i,j] ← 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] ⩽ D^(k−1)[i,k] + D^(k−1)[k,j]
12                  D^(k)[i,j] ← D^(k−1)[i,j]
13                  Π^(k)[i,j] ← Π^(k−1)[i,j]
14              else
15                  D^(k)[i,j] ← D^(k−1)[i,k] + D^(k−1)[k,j]
16                  Π^(k)[i,j] ← Π^(k−1)[k,j]
17  return (D^(n), Π^(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(Π, i, j)
1  if i = j
2      print i
3  else
4      if Π[i,j] = Nil
5          print no path
6      else
7          Print APShP(Π, i, Π[i,j])
8          print j
```

Пример за работата на Floyd-Warshall paths има в [31, стр. 696]. Там са показани всички матрици $D^{(0)}$, $\Pi^{(0)}$, ..., $D^{(n)}$, $\Pi^{(n)}$, които се получават при работата на алгоритъма върху графа на стр. 690.

12.4 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.

По същество това са комбинаторни задачи за мощности на множества. Тези задачи може да бъдат формулирани като задачи за съставяне на алгоритми, връщащи числа и изградени по схемата **Динамично Програмиране**, но в същината си това са комбинаторни задачи.

Една забележка, отнасяща се до всички тези алгоритми. При тях входът се състои от две числа n и k, като $0 \leqslant k \leqslant n$. Както винаги, когато входът се състои от константен брой

числа, възниква въпросът как дефинираме сложността. Очевидно не в големината на входа, която е $\Theta(1)$[†]. В случая "ефикасен" означава "приемливо бърз на практика за разумно големи стойности на n и k". Това да съдържа много субективизъм и е неформално, но не е безсмислица.

Формално прецизно би било да започнем да разглеждаме **размерите на кодирането на n и k**, но ние няма да правим това; ако мерим с този аршин и n и k са кодирани бинарно, алгоритмите биха били експоненциални.

Друга възможност е да мерим с големината на n; в такъв случаи тези алгоритми са със сложност $O(n^2)$. Това е все едно да допуснем, че n е кодирано унарно.

12.4.1 Биномни коефициенти

Задача 47: Брой на подмножества

Конструирайте ефикасен алгоритъм, който по дадени n и k връща броя на k-елементните подмножества на n-елементно множество, където $k, n \in \mathbb{N}$.

И така, иска се да се сметне $\binom{n}{k}$. Добре известно е, че

$$\binom{n}{k} = \binom{n}{n-k} = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-k+1)}{k!} \tag{12.9}$$

така че биномния коефициент може да бъде изчислен, като първо намерим кое от k и $n-k$ е не по-голямо от другото и после го ползваме в тази формула.

Има обаче и друг начин да се изчисли биномните коефициенти, който по същество е алгоритъм по схемата **Динамично Програмиране**. Следната формула е добре известна.

$$\binom{n}{k} = \begin{cases} 0, & \text{ако } k > n \\ 1, & \text{ако } k = 0 \text{ или } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{ако } 0 < k < n \end{cases} \tag{12.10}$$

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.

(12.10) 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, (12.10) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

[†]Това прави сложността по време недефинирана или безкрайност; сравнете с Подсекция 2.2.5.

n \ k	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 над него (56) и числото над него вляво (28).

Нулите над диагонала са само с теоретично значение. На практика няма никакъв смисъл да се запълва областта над диагонала.

Дали е смислено да се изчислява биномния коефициент по този начин на практика? За еднократно изчисляване, за каквото става дума в Задача 47, не. За еднократно изчисление е по-добре да се ползва формулата от (12.9). Ако обаче ни се налага да пресмятаме много биномни коефициенти, може да има смисъл да се създаде таблица като Таблица 12.1 с подходящи размери и после биномните коефициенти да се вземат от нея.

Допълнение 49: Имплементацията на `binomial(n,k)` в Maple(TM)

Защо на практика е по-добре да биномният коефициент да се пресмята като дроб посредством (12.9), а не чрез таблица като Таблица 12.1? Това е очевидно от общи съображения дори без детайлен анализ, но има и друг силен аргумент: имплементацията на функцията `binomial(n,k)`, която пресмята $\binom{n}{k}$, в средата Maple(TM). Тъй като Maple е правен от корифеи в областта на математическия софтуер, може спокойно да вярваме, че тяхната имплементация е най-доброто, което може да се направи на практика.
А как да разберем как е имплементирана тази функция в Maple? Maple е писан на C и на Maple, като има ядро на C, което осигурява достатъчно изчислителни примитиви, и около него има богата и разнообразна обвивка от функции, написани на езика на Maple. Потребителят може да види сорса (написан на Maple) на такава функция чрез командата `eval`, ако преди това нивото на "вербозност" е било повишено на 2. Ето пълен екранен print-out от Maple сесия, показваща сорса на `binomial`.

```
"C:\Program Files\Maple 2018\bin.X86_64_WINDOWS\cmaple"
    |\^/|     Maple 2018 (X86 64 WINDOWS)
```

[†]По отношение на Таблица 12.1, изчислението, визуално говорейки, е отгоре надолу, но правилният термин е "bottom-up" въпреки това.

```
._|\|   |/|_. 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.
> interface(verboseproc = 2):
> eval(binomial);
proc(a::algebraic, b::algebraic)
local res, s, c, i, B, AmB, A, fl_res;
global `binomial/a`, `binomial/b`, `binomial/c`;
option builtin = HFloat_binomial, `Copyright (c) 1996 Waterloo Maple Inc. All rights reserved.`;
    if nargs <> 2 then error "expecting two arguments but got %1", nargs end if;
    if type(a, 'complex(float)') or type(b, 'complex(float)') then
        fl_res := evalf('binomial(a, b)'); if type(fl_res, 'complex(float)') then return fl_res end if
    end if;
    B := normal(b);
    if `tools/type`(B, 0) then res := 1
    else
        AmB := normal(a - B);
        if `tools/type`(AmB, 0) then res := 1
        elif `tools/type`(AmB, -1) then
            try s := is(B = 0) catch: s := FAIL end try;
            if s = true then res := 1 elif s = false and coulditbe(B = 0) = false then res := 0 end if
        end if
    end if;
    if not assigned(res) then
        A := normal(a);
        if type(A, 'infinity') or type(B, 'infinity') then res := 'binomial'(a, b)
        elif type(A, 'undefined') then
            res := NumericTools:-ThrowUndefined(a, 'preserve' = `if`(type(B, 'numeric'), 'real', 'none'))
        elif type(B, 'undefined') then
            res := NumericTools:-ThrowUndefined(b, 'preserve' = `if`(type(a, 'numeric'), 'real', 'none'))
        elif hastype([A, B], 'undefined') then res := 'binomial'(a, b)
        elif `tools/type`(B - 1, 0) or `tools/type`(AmB - 1, 0) then
            if `tools/type`(A, 'negint') then return NumericEvent('invalid_operation', A) else res := A end if
        elif `tools/type`(A, 0) then res := piecewise(b = 0, 1, sin(Pi*b)/(Pi*b))
        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;
    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 **i**nteger **quo**tient, 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 (12.9), последвано от деление. Временният резултат `(a - i + 1)*c` се дели на `i` винаги, така че няма загуба на прецизност заради целочисленото деление.

12.4.2 Числа на Stirling от втори род

Нотация 10: "$\left\{{n \atop k}\right\}$" означава числото на Stirling от втори род n-подмножество-k

Числото на Stirling от втори род n-подмножество-k бележим с $\left\{{n \atop k}\right\}$.

Задача 48: 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 $\left\{ n \atop k \right\}$. За числата на Stirling от втори род е известна следната формула:

$$\left\{ n \atop k \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^j \binom{k}{j} (k-j)^n \tag{12.11}$$

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.

$$\left\{ n \atop k \right\} = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ k\left\{ n-1 \atop k \right\} + \left\{ n-1 \atop k-1 \right\}, & \text{ако } 0 < k < n \end{cases} \tag{12.12}$$

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 $\left\{ n-1 \atop k-1 \right\}$, 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 $\left\{ n-1 \atop k \right\}$; има k възможности за това, къде да го "присъединим".

(12.12) 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, (12.12) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

n \ k	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	1 701	1 050	266	28	1	0	0
9	0	1	255	3 025	7 770	6 951	2 646	462	36	1	0
10	0	1	511	9 330	34 105	42 525	22 827	5 880	750	45	1

Таблица 12.2: Таблица за изчисляване на числа на Stirling от втори род.

12.4.3 Числа на Stirling от първи род

Нотация 11: "$\left[{n \atop k}\right]$" означава числото на Stirling от първи род n-цикъл-k

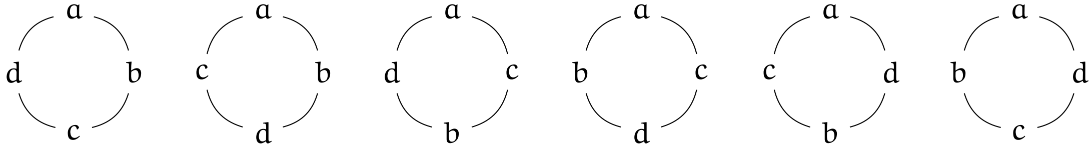
Числото на Stirling от първи род n-цикъл-k бележим с $\left[{n \atop k}\right]$.

Задача 49: 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 87: 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 $\left[{n \atop k}\right]$.

Ясно е, че в общия случай $\left[{n \atop k}\right]$ е много по-голямо от $\left\{{n \atop k}\right\}$. Примерно, $\left\{{4 \atop 1}\right\} = 1$, защото начините да разбием 4-елементно множество на един дял са само един (начин). Докато, както вече

видяхме, $\left[{4 \atop 1}\right] = 6$. Лесно се обобщава, че $\left[{n \atop 1}\right] = (n-1)!$ за всяко $n > 0$, докато $\left\{{n \atop 1}\right\} = 1$ за всяко $n > 0$.

За числата на Stirling от първи род е известна формулата [29, стр. 51]:

$$\begin{bmatrix} n \\ k \end{bmatrix} = \sum_{0 \leqslant j \leqslant h \leqslant n-k} (-1)^{j+h} \binom{h}{j} \binom{n-1+h}{n-k+h} \binom{2n-k}{n-k-h} \frac{(h-j)^{n-k+h}}{h!} \tag{12.13}$$

Нейното извеждане обаче е много по-сложно от (12.9) или (12.11).

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.

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}, & \text{ако } 0 < k < n \end{cases} \tag{12.14}$$

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 располагания в цикли е 1, дори когато множеството е празното множество, а броят на циклите е (съответно) нула;

- $k = 0$, $n > 0$ означава располагания 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 располаганията

- на тези, в които той е сам в цикъл, а те са $\left[{n-1 \atop k-1}\right]$, защото те са колкото располаганията на $(n-1)$-елементно множество в $k-1$ цикъла;

- и на тези, в които той бива "вмъкнат" в някой вече съществуващ цикъл на располагане $(n-1)$-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 $(n-1)$.

(12.14) 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, (12.14) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

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 (12.9) и (12.11), съответно, отколкото чрез алгоритмите по схемата **Динамично Програмиране**, основани съответно на (12.10) и (12.12),

- то числата на Stirling от първи род се пресмятат по-бързо чрез алгоритъм по схемата **Динамично Програмиране**, основан на (12.14), отколкото чрез засуканата формула (12.13).

$n \backslash k$	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	1 764	1 624	735	175	21	1	0	0	0
8	0	5 040	13 068	13 132	6 769	1 960	322	28	1	0	0
9	0	40 320	109 584	118 124	67 284	22 449	4 536	546	36	1	0
10	0	362 880	1 026 576	1 172 700	723 680	269 325	63 273	9 450	870	45	1

Таблица 12.3: Таблица за изчисляване на числа на Stirling от първи род.

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.

12.4.4 Брой на целочислени разбивания

Задача 50: 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 35. *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 отношения 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 [87, стр. 399, (38)].

Нотация 12: “$\left|{n \atop k}\right|$” означава броя на целочислените разбивания на n на k части

Броят на целочислените разбивания на n на точно k части бележим с $\left|{n \atop k}\right|$.

Както става ясно от примера на стр. 189, $\left|{4 \atop 0}\right| = 0$, $\left|{4 \atop 1}\right| = 1$, $\left|{4 \atop 2}\right| = 2$, $\left|{4 \atop 3}\right| = 1$ и $\left|{4 \atop 4}\right| = 1$. Очевидно $\sum_{k=0}^{n} \left|{n \atop k}\right| = p(n)$.

За броя на целочислените разбивания е известна следната формула [87, стр. 399, (39)], която

задава и алгоритъм по схемата **Динамично Програмиране** за тяхното изчисляване.

$$\begin{vmatrix} n \\ k \end{vmatrix} = \begin{cases} 1, & \text{ако } k = n \\ 0, & \text{ако } (k = 0 \text{ и } n > 0) \text{ или } n < k \\ \begin{vmatrix} n-k \\ k \end{vmatrix} + \begin{vmatrix} n-1 \\ k-1 \end{vmatrix}, & \text{ако } 0 < k < n \end{cases} \tag{12.15}$$

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 при $0 < k < n$ е следната. 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 $\begin{vmatrix} n-1 \\ k-1 \end{vmatrix}$, 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 $\begin{vmatrix} n-k \\ k \end{vmatrix}$. 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 88: Диаграма на Ferrers

При дадени естествени n и k, такива че $1 \leqslant k \leqslant n$, на всяко целочислено разбиване $\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_k$ 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

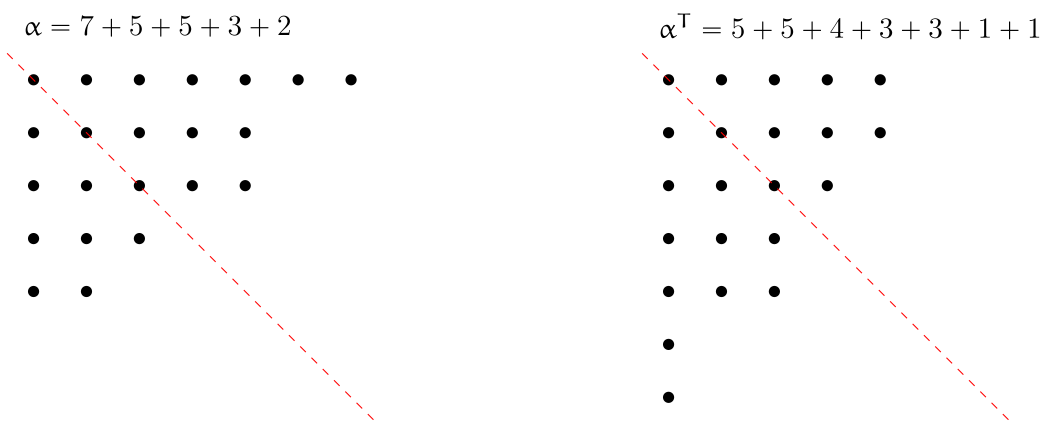
• • • • • • •
• • • • •
• • • • •
• • •
• •

Диаграмите на Ferrers са много удобно средство за онагледяване на целочислени разбивания. Да си припомним формулата $\left|\begin{matrix}n\\k\end{matrix}\right| = \left|\begin{matrix}n-k\\k\end{matrix}\right| + \left|\begin{matrix}n-1\\k-1\end{matrix}\right|$ при $0 < k < n$. Както вече бе отбелязано, събираемото $\left|\begin{matrix}n-k\\k\end{matrix}\right|$ 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* [87, 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 70). 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 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 челочислените разбивания на n, чиято най-голяма част е точно k.

Наблюдение 58

$\left|{n \atop k}\right|$ може алтернативно да се дефинира като броят на целочислените разбивания на n, чиято най-голяма част е точно k.

(12.15) 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, (12.15) тривиално задава алгоритъм по схемата **Динамично Програмиране**.

$n \backslash k$	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 (12.10) и числата на Stirling (уравненения 12.12 и 12.14) 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 12.15), 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 $\left|{n \atop k}\right|$ с Maple, може да го направим с `numelems(partition(n,k)) - numelems(partiton(n,k-1))`.

12.5 Задачи за оптимални скобувания на редици (Динамично Програмиране с триъгълна таблица)

В тази секция ще разгледаме задачи, във всяка от които присъства някаква редица, за която търсим оптимална декомпозиция, **условно** наречена "скобуване". Редицата не се мени по време на работата на алгоритъма.

Да си припомним, че според цитата на Skiena на стр. 476, "Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order". Наистина, всички тези задачи се решават ефикасно с алгоритми по схемата **Динамично Програмиране** с двумерна таблица. В таблицата обаче се полза (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 два непразни подредици, и така нататък, докато не бъдат достигнати началните условия.

12.5.1 Matrix-Chain Multiplication

Започваме с пример. Трябва да извършим матричното умножение $\mathsf{A} \cdot \mathsf{B} \cdot \mathsf{C} \cdot \mathsf{D}$, използвайки стандартния алгоритъм за умножение на матрици:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_{\mathsf{A}} \cdot \underbrace{\begin{bmatrix} 12 & -3 \\ -1 & 1 \\ 8 & 0 \\ -6 & 9 \\ 5 & 11 \end{bmatrix}}_{\mathsf{B}} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_{\mathsf{C}} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{\mathsf{D}} \tag{12.16}$$

Един възможен начин да извършим това е:

$$\mathsf{A}\cdot\mathsf{B}\cdot\mathsf{C}\cdot\mathsf{D} = \underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(\mathsf{A}\cdot\mathsf{B})} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_{\mathsf{C}} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{\mathsf{D}} =$$

$$\underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{((\mathsf{A}\cdot\mathsf{B})\cdot\mathsf{C})} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{\mathsf{D}} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{(((\mathsf{A}\cdot\mathsf{B})\cdot\mathsf{C})\cdot\mathsf{D})}$$

Да видим колко скаларни умножения са извършени при това матрично умножение. Ако умножим матрици $\mathsf{M}_{x\times y} \cdot \mathsf{N}_{y\times z}$, ние извършваме $x \cdot y \cdot z$ скаларни умножение. В случая, A е 2×5, B е 5×2, така че матричното умножение $(\mathsf{A} \cdot \mathsf{B})$ "струва" $2 \cdot 5 \cdot 2 = 20$ скаларни умножения. $(\mathsf{A} \cdot \mathsf{B})$ е 2×2, а C е 2×6, така че матричното умножение $((\mathsf{A} \cdot \mathsf{B}) \cdot \mathsf{C})$ "струва" $2 \cdot 2 \cdot 6 = 24$ скаларни умножения. $((\mathsf{A} \cdot \mathsf{B}) \cdot \mathsf{C})$ е 2×6, а D е 6×2, така че матричното умножение $(((\mathsf{A} \cdot \mathsf{B}) \cdot \mathsf{C}) \cdot \mathsf{D})$ "струва" $2 \cdot 6 \cdot 2 = 24$ скларни умножения. Като цяло, умножението на четирите матрици по този начин "струва" $20 + 24 + 24 = 68$ скаларни умножения.

Друг начин да умножим матриците е

$$\underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(\mathsf{A}\cdot\mathsf{B})} \cdot \underbrace{\begin{bmatrix} 30 & -328 \\ -56 & -44 \end{bmatrix}}_{(\mathsf{C}\cdot\mathsf{D})} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{((\mathsf{A}\cdot\mathsf{B})\cdot(\mathsf{C}\cdot\mathsf{D}))}$$

Крайният резултат е същият. Това не е изненада, понеже умножението на матрици е асоциативно. Но да видим колко скаларни умножения ни струва този начин на умножение на матриците. $(\mathsf{A} \cdot \mathsf{B})$ струва, както вече видяхме, 20 скаларни умножение. $(\mathsf{C} \cdot \mathsf{D})$ струва $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. $((\mathsf{A} \cdot \mathsf{B}) \cdot (\mathsf{C} \cdot \mathsf{D}))$ струва $2 \cdot 2 \cdot 2 = 8$ скаларни умножения. Като цяло, броят на скаларните умножения е $20 + 24 + 8 = 52$. Което 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 скаларни умножения са извършени. B е 5×2 и C е 2×6, така че $(B \cdot C)$ струва $5 \cdot 2 \cdot 6 = 60$ скаларни умножения. $(A \cdot (B \cdot C))$ струва $2 \cdot 5 \cdot 6 = 60$ скаларни умножения. $((A \cdot (B \cdot C)) \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. Като цяло, броят на скларните умножения е $60 + 60 + 24 = 144$.

Идентифицираме сложността на умножението на веригата от матрици с броя на скаларните 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, но алгоритмично (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 скаларните 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 50: 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 израза при $n = 3$:

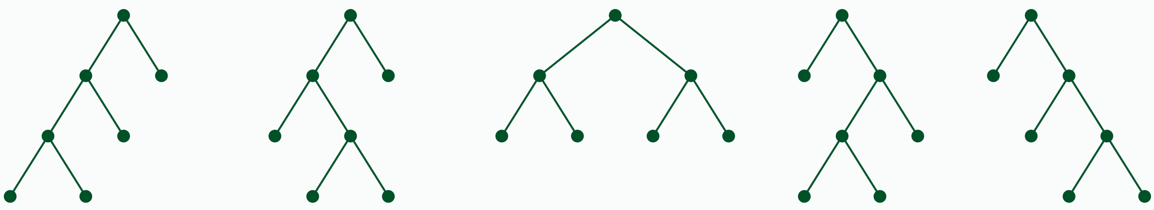
((()))   ()(())   ()()()   (())()   (()())

Забележете, че това не е същото като скобуване на израз с n множители. Тук единствените букви са скобите – множители няма.

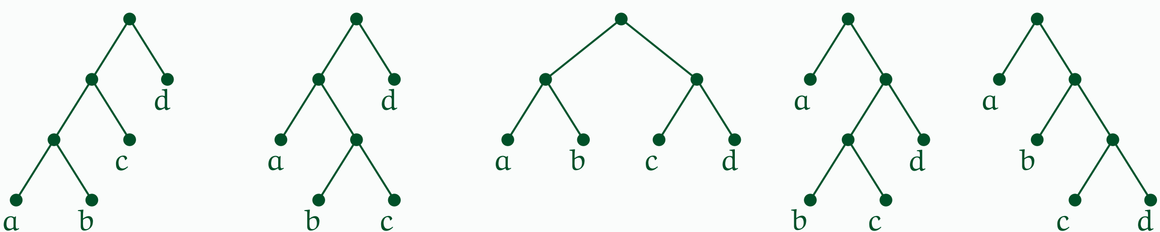
Лема 58

Съществува биекция между скобуваните изрази с $n + 1$ множители и балансираните изрази с $2n$ скоби.

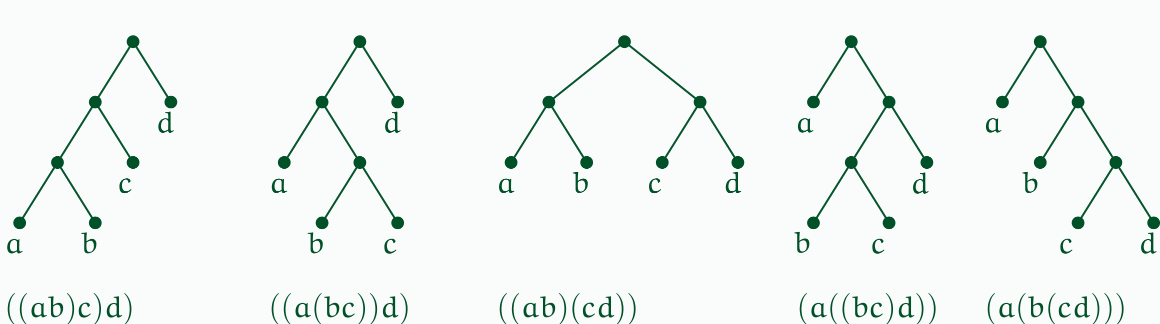
Доказателство. Първо ще покажем с пример, че съществува биекция между скобуваните изрази с $n + 1$ множители и пълните подредени двоични дървета с $n + 1$ листа. Нека $n = 3$. Тогава $n + 1 = 4$ и дърветата с анонимни върхове са тези:



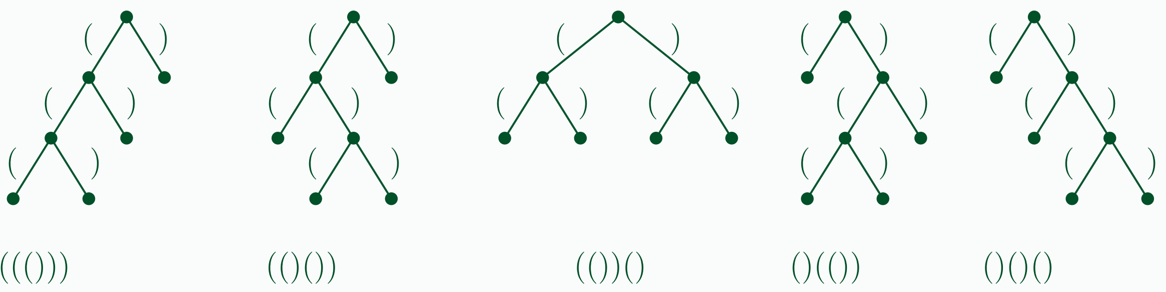
Да сложим имена на листата a, b, c и d, така че при обхождане в preorder (или postorder, няма значение), те се появяват в този ред:



Ето и съответните скобувани изрази:



Показахме 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 58. □

Лема 59

Редицата от числата на Catalan е решението на следното рекурентно уравнение:

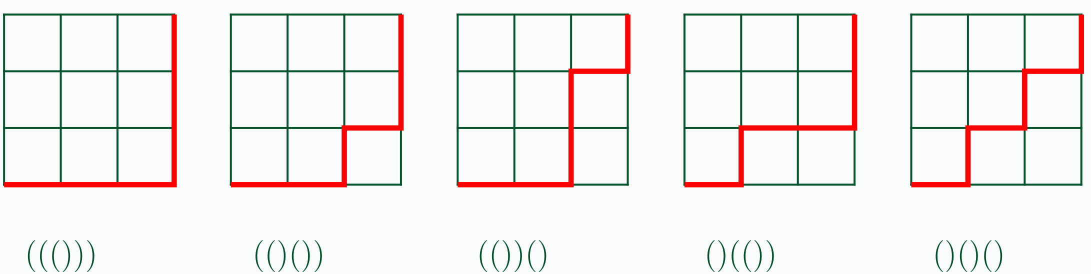
$$C_n = \begin{cases} 1, & \text{ако } n = 0, \\ \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i}, & \text{ако } n > 0 \end{cases} \tag{12.17}$$

Доказателство. Ще покажем с индукция по n, че (12.17) брои скобуваните изрази с $n+1$ множители. Но съгласно Лема 58 те са колкото балансираните изрази с $2n$ скоби, откъдето желаният резултат следва веднага.
Ако $n = 0$, (12.17) дава 1, а добре скобуваният израз с $0+1$ множителя е само един: (a), ако игнорираме идентичността на множителя. ✓

Ако $n \geqslant 1$, става дума за $n+1$ множители. В линейната наредба между тях има $(n+1)-1 = n$ "празнини", във всяка от които може да попада 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 (12.17). □

Рекурентното уравнение 12.17 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?

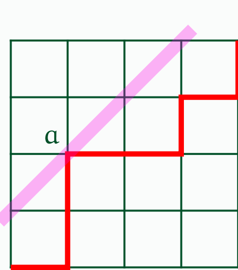
Ще изведем решение на уравнение 12.17 с елементарни комбинаторни съображения. Това уравнение брои балансираните изрази с $2n$ скоби. Ако отпадне изискването да са балансирани и остане само това, че са n скоби от видá "(" и още n скоби от видá ")", ние знаем колко са тези изрази: $\binom{n+n}{n} = \binom{2n}{n}$. 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:



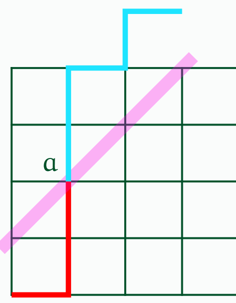
((())) (()()) (())() ()(()) ()()()

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

$$\binom{2n}{n} - \binom{n-1+n+1}{n-1} = \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!} = \frac{(2n)!}{n!(n-1)!}\left(\frac{1}{n} - \frac{1}{n+1}\right) = \frac{(2n)!}{n!(n-1)!}\left(\frac{n+1-n}{n(n+1)}\right) = \frac{(2n)!}{n!n!}\left(\frac{1}{n+1}\right) = \frac{1}{n+1}\binom{2n}{n}$$

Показахме, че $C_n = \frac{1}{n+1}\binom{2n}{n}$. Това е решението на (12.17).

Да видим асимптотиката на C_n. От Теорема 23 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

$$A_1 \cdot A_2 \cdot \cdots \cdot A_n$$

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.

Изч. Задача 29: 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 $1 \leqslant i \leqslant n$. 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 $A_1 \cdot A_2 \cdot \cdots \cdot A_n$.

Олекотеният вариант. Първо ще характеризираме оптимално решение на задачата. Всяко решение, оптимално или не, има място, на което се извършва последното матрично умножение (в Допълнение 50 използвахме термина "празнина"). Формално, съществува $k \in \{1, \ldots, n-1\}$, такова че последното матрично умножение е $B \times C$, където $B = A_1 \cdot A_2 \cdot \cdots \cdot A_k$ и $C = A_{k+1} \cdot A_{k+2} \cdot \cdots \cdot A_n$. Визуално е добре да си представим последното матрично умножение с нещо като разделител, поставен между A_k и A_{k+1}:

$$\underbrace{(A_1 \cdot A_2 \cdot \cdots \cdot A_k)}_{B} \mid \underbrace{(A_{k+1} \cdot A_{k+2} \cdot \cdots \cdot A_n)}_{C}$$

За тази характеризация ще направим следната рекурсивна декомпозиция. Броят на скаларните умножения за цялото матрично умножение е сумата от следните три събираеми:

- броят на скаларните умножения в матричното произведение $A_1 \cdot A_2 \cdot \cdots \cdot A_k$, тоест, за конструиране на матрицата B;
- броят на скаларните умножения в матричното произведение $A_{k+1} \cdot A_{k+2} \cdot \cdots \cdot A_n$, тоест, за конструиране на матрицата C;
- броят на скаларните умножения за матричното умножение $B \cdot C$.

Разглеждаме сумите за всички $k \in \{1, \ldots, n-1\}$ и вземаме минималната.

Тъй като дължината на всяка от веригите за умножение $A_1 \cdot A_2 \cdot \cdots \cdot A_k$ и $A_{k+1} \cdot A_{k+2} \cdot \cdots \cdot A_n$ е по-малка от n, това е истинска рекурсивна декомпозиция, само че е изказана много общо и нямаща начални условия. Началното условие е много просто: ако веригата от матрици за умножение има дължина 1, тоест, ако $n = 1$, цената е 0, понеже умножение на матрици не се извършва.

Нека осмислим добре рекурсивната декомпозиция. От написаното човек може да остане с впечатление, че при декомпозицията винаги декомпозираме на две подвериги, всяка от които "опира" в единия край на началната верига; а именно, започва с A_1 или завършва с A_n. Това, разбира се, не е вярно. Ако си представим още едно ниво на декомпозиция (означено на следната фигура със син делител), ясно се вижда, че в общия случай подверигите започват с някаква матрица A_i и завършват с някаква матрица A_j, където $1 \leqslant i \leqslant j \leqslant n$:

$$\underbrace{(\underbrace{(A_1 \cdot \cdots \cdot A_\ell)}_{B_1} \mid \underbrace{(A_{\ell+1} \cdot \cdots \cdot A_k)}_{B_2})}_{B} \mid \underbrace{(\underbrace{(A_{k+1} \cdot \cdots \cdot A_m)}_{C_1} \mid \underbrace{(A_{m+1} \cdot \cdots \cdot A_n)}_{C_2})}_{C}$$

Подробната рекурсивна декомпозиция е следната. Нека $M(i,j)$ означава минималния брой скаларни умножения за верижното матрично умножение $A_i \cdot \cdots \cdot A_j$, къдено $1 \leqslant i \leqslant j \leqslant n$.

$$M(i,j) = \begin{cases} 0, & \text{ако } i = j, \\ \min\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j \mid i \leqslant k \leqslant j-1\}, & \text{ако } i < j \end{cases} \tag{12.18}$$

$d_{i-1}d_kd_j$ е броят на скаларните умножения в матричното умножение на матрицата-произведение $A_i \cdot \cdots \cdot A_k$ (тя е с размери $(i-1) \times k$) и на матрицата-произведение $A_{k+1} \cdot \cdots \cdot A_j$ (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 тази рекурсивна декомпозиция. Нека M е квадратен масив $n \times n$. Изчислителното правило е пълен аналог на (12.18):

$$M[i,j] = \begin{cases} 0, & \text{ако } i = j, \\ \min_{i \leqslant k \leqslant j-1} \left(M[i,k] + M[k+1,j] + d_{i-1}d_k d_j \right), & \text{ако } i < j \end{cases} \tag{12.19}$$

Да съобразим как работи изчислението. Базата на рекурсията е $M[i,i] = 0$, за $1 \leqslant i \leqslant n$. С други думи, слагаме нули върху главния диагонал. Въз основа на главния диагонал и изчислителното правило от (12.19) 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(⟨d_0, d_1, ..., d_n⟩: цели положителни числа)
 1   създай M[1..n, 1..n] от числа
 2   for i ← 1 to n
 3       M[i,i] ← 0
 4   for diag ← 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] + d_{i−1} d_i d_j
 8           for k ← i+1 to j − 1
 9               M[i,j] ← min (M[i,j], M[i,k] + M[k+1,j] + d_{i−1} d_k d_j)
10   return M[1,n]
```

$M[i,i]$ на ред 7, разбира се, е 0. Написано е по този начин, за да следва буквално (12.19) и да има прилика с ред 9.

Да видим пример, взет от [31, стр. 376]. Нека $n = 6$ и $\langle d_0, \ldots, d_6 \rangle = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$. 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 (12.19),

$$\begin{aligned} M[1,2] &= \min_{1 \leqslant k \leqslant 1} \left(M[1,k] + M[k+1,2] + d_{1-1}d_k d_2 \right) \\ &= M[1,1] + M[2,2] + d_0 d_1 d_2 = 0 + 0 + 30 \cdot 35 \cdot 15 = 15\,750 \end{aligned}$$

	1	2	3	4	5	6
1	0	15 750				
2		0				
3			0			
4				0		
5					0	
6						0

Аналогично,

$$
\begin{aligned}
M[2,3] &= M[2,2] + M[3,3] + d_1 d_2 d_3 = 0 + 0 + 35 \cdot 15 \cdot 5 = 2\,625 \\
M[3,4] &= M[3,3] + M[4,4] + d_2 d_3 d_4 = 0 + 0 + 15 \cdot 5 \cdot 10 = 750 \\
M[4,5] &= M[4,4] + M[5,5] + d_3 d_4 d_5 = 0 + 0 + 5 \cdot 10 \cdot 20 = 1\,000 \\
M[5,6] &= M[5,5] + M[6,6] + d_4 d_5 d_6 = 0 + 0 + 10 \cdot 20 \cdot 25 = 5\,000
\end{aligned}
$$

Запълваме докрая диагонала над главния диагонал:

	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 (12.19),

$$
\begin{aligned}
M[1,3] &= \min_{1 \leqslant k \leqslant 2} \left(M[1,k] + M[k+1,3] + d_{1-1} d_k d_3 \right) \\
&= \min \left(M[1,1] + M[2,3] + d_0 d_1 d_3, M[1,2] + M[3,3] + d_0 d_2 d_3 \right) \\
&= \min \left(0 + 2\,625 + 5\,250, 15\,750 + 0 + 2\,250 \right) \\
&= \min \left(7\,875, 18\,000 \right) = 7\,875
\end{aligned}
$$

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.

$$
\begin{aligned}
M[2,4] &= \min\left(M[2,2] + M[3,4] + d_1 d_2 d_4, M[2,3] + M[4,4] + d_1 d_3 d_4\right) \\
&= \min\left(0 + 750 + 5\,250, 2\,625 + 0 + 1\,750\right) \\
&= \min\left(6\,000, 4\,375\right) = 4\,375 \\
M[3,5] &= \min\left(M[3,3] + M[4,5] + d_2 d_3 d_5, M[3,4] + M[5,5] + d_2 d_4 d_5\right) \\
&= \min\left(0 + 1\,000 + 1\,500, 750 + 0 + 3\,000\right) \\
&= \min\left(2\,500, 3\,750\right) = 2\,500 \\
M[4,6] &= \min\left(M[4,4] + M[5,6] + d_3 d_4 d_6, M[4,5] + M[6,6] + d_3 d_5 d_6\right) \\
&= \min\left(0 + 5\,000 + 1\,250, 1\,000 + 0 + 2\,500\right) \\
&= \min\left(6\,250, 3\,500\right) = 3\,500
\end{aligned}
$$

	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 $M[1,6]$).

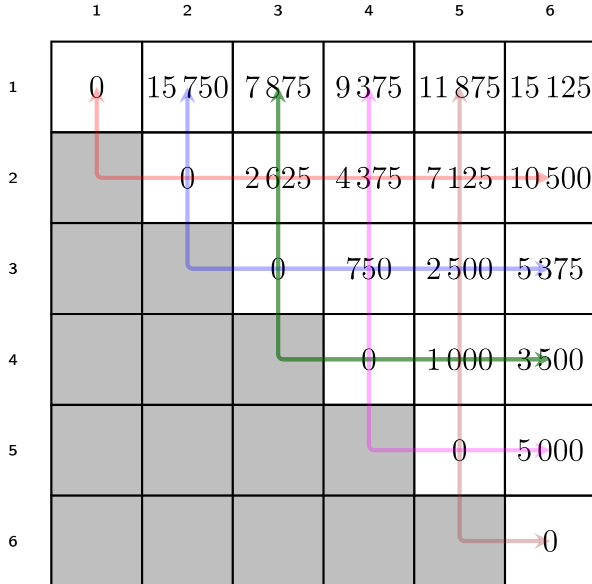
	1	2	3	4	5	6
1	0	15 750	7 875	9 375	11 875	
2		0	2 625	4 375	7 125	10 500
3			0	750	2 500	5 375
4				0	1 000	3 500
5					0	5 000
6						0

$M[1,6]$ се изчислява така:

$$\begin{aligned} M[1,6] = \min(\, & M[1,1] + M[2,6] + d_0 d_1 d_6, \\ & M[1,2] + M[3,6] + d_0 d_2 d_6, \\ & \mathbf{M[1,3] + M[4,6] + d_0 d_3 d_6}, \\ & M[1,4] + M[5,6] + d_0 d_4 d_6, \\ & M[1,5] + M[6,6] + d_0 d_5 d_6) \\ = \min(\, & 0 + 10\,500 + 26\,250, \\ & 15\,750 + 5\,375 + 11\,250, \\ & \mathbf{7\,875 + 3\,500 + 3\,750}, \\ & 9\,375 + 5\,000 + 7\,500, \\ & 11\,875 + 0 + 15\,000) = \min(36\,750, 32\,375, \mathbf{15\,125}, 21\,875, 26\,875) = \mathbf{15\,125} \end{aligned}$$

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**.



Това е и краят на примера.

Да направим няколко пояснения върху псевдокода на Alg Matrix-Chain Multiplication. **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 $M[1, 2]$, $M[2, 3]$, ..., $M[n-1, n]$;
- за диагонал 3, разликата между j и i е 2: той се състои от $M[1, 3]$, $M[2, 4]$, ..., $M[n-2, n]$;
- и така нататък.

И наистина, на ред 6, на j се присвоява точно $i + \text{diag} - 1$.

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 $M[n - \text{diag} + 1, n]$.

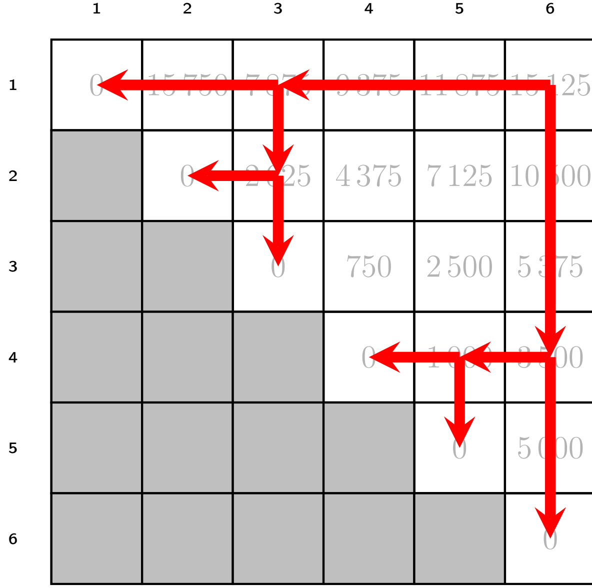
Ние запълваме всеки диагонал в посока от горе-ляво към долу-дясно. Не е задължително да е така. За всяка клетка от даден диагонал, стойността се изчислява чрез стойности, които се вземат само от диагоналите с **по-малки номера** (както и от d стойностите от входа). Ерго, можем да запълваме клетките на 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.

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. 470—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.



Решението, което съответства на тази фигура, е

$$\big((A_1(A_2A_3))\big)\big(((A_4A_5)A_6)\big)$$

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

$$M[i,j] \leftarrow \min_{i \leqslant k \leqslant j-1} \left(\{M[i,k] + M[k+1,j] + d_{i-1}d_kd_j\right)$$

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 $(M[i, j].\text{val}, M[i, j].\text{prev})$, където $M[i, j].\text{val}$ е оптималната цена за умножението на $A_i \cdot \dots \cdot A_j$, а $M[i, j].\text{prev})$ е 0, ако $i = j$, и

```
Alg Matrix-Chain Multiplication-1(⟨d_0, d_1, ..., d_n⟩: цели положителни числа)
 1  създай M[1..n, 1..n] от наредени двойки от числа
 2  for i ← 1 to n
 3      M[i,i].val ← 0
 4      M[i,i].prev ← 0
 5  for diag ← 2 to n
 6      for i ← 1 to n − diag + 1
 7          j ← i + diag − 1
 8          M[i,j].val ← M[i,i].val + M[i+1,j].val + d_{i−1} d_j d_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 + d_{i−1} d_k d_j
12                  M[i,j].val ← M[i,k].val + M[k+1,j].val + d_{i−1} d_k d_j
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..n, 1..n] от (val, prev), i, j)
1  if i = j
2      print "A_i"
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 ")"
```

12.5.2 Minimum Weight Triangulation

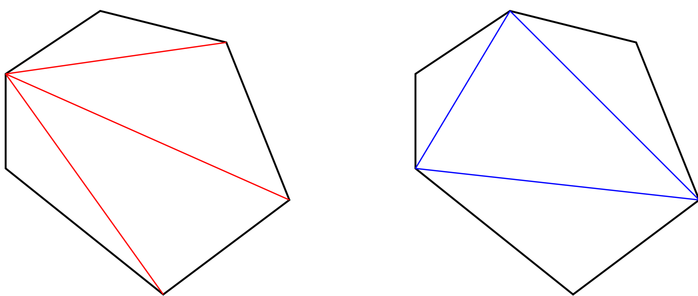
Прост многоъгълник е множество от отсечки, поне три на брой, които не се пресичат две по две, освен може би в крайните точки, и чието обединение е проста затворена крива. Краищата на отсечките са *върховете на многоъгълника. Съседни върхове* са всеки два върха, които са краища на една от тези отсечки. 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 индивидуални отсечки като *изродени многоъгълници*, двуъгълници в този случай, чиито триангулации имат 0 диагонала.

Тегло на триангулация е сумата ог 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 30: Minimum Weight Triangulation

екземпляр: Редица $\langle V_1, \ldots, V_n \rangle$ от точки в равнината, задаваща изпъкнал n-ъгълник.
решение: Триангулация на n-ъгълника с минимално тегло.

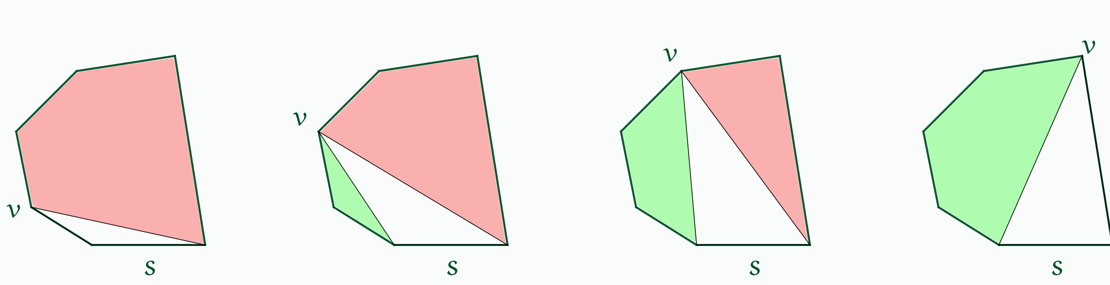
В тези записки ще разгледаме само олекотения вариант на задачата.

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 51: Броят на триангулациите на изпъкнал n-ъгълник

Да си представим изпъкнал n-ъгълник за $n \geqslant 3$ 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

$$T_n = \begin{cases} 1, & \text{ако } n = 2, \\ \sum_{k=2}^{n-1} T_k \cdot T_{n-k+1}, & \text{ако } n > 2 \end{cases} \tag{12.20}$$

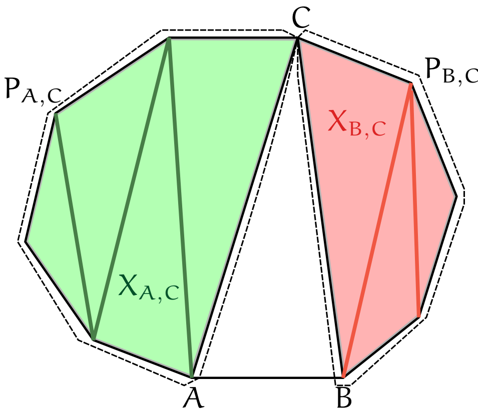
Да сравним (12.20) с (12.17). Веднага се вижда, че $T_n = C_{n-2}$.

Наблюдение 59: Тегло на триангулация, декомпозиция

Нека е даден n-ъгълник P за $n \geqslant 3$ 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

$$|X| = |AC| + |BC| + |X_{A,C}| + |X_{B,C}| \tag{12.21}$$

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 (12.21) наистина брои всички диагонали точно по един път.



Нека входът е редица от точки $\langle V_1, \ldots, V_n \rangle$ в равнината, задаваща изпъкнал n-ъгълник P. Фиксираме посока в равнината, да кажем, по часовниковата стрелка. БОО, нека редицата точки $\langle V_1, \ldots, V_n \rangle$ се среща в посока по часовниковата стрелка, ако обикаляме по периферията на многоъгълника. За всеки i и j, такива че $1 \leqslant i < j \leqslant n$, $P_{i,j}$ е многоъгълникът, състоящ се от отсечките V_iV_{i+1}, $V_{i+1}V_{i+2}$, ..., $V_{j-1}V_j$ и отсечката V_iV_j, която е или страна, или диагонал на многоъгълника. Примерно, $P_{1,n} = P$, като $V_iV_j = V_1V_n$ е страна. Като други примери: $P_{1,2}$ е изроден многоъгълник, съвпадащ със страната V_1V_2, $P_{2,3}$ е изроден многоъгълник, съвпадащ със страната V_2V_3, и така нататък, $P_{n-1,n}$ е изроден многоъгълник, съвпадащ със страната $V_{n-1}V_n$. Страната V_1V_n не може да бъде изразена по този начин, понеже, както казахме, $P_{1,n}$ е целият многоъгълник.

Рекурсивната декомпозиция, базирана на Наблюдение 59, е следната. При дадени i и j, такива че $i+1 < j$, минималната триангулация на $P_{i,j}$ е минимума по всички точки $k \in \{i+1, \ldots, j-1\}$ от сумата на $|V_iV_k|$, $|V_kV_j|$ 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 $1 \leqslant i < j \leqslant n$.

$$T[i,j] = \begin{cases} 0, & \text{ако } j = i+1 \\ \min_{i+1 \leqslant k \leqslant j-1} \left(T[i,k] + T[k,j] + |V_iV_k| + |V_kV_j| \right), & \text{ако } j > i+1 \end{cases} \tag{12.22}$$

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 трябват само дължини на диагонали, а съседни точки не са краища на диагонал.

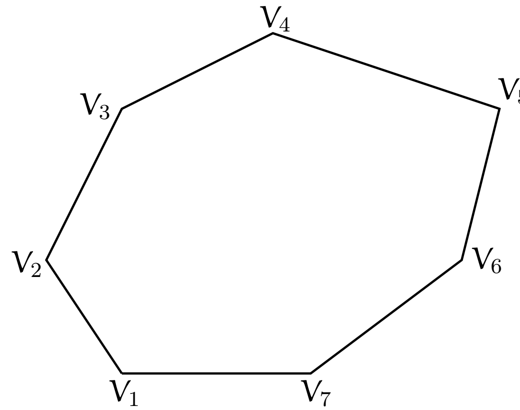
Алгоритъмът за олекотения вариант на задачата е много подобен на ALG MATRIX-CHAIN MULTIPLICATION. Дребна разлика е, че тук главният диагонал не се ползва, а следващите два диагонала вдясно от него съдържат само нули: първият заради началните условия (страните на многоъгълника), а вторият поради това, че триъгълниците имат триангулации с нулево тегло, нямайки диагонали.

[†] А не $j = i$. $P_{i,i}$ е индивидуална точка, а чак толкова изродени многоъгълници не разглеждаме.

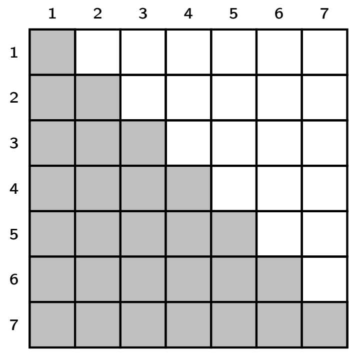
Alg Minimum Weight Triangulation($\langle V_1, V_2, \ldots, V_n \rangle$: редица от точки, образуваща изпъкнал многоъгълник)

```
1  for i ← 1 to n − 1
2      T[i, i + 1] ← 0
3  for gap ← 2 to n − 1
4      for i ← 1 to n − gap
5          j ← i + gap
6          T[i, j] ← T[i, i + 1] + T[i + 1, j] + dist(V_i, V_{i+1}) + dist(V_{i+1}, V_j)
7          for k ← i + 2 to j − 1
8              T[i, j] ← min (T[i, j], T[i, k] + T[k, j] + dist(V_i, V_k) + dist(V_k, V_j))
9  return T[1, n]
```

Да разгледаме малък пример със седмоъгълник:



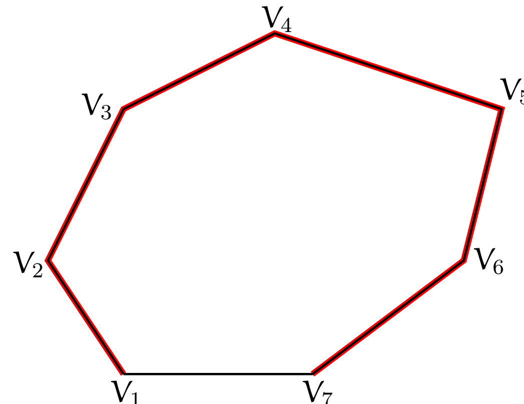
Така изглежда масивът T в началото на алгоритъма. Затъмнени са клетките, които няма да се ползват.



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.



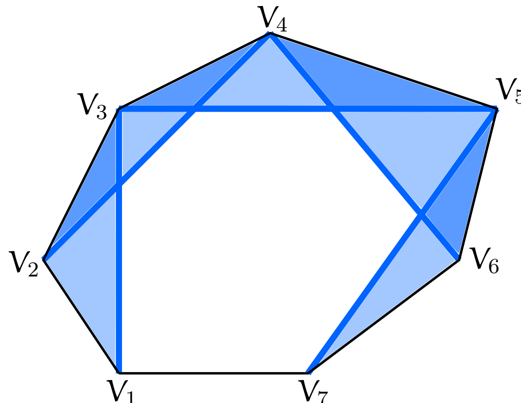
Да разгледаме запълването на следващия диагонал. Имаме $gap = 2$ от ред 3, $i = 1$ от ред 4 и $j = 3$ от ред 5. На ред 6:

$$T[1,3] \leftarrow \underbrace{T[1,2]}_{0} + \underbrace{T[2,3]}_{0} + \underbrace{dist(V_1,V_2)}_{0} + \underbrace{dist(V_2,V_3)}_{0}$$

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							

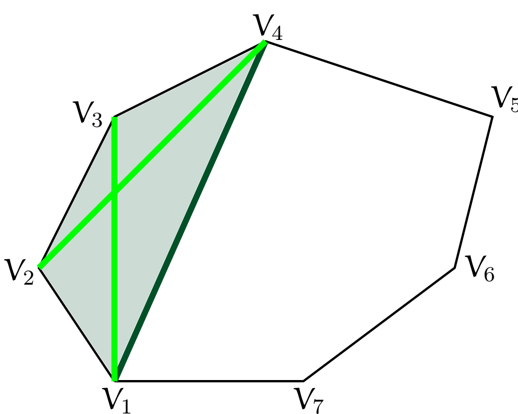
Поставянето на сините нули отговаря на триангулациите на ето тези триъгълници:



Да разгледаме запълването на следващия диагонал. Имаме $\mathsf{gap} = 3$ от ред 3, $\mathsf{i} = 1$ от ред 4 и $\mathsf{j} = 4$ от ред 5. Редове 6–8 реализират това:

$$\begin{aligned}\mathsf{T}[1,4] \leftarrow \min(&\underbrace{\mathsf{T}[1,2]}_{0} + \underbrace{\mathsf{T}[2,4]}_{0} + \underbrace{\mathrm{dist}(\mathsf{V}_1,\mathsf{V}_2)}_{0} + \underbrace{\mathrm{dist}(\mathsf{V}_2,\mathsf{V}_4)}_{42.43 \text{ mm}},\\ &\underbrace{\mathsf{T}[1,3]}_{0} + \underbrace{\mathsf{T}[3,4]}_{0} + \underbrace{\mathrm{dist}(\mathsf{V}_1,\mathsf{V}_3)}_{35 \text{ mm}} + \underbrace{\mathrm{dist}(\mathsf{V}_3,\mathsf{V}_4)}_{0})\end{aligned}$$

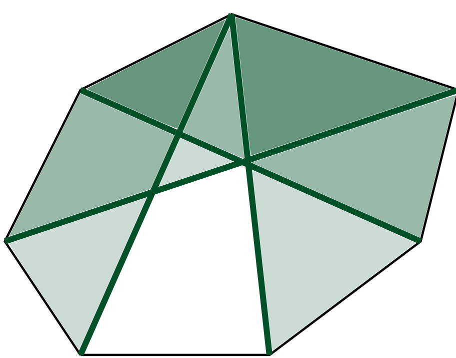
така че $\mathsf{T}[1,4]$ става 35 mm. Във фигурата, това изчисление отговаря на минимална триангулация на зеления четириъгълник $\mathsf{P}_{1,4}$, която е по-малкият от двата диагонала $\mathsf{V}_2\mathsf{V}_4$ и $\mathsf{V}_1\mathsf{V}_3$.



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 gap стане 4, търсим теглото на минимална триангулация на петоълници, три на брой; за всеки петоъгълник взмаме минимума от три стойности, отговарящи на трите междинни върха. Когато gap стане 5, търсим теглото на минимална триангулация на два шестоъгълника; за всеки от тях вземами минимума на четири стойности. И накрая, при gap $= 6$, 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. 513.

12.5.3 Факторизация при неасоциативно умножение

Дадена е азбука Σ и стринг $x = x_1 x_2 \cdots x_n$ над Σ. Нека $|\Sigma| = k$. Дадена е бинарна операция $* : \Sigma^2 \rightarrow \Sigma$. Операцията е наречена "умножение", но това е чисто формално. 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 31: NON-ASSOCIATIVE OPERATION

екземпляр: Азбука Σ и стринг $x = x_1 x_2 \cdots x_n$ над Σ. Бинарна операция $* : \Sigma^2 \to \Sigma$. 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 скаларни 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(Σ, $* : \Sigma^2 \to \Sigma$, $x = x_1 \cdots x_n \in \Sigma^+$, $\sigma \in \Sigma$)

```
1   for i ← 1 to n
2       T[i, i] ← {x_i}
3   for diag ← 2 to n
4       for i ← 1 to n − diag + 1
```

5 $\quad\quad\quad$ $j \leftarrow i + \text{diag} - 1$
6 $\quad\quad\quad$ $T[i,j] \leftarrow \varnothing$
7 $\quad\quad\quad$ **for** $k \leftarrow i$ **to** $j - 1$
8 $\quad\quad\quad\quad$ **foreach** $a \in T[i,k]$
9 $\quad\quad\quad\quad\quad$ **foreach** $b \in T[k+1,j]$
10 $\quad\quad\quad\quad\quad\quad$ $T[i,j] \leftarrow T[i,j] \cup \{a * b\}$
11 $\quad$ **if** $\sigma \in T[1,n]$
12 $\quad\quad$ **return** 1
13 $\quad$ **else**
14 $\quad\quad$ **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 имплементация на алгоритъма, но само до съставянето на цялата T, на Maple(TM), и пускането върху този пример. 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. 513.

12.5.4 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 $A \to X$, 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:

$$x \Rightarrow z \overset{\text{def}}{\leftrightarrow} \exists A \in N\ \exists x_1, x_2 \in (\Sigma \cup N)^* \ (x = x_1 A x_2 \text{ и има правило } A \to \alpha \text{ и } z = x_1 \alpha x_2)$$

Релацията $\overset{*}{\Rightarrow}$ е рефлексивното и транзитивно затваряне на $\Rightarrow$. Ако $x \overset{*}{\Rightarrow} y$, казваме, че има *деривация от* x *до* y *в* Γ. За всеки $y \in \Sigma^*$ казваме, че y *има деривация в* Γ, ако $S \overset{*}{\Rightarrow} y$. *Езикът на граматиката* е $L(\Gamma) = \{y \in \Sigma^* \mid S \overset{*}{\Rightarrow} y\}$.

Нека $\epsilon \notin L(G)$. КСГ е в Нормална Форма на Чомски (пишем кратко КСГНФЧ), ако всички правила са от вида $A \to BC$ или $A \to a$, където A, B и C са нетерминали и a е терминал.

Изч. Задача 32: 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 CYK на имената на създателите си Cocke, Younger и Kasami. Алгоритъмът връща или 1, ако $S \overset{*}{\Rightarrow} y$, 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 $n \geqslant 1$. Ако за всеки $y_i \cdots y_j$, където $1 \leqslant i \leqslant j \leqslant n$, намерим множеството $X_{i,j} = \{A \in N \mid A \overset{*}{\Rightarrow} y_i \cdots y_j\}$, 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 $i \leqslant i' \leqslant j' \leqslant j$ и $j' - i' < j - i$.

Базата на рекурсията са множествата $X_{i,i}$ за $1 \leqslant i \leqslant n$:

$$\forall i \in \{1, 2, \ldots, n\} : X_{i,i} = \{A \in N \mid \text{има правило } A \rightarrow y_i\}$$

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 $k \in \{i, \ldots, j-1\}$) на тези нетерминали, които са левите страни в продукции от вида $A \rightarrow BC$, такива че $B \overset{*}{\Rightarrow} y_i \cdots y_k$ и $C \overset{*}{\Rightarrow} y_{k+1} \cdots y_j$. 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(Γ = ⟨Σ, N, S, R⟩, y ∈ Σ*)
 1  n ← |y|
 2  for i ← 1 to n
 3      M[i, i] ← {A | (A → y_i) ∈ R}
 4  for diag ← 2 to n
 5      for i ← 1 to n − diag + 1
 6          j ← i + diag − 1
 7          M[i, j] ← ∅
 8          for k ← i to j − 1
 9              for B ∈ M[i, k]
10                  for C ∈ M[k + 1, j]
11                      for P ∈ R
12                          if P = (A → BC)
13                              M[i, j] ← M[i, j] ∪ {A}
14  if S ∈ 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 продукциите, имат големини, ограничени от константа, от което следва, че и трите най-вътрешни цикъла "въртят" само константен брой пъти.

12.6 Задачи за оптимални подмножества

12.6.1 2-Partition

Дадено е мултимножество A от цели положителни числа. Във варианта на задача за разпознаване се пита дали A има подмултимножество X, такова че сумата от числата в X е равна на сумата на числата в $A \setminus X$. Във варианта на оптимизационна задача се търси подмултимножество X, такова че абсолютната стойност на разликата между сумата на числата в X и сумата на числата в $A \setminus X$ е минимална.

Тази задача е частен случай на задачата KNAPSACK (вижте Подсекция 12.6.3), 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 A да е **мулти**множество.

Изч. Задача 33: 2-PARTITION, версия за разпознаване

екземпляр: Крайно непразно множество $A = \{a_1, a_2, \ldots, a_n\}$. Функция $s : A \to \mathbb{N}^+$.
въпрос: Дали съществува $X \subset A$, такова че $\sum_{a \in X} s(a) = \sum_{a \in A \setminus X} s(a)$?

В тези записки разглеждаме задачата във версията за разпознаване. На практика по-полезната версия е оптимизационната (Задача 34). 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 34: 2-PARTITION, оптимизационна версия

екземпляр: Крайно непразно множество $A = \{a_1, a_2, \ldots, a_n\}$. Функция $s : A \to \mathbb{N}^+$.
решение: Множество $X \subset A$, такова че $\left| \sum_{a \in X} s(a) - \sum_{a \in A \setminus X} s(a) \right|$ е минимална.

Пълното име на задачата е SET 2-PARTITION. Както знаем, има и други видове разбивания, примерно целочислено разбиване (Определение 35), поради което има смисъл да се уточни, че става дума за разбиване на множество. Но за краткост ще изпускаме "SET".

Подчертаваме, че терминът "PARTITION" в този контекст няма нищо общо нито с алгоритъма PARTITION на стр. 322, който се вика от SELECT (Подсекция 6.2.2), нито с алгоритъма PARTITION (който се реализира като PARTITION–HOARE или PARTITION–LOMUTO), който се вика от QUICKSORT (Секция 6.4).

Задачата във версия за разпознаване. Да разгледаме $\sum_{a \in A} s(a)$. Очевидно е, че ако $\sum_{a \in A} s(a)$ е нечетно, отговорът е НЕ. Нека $\sum_{a \in A} s(a)$ е четно и нека $B = \frac{1}{2} \sum_{a \in A} s(a)$. 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. Иска се да има **някаква** линейна наредба – точно както казва Skiena на стр. 476 за същността на схемата **Динамично Програмиране**. За всяко $i \in \{0, \ldots, n\}$, A_i означава

множеството $\{a_1, \ldots, a_i\}$. Очевидно $A_0 = \emptyset$ и $A_n = A$. БОО, в тези разсъждения допускаме, че за всяко $i \in \{1, \ldots, n\}$, $s(a_i) \leqslant B$; в противен случай отговорът би бил директно НЕ.

Да характеризираме ДА-екземпляр[†] на задачата. В него съществува подмножество на A, такова, че сумата от размерите на елементите му е точно B. Такова подмножество се състои от елементи, поне един на брой. *// очевидно – е, и?*
Продължаваме: всяко непразно подмножество на A със сума B се състои от множество със сума $B - p$ и още един елемент с размер p. *// това вече е нещо.*
Продължаваме: всяко $X \subseteq A$ със сума B е подмножество на някое A_i, където $i \in \{1, \ldots, n\}$. При това,

- или $a_i \in X$, в който случай $X \backslash \{a_i\}$ е подмножество на A_{i-1} със сума $B - s(a_i)$,
- или $a_i \notin X$, в който случай X е подмножество на A_{i-1}. *// аха, това е!*

За тази характеризация ще направим следната рекурсивна декомпозиция. За краткост на записа въвеждаме следната нотация. Нека $P(i, j)$, където $1 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant B$, е предикат със следния смисъл:

$$P(i, j) \stackrel{\text{def}}{=} \exists X \subseteq A_i : \sum_{a \in X} s(a) = j$$

Очевидно търсеният отговор е ДА тстк $P(n, B) = \mathtt{T}$.

Може да изчислим P ефикасно така.

- Разглеждаме $P(1, j)$, за $0 \leqslant j \leqslant B$. Има точно две подмножества на A_1, сумата от теглата на чиито елементи е от $\{0, \ldots, B\}$. 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 $0 \leqslant j \leqslant B$. Подмножествата на A_i, сумата от теглата на чиито елементи е от $\{0, \ldots, B\}$, се разбиват на:
  - подмножествата на A_{i-1}, сумата от теглата на чиито елементи е някой елемент от $\{0, \ldots, B\}$, и
  - множествата от вида $X \cup \{a_i\}$, където $X \subseteq A_{i-1}$ и сумата от теглата на елементите на X е от $\{0, \ldots, B - s(a_i)\}$.

  Ерго, $P(i, j)$ е истина тстк $P(i - 1, j)$ е истина **или** $s(a_i) \leqslant j$ и $P(i - 1, j - s(a_i))$ е истина. Това **или** е включващо.

Ето схема за изчисление, базирана на тази рекурсивна декомпозиция.

$$P[1, j] = \begin{cases} \mathtt{T}, & \text{ако } j = 0 \text{ или } j = s(a_1) \\ \mathtt{F}, & \text{в противен случай} \end{cases} \tag{12.23}$$

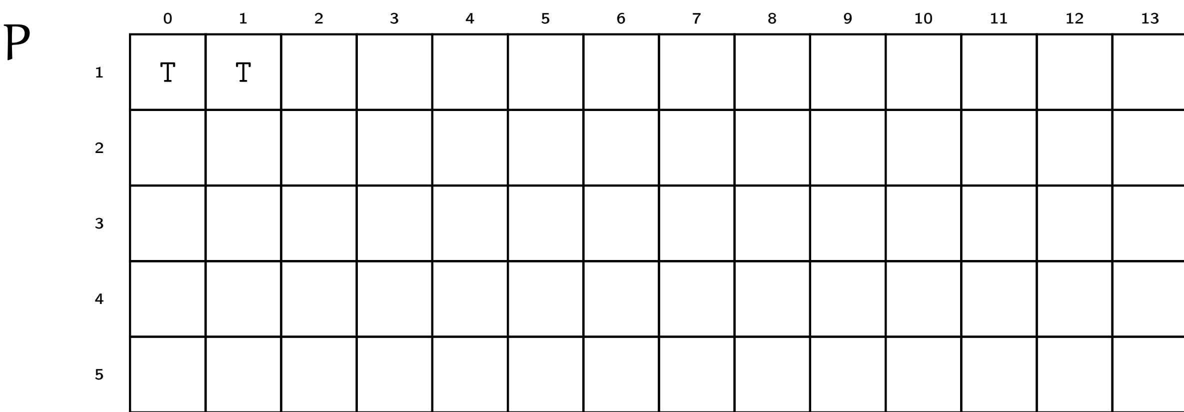
$$P[i, j] = \begin{cases} \mathtt{T}, & \text{ако } P[i-1, j] \text{ или } (s(a_i) \leqslant j \text{ и } P[i-1, j - s(a_i)]) \\ \mathtt{F}, & \text{в противен случай} \end{cases}, \text{ ако } i > 1 \tag{12.24}$$

[†]Това не е оптимизационна задача, така че не говорим за характеризиране на решение. При задачата за най-къси пътища характеризирахме решение, понеже тя е оптимизационна задача. Тук задачана е задача за разпознаване и решенията са само две: ДА и НЕ.

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 $\left(\sum_{i=1}^{n} s(a_i)\right)$
2   **return** F
3 $B \leftarrow \frac{1}{2}\sum_{i=1}^{n} s(a_i)$
4 **if** max $\{s(a_i) \,|\, 1 \leqslant i \leqslant n\} > B$
5   **return** F
6 създай булев масив $P[1 \mathinner{..} n, 0 \mathinner{..} B]$
7 инициализирай P с F
8 $P[1, 0] \leftarrow$ T, $P[1, s(a_1)] \leftarrow$ T
9 **for** $i \leftarrow 2$ **to** n
10   **for** $j \leftarrow 0$ **to** B
11         $P[i, j] \leftarrow P[i-1, j]$ **or** $(s(a_i) \leqslant j$ **and** $P[i-1, j - s(a_i)])$
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 $s(a_1) = 1$.

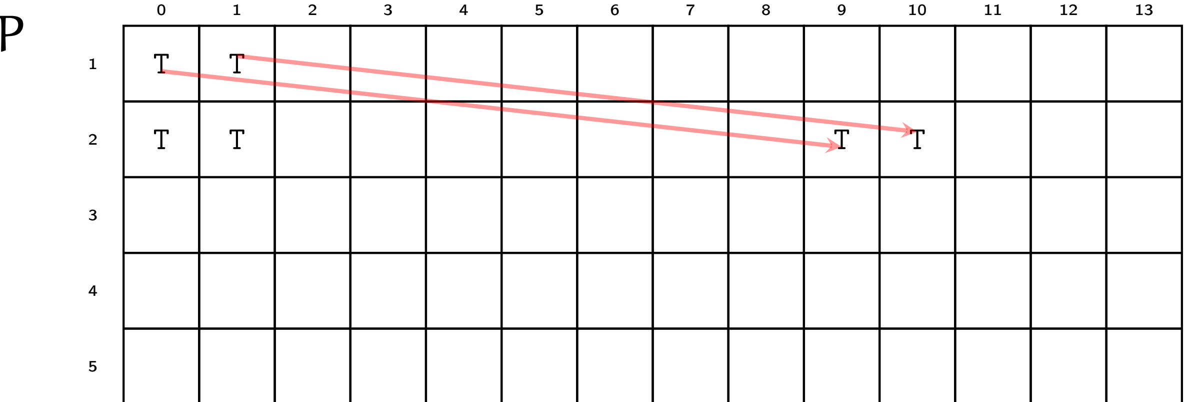


P

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	T												
2														
3														
4														
5														

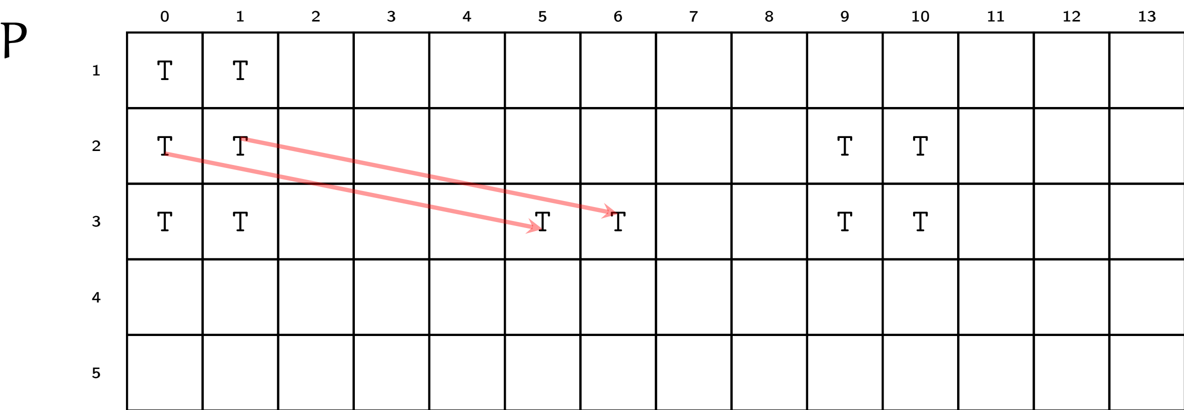
Във втория ред записваме четири T.

- 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.



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 T от P[2, 0] и P[2, 1] "пораждат" съответно T в P[3, 5] и P[3, 6], понеже $s(a_3) = 5$. Обаче T от P[2, 9] и P[2, 10] не "пораждат" T на третия ред, тъй като $9 + 5 > 13$ и $10 + 5 > 13$; това е причината на третия ред T да са само шест. С червени стрелки илюстрираме пораждан ията надясно (с отместване 5).



Да разгледаме четвъртия ред. В него записваме единадесет T.

- T 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 T от P[3, 0] и P[3, 1] "пораждат" съответно T в P[4, 3] и P[4, 4], понеже $s(a_4) = 3$. T от P[3, 5] поражда T в P[4, 8] по същата причина. За T от P[3, 6] не може да кажем, че поражда T в P[4, 9], защото вече сме записали T в P[4, 9]. Двете T в P[3, 9] и P[3, 10] пораждат T съответно в P[4, 12] и P[4, 13]. Петте червени стрелки показват пораждан ията надясно (с отместване 3).

P

	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														

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 $\mathtt{T}$ надясно – $\mathtt{T}$ в $\mathtt{P}[4, 3]$ поражда $\mathtt{T}$ в $\mathtt{P}[5, 11]$, а останалите $\mathtt{T}$ в петия ред са получени с преписване надолу от четвъртия ред.

P

	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 $\mathtt{P}[5, 13] = \mathtt{T}$ след приключване на алгоритъма.

Коректността на алгоритъма е очевидна предвид (12.23) и (12.24). Интерес представлява изследването на сложността му по време. Тя е $\Theta(nB)$. Множителят n е размерът на входа, при допускането, че всяко от числата $s(a_i)$ има размер $\Theta(1)$. Но B какво е? B е полусумата от **големините** на числата. Тези големини може да 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 е безкрайност. Разбира се, този парадоксален извод се появява само при допускането на нашия изчислителен модел, че числата във входа имат константни размери за всички големини; тоест, 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 големини 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 89: 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.

Допълнение 52: 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 казано. 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 ∈ 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, измерено 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 да се опитаме да генерираме всички решения, понеже в най-лошия случай броят им е експоненциален в размера на входа[†]. Следният алгоритъм, който е тривиална модификация на Alg 2-Partition, връща и таблица Q, чрез която можем да направим traceback с друг алгоритъм.

```
Alg 2-Partition, solution(A = {a_1, ..., a_n}, s : A → ℕ⁺)
1   if isodd (∑_{i=1}^{n} s(a_i))
2       return "няма решение"
3   B ← (1/2) ∑_{i=1}^{n} s(a_i)
4   if max {s(a_i) | 1 ≤ i ≤ 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] ← T, P[1, s(a_1)] ← T
9   for i ← 2 to n
10      for j ← 0 to B
11          if P[i − 1, j]
12              P[i, j] ← T, Q[i, j] ← j
13          else if s(a_i) ≤ j and P[i − 1, j − s(a_i)]
14              P[i, j] ← T, Q[i, j] ← j − s(a_i)
15  return ⟨P, Q⟩
```

[†]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 23, $\binom{n}{n/2} \asymp \frac{2^n}{\sqrt{n}}$.

Ето алгоритъм, който връща решение, генерирано с traceback, или индикира, че решение няма.

Generate 2-Partition solution($n, B \in \mathbb{N}^+$, $A = \{a_1, \dots, a_n\}$, $s : A \to \mathbb{N}^+$, $P[1..n, 0..B]$: 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
6       j ← B
7       for i ← n downto 2
8           if Q[i, j] ≠ j
9               res ← res ∪ s(a_i)
10              j ← Q[i, j]
11      if Q[2, j] ≠ 0
12          res ← res ∪ s(a_1)
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 подходили 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** $i \leftarrow n$ **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 traceback-а съответства наследните 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.

12.6.2 2 Equal Sum Subsets (2-ESS)

Тази задача е обобщение на 2-PARTITION. Сега се пита дали има две непразни подмножества X и Y на A, които имат една и съща сума от размерите на елементите. БОО, $X \cap Y = \emptyset$, 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 $X \uplus Y = A$, то задачата става 2-PARTITION, поради което казваме, че 2-ESS обобщава 2-PARTITION.

Изч. Задача 35: 2-ESS, версия за разпознаване

екземпляр: Множество $A = \{a_1, a_2, \ldots, a_n\}$. Функция $s : A \to \mathbb{N}^+$.
въпрос: Дали съществуват непразни $X, Y \subset A$, такива че $X \cap Y = \emptyset$ и $\sum_{a \in X} s(a) = \sum_{a \in Y} s(a)$?

Изч. Задача 36: 2-ESS, оптимизационна версия

екземпляр: Множество $A = \{a_1, a_2, \ldots, a_n\}$. Функция $s : A \to \mathbb{N}^+$.
решение: Непразни $X, Y \subset A$, такива че $X \cap Y = \emptyset$ и разликата $|\sum_{a \in X} s(a) - \sum_{a \in Y} s(a)|$ е минимална.

Тук разглеждаме само версията за разпознаване. Задачата е **NP**-пълна [147].

Наблюдение 60

Ако $\sum_{a \in A} s(a) < 2^n - 1$, 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 $\sum_{a \in C} s(a) = \sum_{a \in D} s(a)$. 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 $\sum_{a \in C'} s(a) = \sum_{a \in D'} s(a)$.
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.

Въпреки че 2-ESS е обобщение на 2-Partition, алгоритъмът за 2-ESS, който ще конструираме, не решава 2-Partition.

Нека $B = \left\lfloor \frac{1}{2} \sum_{a \in A} s(a) \right\rfloor$. За разлика от 2-Partition, сега не държим $\sum_{a \in A} s(a)$ да е четно. Също като в 2-Partition, нека $A_i = \{a_1, \ldots, a_i\}$, за $0 \leqslant i \leqslant n$. Нека $Q(t, i, j)$, за $1 \leqslant t \leqslant n$ и $0 \leqslant i, j \leqslant B$, 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 $k \in \{1, \ldots, B\}$, такова че $Q(n, k, k) = T$.

Може да изчислим ефикасно Q така.

- Разглеждаме $Q(1, i, j)$, за $0 \leqslant i, j \leqslant B$. Има точно три наредени двойки $(X, Y) \subseteq A_1 \times A_1$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \ldots, B\} \times \{0, \ldots, B\}$. 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 $0 \leqslant i, j \leqslant B$. Наредените двойки $(X, Y) \in A_t \times A_t$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \ldots, B\} \times \{0, \ldots, B\}$, се разбиват на:
  - наредените двойки $(X, Y) \in A_{t-1} \times A_{t-1}$, такива че $X \cap Y = \emptyset$ и $(\sum_{a \in X} s(a), \sum_{a \in Y} s(a)) \in \{0, \ldots, B\} \times \{0, \ldots, B\}$, 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 $(\sum_{a \in Z} s(a), \sum_{a \in Y} s(a)) \in \{0, \ldots, B - s(a_t)\} \times \{0, \ldots, B\}$, 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 $(\sum_{a \in X} s(a), \sum_{a \in Z} s(a)) \in \{0, \ldots, B\} \times \{0, \ldots, B - s(a_t)\}$.

  Ерго, $Q(t, i, j)$ е истина тстк $Q(t-1, i, j)$ е истина **или** ($s(a_t) \leqslant i$ и $Q(t-1, i - s(a_t), j)$ е истина) **или** ($s(a_t) \leqslant j$ 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 "$Q^{(t)}[i, j]$".

$$Q^{(1)}[i,j] = \begin{cases} \mathtt{T}, & \text{ако } i = j = 0 \vee (i = s(a_1) \wedge j = 0) \vee (i = 0 \wedge j = s(a_1)) \\ \mathtt{F}, & \text{в противен случай} \end{cases} \tag{12.25}$$

$$Q^{(t)}[i,j] = \begin{cases} \mathtt{T}, & \text{ако } Q^{(t-1)}[i,j] \vee (i \geqslant s(a_t) \wedge Q^{(t-1)}[i - s(a_t), j]) \vee (j \geqslant s(a_t) \wedge Q^{(t-1)}[i, j - s(a_t)]) \\ \mathtt{F}, & \text{в противен случай} \end{cases}, \text{ ако } t > 1 \tag{12.26}$$

Би трябвало да е ясно как да конструираме алгоритъм от (12.25) и (12.26). 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 $i \in \{1, \ldots, n\}$ е изпълнено $Q^{(n)}[i,i] = \mathtt{T}$; 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 $\mathtt{T}$, отговарящо на празното решение, което не ни интересува—за да реши дали да върне $\mathtt{T}$ или $\mathtt{F}$.

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 $Q^{(1)}, \ldots, Q^{(4)}$ като четири отделни таблици, като $Q^{(1)}$ е базата (12.25), а всяка от следващите, да кажем $Q^{(t)}$, се попълва съгласно (12.26), използвайки предната таблица и съответното $s(a_t)$.

За да получим $Q^{(1)}$, слагаме $\mathtt{T}$ в клетка $[0,0]$ и, тъй като $s(a_1) = 2$, още по едно $\mathtt{T}$ 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 $\mathtt{T}$ на разстояние 3, по хоризонтал и вертикал, от досега сложените $\mathtt{T}$. Тогава $Q^{(2)}$ е (новите $\mathtt{T}$ 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 T от $Q^{(2)}$. После, тъй като $s(a_3) = 4$, слагаме по едно T на разстояние 4, по хоризонтал и вертикал, от досега сложените T. 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					

Аналотично 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 тстк има поне едно T върху главния диагонал. В случая това е изпълнено, така че за нашия пример алгоритъмът връща ДА.

Забележете, че T в $Q^{(4)}[5, 5]$ отговаря на решението $\{\{2, 3\}, \{5\}\}$, а T в $Q^{(4)}[7, 7]$ отговаря на решението $\{\{2, 5\}, \{3, 4\}\}$.

12.6.3 Задачата за раницата (Knapsack)

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.

В оптимизационната версия на задачата се иска да се намери подмножество от предмети с максимална сумарна цена, чието сумарно тегло не надвишава капацитета на раницата.

Ако предметите не са атомарни, а са неограничено делими, можем да кажем с известна доза условност, че е дадена *континуална* версия на задачата[†]. 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 относителната цена на делимите

[†]Още я наричат FRACTIONAL KNAPSACK.

предмети—да кажем, златният прах на килограм е по-скъп от сребърния, който на свой ред е по-скъп от бронзовия на килограм—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.

Ако предметите са атомарни, да кажем, 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 печелившa стратегия; сега трябва да сме “глобално алчни”, за да напълним раницата с максимално сумарно скъпи неща, без да я скъсаме.

Ето пример за това, че въпросната алчна 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 тегло е $5/5 = 1$. Ерго, алчната идея слага a_1 в раницата, в раницата остава свободен капацитет $10 - 6 = 4$ и това е краят, 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 $5 + 5 = 10$.

Дискретната версия на задачата е широко известна в поне три варианта, които ще разгледаме сега.

12.6.3.1 Unbounded Knapsack

Изч. Задача 37: UNBOUNDED KNAPSACK

екземпляр: Множество от видове предмети $A = \{a_1, \ldots, a_n\}$, като има неограничено много предмети от всеки вид. За всеки вид a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.
решение: $(x_1, \ldots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^{n} x_i w(a_i) \leqslant C \tag{12.27}$$

$$\sum_{i=1}^{n} x_i v(a_i) \text{ е максимална} \tag{12.28}$$

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, …, a_n, а от всеки вид има неограничено много[†] предмети за крадене. Предметите от всеки вид a_i са неразличими помежду си и имат една и съща стойност $v(a_i)$ и едно и също тегло $w(a_i)$. Наредената n-орка от естествени числа $(x_1, \ldots, x_n)$ има смисъл на това, по колко броя от всеки вид ще бъдат взети в раницата. (12.27) и (12.28) 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 стойност.

[†]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.

Олекотеният вариант. Да направим характеризация на някое оптимално решение X. Очевидно 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, а сумарната цена е максимална. Очевидно Y е оптимално решение за раница с капацитет $C - w(a_i)$. *// аха, това е!*

Забележете, че характеристиката е капацитетът. Принципът на оптималността (Определение 86) е спазен, понеже оптимално решение за характеристика-капацитет C се получава от оптимално решение за характеристика-капацитет C', където $C' < C$.

Да помислим за рекурсивна декомпозиция. Да кажем, че числата $0, 1, \ldots, C-1$ са *редуцираните капацитети*. Ако за всеки редуциран капацитет D знаем някое оптимално решение Y_D[†], можем да намерим оптимално решение за пълния капацитет C, като за всяко D и всеки вид предмет a_i опитаме да добавим предмет от вид a_i към Y_D. Трябва да игнорираме получените решения, чието тегло надхвърля C, а от останалите да вземем решението с максимална цена. Това е по всички редуцирани капацитети и по всички видове предмети.

Следното решение на задачата е добре известно. Нека $M[j]$ е максималната стойност на предмети, които можем да сложим в раница с капацитет j, за всички $j \in \{0, \ldots, C\}$. Ако можем да пресмятаме $M[j]$ чрез $M[j-1], \ldots, M[0]$, ще получим решението $M[C]$. Схемата за изчисление е следната.

$$M[j] = \begin{cases} 0, & \text{ако } j = 0 \\ \max\left(M[j-1], \max_{i \in \{1,\ldots,n\} \text{ и } w(a_i) \leqslant j} \left(M[j - w(a_i)] + v(a_i)\right)\right), & \text{ако } j > 0 \end{cases}$$

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 a_i от всеки вид, чието тегло не надхвърля j, слагаме този a_i, който максимизира сумарната цена. Но, за да сложим предмет от вид a_i в раница с капацитет j, трябва да има свободен капацитет $j - w(a_i)$. Максимумът, който можем да постигнем, слагайки предмет от вид a_i, е максимумът за раница с капацитет $j - w(a_i)$ плюс стойността $v(a_i)$ на предмета.

Ако искаме пълно решение на задачата, което ни дава не просто числен отгово, а освен това и по колко предмети от всеки вид да сложим в раницата, можем да записваме за всяко j, $1 \leqslant j \leqslant C$, дали $M[j]$ е получено като $M[j-1]$ или по другия начин. Ако е по другия начин, а именно като максимум по всички i, такива че $w(a_i) \leqslant j$, записваме i, за което се постига максимална стойност на $M[j - w(a_i)] + v(a_i)$. От тази информация може лесно да генерираме пълно решение във време $O(C)$.

[†]Което се побира в раница с капацитет D.

Сложността по време е $\Theta(nC)$, а по памет е $\Theta(C)$. Това е поредният псевдополиомиален алгоритъм, който разглеждаме. Ако числата $w(a_i)$ са малки, масивът M е малък и алгоритъмът е ефикасен.

12.6.3.2 0-1 Knapsack

Изч. Задача 38: 0-1 Knapsack

екземпляр: Множество от предмети $A = \{a_1, \ldots, a_n\}$. За всяко a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.
решение: Подмножество $X \subseteq A$, такова че:

$$\sum_{a \in X} w(a) \leqslant C$$
$$\sum_{a \in X} v(a) \text{ е максимална}$$

Бихме могли да дефинираме и тази версия на задачата чрез вектор $(x_1, \ldots, x_n)$, 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 обща стойност и сумарен размер, не по-голям от капацитета.

В тази версия на задачата—също както и в предишната—не е добра идея да правим рекурсивна декомпозиция само по подмножествата от първите i елемента на A (има се предвид $\{a_1, \ldots, a_i\}$). Читателят лесно може да измисли пример, в който оптимално решение за, да кажем, $\{a_1, \ldots, a_6\}$, не ползва оптимално решение за $\{a_1, \ldots, a_5\}$. Ще направим двумерна рекурсивна декомпозиция, подобна на решението на 2-Partition. Нека $M[i, j]$ е (стойността на) оптимално решение за раница с капацитет j, ползващо предмети a_1, ..., a_i, където $0 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant C$. Решението е $M[n, C]$. Схемата за изчисление е следната.

$$\begin{aligned}
&M[i, 0] = 0, \text{ за } 1 \leqslant i \leqslant n \quad // \text{ нулев капацитет}\\
&M[0, j] = 0, \text{ за } 1 \leqslant j \leqslant C \quad // \text{ няма предмети за слагане}\\
&M[i, j] = M[i-1, j], \text{ ако } i > 0 \text{ и } j > 0 \text{ и } w(a_i) > j \quad // \; a_i \text{ не се побира при капацитет } j\\
&M[i, j] = \max\big(\underbrace{M[i-1, j]}_{\text{не вземаме } a_i}, \underbrace{M[i-1, j - w(a_i)] + v(a_i)}_{\text{вземаме } a_i}\big), \text{ ако } i, j > 0 \text{ и } w(a_i) \leqslant j \qquad (12.29)
\end{aligned}$$

Случай (12.29) е основата на декомпозицията и може би единственият, чиято обосновка не е напълно очевидна. Имаме две възможности.

- Да не вземем предмета a_i. Стойността на такова решение ни е известна и тя е $M[i-1, j]$.
- Да вземем предмета a_i. При това обаче трябва да има капацитет за него $w(a_i)$, така че гледаме колко най-много можем да сложим, оставяйки си свободен капацитет $w(a_i)$, и какво ще получим при това. Тази стойност се пресмята тривиално и тя е $M[i - 1, j - w(a_i)] + v(a_i)$.

Избираме по-добрата от двете възможности, тоест, максимума.

Алгоритъм, реализиращ това изчисление, е следният.

```
0-1 KNAPSACK(A = {a_1, ..., a_n}, v : A → ℝ⁺, w : A → ℕ⁺, C ∈ 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(a_i) > 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(a_i)] + v(a_i))
11  return M[n, C]
```

Коректността на алгоритъма е очевидна. Сложността както по време, така и по памет, е $\Theta(nC)$.

12.6.3.3 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 [103].

Изч. Задача 39: BOUNDED KNAPSACK

екземпляр: Множество от видове предмети $A = \{a_1, \dots, a_n\}$. За всеки вид a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$, 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

$$\sum_{i=1}^{n} x_i w(a_i) \leqslant C \tag{12.30}$$

$$0 \leqslant x_i \leqslant b_i, \text{ за } 1 \leqslant i \leqslant n \tag{12.31}$$

$$\sum_{i=1}^{n} x_i v(a_i) \text{ е максимална} \tag{12.32}$$

БОО, нека $\sum_{i=1}^{n} b_i w(a_i) > C$, защото в противен случай имаме тривиално решение $x_i = b_i$ за $1 \leqslant i \leqslant n$.

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:

$$\langle \{\widehat{a}_1, \dots, \widehat{a}_m\}, (\widehat{v}_1, \dots, \widehat{v}_m), (\widehat{w}_1, \dots, \widehat{w}_m), C \rangle$$

За всеки вид a_i, $1 \leqslant i \leqslant n$, 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 (стойност, тегло) са

$$(v_i, w_i), (2v_i, 2w_i), (4v_i, 4w_i), \ldots, (\lfloor \log_2 b_i \rfloor v_i, \lfloor \log_2 b_i \rfloor w_i)$$

- второ, ако b_i не е число от вида "точна степен на двойката минус единица", още един предмет със стойност

$$b_i v_i - (v_i + 2v_i + 4v_i + \cdots \lfloor \log_2 b_i \rfloor v_i)$$

и размер

$$b_i w_i - (w_i + 2w_i + 4w_i + \cdots + \lfloor \log_2 b_i \rfloor w_i)$$

Очевидно броят на предметите в $\Pi_{0,1}$, съответни на a_i, е $\lceil \log_2(b_i + 1) \rceil$, като сумата от стойностите 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 $\widehat{x}_{i,1}, \ldots, \widehat{x}_{i,\lceil \log_2(b_i+1) \rceil}$ от решението на $\Pi_{0,1}$. Нещо повече, за $1 \leqslant h \leqslant \lceil \log_2(b_i + 1) \rceil$, булевата променлива $\widehat{x}_{i,h}$ "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 $x_i = \sum_{h=1}^{\lceil \log_2(b_i+1) \rceil} n_h \widehat{x}_{i,h}$ може да приеме всяка стойност от $\{0, \ldots, b_i\}$.

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 [103, 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 стойност

$$6 \cdot 10 + 1 \cdot 15 + 0 \cdot 11 = 75$$

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. Стойностите 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 стойности и теглата 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

$$(x_1', x_1'', x_1''', x_2', x_2'', x_2''', x_3', x_3'') = (1\,1\,1\,1\,0\,0\,0\,0)$$

Имената на променливите x съответстват по очевиден начин на предметите. Решението $(1\,1\,1\,1\,0\,0\,0\,0)$ 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.

12.7 Задачи за планиране (Scheduling)

При тези задачи са дадени някакви дейности, всяка от които се дефинира чрез начало, край и може би тегло, и се иска да се намери оптимално подмножество или оптимално разбиване на множеството от дейностите.

12.7.1 Задачата за безработния актьор (Interval Scheduling)

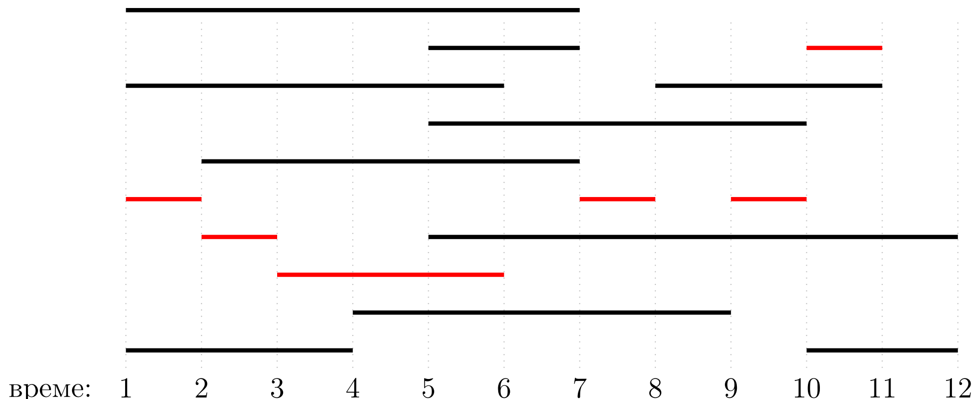
Тази задача е известна още под името “ACTIVITY-SELECTION” [31, стр. 415]. Дадено е непразно множество $A = \{a_1, \ldots, a_n\}$ от отворени интервали, като $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leqslant i \leqslant n$. Ако мислим за интервалите като за дейности във времето, дейност a_i се дефинира чрез начално време s_i и крайно време f_i. Два различни интервала a_i и a_j са *съвместими*, ако имат празно сечение; тоест, ако $f_i < s_j$ или $f_j < s_i$. В противен случай те са *несъвместими*. Казваме, че подмножество от интервали е *безконфликтно*, ако всеки два интервала от него са съвместими. Забележете, че при $f_i = s_j$ интервалите 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 40: Interval Scheduling

екземпляр: Множество от отворени интервали A.
решение: Подмножество $A' \subseteq A$ от взаимно съвместими интервали 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 12.7.1.1 ще стане ясно, че това наистина е така (че няма по-голямо безконфликтно множество).



12.7.1.1 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 засичат с този клип, отпадат като възможности. Измежду останалите клипове отново взема този с най-ранно начално време, и така нататък.

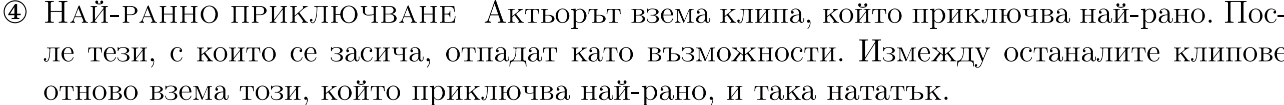
Тази идея не работи в смисъл, че има примери, в които дава неоптимални решения. Ето контрапример с илюстрация. Вземайки първата започваща работа (червения интервал), той ще изтърве четири други, а всички тях заплащането е едно и също.

② Първо най-късият интервал Актьорът взема клипа, който е най-къс. След това всички клипове, които са засичат с този клип, отпадат като възможности. Измежду останалите клипове отново взема този, който е най-къс, и така нататък.

Тази идея също не работи. Ето контрапример с илюстрация. Вземайки най-късия във времето ангажимент (червения интервал), той ще изтърве два други.

③ Първо интервалът с най-малко конфликти Актьорът взема клипа, за който има най-малко други клипове, с които се засича. После тези, с които се засича, отпадат като възможности. Измежду останалите клипове отново взема този, който се засича с най-малко клипове, и така нататък.

Тази идея също не работи. Ето контрапример с илюстрация. Вземайки клипа с най-малко засичания (червения интервал; той се засича само с два други клипа), той губи възможността да вземе оптималното решение от четири клипа (в синьо).



④ Най-рано приключване Актьорът взема клипа, който приключва най-рано. После тези, с които се засича, отпадат като възможности. 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 65: Най-рано приключване е оптимално за Interval Scheduling

За всеки екземпляр A на Interval Scheduling, подмножеството $A' \subseteq A$, конструирано съгласно стратегията най-рано приключване, е максимално множество от два по два съвместими интервали.

Доказателство: Това, че стратегията най-рано приключване конструира подмножество от интервали, между които няма несъвместима двойка, е очевидно. Ще докажем, че конструираното множество е **максимално** множество от два по два съвместими интервали. Да допуснем противното. Нека $A = \{a_1, \ldots, a_n\}$ е контрапример. Да си представим работата на алчен алгоритъм AlgIS за Interval Scheduling, изграден върху стратегията най-рано приключване, върху вход A. Изходът A' не е максимално по мощност безконфликтно множество, съгласно допускането. Нека $A' = \{a_{i_1}, \ldots, a_{i_k}\}$, където $\{i_1, \ldots, i_k\} \subset \{1, \ldots, n\}$, и БОО нека интервалите "влизат" в A' точно в този ред: първо a_{i_1}, после a_{i_2}, и така нататък, накрая a_{i_k}.

Да съобразим, че, слагайки интервалите по този начин в A', алгоритъмът AlgIS греши с поне един интервал. Какво означава това? По допускане, съществува безконфликтно множество $A'' \subseteq A$, такова че $|A''| > |A'|$. Ще докажем, че е невъзможно $A' \subset A''$. Ако допуснем, че $A' \subset A''$, трябва A'' да се състои от интервалите в A' и още поне един интервал a_m, който е или вляво от най-левия интервал на A', или някъде между интервалите на A', или вдясно от най-десния интервал на A'. И трите случая са невъзможни – лесно се вижда, че AlgIS би сложил и a_m в A'. И така, $A' \not\subset A''$, което е същото като да има интервал в A', който не се съдържа в A''. За всеки такъв интервал казваме, че с него AlgIS е *сгрешил*.

Нека a_{i_r} е първият интервал, с който AlgIS е сгрешил. Тогава интервалите, сложени от AlgIS преди a_{i_r}, а именно $a_{i_1}, \ldots, a_{i_{r-1}}$, се намират и в A''. Нека интервалът в A'', който се намира след $a_{i_{r-1}}$, се нарича a_t, като очевидно $t \neq i_r$. Ключовото наблюдение е, че краят f_{i_r} на a_{i_r} се намира вляво от края f_t на a_t или съвпада с него, но не може да е вдясно от a_t. Това е напълно очевидно предвид факта, че AlgIS е изграден върху стратегията най-ранно приключване. Следната фигура илюстрира ситуацията:

A' : a_{i_1} .... $a_{i_{r-1}}$ a_{i_r} ....

A'' : a_{i_1} .... $a_{i_{r-1}}$ a_t ....

Интервалите a_{i_r} и a_t може да не са разположени точно така един спрямо друг, но със сигурност $f_{i_r} \leqslant f_t$, в противен случай AlgIS щеше да сложи a_t, а не a_{i_r} в A'; забележете, че както a_{i_r}, така и a_t са съвместими с всеки от $a_{i_1}, \ldots, a_{i_{r-1}}$.

Веднага се вижда, че AlgIS не може да е сгрешил с a_{i_r}, защото, ако заменим a_t с a_{i_r} в A'', то остава валидно решение:

A'' : a_{i_1} .... $a_{i_{r-1}}$ a_{i_r} ....

С други думи, A'' няма какво да загуби, получавайки a_{i_r} наместо a_t, тъй като $f_{i_r} \leqslant f_t$. □

Допълнение 53: 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 итерираме 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 случай, U е максимално по включване подмножество на S спрямо свойството π и алгоритъмът връща 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 65.

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.

[a] Да 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 задачата, базирайки се на НАЙ-РАННО ПРИКЛЮЧВАНЕ.

INTERVAL SCHEDULING, GENERAL($A = \{a_1, \ldots, a_n\}$, където $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leqslant i \leqslant n$)

1. сортирай A по f-стойностите, резултатът е $\langle a'_1, \ldots, a'_n \rangle$
2. $A' \leftarrow \varnothing$
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($A = \{a_1, \ldots, a_n\}$, където $n \geqslant 1$, $a_i = (s_i, f_i)$ и $s_i < f_i$ за $1 \leqslant i \leqslant n$)

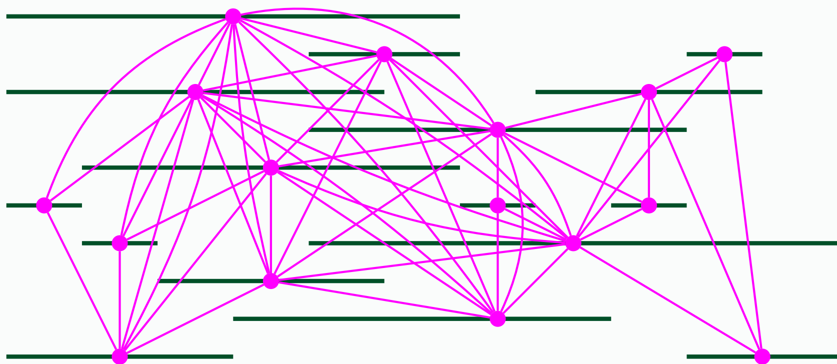
1. сортирай A по f-стойностите, резултатът е $\langle a'_1, \ldots, a'_n \rangle$
2. $A' \leftarrow \{a'_1\}$
3. $f^* \leftarrow f'_1$
4. **for** $i \leftarrow 2$ **to** n
5.   **if** $f^* \leqslant s'_i$
6. $A' \leftarrow A' \cup \{a'_i\}$
7. $f^* \leftarrow f'_i$
8. **return** A'

Коректността се обосновава с Теорема 65. 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.

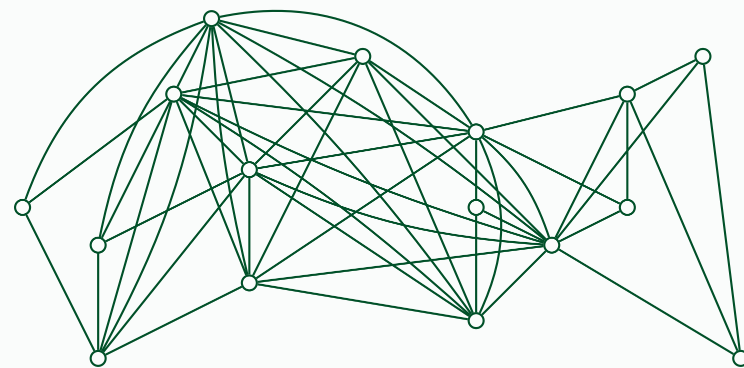
Допълнение 54: МАКСИМАЛНА АНТИКЛИКА ВЪРХУ ИНТЕРВАЛНИ ГРАФИ

Задачата МАКСИМАЛНА АНТИКЛИКА е дефинирана на стр. 589. Тук ще покажем, че алгоритъмът INTERVAL SCHEDULING в някакъв смисъл решава МАКСИМАЛНА АНТИК-

лика. Нека е дадено множество интервали $A = \{a_1, \ldots, a_n\}$, 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. 545 и да си представим въпросния графа, нарисуван върху рисунката на интервалите.



Този граф съответства точно на застъпванията на интервалите. Нека го разгледаме самостоятелно.



Графите, които определият по такъв начин застъпванията в множество от интервали, се наричат интервални графи.

Определение 90: Интервален граф

Граф $G = (V, E)$ се нарича *интервален граф*, ако съществува множество от интервали $\mathcal{I} = \{i_1, \ldots, i_n\}$, 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 $\phi(u) \cap \phi(v) = \emptyset$.

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. [55, 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 [55, 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 [24].

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. 589); 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 12.9.1, 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 преставяне 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.

12.7.1.2 Решение по схемата Динамично Програмиране

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 краят, 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. Тази задача не може да се реши ефикасно с леко обобщение на алчния алгоритъм от предишната подсекция. Ще видим реше-

ние, конструирано по схемата **Динамично Програмиране**, което има същата сложност по време $\Theta(n \lg n)$ като алчното решение, но е несравнимо по-изтъчнено. Естествено, решението с динамично програмиране решава и задачата без тегла, като дава едно и също тегло на всеки интервал.

Нека множеството от интервалите е $A = \{a_1, \ldots, a_n\}$, а сортираната им редица по f-стойности е $S = (a'_1, \ldots, a'_n)$. Това означава, че

$$f'_1 \leqslant f'_2 \leqslant \cdots \leqslant f'_n$$

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 $\phi : \{1, \ldots, n\} \rightarrow \{0, \ldots, n-1\}$. За всяко $i \in \{1, \ldots, n\}$, нека $C(i)$ е множеството от интервалите в S, които са вляво от i и са съвместими с i. Формалната дефиниция на ϕ е

$$\forall i \in \{1, \ldots, n\} : \phi(i) \stackrel{\text{def}}{=} \begin{cases} \max\{j \mid a'_j \in C(i)\}, & \text{ако } C(i) \neq \emptyset \\ 0, & \text{ако } C(i) = \emptyset \end{cases}$$

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 $\phi[1 \mathinner{..} n]$. Схемата за изчисление е:

$$\text{opt}[0] = 0$$
$$\text{opt}[i] = \max\left(\text{opt}[i-1], w'_i + \text{opt}[\phi[i]]\right), \text{ ако } i > 0$$

Следният итеративен алгоритъм решава ефикасно задачата.

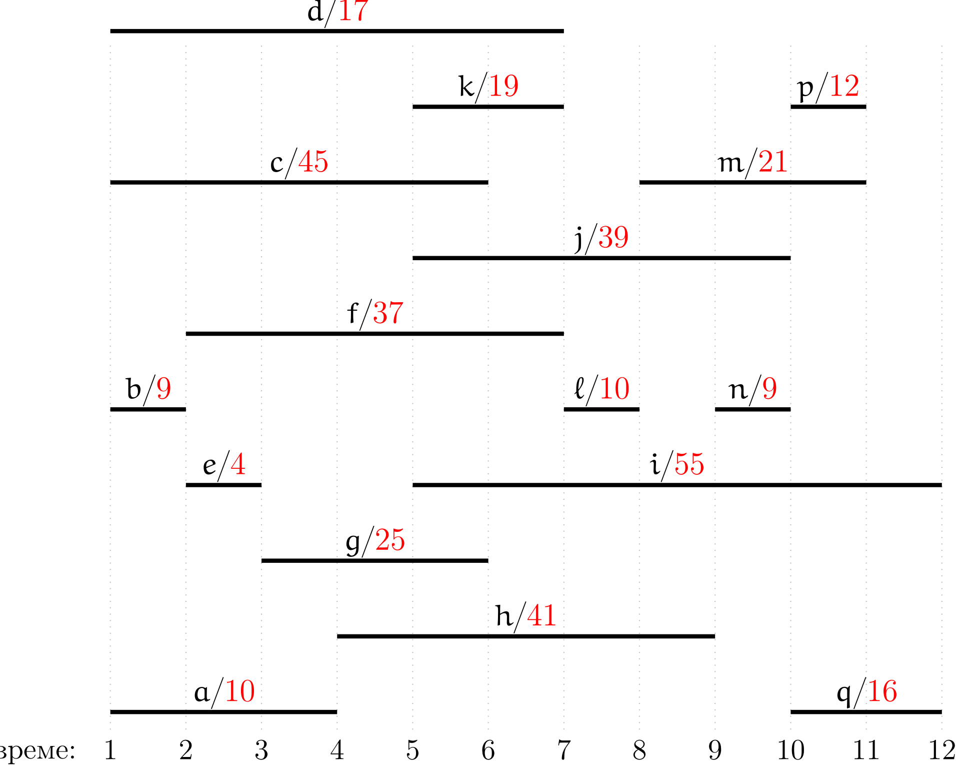
Weighted Interval Scheduling($A = \{a_1, \ldots, a_n\}$, където $n \geqslant 1$, $a_i = (s_i, f_i)$, $s_i < f_i$ и a_i има тегло w_i, за $1 \leqslant i \leqslant n$)

```
1   сортирай A по f-стойностите, резултатът е ⟨a'_1, ..., a'_n⟩
2   opt[0] ← 0
3   for i ← 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 $\phi[1 \mathinner{..} n]$ 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 $\phi[1 \mathinner{..} n]$ и оттам, квадратична сложност на целия алгоритъм. За щастие, стойностите на масива $\phi[1 \mathinner{..} n]$ може да се пресметнат във време $O(n \lg n)$ така: за всяко a'_i, с двоично търсене намираме максималното j, такова че $f'_j \leqslant s'_i$. 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 $\phi[1 \mathinner{..} n]$ във време $O(n \lg n)$, очевидно имаме алгоритъм със сложност по време $\Theta(n \lg n)$.

Да разгледаме пример. Интервалите са именувани $\mathfrak{a}, \ldots, \mathfrak{q}$ (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 $\phi[1 \ldots 16]$:

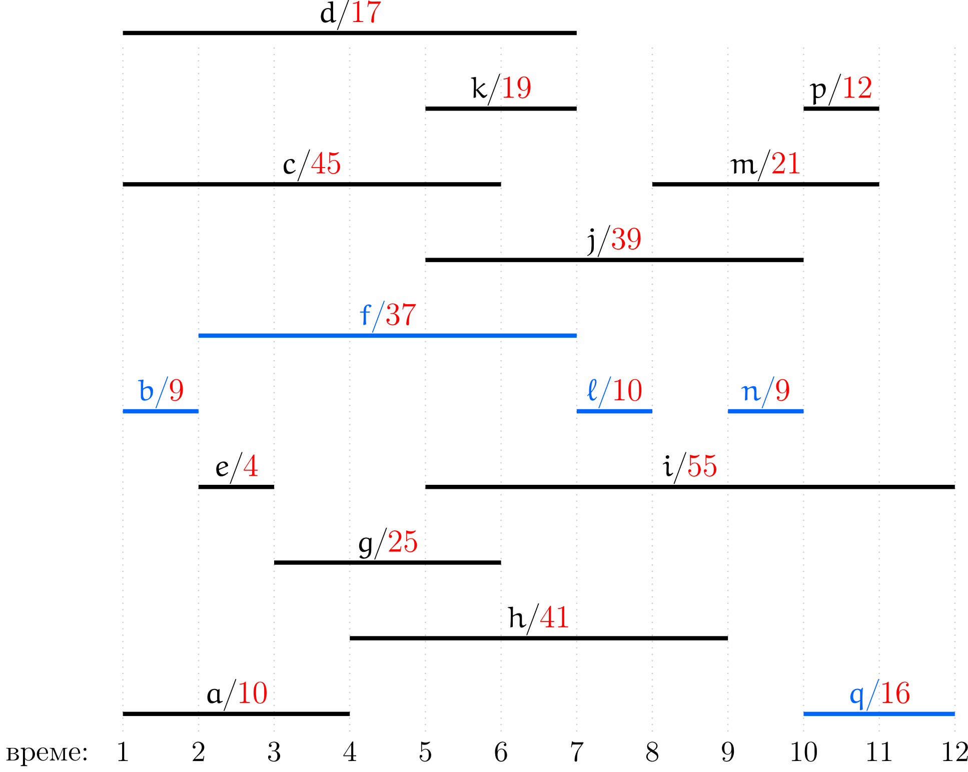
x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\phi(x)$	0	1	0	2	0	1	3	0	8	3	10	3	9	12	12	3

Ето и изчисленията за $opt[16]$:
$opt[1] = \max(0, 9 + opt[0]) = 9.$ ✓
$opt[2] = \max(9, 4 + opt[1]) = 13.$ ✓
$opt[3] = \max(13, 10 + opt[0]) = 13.$
$opt[4] = \max(13, 25 + opt[2]) = 38.$ ✓
$opt[5] = \max(38, 45 + opt[0]) = 45.$ ✓
$opt[6] = \max(45, 37 + opt[1]) = 46.$ ✓
$opt[7] = \max(46, 19 + opt[3]) = 46.$
$opt[8] = \max(46, 17 + opt[0]) = 46.$
$opt[9] = \max(46, 10 + opt[8]) = 56.$ ✓
$opt[10] = \max(56, 41 + opt[3]) = 56.$
$opt[11] = \max(56, 9 + opt[10]) = 65.$ ✓

$\mathrm{opt}[12] = \max(65, 39 + \mathrm{opt}[3]) = 65.$
$\mathrm{opt}[13] = \max(65, 21 + \mathrm{opt}[9]) = 77.$ ✓
$\mathrm{opt}[14] = \max(77, 12 + \mathrm{opt}[12]) = 77.$
$\mathrm{opt}[15] = \max(77, 16 + \mathrm{opt}[12]) = 81.$ ✓
$\mathrm{opt}[16] = \max(81, 55 + \mathrm{opt}[3]) = 81.$
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 opt[16], с червен тик е отбелязан всеки ред, в който стойността е получена от втория аргумент на max, а именно $\boxed{w'_i + \mathrm{opt}[\phi[i]]}$ 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 opt[15]. Виждаме, че q участва в оптималното решение и освен това opt[15] е получен като $\boxed{16 + \mathrm{opt}[12] = 81}$. Отиваме на реда на opt[12]. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на opt[11]. Тогава n участва в оптималното решение и освен това opt[11] е получен като $\boxed{9 + \mathrm{opt}[10] = 65}$. Отиваме на реда на opt[10]. Той няма тик, така че се връщаме до първия ред назад с тик, който е реда на opt[9]. Тогава ℓ участва в оптималното решение и освен това opt[9] е получен като $\boxed{10 + \mathrm{opt}[8] = 56}$. Отиваме на реда на opt[8]. Той няма тик, така че се връщаме до първия ред назад с тик, който е редът на opt[6]. Тогава f участва в оптималното решение и освен това opt[6] е получен като $\boxed{37 + \mathrm{opt}[1] = 46}$. Отиваме на реда на 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.



12.8 Задачи върху редици и стрингове

Както обикновено, редиците са крайни (Конвенция 3). *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.

12.8.1 Longest Increasing Subsequence

Определение 91: Поредица и растяща поредица

Нека е дадена редица $S = (a_1, a_2, \ldots, a_n)$. *Поредица в* S е всяка редица $(a_{i_1}, a_{i_2}, \ldots, a_{i_k})$, такава че $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$. *Растяща поредица в* S е всяка поредица $(a_{i_1}, a_{i_2}, \ldots, a_{i_k})$, такава че $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$. *Максимална растяща поредица в* S е растяща поредица в S с максимална дължина.

Изч. Задача 41: Longest Increasing Sequence

екземпляр: Редица $S = (a_1, a_2, , \ldots, a_n)$.
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 преминаване, 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?

12.8.1.1 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 $k \in \{1, \ldots, n\}$.

Нека да мислим за рекурсивна декомпозиция за тази характеризация. Нека

$$
\begin{aligned}
S^1 &= (a_1) \\
S^2 &= (a_1, a_2) \\
S^3 &= (a_1, a_2, a_3) \\
&\cdots \\
S^n &= (a_1, a_2, \ldots, a_n) = S
\end{aligned}
$$

Нека ℓ_k е дължината на максимална растяща поредица в S^k, завършваща на a_k, за $1 \leqslant k \leqslant n$. Ако имаме тези ℓ_k, то ние сме готови! Тогава отговорът е $\max\{\ell_k \mid 1 \leqslant k \leqslant n\}$, тъй като всяка максимална растяща поредица завършва на някой елемент.

Как да получим ℓ_k от $\ell_{k-1}, \ldots, \ell_1$? Само числата $\ell_{k-1}, \ldots, \ell_1$ не са достатъчни, но двойките $(\ell_{k-1}, a_{k-1}), \ldots, (\ell_1, a_1)$ 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, 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

$$\forall k \in \{1, \ldots, n\} : R_k = \{j \in \{1, \ldots, k-1\} \mid a_j < a_k\}$$

Тогава, за $1 \leqslant k \leqslant n$:

$$
\ell_k = \begin{cases} \max\{\ell_j \mid j \in R_k\} + 1, & \text{ако } R_k \neq \emptyset \\ 1, & \text{ако } R_k = \emptyset \end{cases}
$$

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 $k \in \{1, \ldots, n\}$ 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, ..., a_n))
1  ℓ[1] ← 1
2  π[1] ← nil
3  for k ← 2 to n
4      ℓ[k] ← 1
5      π[k] ← nil
6      for j ← 1 to k − 1
7          if ℓ[k] < ℓ[j] + 1 and a[k] > a[j]
8              ℓ[k] ← ℓ[j] + 1
9              π[k] ← j
10 return (max_{1≤k≤n} {ℓ[k]}, π)
```

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
ℓ[k]	1	1	1	2	2	3	4	2	5	6	5	6	7	7
π[k]	nil	nil	nil	3	3	5	6	3	7	9	7	11	10	12

Едно оптимално решение е маркирано в синьо.

12.8.1.2 Дин. Прогр. със сложност $\Theta(n \lg n)$

Оригиналната статия с решение на задачата LONGEST INCREASING SUBSEQUENCE на Fredman от 1972 г. [49] 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 Pucci* от Университета на Padova, Италия. Авторът на тези лекционни записки е на мнение, че има дребна формална грешка в рекурсивната декомпозиция на професор Pucci, поради което изложението тук се различава от неговото. Също така, тук се ползва различна нотация.

В Подподсекция 12.8.1.1 характеризирахме оптималното решение като поредица, завършваща с някакъв елемент a_k. Сега ще характеризираме по друг начин. Нека S^k има смисълът от Подподсекция 12.8.1.1. За всяко $k \in \{1, \ldots, n\}$, 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 отпускаме 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 61

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 "**м**аксимална **р**астяща **п**оредица с **м**инимален **з**авършек". 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 $(a_{i_1}, \ldots, a_{i_j})$.

- Ако $a_{i_j} < a_{k+1}$, със сигурност $(a_{i_1}, \ldots, a_{i_j}, a_{k+1})$ 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 $(a_{i_1}, \ldots, a_{i_j}, a_{k+1})$ 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 $(a_{t_1}, \ldots, a_{t_{j-1}})$, такава че $a_{t_{j-1}} < a_{k+1}$. Забележете, че $(a_{t_1}, \ldots, a_{t_{j-1}})$ не е максимална растяща поредица в S^k, имайки дължина само $j - 1$, но в S^{k+1}, $(a_{t_1}, \ldots, a_{t_{j-1}}, a_{k+1})$ е растяща поредица с дължина j. Тогава и $(a_{i_1}, \ldots, a_{i_j})$, и $(a_{t_1}, \ldots, a_{t_{j-1}}, a_{k+1})$ са максимални растящи поредици в S^{k+1}, но $(a_{t_1}, \ldots, a_{t_{j-1}}, a_{k+1})$ е МРПМЗ, а $(a_{i_1}, \ldots, a_{i_j})$ 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 $j, k \in \{1, \ldots, n\}$, 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 60

В текущия контекст, нека за някое j, L_j^k и L_{j+1}^k са непразни. Нека $L_j^k = (a_{i_1}, \ldots, a_{i_j})$ и $L_{j+1}^k = (a_{t_1}, \ldots, a_{t_j}, a_{t_{j+1}})$, 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 $(a_{t_1}, \ldots, a_{t_j})$ 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:

$$L_1^k = (\min(S^k)), \quad \text{за } 1 \leqslant k \leqslant n \tag{12.33}$$

$$L_j^k = (\,), \quad \text{за } 1 \leqslant k < j \leqslant n \tag{12.34}$$

Коректността им е очевидна. Самата рекурсия изглежда така. При $2 \leqslant k \leqslant n$ и $2 \leqslant j \leqslant k$:

$$L_j^k = L_j^{k-1}, \qquad \text{ако } \ell_j^{k-1} < a_k \tag{12.35}$$

$$L_j^k = L_j^{k-1}, \qquad \text{ако } a_k < \ell_j^{k-1} \text{ и } a_k < \ell_{j-1}^{k-1} \tag{12.36}$$

$$L_j^k = \text{concat}(L_{j-1}^{k-1}, a_k), \qquad \text{ако } \ell_{j-1}^{k-1} < a_k < \ell_j^{k-1} \tag{12.37}$$

Ето обосновката.

- В (12.35), 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 (12.35), 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 (12.36), $a_k < \ell_{j-1}^{k-1}$.

    * 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 $\ell_{j-1}^{k-1} < a_k < \ell_j^{k-1}$ (12.37), 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. 554.

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 (12.33), (12.34), (12.35), (12.36) и (12.37), получаваме следните поредици (само непразните поредици са показани):

$L_1^1 = (14)$
$L_1^2 = (9)$
$L_1^3 = (-5)$
$L_1^4 = (-5)$
$L_2^4 = (-5, 11)$
$L_1^5 = (-5)$
$L_2^5 = (-5, -2)$
$L_1^6 = (-5)$
$L_2^6 = (-5, -2)$
$L_3^6 = (-5, -2, 0)$
$L_1^7 = (-5)$
$L_2^7 = (-5, -2)$
$L_3^7 = (-5, -2, 0)$
$L_4^7 = (-5, -2, 0, 3)$
$L_1^8 = (-5)$
$L_2^8 = (-5, -4)$
$L_3^8 = (-5, -2, 0)$
$L_4^8 = (-5, -2, 0, 3)$

$L_1^9 = (-5)$
$L_2^9 = (-5, -4)$
$L_3^9 = (-5, -2, 0)$
$L_4^9 = (-5, -2, 0, 3)$
$L_5^9 = (-5, -2, 0, 3, 12)$
$L_1^{10} = (-5)$
$L_2^{10} = (-5, -4)$
$L_3^{10} = (-5, -2, 0)$
$L_4^{10} = (-5, -2, 0, 3)$
$L_5^{10} = (-5, -2, 0, 3, 12)$
$L_6^{10} = (-5, -2, 0, 3, 12, 19)$
$L_1^{11} = (-5)$
$L_2^{11} = (-5, -4)$
$L_3^{11} = (-5, -2, 0)$
$L_4^{11} = (-5, -2, 0, 3)$
$L_5^{11} = (-5, -2, 0, 3, 6)$
$L_6^{11} = (-5, -2, 0, 3, 12, 19)$

$L_1^{12} = (-5)$
$L_2^{12} = (-5, -4)$
$L_4^{12} = (-5, -2, 0, 3)$
$L_5^{12} = (-5, -2, 0, 3, 6)$
$L_6^{12} = (-5, -2, 0, 3, 6, 8)$
$L_1^{13} = (-5)$
$L_2^{13} = (-5, -4)$
$L_3^{13} = (-5, -2, 0)$
$L_4^{13} = (-5, -2, 0, 3)$
$L_5^{13} = (-5, -2, 0, 3, 6)$
$L_6^{13} = (-5, -2, 0, 3, 6, 8)$
$L_7^{13} = (-5, -2, 0, 3, 6, 8, 22)$
$L_1^{14} = (-5)$
$L_2^{14} = (-5, -4)$
$L_3^{14} = (-5, -2, 0)$
$L_4^{14} = (-5, -2, 0, 3)$
$L_5^{14} = (-5, -2, 0, 3, 6)$
$L_6^{14} = (-5, -2, 0, 3, 6, 8)$
$L_7^{14} = (-5, -2, 0, 3, 6, 8, 10)$

Виждаме добра илюстрация на Лема 60: 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 $k \in \{1, \ldots, 14\}$, за всяко $j \in \{1, \ldots, 14\}$, **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:

$$
\begin{aligned}
k = 1 :&\quad L^1_1 = (14),\ L^1_2 = (\,),\ L^1_3 = (\,),\ \ L^1_4 = (\,), \ldots,\ L^1_{14} = (\,)\\
k = 2 :&\quad L^2_1 = (9),\ L^2_2 = (\,),\ L^2_3 = (\,),\ \ L^2_4 = (\,), \ldots,\ L^2_{14} = (\,)\\
k = 3 :&\quad L^3_1 = (-5),\ L^3_2 = (\,),\ L^3_3 = (\,),\ \ L^3_4 = (\,), \ldots,\ L^3_{14} = (\,)\\
k = 4 :&\quad L^4_1 = (-5),\ L^4_2 = (-5, 11),\ L^4_3 = (\,),\ \ L^4_4 = (\,), \ldots,\ L^4_{14} = (\,)\\
k = 5 :&\quad L^5_1 = (-5),\ L^5_2 = (-5, -2),\ L^5_3 = (\,),\ \ L^5_4 = (\,), \ldots,\ L^5_{14} = (\,)\\
k = 6 :&\quad L^6_1 = (-5),\ L^6_2 = (-5, -2),\ L^6_3 = (-5, -2, 0),\ L^6_4 = (\,),\ \ldots,\ L^6_{14} = (\,)\\
&\quad \ldots\\
k = 13 :&\quad L^{13}_1 = (-5),\ \ldots,\ L^{13}_7 = (-5, -2, 0, 3, 6, 8, 22),\ L^{13}_8 = (\,),\ \ldots,\ L^{13}_{14} = (\,)\\
k = 14 :&\quad L^{14}_1 = (-5),\ \ldots,\ L^{14}_7 = (-5, -2, 0, 3, 6, 8, 10),\ L^{14}_8 = (\,),\ \ldots,\ L^{14}_{14} = (\,)
\end{aligned}
$$

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 $L^2_1 = (9)$ се различават, а за $j \in \{2, \ldots, 14\}$ имаме $L^1_j = L^2_j = (\,)$,
- при прехода от $k = 13$ към $k = 14$ е вярно, че L^{13}_7 и L^{14}_7 се различават, а за $j \in \{1, \ldots, 6, 8, \ldots, 14\}$ имаме $L^{13}_j = L^{14}_j$.

Това не е случайно. В *ресурса на професор Geppino Pucci*, на страница 4 се казва (написано в текущата терминология):

> *...тъй като завършеците са в нарастващ ред съгласно Лема 60, 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 66: Обосновка на бързия алг. за LONGEST INCREASING SEQUENCE

За всяко изпълнение на рекурсивния алгоритъм, определен от (12.33), (12.34), (12.35), (12.36) и (12.37), за всяко $k \in \{2, \ldots, n\}$ съществува точно едно $j \in \{1, \ldots, n\}$, такова че $L^{k-1}_j \neq L^k_j$.

Доказателство: Да разгледаме отново (12.35), (12.36) и (12.37). Тъй като редицата

$$(\ell^{k-1}_1, \ell^{k-1}_2, \ldots, \ell^{k-1}_k)$$

е сортирана съгласно Лема 60 и освен това числата в S са две по две различни, наистина има точно едно j, такова че a_k попада между ℓ^{k-1}_{j-1} и ℓ^{k-1}_j. Това е в сила

- и когато $\ell^{k-1}_j < \infty$, 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 (12.37), 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 (12.35) или (12.36). $\square$

Например да разгледаме списъка от непразните поредици на стр. 559.

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

$$14,\ \infty,\ \ldots,\ \infty$$

Минавайки към $k = 2$, имаме $S_2 = 9$, което попада върху 14, и редицата от завършеците е

$$9,\ \infty,\ \ldots,\ \infty$$

Минавайки към $k = 3$, имаме $S_3 = -5$, което попада върху 9, и редицата от завършеците е

$$-5,\ \infty,\ \ldots,\ \infty$$

Минавайки към $k = 4$, имаме $S_4 = 11$, което попада върху най-лявата ∞, и редицата от завършеците е

$$-5,\ 11, \infty,\ \ldots,\ \infty$$

Минавайки към $k = 5$, имаме $S_5 = -2$, което попада върху 11, и редицата от завършеците е

$$-5,\ -2, \infty,\ \ldots,\ \infty$$

Минавайки към $k = 6$, имаме $S_6 = 0$, което попада върху най-лявата ∞, и редицата от завършеците е

$$-5,\ -2,\ 0,\ \infty,\ \ldots,\ \infty$$

И така нататък.

не е довършено...

12.8.1.3 Patiencesort

Статията на Aldous и Diaconis [5] съдържа много подробна информация за теоретичната основа на тази сортировка, както и сложни и нетривиални резултати за асимптотиките на максималните растящи поредици.

12.8.1.4 Building Bridges

12.8.2 Longest Common Subsequence

Сега разглеждаме редици над някаква крайна азбука Σ. Както се разбрахме в началото на Подсекция 12.8, такива редици се наричат стрингове.

Изч. Задача 42: Longest Common Sequence

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 $0 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant m$, 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 $i \in \{1, \ldots, n\}$ и някакво $j \in \{1, \ldots, m\}$. 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:

$$X^i = \mathtt{c\,d\,e\,f}$$
$$Y^j = \mathtt{g\,h\,i\,c}$$

Очевидно $X^{i-1} = \mathtt{cde}$ и $Y^{j-1} = \mathtt{ghi}$, така че $Z = \epsilon$. А максималната обща поредица за X^i и Y^j е $\mathtt{c}$, като $\mathtt{c}$ 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,

$$M(i, j) = \begin{cases} 0, & \text{ако } i = 0 \text{ или } j = 0, \\ M(i-1, j-1) + 1, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i = y_j \\ \max\{M(i, j-1), M(i-1, j)\}, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i \neq y_j \end{cases} \tag{12.38}$$

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 $0 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant m$ 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 с ясен смисъл: `diag` означава, че сме дошли от $[i-1,j-1]$, `left` означава, че 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 $i \geqslant 1$ и $j \geqslant 1$ и $x_i = y_j$, то $M[i,j]$.dir = `diag`,
  - ако $i \geqslant 1$ и $j \geqslant 1$ и $x_i \neq y_j$ и $M[i,j-1]$.val $\geqslant M[i-1,j]$.val, то $M[i,j]$.dir = `left`,
  - ако $i \geqslant 1$ и $j \geqslant 1$ 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 ∈ ℕ⁺, X[1..n] и Y[1..m]: стрингове над Σ)
 1  създай M[0..n, 0..m]
 2  for i ← 0 to n
 3      M[i,0].val ← 0
 4      M[i,0].dir ← Nil
 5  for j ← 1 to m
 6      M[0,j].val ← 0
 7      M[0,j].dir ← Nil
 8  for i ← 1 to n
 9      for j ← 1 to m
10          if X[i] = Y[j]
11              M[i,j].val ← M[i−1,j−1].val + 1
12              M[i,j].prev ← diag
13          else if M[i−1,j].val ⩾ M[i,j−1].val
14              M[i,j].val ← M[i−1,j].val
15              M[i,j].prev ← up
16          else
17              M[i,j].val ← M[i,j−1].val
18              M[i,j].prev ← left
19  return M
```

Ето малък пример. Нека азбуката е $\{$`A`, `B`, `C`$\}$ и

X = `CCBACCB`

Y = `ABCABCABCABC`

Таблицата е 8×13, индексирана от 0 по редове и колони. Началните условия са:

- що се отнася до числата, всички клетки в нулевия ред и нулевата колона съдържат 0;
- що се отнася до посоките, във всеки елемент на нулевия ред и на нулевата колона посоката е Nil, но тези Nil-ове не са нарисувани.

M

Y

X

		ϵ 0	A 1	B 2	C 3	A 4	B 5	C 6	A 7	B 8	C 9	A 10	B 11	C 12
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	0												
C	2	0												
B	3	0												
A	4	0												
C	5	0												
C	6	0												
B	7	0												

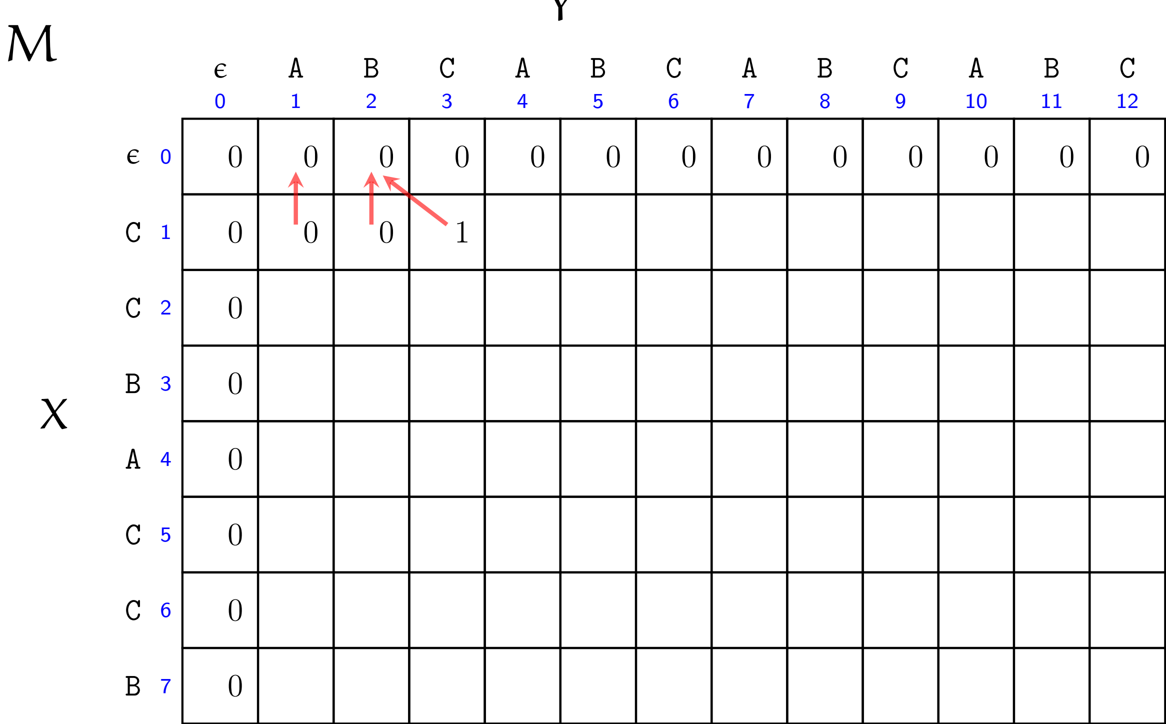
Да видим $M[1,1]$ какво е. Разглеждаме $i = j = 1$. Тъй като $x_1 = \texttt{C}$ и $y_1 = \texttt{A}$, условието на ред 10 е лъжа и изпълнението отива на ред 13. $M[i-1,j].\text{val}$ е $M[0,1].\text{val} = 0$, $M[i,j-1].\text{val}$ е $M[1,0].\text{val} = 0$, условието е истина и се изпълняват редове 14 и 15, поради което $M[1,1].\text{val}$ получава стойност $M[i-1,j].\text{val} = M[0,1].\text{val} = 0$, а $M[1,1].\text{prev}$ става up.

Абсолютно аналогично, при $i = 1$ и $j = 2$ имаме $x_1 = \texttt{C}$ и $y_2 = \texttt{B}$, така че $M[1,2].\text{val}$ получава стойност $M[i-1,j].\text{val} = M[0,2].\text{val} = 0$, а $M[1,2].\text{prev}$ става up.

M, Y, X

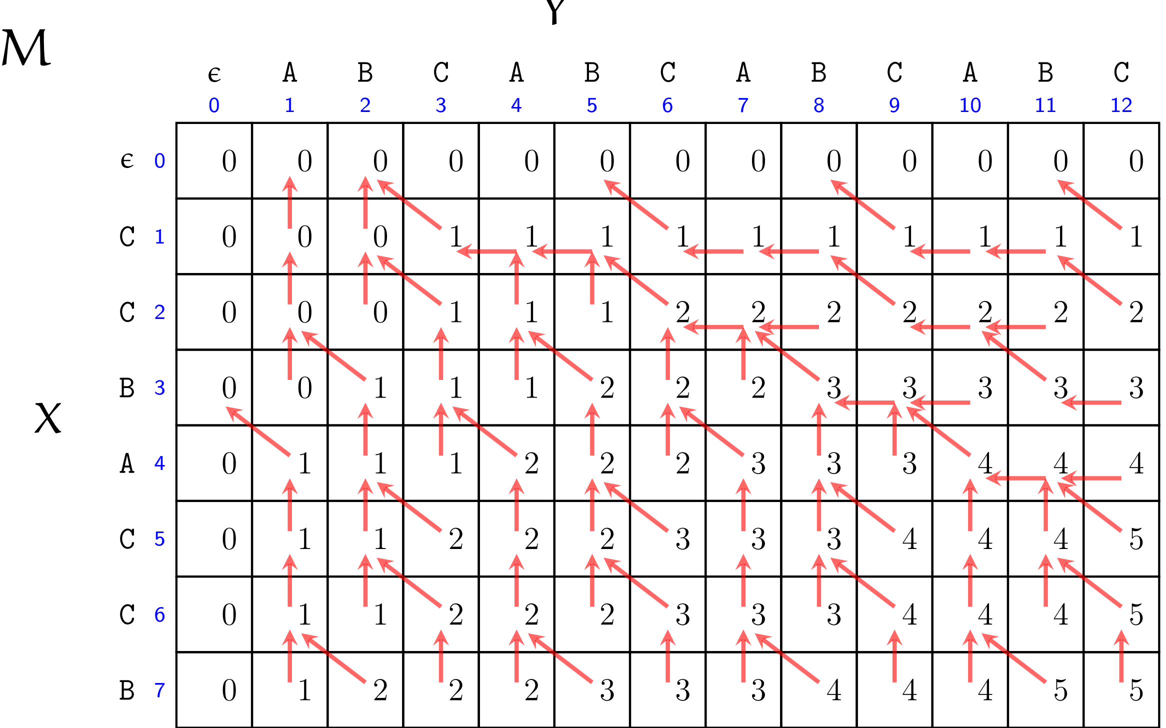
	ϵ 0	A 1	B 2	C 3	A 4	B 5	C 6	A 7	B 8	C 9	A 10	B 11	C 12
ϵ 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0										
C 2	0												
B 3	0												
A 4	0												
C 5	0												
C 6	0												
B 7	0												

Да видим $M[1,3]$ какво е. Разглеждаме $i = j = 3$. Тъй като $x_1 = \texttt{C}$ и $y_3 = \texttt{C}$, 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`.

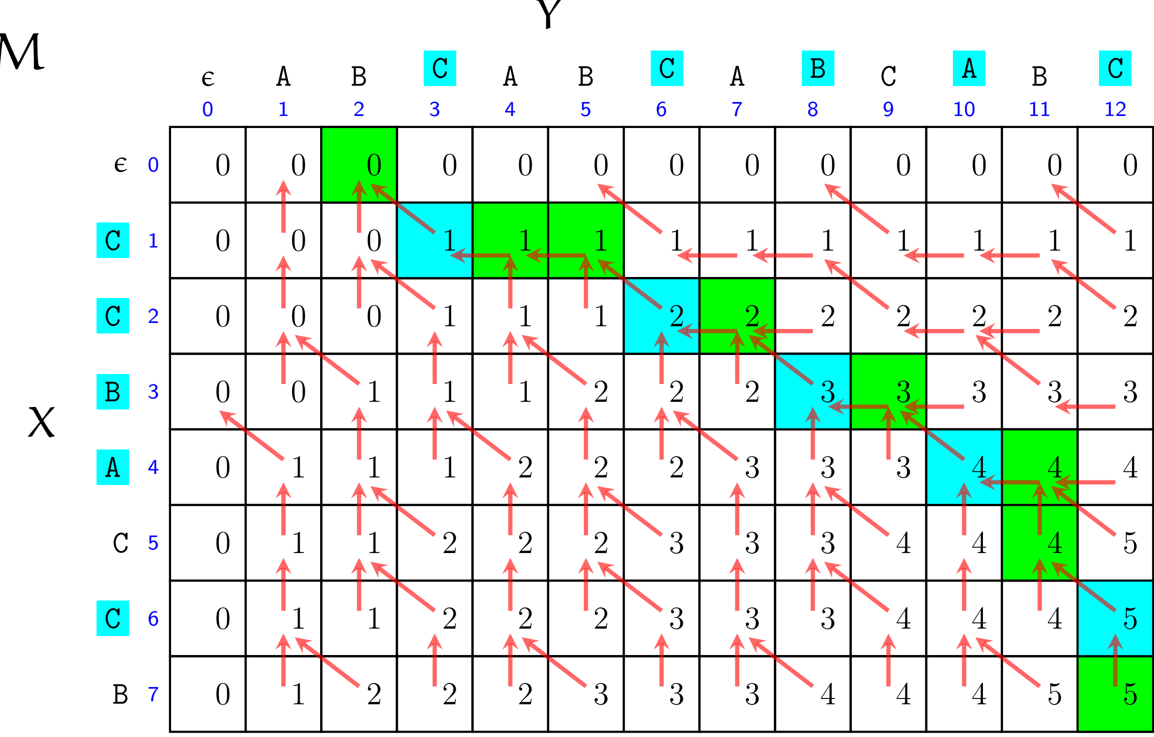


	ϵ 0	A 1	B 2	C 3	A 4	B 5	C 6	A 7	B 8	C 9	A 10	B 11	C 12
ϵ 0	0	0	0	0	0	0	0	0	0	0	0	0	0
C 1	0	0	0	1									
C 2	0												
B 3	0												
A 4	0												
C 5	0												
C 6	0												
B 7	0												

По аналогичен начин запълваме цялата таблица.



Разполагайки с пълната таблица, лесно може да генерираме решение с обратно проследяване. Ще следваме указателите prev, изобразени с червени стрелки, тръгвайки от клетка $M[7, 12]$, докато не срещнем клетка, чийто указател е `Nil` (без изображение на рисунката). Ето клетките, през които се минава, показани в зелено или циан. Циановите клетки съответсват 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 съответсват 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`).



Оптималното решение, което получихме, е поредицата `CCBAC`. Ето я в контекста на двата

стринга:

$X = \mathtt{CCBACCB}$

$Y = \mathtt{ABCABCABCABC}$

12.8.3 Sequence Alignment

Става дума за добре известна задача от областта на изчислителната биология. Дадени са две ДНК или две РНК или два протеина. Иска се да се намери подобието 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. Иска 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 стринговета, 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 [65]:

> *Among the most useful computer-based tools in modem 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 $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Нека двата стринга, в които търсим общи подстрингове, са X и Y:

$$
\begin{array}{lcl}
X & = & \mathtt{C\ C\ A\ A\ C\ C\ G\ G\ A\ T\ G\ C\ T\ A} \\
Y & = & \mathtt{A\ A\ C\ C\ G\ G\ A\ T\ G\ G\ C\ T\ A}
\end{array}
$$

Примерът е достатъчно малък и недвусмислен, така че общите подстрингове се идентифицират на око веднага, като няма съмнение кои са те:

$$
\begin{array}{lcl}
X & = & \mathtt{C\ C\ A\ A\ C\ C\ G\ G\ A\ T\ G\ C\ T\ A} \\
Y & = & \mathtt{A\ A\ C\ C\ G\ G\ A\ T\ G\ G\ C\ T\ A}
\end{array}
$$

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 $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \sqcup\}$. Ето как изглеждат X и Y с вмъкнати на подходящи места шпации, което ги превръща в съответно X' и Y':

$$
\begin{array}{lcccccccccccccccc}
X' & = & \mathtt{C} & \mathtt{C} & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{G} & \sqcup & \mathtt{C} & \mathtt{T} & \mathtt{A} \\
Y' & = & \sqcup & \sqcup & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{A} & \mathtt{T} & \mathtt{G} & \mathtt{G} & \mathtt{C} & \mathtt{T} & \mathtt{A}
\end{array}
$$

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:

$$
\begin{array}{lccccccccccccccccccc}
X'' & = & \mathtt{C} & \mathtt{C} & \sqcup & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{A} & \sqcup & \mathtt{T} & \mathtt{G} & \sqcup & \mathtt{C} & \mathtt{T} & \mathtt{A} \\
Y'' & = & \sqcup & \sqcup & \sqcup & \mathtt{A} & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{A} & \sqcup & \mathtt{T} & \mathtt{G} & \mathtt{G} & \mathtt{C} & \mathtt{T} & \mathtt{A}
\end{array}
$$

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 $\Sigma \setminus \{\sqcup\}$. *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 $X = \mathtt{ACCT}$, следните стрингове са попълвания на X: $\boxed{\mathtt{A}\sqcup\mathtt{C}\sqcup\mathtt{C}\sqcup\mathtt{T}}$, $\boxed{\sqcup\sqcup\mathtt{AC}\sqcup\sqcup\sqcup\mathtt{CT}\sqcup}$ и $\boxed{\mathtt{ACCT}}$.

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 $X = \mathtt{AGTCGTC}$ и $Y = \mathtt{GACTCAGG}$.

- Следното подравняване има тегло -6. Под всяка позиция в попълнените стрингове е написано нейното тегло.

$$
\begin{array}{cccccccccc}
\mathtt{A} & \mathtt{G} & \sqcup & \mathtt{T} & \sqcup & \mathtt{C} & \sqcup & \mathtt{G} & \mathtt{T} & \mathtt{C} \\
\sqcup & \mathtt{G} & \mathtt{A} & \mathtt{C} & \mathtt{T} & \mathtt{C} & \mathtt{A} & \mathtt{G} & \sqcup & \mathtt{G} \\
\hline
-2 & +2 & -2 & -1 & -2 & +2 & -2 & +2 & -2 & -1
\end{array}
$$

  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 43: 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 минимално 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 $0\leqslant i\leqslant n$ и $0\leqslant j\leqslant m$, 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 $1\leqslant i\leqslant n$, не са нули, и $M(0,j)$, за $1\leqslant j\leqslant m$, не са нули. Да разсъждаваме за $M(i,0)$ за $i\geqslant 1$. Тъй като подравняваме (под)стрингове с **еднаква** дължина, $M(i,0)$ е теглото на подравняване на два стринга с държина 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 попълваме 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 $1\leqslant i\leqslant n$. Аналогично, $M(0,j)=-2j$ за $1\leqslant j\leqslant m$.

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:

$$M(i,j) = \begin{cases} 0, & \text{ако } i = 0 \text{ и } j = 0, \\ -2i, & \text{ако } i \geqslant 1 \text{ и } j = 0, \\ -2j, & \text{ако } i = 0 \text{ и } j \geqslant 1, \\ \max\{M(i-1,j-1)+\delta, M(i-1,j)-2, M(i,j-1)-2\}, & \text{ако } i \geqslant 1 \text{ и } j \geqslant 1. \end{cases} \quad (12.39)$$

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, ..., $-2n$, и ред 0, запълнен, отляво надясно, с 0, -2, -4, ..., $-2m$. 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 $x_1 \cdots x_i$ и $y_1 \cdots y_j$;
- $M[i,j]$.dir: елемент на $\{\texttt{up}, \texttt{left}, \texttt{diag}, \texttt{Nil}\}$, който указва предната клетка – тази, от която сме "влезли" в $[i,j]$. Имената на елементите би трябвало да с ясен смисъл: `diag` означава, че сме дошли от $[i-1,j-1]$, `left` означава, че 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 $M[i,j].\text{dir} = \texttt{Nil}$,
    - ако $i = 0$ и $j \geqslant 1$, то $M[i,j].\text{dir} = \texttt{left}$,
    - ако $i \geqslant 1$ и $j = 0$, то $M[i,j].\text{dir} = \texttt{up}$,
    - ако $i \geqslant 1$ и $j \geqslant 1$, 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 $M[i,j].\text{dir} = \texttt{diag}$, 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 $M[i,j].\text{dir} = \texttt{left}$,
        * ако $\max(M[i-1,j-1]+\delta, M[i-1,j]-2, M[i,j-1]-2\} = M[i-1,j]-2$, то $M[i,j].\text{dir} = \texttt{up}$,

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 ← 0
3   M[0,0].dir ← Nil
4   for i ← 1 to n
5       M[i,0].val ← −2i
6       M[i,0].dir ← up
7   for j ← 1 to m
8       M[0,j].val ← −2j
9       M[0,j].dir ← left
10  for i ← 1 to n
11      for j ← 1 to m
12          if X[i] = Y[j]
13              δ ← 2
14          else
15              δ ← −1
16          themax ← max(M[i−1,j−1].val + δ, M[i−1,j].val − 2, M[i,j−1].val − 2)
17          if themax = M[i−1,j−1].val + δ
18              M[i,j].val ← M[i−1,j−1].val + δ
19              M[i,j].dir ← diag
20          else if themax = M[i−1,j].val − 2
21              M[i,j].val ← M[i−1,j].val − 2
22              M[i,j].dir ← up
23          else
24              M[i,j].val ← M[i,j−1].val − 2
25              M[i,j].dir ← left
26  return M
```

Теглото на оптималното подравняване е $M[n, m]$.val. Ако искаме и накое 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 ← n, q ← m
3  while True do {
4      if p = 0 and q = 0
5          break
6      switch
7          case M[p, q].dir = diag
8              prepend(buffX, X[p])
9              prepend(buffY, Y[q])
10             p--, q--
11             break
12         case M[p, q].dir = left
13             prepend(buffX, ␣)
14             prepend(buffY, Y[q])
15             q--
16             break
17         case M[p, q].dir = up
18             prepend(buffX[k], X[p])
19             prepend(buffY, ␣)
20             p--
21             break
22     }
23 return buffX, buffY
```

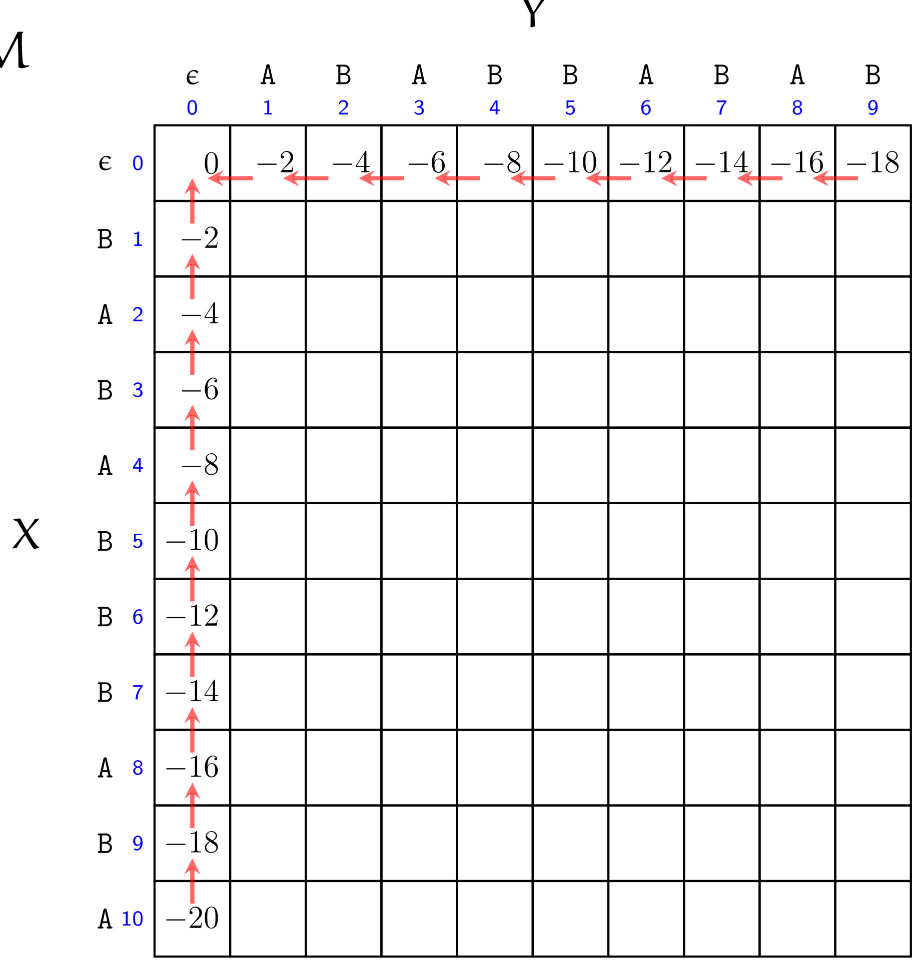
Ето малък пример. Нека азбуката е $\{A, B\}$ и

$$X = \texttt{B A B A B B B A B A}$$

$$Y = \texttt{A B A B B A B A B}$$

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.



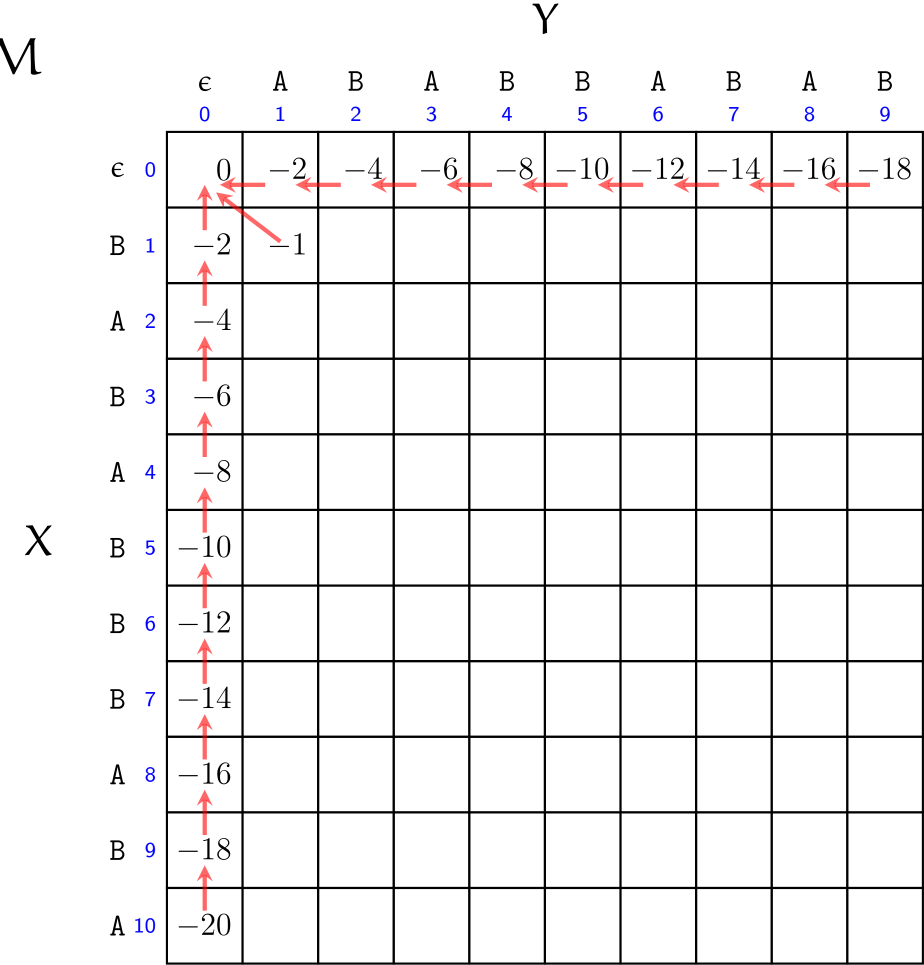
Да видим $\mathsf{M}[1,1]$ какво е. Тъй като $\mathsf{X}[1] = \mathtt{B}$ и $\mathsf{Y}[1] = \mathtt{A}$, то $\delta = -1$ и съгласно (12.39),

$$\mathsf{M}[1,1] = \max\{\mathsf{M}[0,0] - 1, \mathsf{M}[1,0] - 2, \mathsf{M}[0,1] - 2\}$$

Имайки предвид, че $\mathsf{M}[0,0] = 0$, $\mathsf{M}[1,0] = -2$ и $\mathsf{M}[0,1] = -2$, това става

$$\mathsf{M}[1,1] = \max\{0 - 1, -2 - 2, -2 - 2\} = \max\{-1, -4, -4\} = -1$$

Тогава $\mathsf{M}[1,1].\mathrm{val} = -1$, а $\mathsf{M}[1,1].\mathrm{dir} = \mathrm{diag}$.



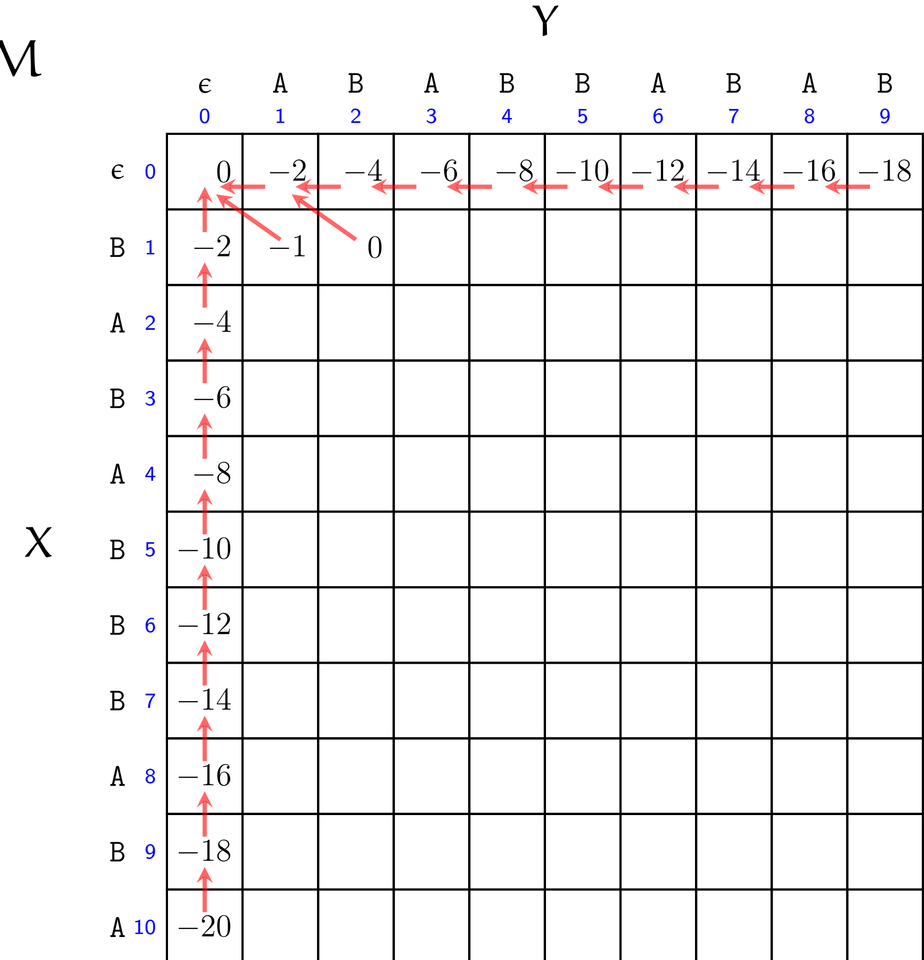
Да видим $\mathsf{M}[1,2]$ какво е. Тъй като $\mathsf{X}[1] = \mathtt{B}$ и $\mathsf{Y}[2] = \mathtt{B}$, то $\delta = 2$ и съгласно (12.39),

$$\mathsf{M}[1,2] = \max\{\mathsf{M}[0,1] + 2, \mathsf{M}[0,2] - 2, \mathsf{M}[1,1] - 2\}$$

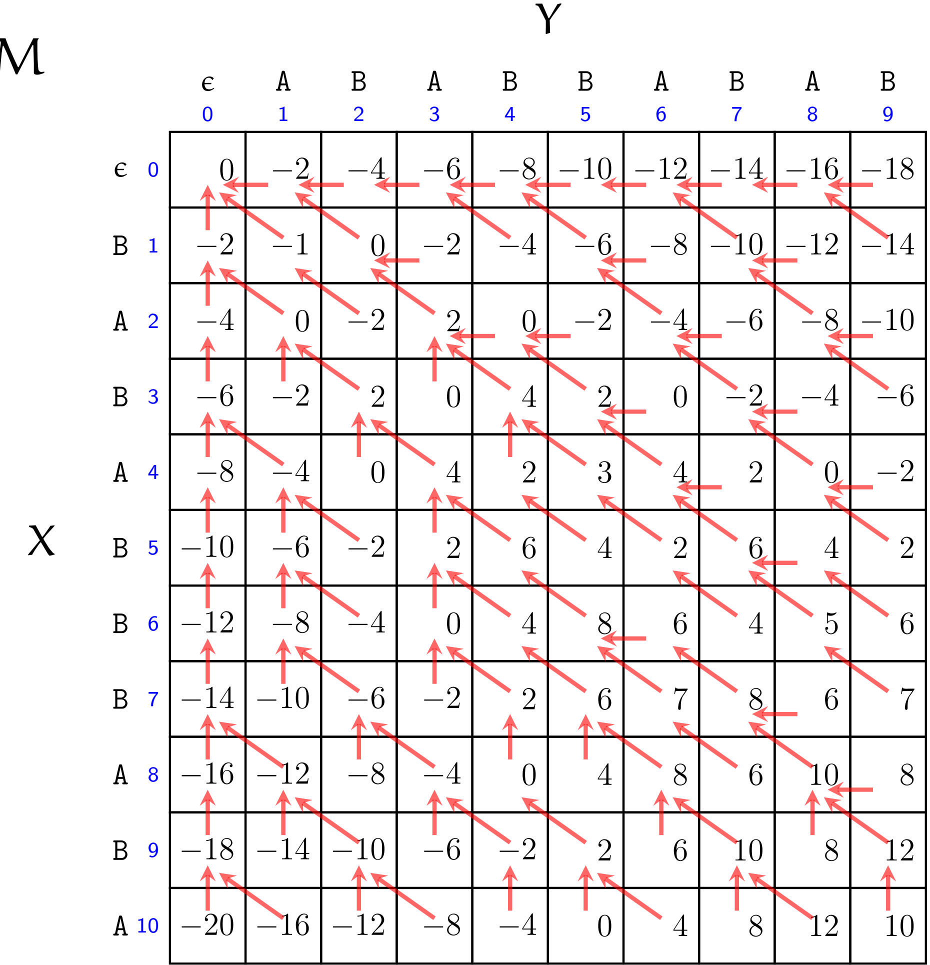
Имайки предвид, че $\mathsf{M}[0,1] = -2$, $\mathsf{M}[0,2] = -4$ и $\mathsf{M}[1,1] = -1$, това става

$$\mathsf{M}[1,2] = \max\{-2+2, -4-2, -1-2\} = \max\{0, -6, -3\} = 0$$

Тогава $\mathsf{M}[1,2].\mathrm{val} = 0$, а $\mathsf{M}[1,1].\mathrm{dir} = \mathrm{diag}$.



Попълваме таблицата докрая и получаваме:

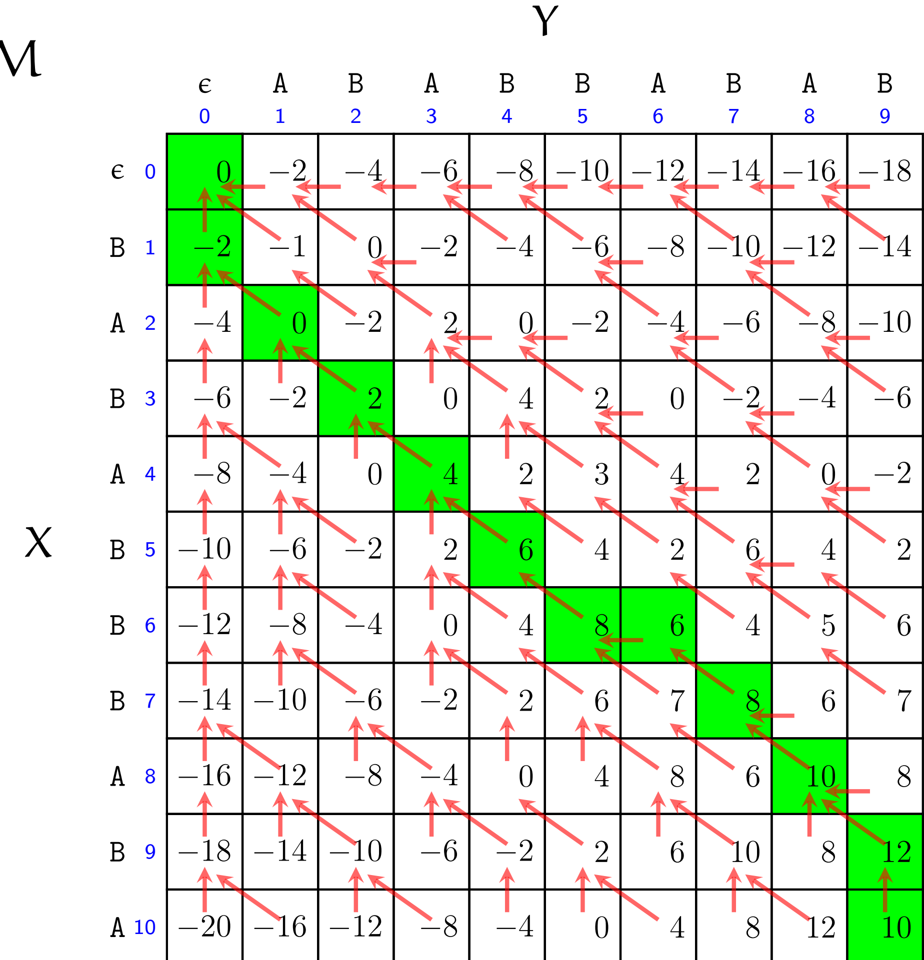


M / X \ Y	ϵ 0	A 1	B 2	A 3	B 4	B 5	A 6	B 7	A 8	B 9
ϵ 0	0	−2	−4	−6	−8	−10	−12	−14	−16	−18
B 1	−2	−1	0	−2	−4	−6	−8	−10	−12	−14
A 2	−4	0	−2	2	0	−2	−4	−6	−8	−10
B 3	−6	−2	2	0	4	2	0	−2	−4	−6
A 4	−8	−4	0	4	2	3	4	2	0	−2
B 5	−10	−6	−2	2	6	4	2	6	4	2
B 6	−12	−8	−4	0	4	8	6	4	5	6
B 7	−14	−10	−6	−2	2	6	7	8	6	7
A 8	−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

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.



M		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

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.

X'	=	B	A	B	A	B	B	␣	B	A	B	A
Y'	=	␣	A	B	A	B	B	A	B	A	B	␣
		−2	+2	+2	+2	+2	+2	−2	+2	+2	+2	−2

Наистина, $-2+2+2+2+2+2-2+2+2+2-2 = 10$.

Допълнение 55: 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. [110].

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 [110, стр. 2]:

> *In the simplest method, MATij is assigned the value, one, if Aj is the same kind of amino acid as Bi; if they are different amino acids, MATij 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 [110] стринговете са протеини, не ДНК, и затова буквите са аминокиселини. Двата протеина са A и B, като Aj и Bi са съответно j-тата и i-тата аминокиселина. MAT е това, което ние наричаме "таблицата на динамичното", а MATij описва резултата от сравнението на префикс на A, съдържащ Aj, и префикс на B, съдържащ Bi.

12.8.4 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 Левенщейн [97]. Левенщейн 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 [109]. 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'
   риспособление // замяна на 'и' с 'а'
   распособление // замяна на 'с' със 'з'
   разп⋎особление // вмъкване на 'р'

разпрос⋎обление // вмъкване на 'т'
разпростобление // замяна на 'о' с 'р'
разпрострбление // замяна на 'б' с 'а'
разпростралеие // 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 *препис* (на английски е *transpript*, вижте [58, стр. 216]), който е стринг над азбуката $\{\mathtt{s}, \mathtt{m}, \mathtt{d}, \mathtt{i}\}$ 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 `dmssmimmisssmmmm` 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 56: Edit Distance срещу Sequence Alignment

Ние вече видяхме в Подсекция 12.8.3 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 [58, 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 `dmssmimmisssmmmm` 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 се 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, подхождаме 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 годинни и това дивергиране е окончателно – те няма да се слеят 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, подхождаме 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 [133, 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 [133, стр. 293, Figure 8.7] и реши сам(а).

Ето и формалното описание на задачата.

Изч. Задача 44: 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 трансформиращи операции, превръщащи X в Y. Трансформиращите 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 12.8.2.

Олекотеният вариант на задачата е да се изчисли само разстоянието между X и Y. Да разгледаме ефикасно решение за него. За всяко $i \in \{0, \ldots, n\}$ и $j \in \{0, \ldots, m\}$, нека $D(i, j)$ е разстоянието между $x_1 \cdots x_i$ и $y_1 \cdots y_j$. Ние търсим $D(n, m)$, но за да го изчислим, ще изчислим всички $D(i, j)$.

Очевидно $D(i, 0) = i$, за $0 \leqslant i \leqslant n$, защото става дума за превръщането на $x_1 \cdots x_i$ в празния стринг по най-икономичен начин, който начин трябва да е i на брой изтривания на букви, а именно на $x_1, \ldots, x_i$. Аналогично, $D(0, j) = j$, за $0 \leqslant i \leqslant m$.

Нека $i \in \{1, \ldots, n\}$ и $j \in \{1, \ldots, m\}$. Да помислим как префиксът $x_1 \cdots x_i$ може да се превърне в префикса $y_1 \cdots y_j$ 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 $x_1 \cdots x_{i-1}$ да бъде превърнат в $y_1 \cdots y_j$ по най-икономичен начин. Цената е оптималната цена на трансформацията на $x_1 \cdots x_{i-1}$ в $y_1 \cdots y_j$ плюс единица.

- Може y_j да бъде добавена към $y_1 \cdots y_{j-1}$, което е една трансформираща операция, ако преди това $x_1 \cdots x_i$ е бил превърнат в $y_1 \cdots y_{j-1}$ по най-икономичен начин. Цената е оптималната цена на трансформацията на $x_1 \cdots x_i$ в $y_1 \cdots y_{j-1}$ плюс единица.

- Оставащата възможност е $x_1 \cdots x_{i-1}$ да бъде превърнат в $y_1 \cdots y_{j-1}$ по най-икономичен начин, след което:

  - ако $x_i = y_j$, няма какво повече да се прави – имаме трансформация на $x_1 \cdots x_i$ в $y_1 \cdots y_j$ на цената на трансформацията на $x_1 \cdots x_{i-1}$ в $y_1 \cdots y_{j-1}$;
  - ако $x_i \neq y_j$, превръщаме x_i в y_j и по този начин имаме трансформация на $x_1 \cdots x_i$ в $y_1 \cdots y_j$ на цената на трансформацията на $x_1 \cdots x_{i-1}$ в $y_1 \cdots y_{j-1}$ плюс единица.

И така:

$$D(i, j) = \begin{cases} i, & \text{ако } j = 0, \\ j, & \text{ако } i = 0 \\ \min\{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta\}, & \text{ако } i \geqslant 1 \text{ и } j \geqslant 1. \end{cases} \tag{12.40}$$

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 $x_1 \cdots x_i$ в $y_1 \cdots y_j$. $D[i, j]$.dir, който е елемент на $\{\texttt{up}, \texttt{left}, \texttt{diag}, \texttt{Nil}\}$, 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 = `Nil`,

- ако $i = 0$ и $j \geqslant 1$, то $D[i, j]$.dir = `left`,

- ако $i \geqslant 1$ и $j = 0$, то $D[i, j]$.dir = `up`,

- ако $i \geqslant 1$ и $j \geqslant 1$, 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 $D[i,j].\text{dir} =$ `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 $D[i,j].\text{dir} =$ `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 $D[i,j].\text{dir} =$ `diag`, без оглед на това дали δ е 0 или 1.

Ето псевдокод, генериращ таблицата.

```
ALG EDIT DISTANCE(Σ: азбука, n, m ∈ ℕ⁺, X[1..n] и Y[1..m]: стрингове над Σ)
 1  създай D[0..n, 0..m]
 2  D[0,0].val ← 0
 3  D[0,0].dir ← Nil
 4  for i ← 1 to n
 5      D[i,0].val ← i
 6      D[i,0].dir ← up
 7  for j ← 1 to m
 8      D[0,j].val ← j
 9      D[0,j].dir ← left
10  for i ← 1 to n
11      for j ← 1 to m
12          if X[i] = Y[j]
13              δ ← 0
14          else
15              δ ← 1
16          themin ← min(D[i-1,j-1].val + δ, D[i-1,j].val + 1, D[i,j-1].val + 1)
17          if themin = D[i-1,j-1].val + δ
18              D[i,j].val ← D[i-1,j-1].val + δ
19              D[i,j].dir ← diag
20          else if themin = D[i-1,j].val + 1
21              D[i,j].val ← D[i-1,j].val + 1
22              D[i,j].dir ← up
23          else
24              D[i,j].val ← D[i,j-1].val + 1
25              D[i,j].dir ← left
26  return D
```

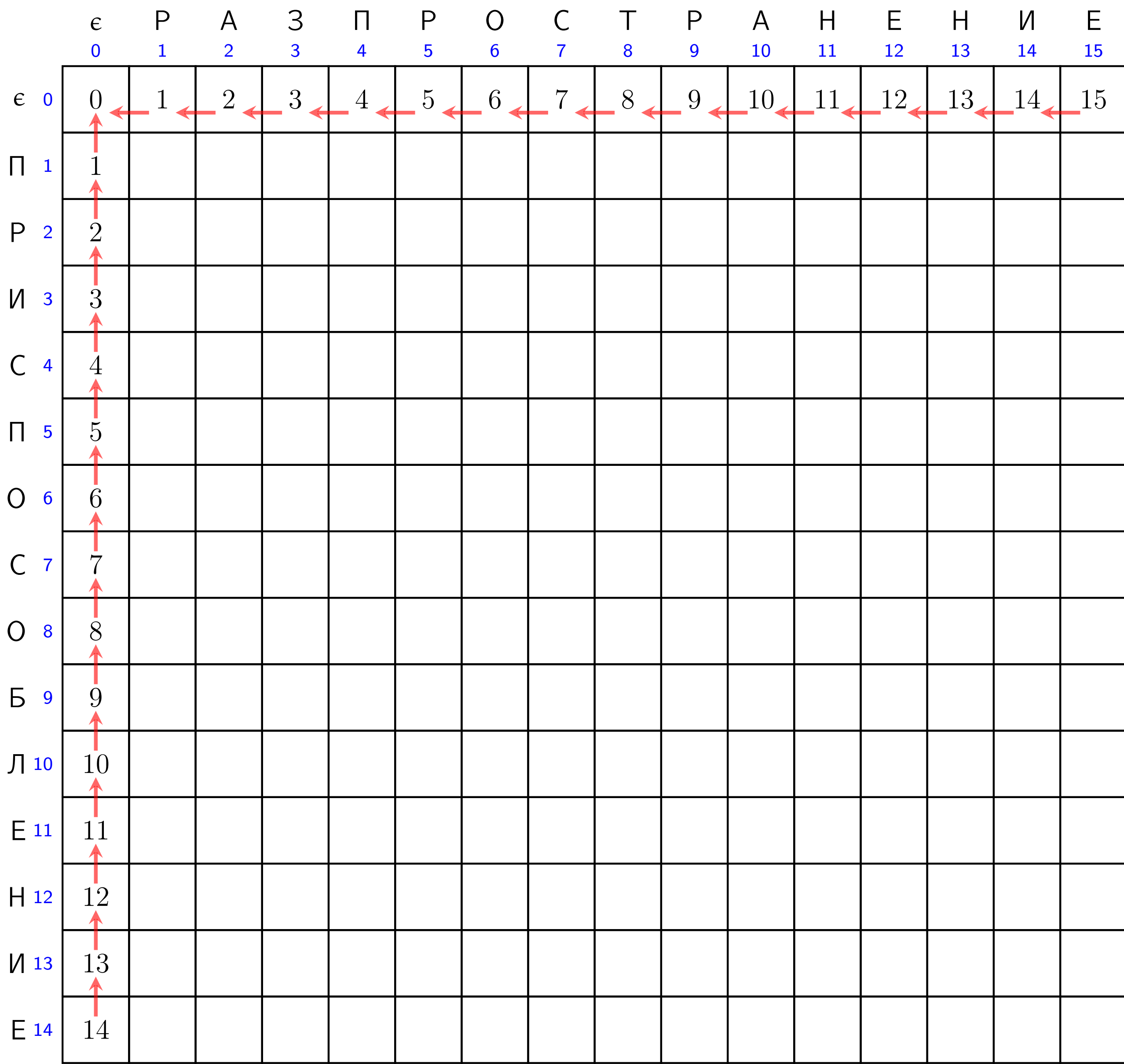
Теглото на оптимално решение е $D[n, m].val$. Ако искаме и накое оптимално решение, записано като препис, ето код, който го генерира.

GENERATE TRANSCRIPT(Σ: азбука, $n, m \in \mathbb{N}^+$, $X[1..n]$ и $Y[1..m]$: стрингове над Σ, двумерен масив $D[0..n, 0..m]$ от (val, dir))

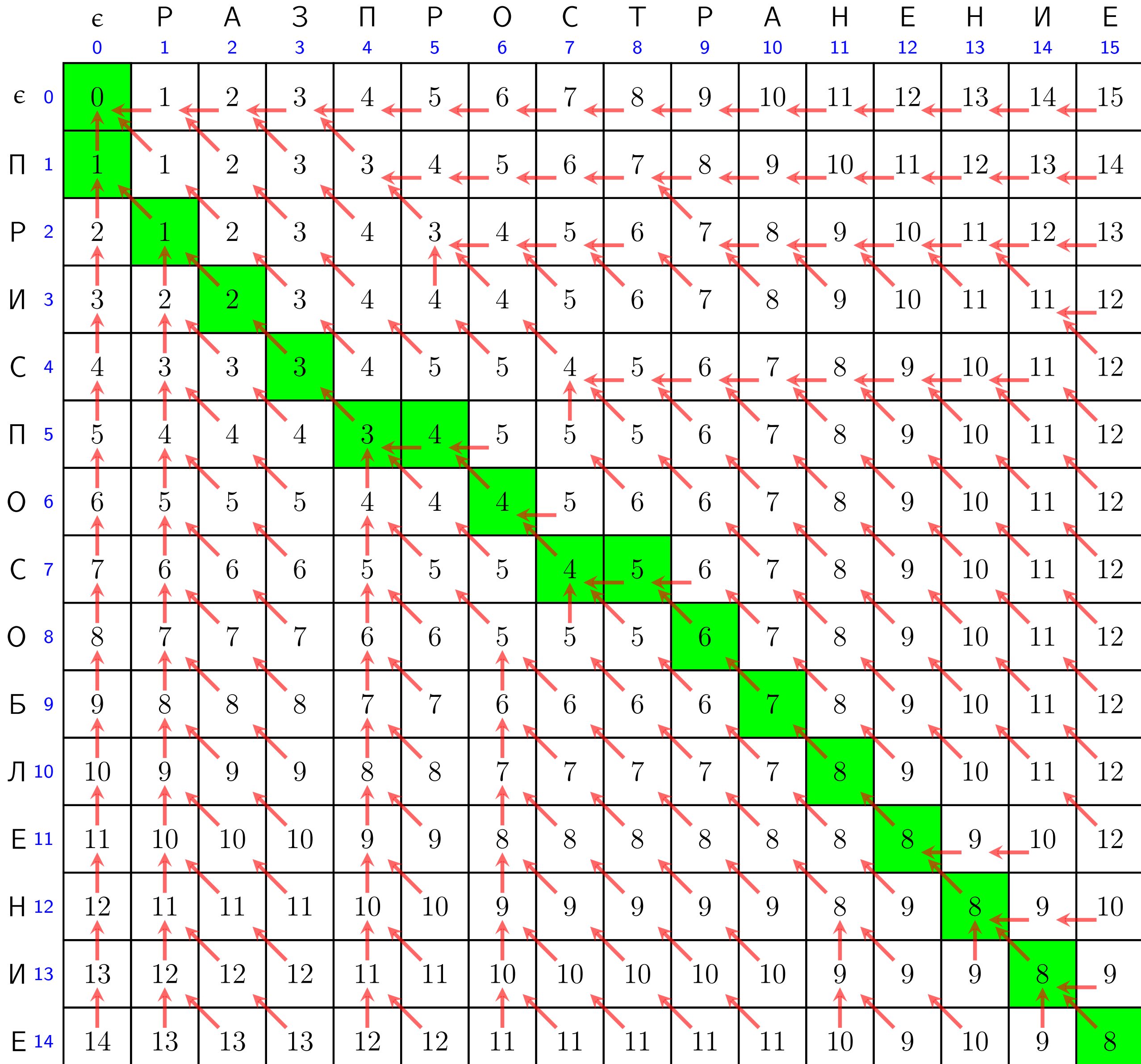
```
 1  t ← max(n, m)
 2  създай празен стринг result
 3  p ← n, q ← m
 4  while TRUE do {
 5      if p = 0 and q = 0
 6          break
 7      switch
 8          case D[p, q].dir = diag
 9              if X[p] = Y[q]
10                  prepend(result, m)
11              else
12                  prepend(result, 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.



Крайната таблица е следната. Решението е акцентирано със зелен фон.



	ϵ 0	Р 1	А 2	З 3	П 4	Р 5	О 6	С 7	Т 8	Р 9	А 10	Н 11	Е 12	Н 13	И 14	Е 15
ϵ 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
П 1	1	1	2	3	3	4	5	6	7	8	9	10	11	12	13	14
Р 2	2	1	2	3	4	3	4	5	6	7	8	9	10	11	12	13
И 3	3	2	2	3	4	4	4	5	6	7	8	9	10	11	11	12
С 4	4	3	3	3	4	5	5	4	5	6	7	8	9	10	11	12
П 5	5	4	4	4	3	4	5	5	5	6	7	8	9	10	11	12
О 6	6	5	5	5	4	4	4	5	6	6	7	8	9	10	11	12
С 7	7	6	6	6	5	5	5	4	5	6	7	8	9	10	11	12
О 8	8	7	7	7	6	6	5	5	5	6	7	8	9	10	11	12
Б 9	9	8	8	8	7	7	6	6	6	6	7	8	9	10	11	12
Л 10	10	9	9	9	8	8	7	7	7	7	7	8	9	10	11	12
Е 11	11	10	10	10	9	9	8	8	8	8	8	8	8	9	10	12
Н 12	12	11	11	11	10	10	9	9	9	9	9	8	9	8	9	10
И 13	13	12	12	12	11	11	10	10	10	10	10	9	9	9	8	9
Е 14	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 `dmssmimmisssmmmm`.

12.9 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 са 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 125), 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 [106].

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 54. 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.

12.9.1 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 кой 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.

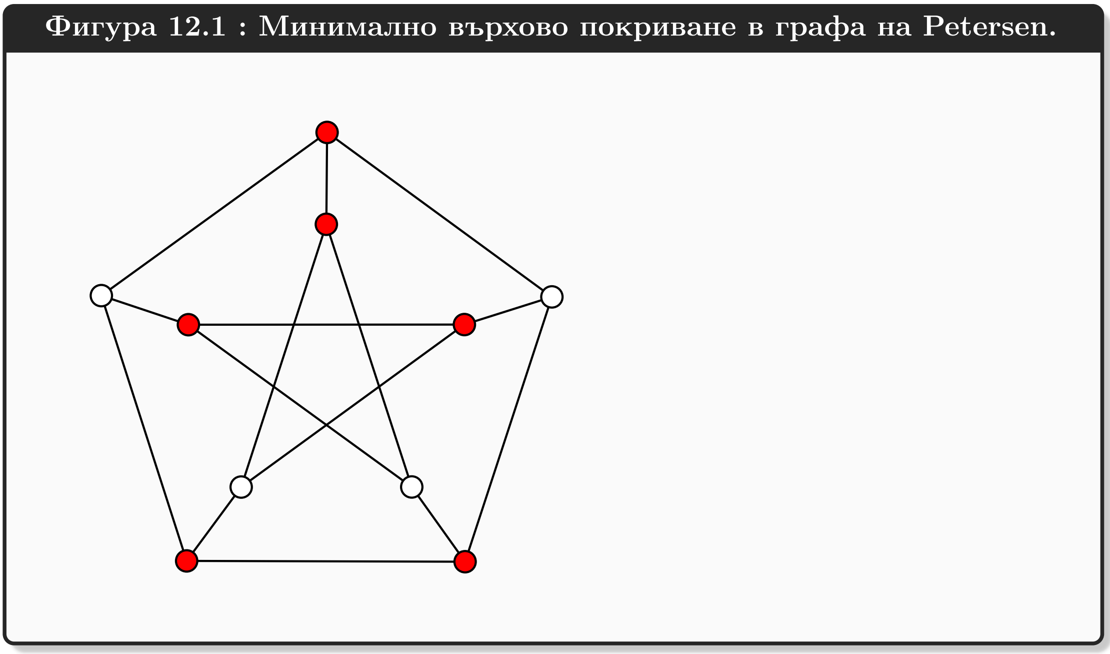
Изч. Задача 45: Vertex Cover

екземпляр: Неориентиран граф $G = (V, E)$.
решение: Подмножество $U \subset V$, такова че $\forall (u, v) \in E : (u \in U$ 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 12.1. 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 12.1 е оптимално. И така, $\tau(P) = 6$.

Фигура 12.1 : 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 46: 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 67: 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 $V \setminus U$. Показахме, че за всяко ребро, поне единият му край е от $V \setminus U$. Тогава $V \setminus U$ е върхово покриване. □

Теорема 67 се илюстрира добре от Фигура 12.1: четирите неоцветени върха 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 57: VERTEX COVER и EDGE COVER са различни задачи

Задачата VERTEX COVER (Задача 45) 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 може да е само 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 [51, стр. 79].

Изч. Задача 47: 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 [51, стр. 190]. Терминът *съчетание* е обяснен в Подподсекция 16.2.7.1.
На стр. 79 в [51] 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-c** (Подсекция 14.5.5 и Определение 125) и, по всеобщото мнение, не е ефикасно решима. Други такива 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 $w_1, \ldots, w_k$, то, ако не сложим v в U, трябва да сложим всички $w_1, \ldots, w_k$; а ако сложим v, можем да забравим за $w_1, \ldots, w_k$, понеже v покрива ребрата $(v, w_1), \ldots, (v, w_k)$.

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**:

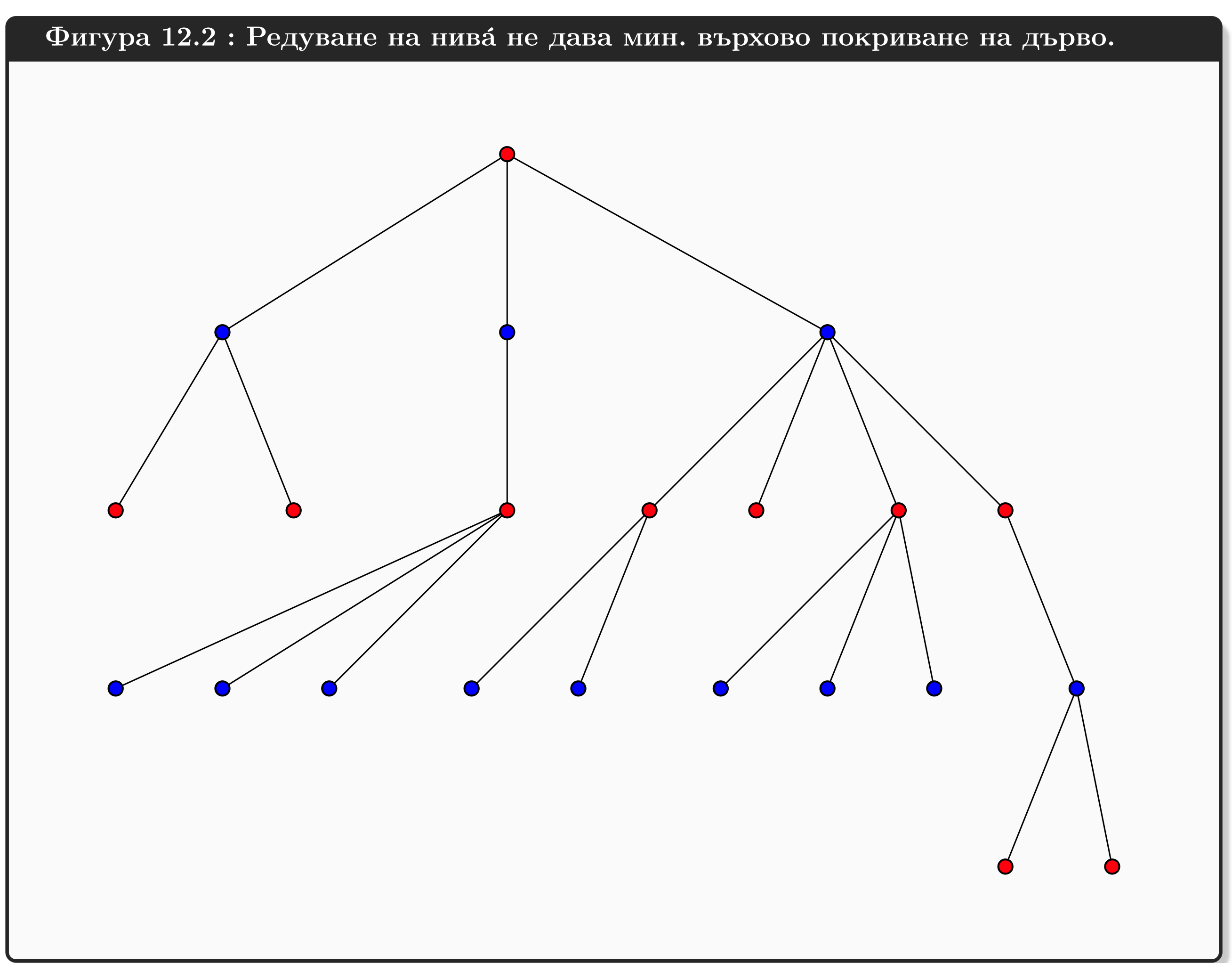
- да пуснем BFS[‡] върху дървото от кой да е връх s,
- да разбием множеството от върховете на $V(T)$ на две подмножества: V_e, тези на четно разстояние от s, и V_o, тези на нечетно разстояние от s,
- и да вземем като решение това от V_e, V_o, което е не по-голямо от другото,

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 12.2 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 нивá спрямо корена дава решение с тегло 10. Но Фигура 12.3 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 12.2 : Редуване на нивá 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 $v \in V(T)$ поддържаме 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 $w_1, \ldots, w_k$. Тогава

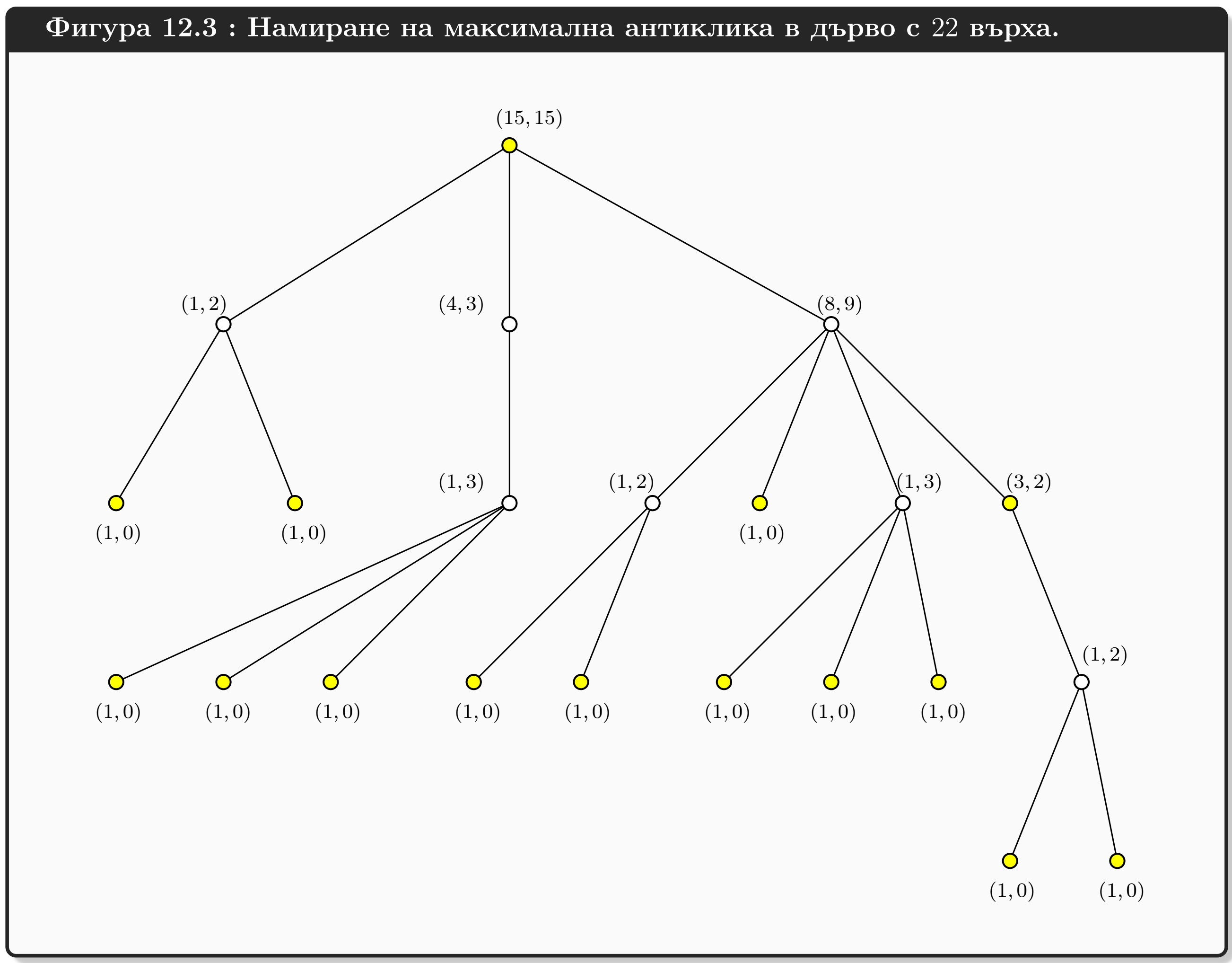
$$v^+ \leftarrow 1 + \sum_{i=1}^{k} w_i^- \tag{12.41}$$

$$v^- \leftarrow \sum_{i=1}^{k} \max\{w_i^-, w_i^+\} \tag{12.42}$$

Фигура 12.3 показва примерно кореново дърво с 22 върха и работата на алгоритъм, изграден върху рекурсивната декомпозиция (12.41) и (12.42). Наредената двойка (15, 15) в корена

означава, че има антиклика с размер 15, съдържаща корена, и антиклика с размер 15, несъдържаща корена.

Фигура 12.3 : 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. 590: алгоритъмът по схемата **Динамично Програмиране** може да се модифицира тривиално така, че да решава Максимална Антиклика в тегловен вариант.

Изч. Задача 48: Independent Set, weigthed

екземпляр: Неориентиран граф $G = (V, E)$, функция $g : V \to \mathbb{R}$.
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 78).

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,

$$v^+ \leftarrow g(v) + \sum_{i=1}^{k} w_i^- \tag{12.43}$$

$$v^- \leftarrow \sum_{i=1}^{k} \max\{w_i^-, w_i^+\} \tag{12.44}$$

По този начин получаваме алгоритъм за задачата в тегловен вариант със същата линейна сложност. Докато алчният алгоритъм на стр. 590 няма очевидно обобщение, което решава тегловния вариант на задачата.

12.9.2 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 76).

Дефинициите на "върхово покриване" и "доминиращо множество" си приличат повърхностно, но всъщност са доста различни. Като екстремен пример да разгледаме пълния 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 49: 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 68: 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. $\lightning$ □

Забележете следната фундаментална разлика между Vertex Cover и Dominating Set. Ако $G = (V, E)$ е граф, $u, v \in V$ не са съседи и $G' = (V, E \cup \{u, v\})$, то $\tau(G) \leqslant \tau(G')$, но $\gamma(G) \geqslant \gamma(G')$. Че $\tau(G) \leqslant \tau(G')$ 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'.

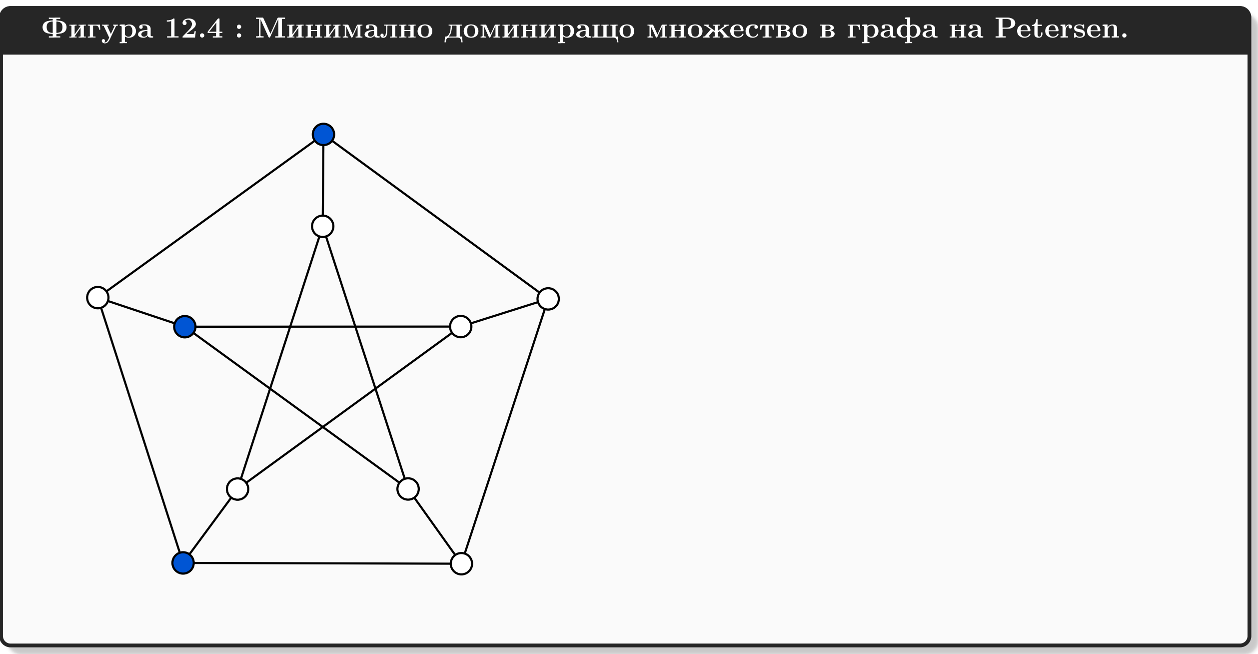
Да се убедим, че $\gamma(G) \geqslant \gamma(G')$. Нека U е произволно доминиращо множество за G. Със сигурност U доминира и G':

- 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 12.4. 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 12.1 е оптимално. И така, $\gamma(P) = 3$. Виждаме, че конверсното твърдение на Теорема 68 не е вярно: трите върха на Фигура 12.4 не покриват графа на Petersen, който “иска” поне шест върха, за да бъде покрит, както отбелязахме на стр. 588.

Фигура 12.4 : 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 множества с празно сечение. На английски се казва *disjoint union*.

Забележете, че ако $A_1, \ldots, A_k$ са непразни множества и $B = A_1 \uplus \cdots \uplus A_k$, то $\{A_1, \ldots, A_k\}$ 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 $B = A_1 \uplus \cdots \uplus A_k$, то $|B| = \sum_{i=1}^{k} A_i$. Ако обаче само знаем, че $B = A_1 \cup \cdots \cup A_k$, то трябва да използваме принципа на включването и изключването, за да намерим $|B|$.

В Определение 71 въведохме “argmin”. Тук ползваме “argmin” с незначителна формална злоупотреба.

$$D(H) = \operatorname{argmin}\{|D^+(H)|, |D^-(H)|\} \tag{12.45}$$

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

$$D(H) = \underset{U \in D^+(H) \uplus D^-(H)}{\operatorname{argmin}} |U| \qquad (12.46)$$

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 (12.45), а не формално изрядни изрази като (12.46).

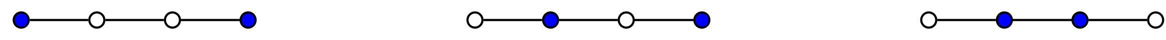
(12.45) се интерпретира така: ако разполагаме с **едно** оптимално доминиращо множество, съдържащо корена, и **едно** оптимално доминиращо множество, несъдържащо корена, то не по-голямото от тях е **едно** оптимално решение за H.

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, ..., 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 $D(T)$.

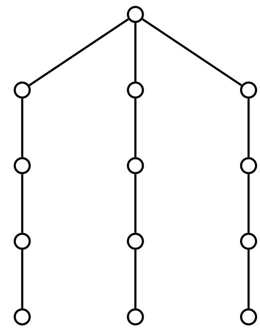
- Да допуснем, че $r \in D(T)$. Търсим $D^+(T)$. Очевидно $\{r\} \uplus \biguplus_{i=1}^{k} D(T[\![w_i]\!])$ е доминиращо множество за дървото T, съдържащо корена. Но дали е оптимално решение за T? Не непременно. Това, че r задължително участва в доминиращо множество ни дава свободата за всяко $T[\![w_i]\!]$ да вземаме 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 вземаме "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.

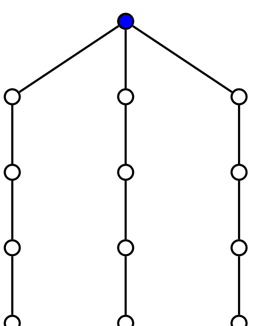
  Второ, забелязваме, че два върха могат да доминират пътя по някой от тези три начина:

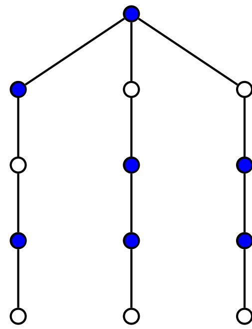
  Трето, конструираме самия екземпляр. Както обикновено, коренът е нарисуван най-горе.

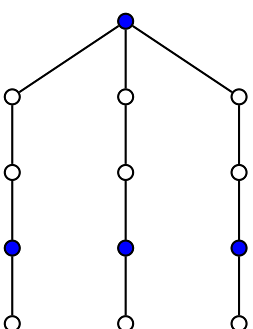
Разглеждаме случая, в който коренът е в доминиращото множество.



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.

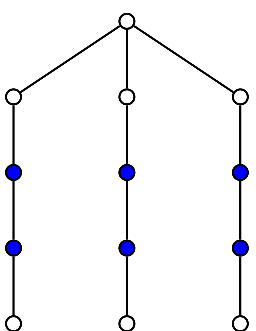
Ще въведем още една нотация. Нека H е произволно кореново дърво. "$\widetilde{D}(H)$" означава някое множество с минимална мощност, което доминира всеки връх на H, може би с изключение на корена.

Забележете, че $|\widetilde{D}(H)| \leqslant |D(H)|$, като е възможно строго неравенство, пример за което видяхме преди малко (дървото е път, вкоренен в единия си край).

И така,

$$\mathsf{D}^+(\mathsf{T}) = \{r\} \uplus \biguplus_{i=1}^{k} \widetilde{\mathsf{D}}(\mathsf{T}[\![w_i]\!]) \tag{12.47}$$

- Да допуснем, че $r \notin \mathsf{D}(\mathsf{T})$. Човек би се изкушил да каже, че решението е $\biguplus_{i=1}^{k} \mathsf{D}(\mathsf{T}[\![w_i]\!])$, 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 $\mathsf{D}^+(\mathsf{T}[\![w_i]\!])$? За това, за което получаваме оптимално решение. Накратко,

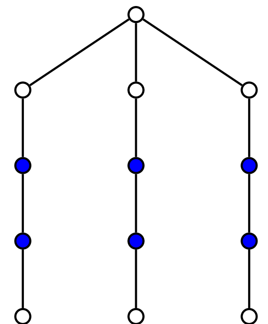
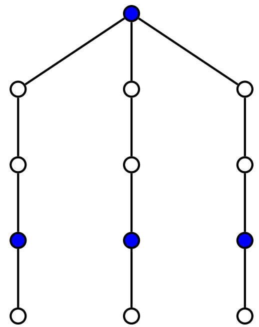
$$\mathsf{D}^-(\mathsf{T}) = \operatorname{argmin}\left\{\left|\mathsf{D}^+(\mathsf{T}[\![w_i]\!]) \uplus \biguplus_{j\in\{1,\ldots,k\}\setminus\{i\}} \mathsf{D}(\mathsf{T}[\![w_j]\!])\right| \,:\, 1 \leqslant i \leqslant k\right\} \tag{12.48}$$

Получихме изрази за $\mathsf{D}^+(\mathsf{T})$ и $\mathsf{D}^-(\mathsf{T})$, съответно (12.47) и (12.48), но с цената на въвеждането на $\widetilde{\mathsf{D}}(\mathsf{T})$. Налага са да изразим и него. Припомняме си, че $\widetilde{\mathsf{D}}(\mathsf{T})$ 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 $\widetilde{\mathsf{D}}(\mathsf{T}) = \biguplus_{i=1}^{k} \mathsf{D}(\mathsf{T}[\![w_i]\!])$. И би сбъркал. Със сигурност множеството $\biguplus_{i=1}^{k} \mathsf{D}(\mathsf{T}[\![w_i]\!])$ 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 $\widetilde{\mathsf{D}}(\mathsf{T})$, което съдържа корена. Това, че коренът се съдържа, ни "развързва ръцете" за децата му $w_1, \ldots, w_k$ да вземаме не D, а $\widetilde{\mathsf{D}}$. С други думи, ако коренът е в $\widetilde{\mathsf{D}}(\mathsf{T})$, можем да си позволим за поддърветата, вкоренени в децата, да вземаме 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,

$$\widetilde{D}(T) = \operatorname{argmin}\left\{\left|\biguplus_{i=1}^{k} D(T[\![w_i]\!])\right|, \left|\{r\} \uplus \biguplus_{i=1}^{k} \widetilde{D}(T[\![w_i]\!])\right|\right\} \tag{12.49}$$

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.

$$
\begin{aligned}
D(T) &= \operatorname{argmin}\{|D^+(T)|, |D^-(T)|\}\\
D^+(T) &= \{r(T)\} \uplus \biguplus_{w \in c(T)} \widetilde{D}(T[\![w]\!])\\
D^-(T) &= \operatorname{argmin}\left\{\left|D^+(T[\![w]\!]) \uplus \biguplus_{z \in c(H) \setminus \{w\}} D(T[\![w]\!])\right| \,:\, w \in c(H)\right\}\\
\widetilde{D}(T) &= \operatorname{argmin}\left\{\left|\biguplus_{i=1}^{k} D(T[\![w_i]\!])\right|, \left|\{r\} \uplus \biguplus_{i=1}^{k} \widetilde{D}(T[\![w_i]\!])\right|\right\}
\end{aligned}
$$

Тази система може да се опрости значително. D се изразява чрез D^+ и D^-, но

- D^+ се изразява чрез $\widetilde{D}$,
- D^- се изразява чрез D^+ и D,
- $\widetilde{D}$ се изразява чрез D и $\widetilde{D}$.

Очевидно можем да изразим всичко само чрез D и $\widetilde{\mathsf{D}}$ по този начин:

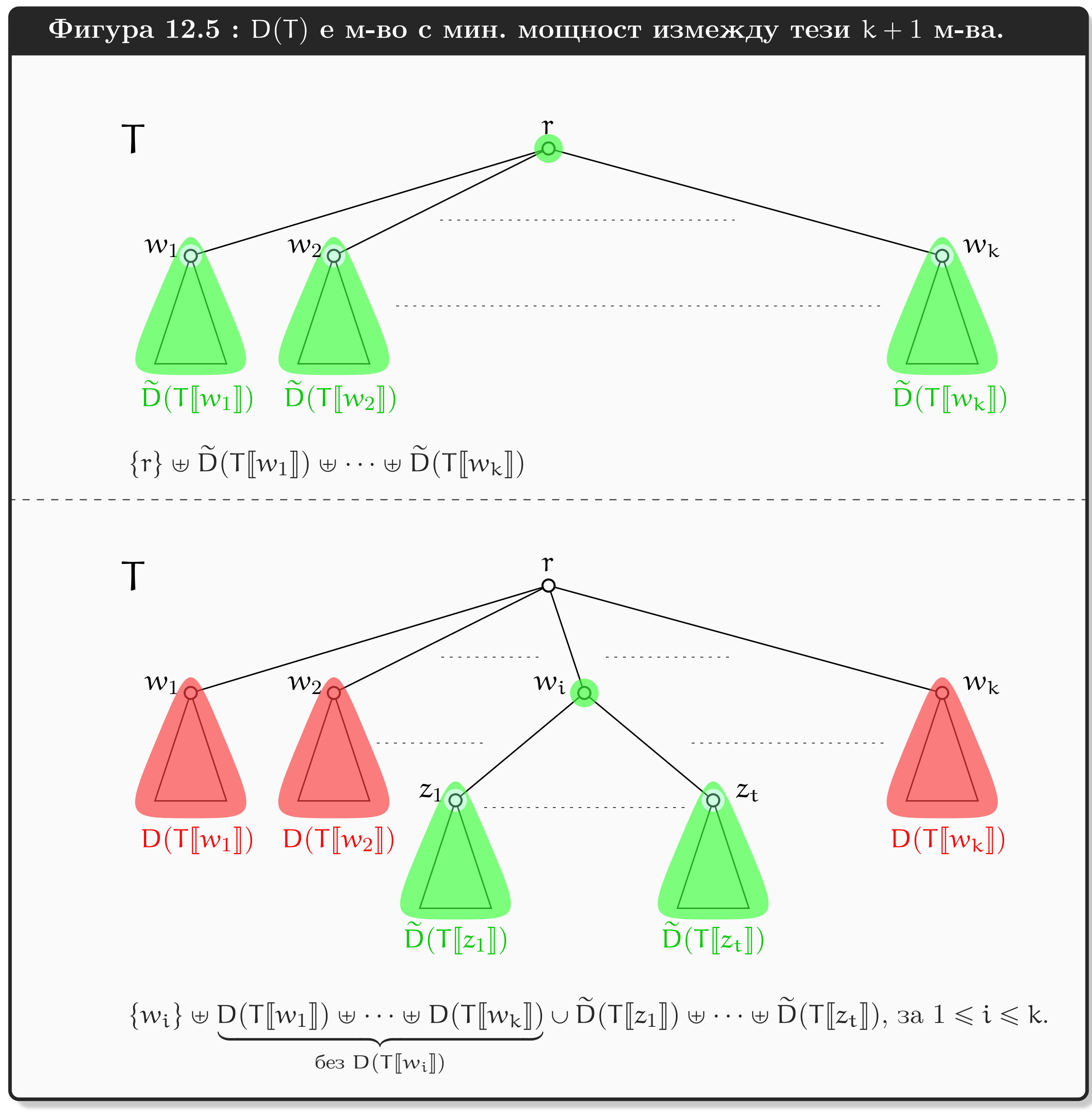
$$\mathsf{D}(\mathsf{T}) = \operatorname{argmin}\left\{ \left| \{r\} \uplus \biguplus_{w \in c(\mathsf{T})} \widetilde{\mathsf{D}}(\mathsf{T}[\![w]\!]) \right|, \operatorname{argmin}\left\{ \left| \{w\} \uplus \biguplus_{z \in c(\mathsf{T}[\![w]\!])} \widetilde{\mathsf{D}}(\mathsf{T}[\![z]\!]) \uplus \biguplus_{a \in c(\mathsf{H}) \setminus \{w\}} \mathsf{D}(\mathsf{T}[\![a]\!]) \right| \,:\, w \in c(\mathsf{H}) \right\} \right\} \tag{12.50}$$

$$\widetilde{\mathsf{D}}(\mathsf{T}) = \operatorname{argmin}\left\{ \left| \biguplus_{w \in c(\mathsf{T})} \mathsf{D}(\mathsf{T}[\![w]\!]) \right|, \left| \{r\} \uplus \biguplus_{w \in c(\mathsf{T})} \widetilde{\mathsf{D}}(\mathsf{T}[\![w]\!]) \right| \right\} \tag{12.51}$$

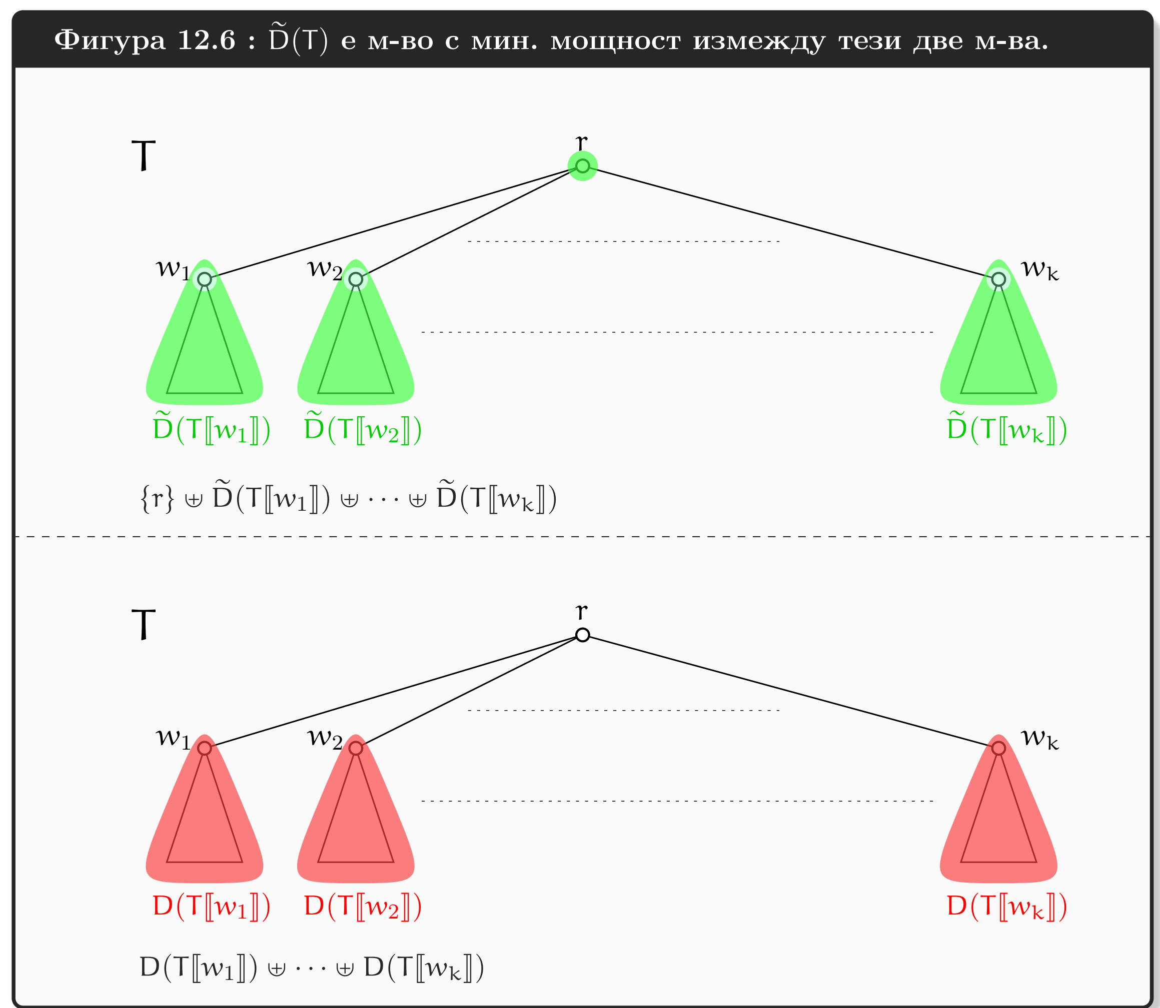
С червено и синьо е показано как D^+ и D^- биват замествани, за да изчезнат и да останат само D и $\widetilde{\mathsf{D}}$.

Фигура 12.5 илюстрира двата операнда в минимума в 12.50.

Фигура 12.5 : $\mathsf{D}(\mathsf{T})$ **е м-во с мин. мощност измежду тези** $k+1$ **м-ва.**



Фигура 12.6 илюстрира двата операнда в минимума в 12.51.



Фигура 12.6 : $\widetilde{D}(T)$ **е м-во с мин. мощност измежду тези две м-ва.**

В светлината на опростените изрази, искаме началните условия само за D и $\widetilde{D}$. И така, ако $T = (\{r\}, \varnothing)$, то

$$D(T) = r, \widetilde{D}(T) = \varnothing \qquad (12.52)$$

Тежкия вариант на задачата се решава от рекурсивна декомпозиция, състояща се от (12.50), (12.51) и (12.52). 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), с неясен произход (поне за автора на записките), което го прави фолклор, и е горе-долу следното.

DS ON TREES($T = (V, E)$: дърво)
1 $A[1, \ldots, n]$, $B[1, \ldots, n]$: 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 (12.52), (12.50) и (12.51):

- 6 отговаря на (12.52),
- 8 отговаря на (12.50),
- 9 отговаря на (12.51).

Това, че 6 отговаря на (12.52) и 9 отговаря на (12.51), е очевидно. Че 8 отговаря на (12.50) 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 (12.50) и (12.51) ползват само обединения на множества без общи елементи, така че появите на $\uplus$ директно се превеждат в суми.

Задачата във вариант с тегла. Задачата е следната.

Изч. Задача 50: Dominating Set, weighted

екземпляр: Неориентиран граф $G = (V, E)$, функция $g : V \to \mathbb{R}$.
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 $A[1, \ldots, n]$, $B[1, \ldots, n]$: 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** $A[r]$

12.10 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 62 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 12.10.1), 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 12.10.2. 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 играга 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).

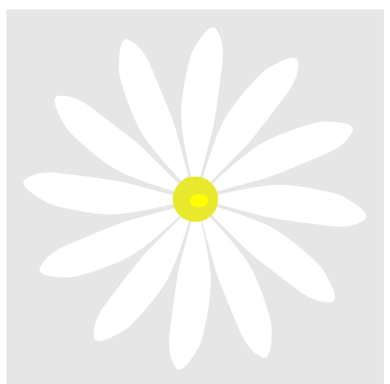
Ако даден играч губи задължително при оптимална игра на противника—какъвто е случаят с Албена в играта с пълна маргаритка—няма смисъл да говорим за оптимална статегия за него или нея.

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 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".

12.10.1 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.

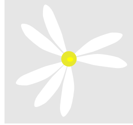
Лесният вариант: "пълна" маргаритка. Доколкото е известно на автора на тези записки, задачата е популярна във варианта, в който се започва от маргаритка, от която не е късано нищо ("пълна" маргаритка).

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 венчелистчета 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 C 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 нарисувана 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.

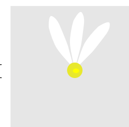
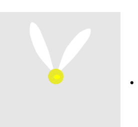
Конфигурациите $\langle 1 \rangle$ и $\langle 2 \rangle$ са базовите конфигурации. Както отбелязахме, те са печеливши. Всяка конфигурация C с общо n венчелистчета се получава

- 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.

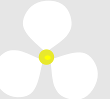
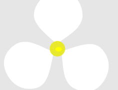
На практика вече имаме рекурсивна декомпозиция по броя на венчелистчетата. Нека n е броят на венчелистчетата в задачата. Можем да пресметнем рекурсията по схемата **Динамично Програмиране**, като систематично, за $k = 1, 2, \ldots, n$ намерим всички конфигурации с k венчелистчета и за всяка от тях, която 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 .

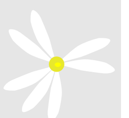
И двете конфигурации се описват от редицата $\langle 3 \rangle$. При редици с една единствена компонента се налага да различаваме някак ситуация с нула "празнини" (маргаритка, от която не е късано) от ситуация с една "празнина" (има късани венчелистчета).

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.

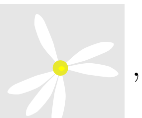
Изключвайки тази особеност, ясно е, че всяка конфигурация на n венчелистчета е едно целочислено разбиване на n (Определение 35), така че можем да мислим в термините на целочислени разбивания. Вместо да говорим за конфигурации от венчелистчета, ще говорим за целочислени разбивания. Можем да изградим триъгълна таблица с информация за разбиванията до каквато стойнст на n пожелаем. Таблица 12.5 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.

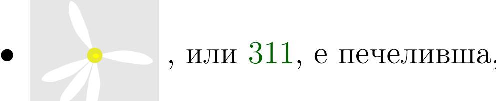
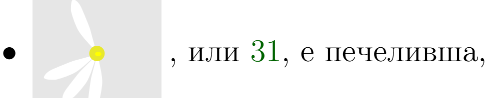
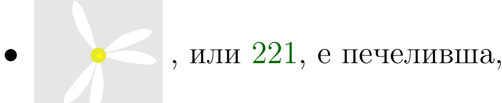
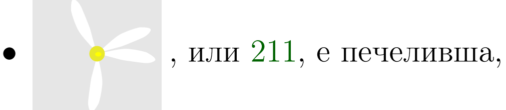
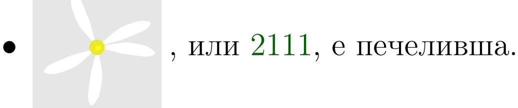
n	целочислените разбивания на n															
1	$\tilde{1}$	1														
2	$\tilde{2}$	2	11													
3	$\tilde{3}$	3	21	111												
4	$\tilde{4}$	4	31	22	211	1111										
5	$\tilde{5}$	5	41	32	311	221	2111	11111								
6	$\tilde{6}$	6	51	42	33	411	321	222	2211	3111	21111	111111				
7	$\tilde{7}$	7	61	52	43	511	421	331	322	4111	3211	2221	31111	22111	211111	1111111

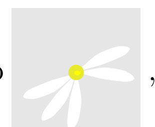
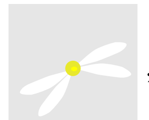
Таблица 12.5: Печеливши и губещи целочислени разбивания.

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 12.5 и Таблица 12.4. В някакъв смисъл, Таблица 12.4 е кондензиран вариант на Таблица 12.5: ако игнорираме колоната с тилдите в Таблица 12.5, останалата част на Таблица 12.5 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 12.5 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.

12.10.2 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). Иска 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 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 12.5 надолу, дължината на n-ия ред ще е от порядъка на експонента от $\sqrt{n}$, като, примерно, 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 $1 \leqslant i < j \leqslant n$. 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 $\text{opt}[1, \dots, n][1, \dots, n]$. Решението е $\text{opt}[1][n]$.

Нека масив $M[1, \dots, n]$ съдържа редицата от монети; тоест, $M[i]$ е стойността на i-тата монета отляво, за $1 \leqslant i \leqslant n$.

Дъното на рекурсията се реализира чрез

$$\text{opt}[i, i+1] = \max\{M[i], M[i+1]\}$$

за $1 \leqslant i < n$. 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,

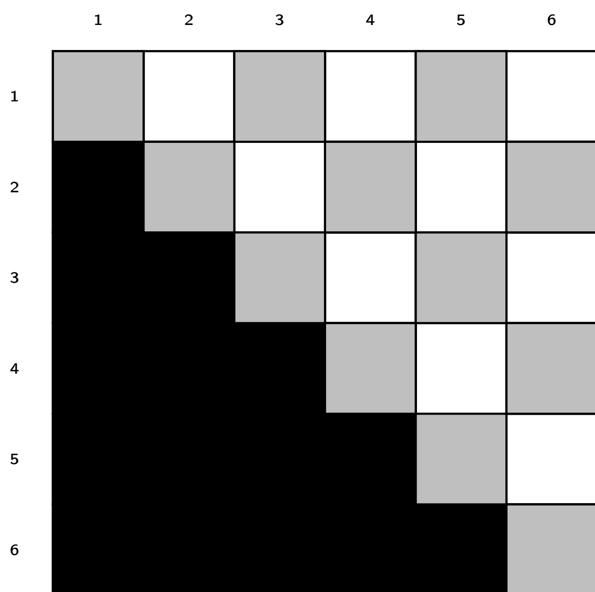
$$M[j] + \min\{\text{opt}[i, j-2], \text{opt}[i+1, j-1]\}$$

Албена трябва да избере максимума от тези. И така:

$$\text{opt}[i, j] = \max\big\{M[i] + \min\{\text{opt}[i+1, j-1], \text{opt}[i+2, j]\}, M[j] + \min\{\text{opt}[i, j-2], \text{opt}[i+1, j-1]\}\big\}$$

Да илюстрираме работата на алгоритъма, изграден върху тази рекурсивна декомпозиция.

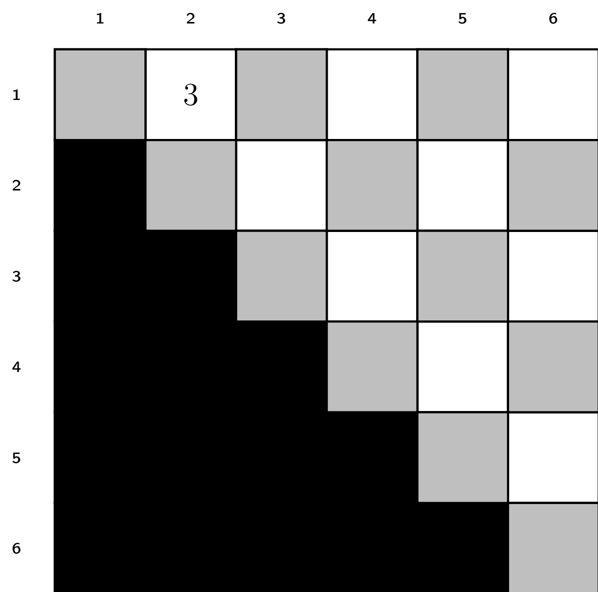
Нека входът отново е $\langle 1, 3, 6, 3, 1, 3 \rangle$. Работната матрица е квадратна, като клетките под главния диагонал не се ползват по очевидни причини (те са в черно на илюстрацията), а в останалата част се ползват само клетките $[i, j]$, за които $j - i$ е нечетно; тези, за които $j - i$ е четно (те са в сиво на илюстрацията), отговарят на ходове на Борис, които са нерелевантни за алгоритъма за Албена.



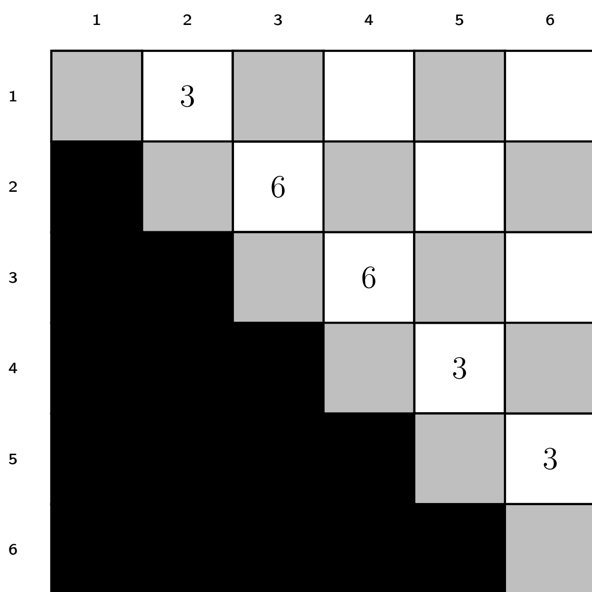
Да разгледаме началните условия

$$\text{opt}[i, i+1] = \max\{M[i], M[i+1]\}$$

за $1 \leqslant i \leqslant 5$. Тогава $\text{opt}[1, 2] = \max\{1, 3\} = 3$, така че имаме:



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:



С това приключихме с началните условия.

За клетка $[1, 4]$, стойността се получава като

$$\begin{aligned}
\mathrm{opt}[1,4] &= \max\Big\{1 + \min\{\mathrm{opt}[2,3], \mathrm{opt}[3,4]\}, 3 + \min\{\mathrm{opt}[1,2], \mathrm{opt}[2,3]\}\Big\} \\
&= \max\Big\{1 + \min\{6,6\}, 3 + \min\{3,6\}\Big\} \\
&= \max\Big\{1 + 6, 3 + 6\Big\} \\
&= 7
\end{aligned}$$

Наистина, върху вход $\langle 1, 3, 6, 3\rangle$, Албена може да вземе най-много 7. Аналогично,

$$\begin{aligned}
\mathrm{opt}[2,5] &= \max\Big\{3 + \min\{\mathrm{opt}[3,4], \mathrm{opt}[4,5]\}, 1 + \min\{\mathrm{opt}[2,3], \mathrm{opt}[3,4]\}\Big\} \\
&= \max\Big\{3 + \min\{6,3\}, 1 + \min\{6,6\}\Big\} \\
&= \max\Big\{3 + 3, 1 + 6\Big\} \\
&= 7
\end{aligned}$$

и

$$\begin{aligned}
\mathrm{opt}[3,6] &= \max\Big\{6 + \min\{\mathrm{opt}[4,5], \mathrm{opt}[5,6]\}, 3 + \min\{\mathrm{opt}[3,4], \mathrm{opt}[4,5]\}\Big\} \\
&= \max\Big\{6 + \min\{3,3\}, 3 + \min\{6,3\}\Big\} \\
&= \max\Big\{6 + 3, 3 + 3\Big\} \\
&= 9
\end{aligned}$$

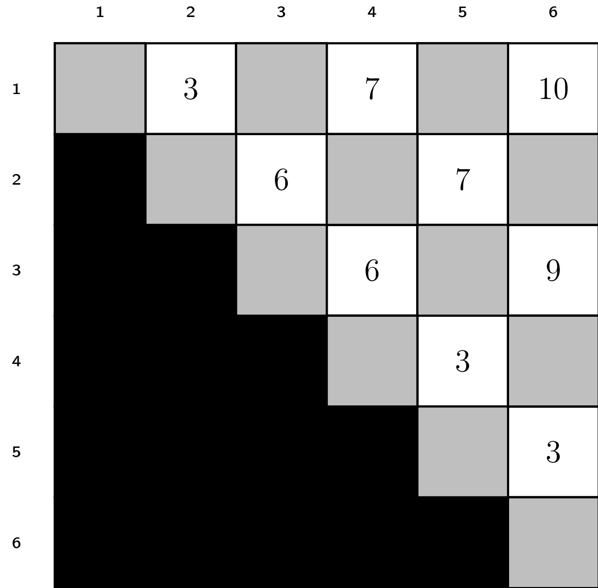
Имаме

	1	2	3	4	5	6
1		3		7		
2			6		7	
3				6		9
4					3	
5						3
6						

Окончателно,

$$\begin{aligned}
\text{opt}[1,6] &= \max\Big\{1 + \min\{\text{opt}[2,5], \text{opt}[3,6]\}, 3 + \min\{\text{opt}[1,4], \text{opt}[2,5]\}\Big\} \\
&= \max\Big\{1 + \min\{7,9\}, 3 + \min\{7,7\}\Big\} \\
&= \max\Big\{1+7, 3+7\Big\} \\
&= 10
\end{aligned}$$

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.

12.11 Мемоизация

Името на тази секция не е сгрешено. Не става дума за мемо**р**изация, 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 (12.10). Като среда ще ползваме Maple (TM).

Нека изчисляваме $\binom{30}{15}$, който е 155 117 520. Съгласно (12.10), 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 биномен 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 изчислението с мемоизация в Maple.

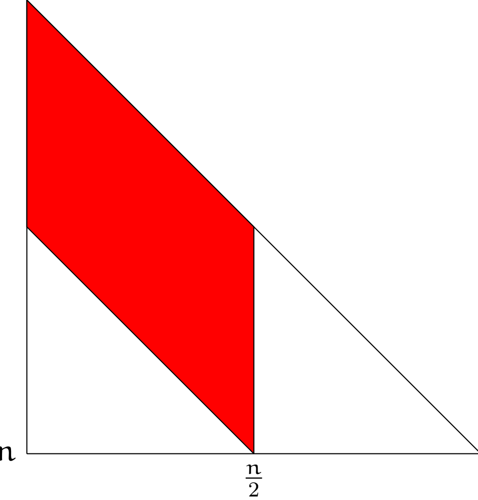
Стандартният начин за измерване на чистото време за изчисление в Maple (TM) не е с гледане на часовника, а с използване на функцията `profile` с аргумент-името на функцията или процедурата, която искаме да профилираме. После с `showprofile` се извежда подробна информация за изразходваното изчислително време и памет. Но в примера с биномните 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.

12.12 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. [133, 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 51: 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 дъжината 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 зацикли.

Коректна рекурсия е следната. Нека $\mathsf{LP}(i, j, P)$ 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.

$$\mathsf{LP}(i, j, P \oplus x) = \max_{(x,j) \in E, j \notin P} \{\mathsf{L}(i, x, P) + w(x, j)\}$$

"$P \oplus x$" означава редицата P с "долепен" x в края. $\mathsf{L}(i, x, P)$ 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 неефикасна. От общи съображения може да кажем, че има не повече от $(n - 3)!$ възможности за P[†], 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 $\widetilde{\mathsf{L}}(i, j, S)$ със смисъл на дължина на най-къс път от i до j, в който S е множеството от междинните върхове, може да кажем:

$$\widetilde{\mathsf{L}}(i, j, S \cup \{x\}) = \max_{(x,j) \in E, j \notin S} \{\widetilde{\mathsf{L}}(i, x, S) + w(x, j)\}$$

Пространството от състоянията става от порядъка на 2^n, което е известно подобрение спрямо факториела. Въпреки това, алгоритъмът, съответен на рекурсията, продължава да е неефикасен.

Наблюдение 62

Схемата **Динамично Програмиране** не е всемогъща. Тя има своите естествени ограничения. За задачачи 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 Π_1 до задача Π_2, по този начин показвайки, че, ако Π_1 е трудна, то и Π_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 92: Долна и горна граница на сложността на задача

Нека Π е произволна изчислителна задача. *Долна граница върху сложността на* Π е всяка функция $\phi(n)$, такава че всеки алгоритъм, който решава Π, работи във време $\Omega(\phi(n))$. *Горна граница върху сложността на* Π е всяка функция $\psi(n)$, такава че поне един алгоритъм, който решава Π, 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 94), 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 58: 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 елементи 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 [31, стр. 40].

Очевидно е, че в процеса на работата на алгоритъма, всяка извършена транспозиция може да намали броя на инверсиите (Определение 55) 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 елементи), то само "ликвидирането" на инверсиите налага квадратичен брой транспозиции. Масивът не може да е сортиран, ако в него има инверсии (Наблюдение 39), 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; примерно, 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 51: The Balance Puzzle

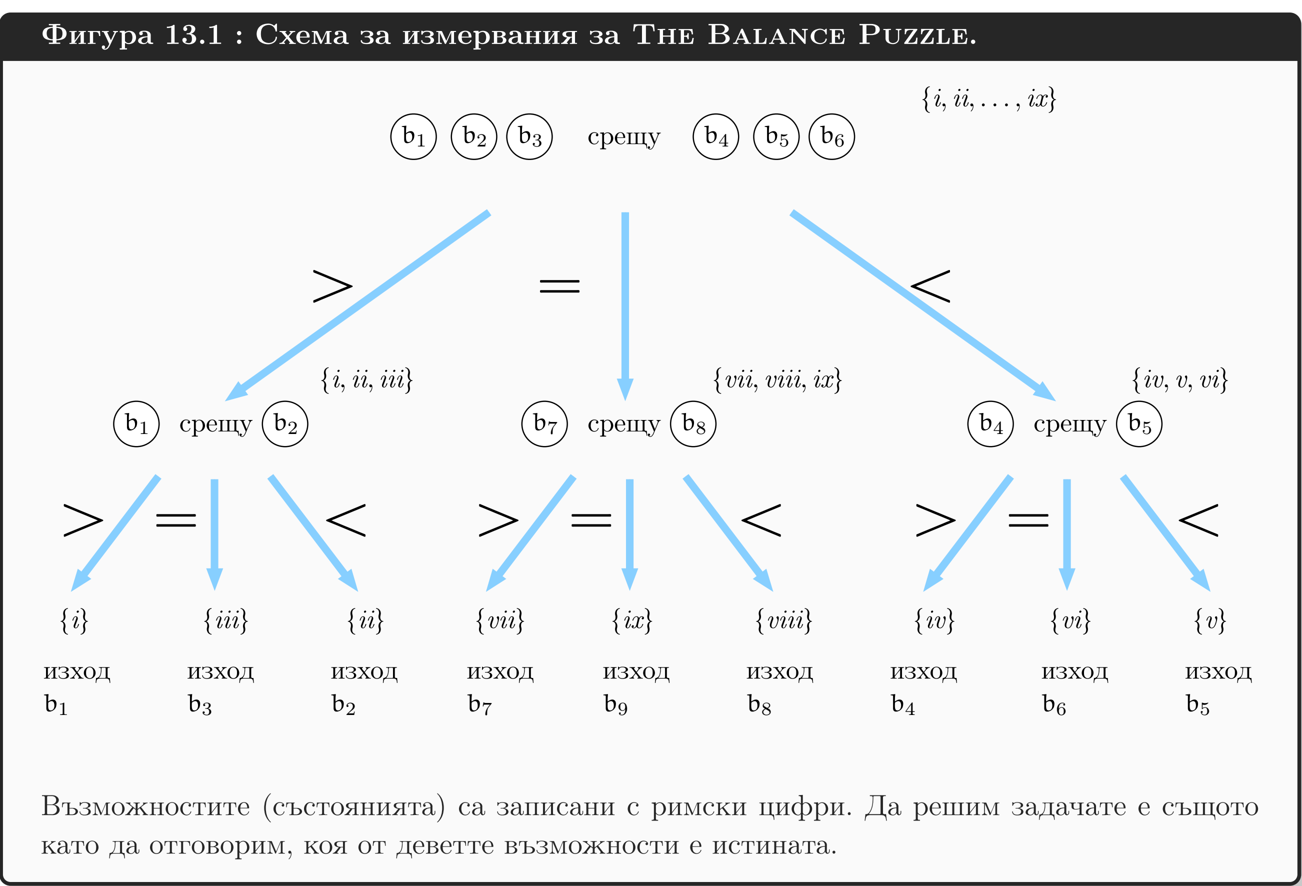
Дадени са 9 номерирани предмета, да речем топки. Осем от тях имат едно и също тегло, а една топка е по-тежка (от коя да е от осемте). Нашата цел е да идентифицираме тежката топка, като използваме везни. Везните нямат стандартни теглилки, така че можем правим единствено измервания от вида: някои топки 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 $\mathfrak{b}_1, \ldots, \mathfrak{b}_9$. С едно използване на везната мерим $\mathfrak{b}_1$, $\mathfrak{b}_2$ и $\mathfrak{b}_3$ срещу $\mathfrak{b}_4$, $\mathfrak{b}_5$ и $\mathfrak{b}_6$. Има точно три възможни изхода.

- Ако $\mathfrak{b}_1$, $\mathfrak{b}_2$ и $\mathfrak{b}_3$ са по-тежки от $\mathfrak{b}_4$, $\mathfrak{b}_5$ и $\mathfrak{b}_6$, използваме везната втори път с $\mathfrak{b}_1$ срещу $\mathfrak{b}_2$. Отново има точно три възможни изхода.
  - Ако $\mathfrak{b}_1$ е по-тежка от $\mathfrak{b}_2$, връщаме "$\mathfrak{b}_1$ *е тежката топка*".
  - Ако $\mathfrak{b}_2$ е по-тежка от $\mathfrak{b}_1$, връщаме "$\mathfrak{b}_2$ *е тежката топка*".
  - Ако $\mathfrak{b}_1$ и $\mathfrak{b}_2$ са еднакво тежки, връщаме "$\mathfrak{b}_3$ *е тежката топка*".
- Ако $\mathfrak{b}_4$, $\mathfrak{b}_5$ и $\mathfrak{b}_6$ са по-тежки от $\mathfrak{b}_1$, $\mathfrak{b}_2$ и $\mathfrak{b}_3$, използваме везната втори път с $\mathfrak{b}_4$ срещу $\mathfrak{b}_5$. Отново има точно три възможни изхода.
  - Ако $\mathfrak{b}_4$ е по-тежка от $\mathfrak{b}_5$, връщаме "$\mathfrak{b}_4$ *е тежката топка*".
  - Ако $\mathfrak{b}_5$ е по-тежка от $\mathfrak{b}_4$, връщаме "$\mathfrak{b}_5$ *е тежката топка*".
  - Ако $\mathfrak{b}_4$ и $\mathfrak{b}_5$ са еднакво тежки, връщаме "$\mathfrak{b}_6$ *е тежката топка*".
- Ако $\mathfrak{b}_1$, $\mathfrak{b}_2$ и $\mathfrak{b}_3$, от една страна, и $\mathfrak{b}_4$, $\mathfrak{b}_5$ и $\mathfrak{b}_6$, от друга страна, са еднакво тежки, използваме везната втори път с $\mathfrak{b}_7$ срещу $\mathfrak{b}_8$. Отново има точно три възможни изхода.
  - Ако $\mathfrak{b}_7$ е по-тежка от $\mathfrak{b}_8$, връщаме "$\mathfrak{b}_7$ *е тежката топка*".q
  - Ако $\mathfrak{b}_8$ е по-тежка от $\mathfrak{b}_7$, връщаме "$\mathfrak{b}_8$ *е тежката топка*".
  - Ако $\mathfrak{b}_7$ и $\mathfrak{b}_8$ са еднакво тежки, връщаме "$\mathfrak{b}_9$ *е тежката топка*".

Фигура 13.1 илюстрира тази схема.

Фигура 13.1 : Схема за измервания за The Balance Puzzle.



Възможностите (състоянията) са записани с римски цифри. Да решим задачате е същото като да отговорим, коя от деветте възможности е истината.

Много полезно е да мислим в термините на възможности, или възможни състояния. Поначало има точно девет възможности: или $\mathfrak{b}_1$ е тежката топка, или $\mathfrak{b}_2$ е тежката топка, и така нататък, или $\mathfrak{b}_9$ е тежката топка. Нека запишем тези девет състояния със, съответно, i, ii, ..., ix. В началото множеството от състоянията е $\{i, ii, \ldots, ix\}$. 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 различи 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 63

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 **<u>нищо общо</u>** 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 **<u>е</u>** достижима,

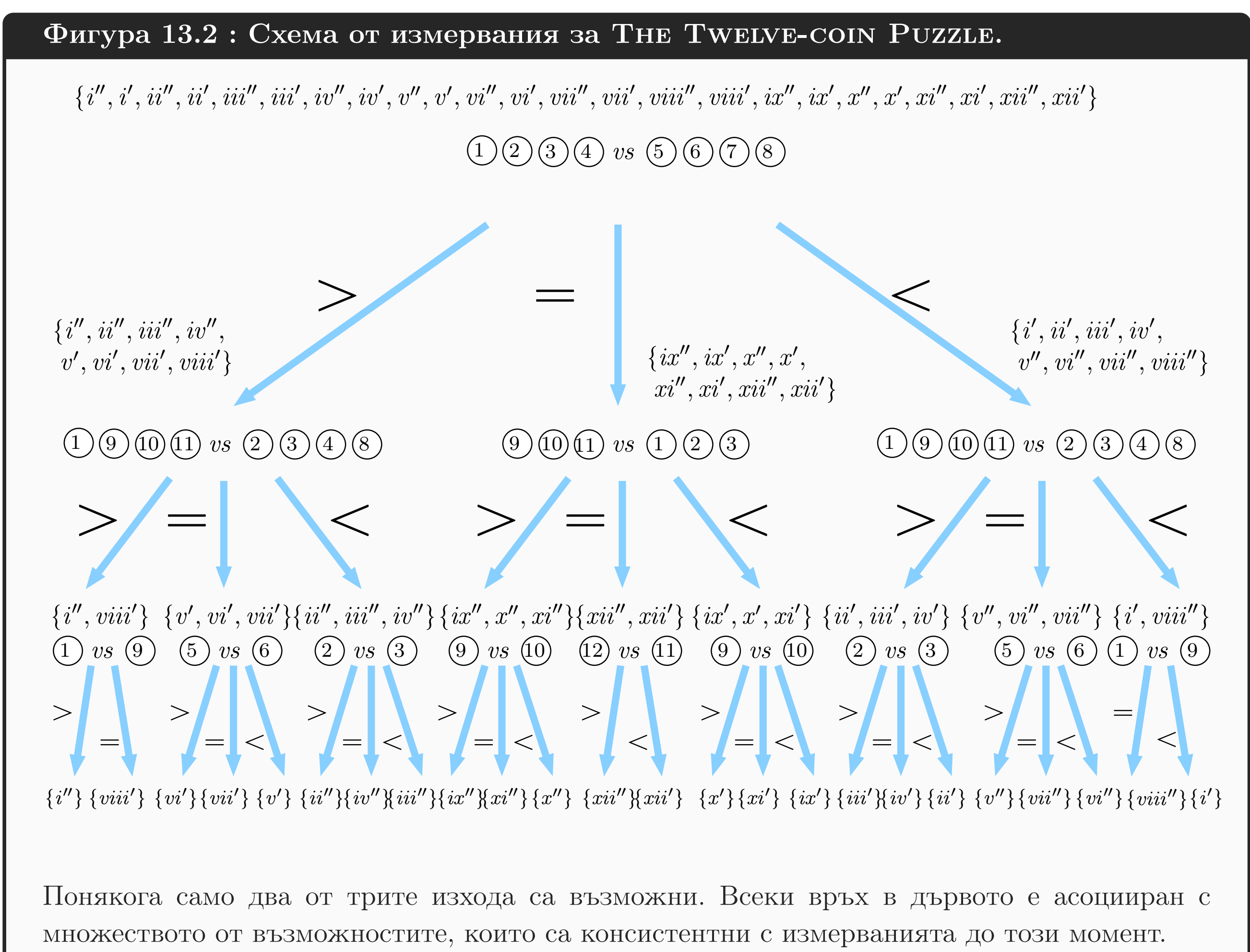
[†]"Тернарно" кореново дърво е такова, в което всеки връх има най-много три деца.

но това следва от показаната конкретна схема. Аргументът за долна граница казва само, че с по-малко измервания няма да стане. □

Задача 52: 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 51. Монетите са различими на външен вид, примерно са номерирани $c_1, \ldots, c_{12}$.
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.

Понякога само два от трите изхода са възможни. Всеки връх в дървото е асоцииран с множеството от възможностите, които са консистентни с измерванията до този момент.

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 $\geqslant 24$ 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 $\geqslant 5$, или множеството от останалите монети има кардиналност $\geqslant 5$.
- Множество с кардиналност $\geqslant 5$ има $\geqslant 10$ възможности за странната монета.
- Няма начин да определим една възможност измежду $\geqslant 10$ възможности, използвайки $\leqslant 2$ измервания, защото при не повече от три изхода от измерване, две последователни измервания могат да различат най-много 9 възможности.

Показахме 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. □

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 [32] това понятие е обяснено по следния начин.

Определение 93: Comparison sort [32, стр. 191]

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leqslant a_j$, $a_i = a_j$, $a_i \geqslant a_j$, 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 93 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. Същественото 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 94: Сортиращ алгоритъм, базиран на сравнения

Сортиращ алгоритъм, базиран на сравнения, е всеки сортиращ алгоритъм, който извършва сравнения от вида $a_i \overset{?}{<} a_j$ 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 Глава 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 64

Да сортираме входа със сравнения в някакъв смисъл е същото като да определим, само в резултат на изходите от сравненията, коя от пермутациите е истината.

Сравненията стават само на ред 4 в $\text{key} \overset{?}{<} A[j]$. Първото сравнение е $a_2 \overset{?}{<} a_1$. 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 $a_3 \overset{?}{<} a_1$. 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 $a_3 \overset{?}{<} a_2$.

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:

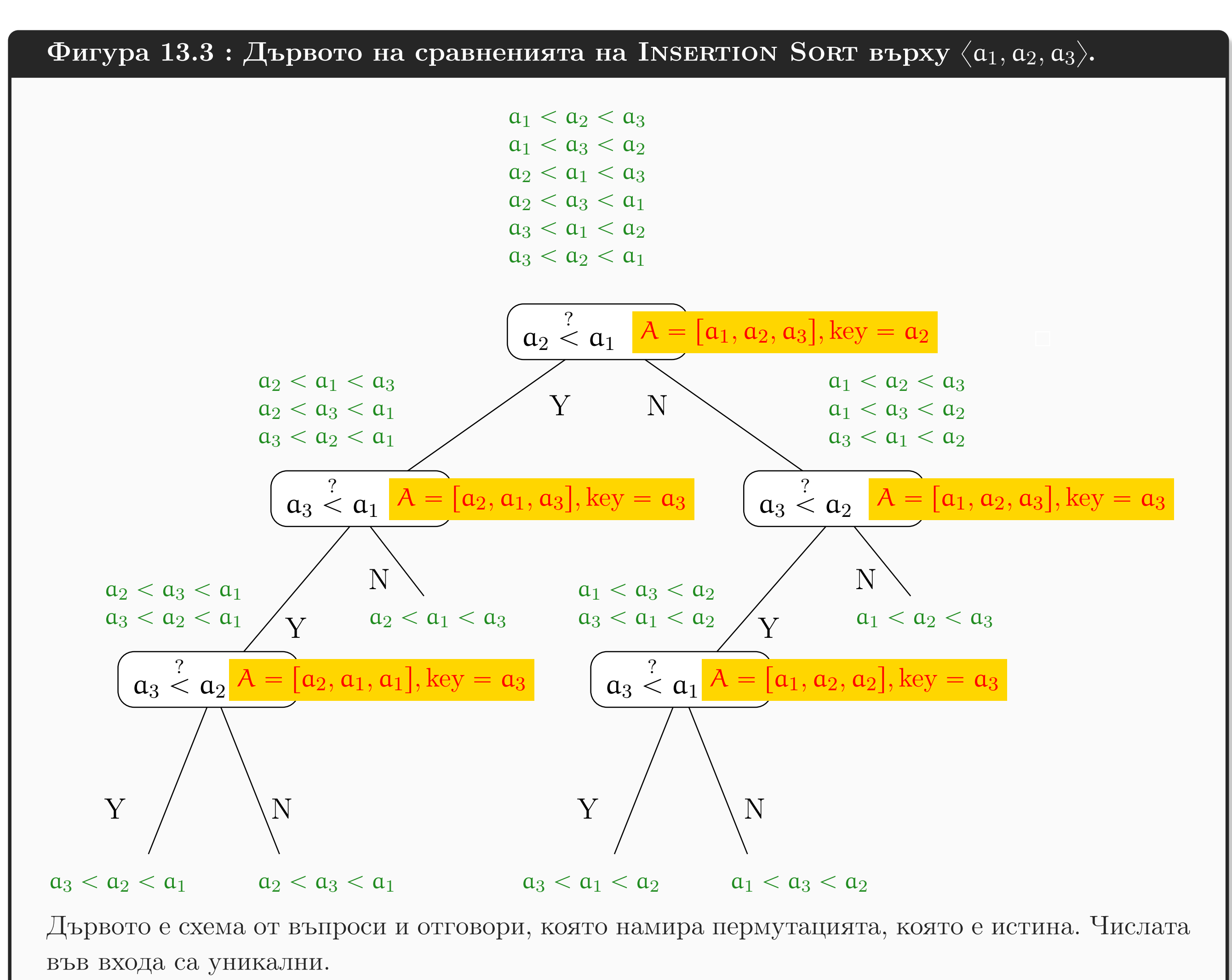
$$a_2 < a_1 < a_3$$

Да си припомним, че при сравнението $a_3 \overset{?}{<} a_1$ 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 алгоритъма; 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 $\langle a_1, a_2, a_3 \rangle$.



Дървото е схема от въпроси и отговори, която намира пермутацията, която е истина. Числата във входа са уникални.

Допълнение 59: 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:

$$
\begin{array}{ll}
a_1 < a_2 < a_3 & a_1 = a_2 < a_3 \\
a_1 < a_3 < a_2 & a_3 < a_1 = a_2 \\
a_2 < a_1 < a_3 & a_1 = a_3 < a_2 \\
a_2 < a_3 < a_1 & a_2 < a_1 = a_3 \\
a_3 < a_1 < a_2 & a_2 = a_3 < a_1 \\
a_3 < a_2 < a_1 & a_1 < a_2 = a_3 \\
 & a_1 = a_2 = a_3
\end{array}
$$

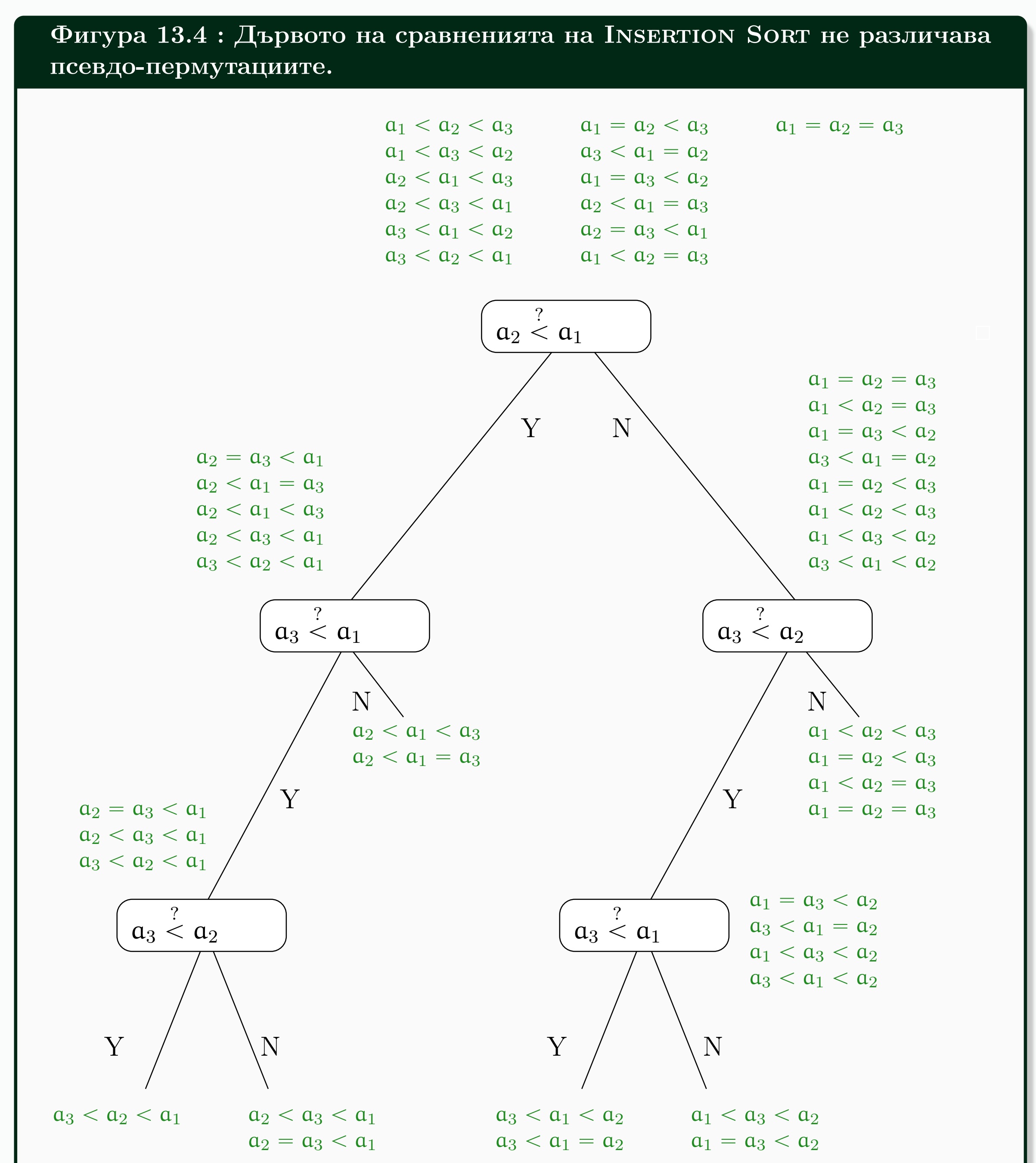
Това са слабите наредби от Допълнение 13 за $n = 3$. Както знаем от Допълнение 13, броят им е $P_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} k!$ при n елемента. Наистина, за $n = 3$ имаме

$$P_3 = \sum_{k=1}^{3} \left\{ {3 \atop k} \right\} k! = \left\{ {3 \atop 1} \right\} \cdot 1! + \left\{ {3 \atop 2} \right\} \cdot 2! + \left\{ {3 \atop 3} \right\} \cdot 3! = 1 \cdot 1 + 3 \cdot 2 + 1 \cdot 6 = 1 + 6 + 6 = 13$$

Ще наричаме тези състояния *псевдо-пермутациите*, а не “слабите наредби”, за да има аналогия с термина “пермутациите” на стр. 627. Терминът “псевдо-пермутации” е взет от статията [89].

Дървото на сравненията на INSERTION SORT за $n = 3$ с псевдо-пермутациите е показано на Фигура 13.4.

Фигура 13.4 : Дървото на сравненията 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 долната 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..n]: array of integers)
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])
```

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 48), 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 $\binom{n}{2}$[†]. 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 "$\varnothing$", 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; $\varnothing$ се асоциира с отговора, който е несъвместим с прежде получения отговор.

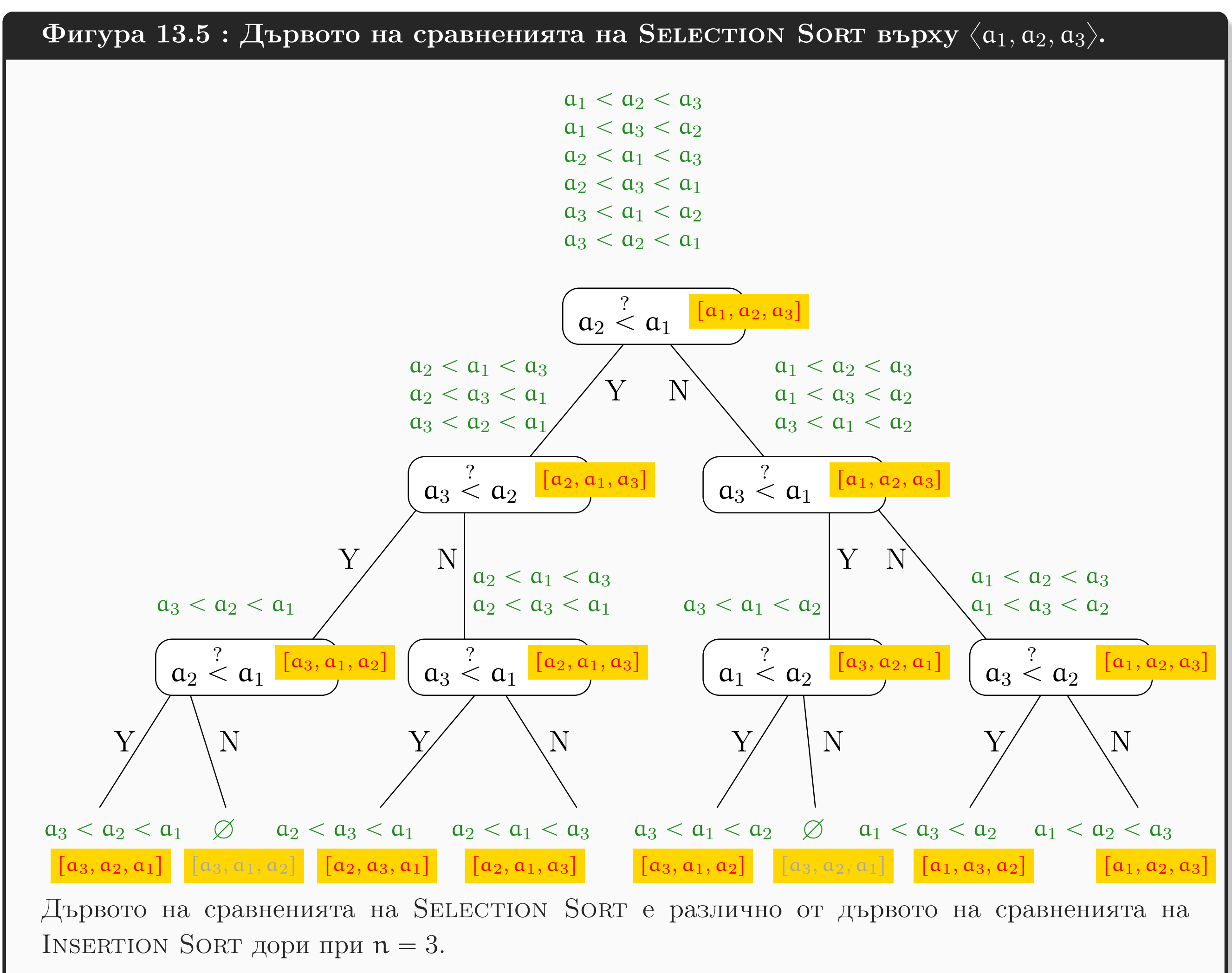
Във всеки връх на дървото е записано с червени букви на жълт фон съдържанието на масива A, като

- 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 сиво.

[†] 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 алгоритъми 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 [31, 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 51) и The Twelve-coin Puzzle (Задача 52) са податливи на анализ чрез дървета за вземане на решение, само че тернарни. И така, дърветата 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 65

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 "дърво**то** 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 95: Пермутациите спрямо дърветата за вземане на решение

Нека са дадени числа $a_1, \ldots, a_n$, две по две различни. *Пермутациите на* $a_1, \ldots, a_n$ са $n!$ на брой конюнкции от неравенства, където всяка конюнкция е над $n-1$ неравенства и е от вида

$$a_{i_1} < a_{i_2} < \cdots < a_{i_n}$$

за някоя същинска пермутация $\langle i_1, i_2, \ldots, i_n \rangle$ на $\{1, 2, \ldots, n\}$.

Щом числата са две по две различни, точно една от пермутациите е истина.

Определение 95 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 96: Сортиращо дърво за вземане на решение

Нека са дадени числа $a_1, \ldots, a_n$, две по две различни. *Сортиращо дърво за вземане на решение* е всяка дървовидна схема от сравнения от вида $a_i \overset{?}{<} a_j$ и техните отговори, където $1 \leqslant i < j \leqslant n$, 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 96. Можем да мислим, че листата с по нула пермутации са изтрити, 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 сравнение.

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 [125, стр. 16]). В контраст на това, машината с произволен достъп [125, стр. 19] е униформен изчислителен модел, защото за всяка големина на входа тази машината е една и съща. Машината на Turing (Определение 106) също е униформен изчислителен модел.

13.2.2.6 За структурата на сортиращите дървета за вземане на решение

Определение 97: Дърво за вземане на решение различава пермутациите

Нека $a_1, \ldots, a_n$ 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 $\leqslant 1$ пермутация.

Наблюдение 66: Сортиращите дървета за вземане на решение различават пермутациите

По определение (Определение 96), сортиращите дървета за вземане на решение различават пермутациите.

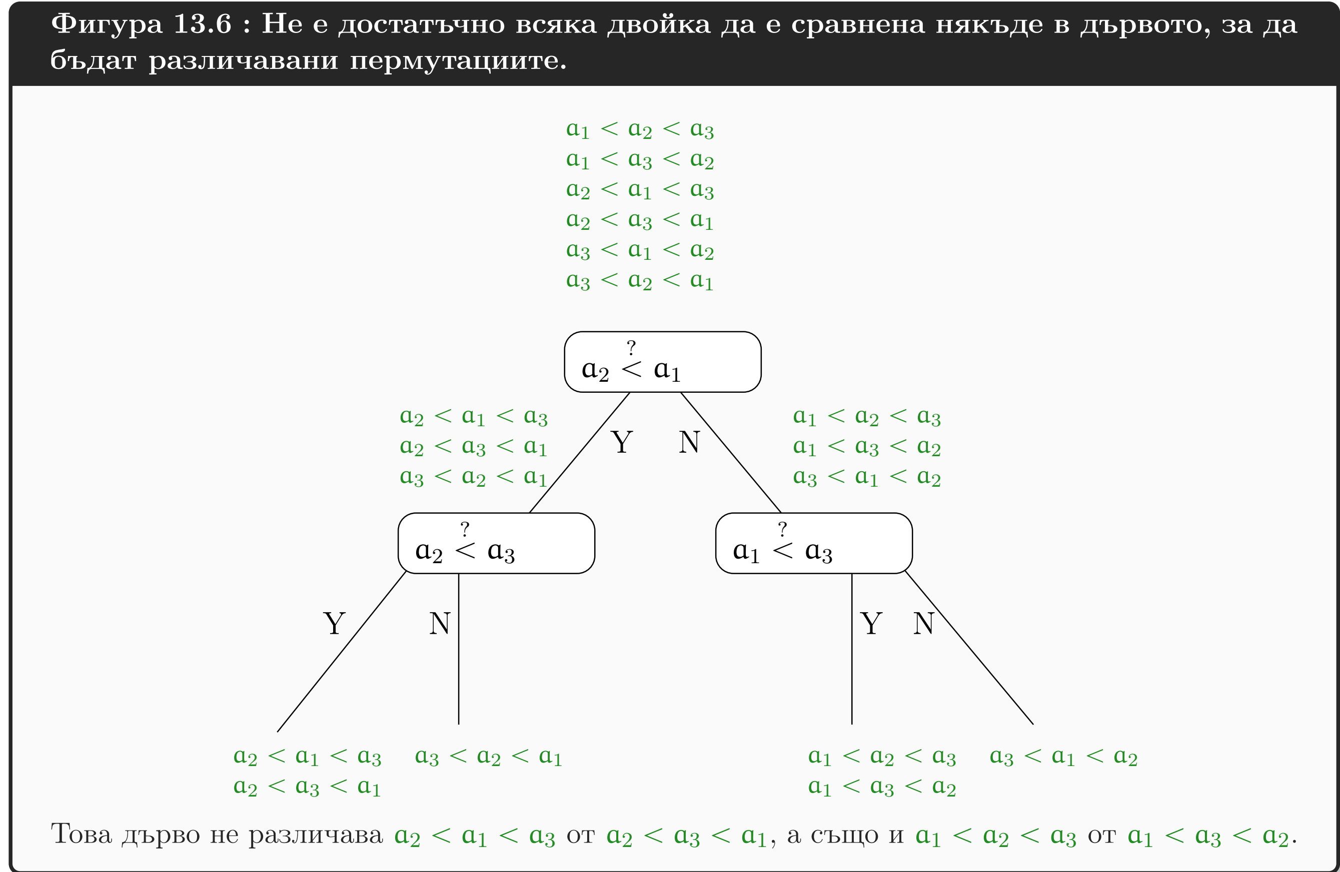
Каква е структурата на дърво T, което хем различава пермутациите, хем е възможно най-малко? Следните две наблюдения показват, че отговорът не е тривиален.

Наблюдение 67

За да различава T пермутациите, не е достатъчно T да прави сравнение $a_i \overset{?}{<} a_j$, за всеки i и j, такива че $1 \leqslant i < j \leqslant n$, **в някой връх**.

С други думи, не е достатъчно за всяко $\{i, j\} \subseteq \{1, \ldots, n\}$ да има поне един вътрешен връх с въпроса $a_i \overset{?}{<} a_j$. 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 и 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 $a_1 \overset{?}{<} a_3$, 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 68

За да различава T пермутациите, не е необходимо по всеки път от корена до листо да се прави сравнение $a_i \overset{?}{<} a_j$ за всеки i и j, такива че $1 \leqslant i < j \leqslant n$.

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 $\lim_{n\to\infty} \frac{2^{\binom{n}{2}}}{n!} = \infty$. Това на свой ред се извежда тривиално с логаритмуване на числителя и знаменателя, което води до $\lim_{n\to\infty} \frac{\Theta(n^2)}{\Theta(n \lg n)}$, 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 69: 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 96), всяко дърво T за вземане на решение, което сортира n уникални елементи, има $n!$ листа. От курса Дискретни Структури знаем, че щом T е пълно двоично дърво, в сила е $h \geqslant (\log_2 \ell) - 1$, където ℓ е броят на листата. Щом $\ell \geqslant n!$, в сила е $h \geqslant (\log_k n!) - 1$. Но съгласно Теорема 21, $\log_2 n! \asymp n \lg n$. □

Следствие 22: $\Omega(n \lg n)$ 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 69, $h_{min}(n) = \Omega(n \lg n)$. Тогава $\Omega(n \lg n)$ е долна граница за сортиране, базирано на сравнения, ако елементите са уникални.

Следствие 23: $\Omega(n \lg n)$ 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. Горе 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 интепретира 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 70: Сортиращите дървета сравняват всяка двойка съседни по големина елементи

Нека са дадени числата $a_1, \ldots, a_n$, две по две различни. Нека $\langle i_1, i_2, \ldots, i_n \rangle$ е уникалната пермутация на $1, 2, \ldots, n$, такава че

$$a_{i_1} < a_{i_2} < \cdots < a_{i_n}$$

Нека $A = \langle a_1, \ldots, a_n \rangle$. 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 $a_p \overset{?}{<} a_q$, или сравнението $a_q \overset{?}{<} a_p$. 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 $k \in \{1, \ldots, n-1\}$, дървото T не сравнява a_{i_k} с $a_{i_{k+1}}$ и въпреки това връща коректно пермутацията

$$\pi = a_{i_1} < a_{i_2} < \cdots < a_{i_{k-1}} < a_{i_k} < a_{i_{k+1}} < a_{i_{k+2}} < \cdots < a_{i_n}$$

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

$$a_{i_1} < a_{i_2} < \cdots < a_{i_{k-1}} < a_{i_{k+1}} < a_{i_k} < a_{i_{k+2}} < \cdots < a_{i_n}$$

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 $a_p \overset{?}{<} a_q$ 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 $\boldsymbol{\pi}$. Тогава T не е коректно дърво за вземане на решение. □

Заслужава си да се подчертае следното за Теорема 70. 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 същностно важно за дървото, въпреки че дървото "не знае" предварително кои са тези двойки.

Следствие 24: За всеки път p от корена до листо в сортиращо дърво, $|p| \geqslant n - 1$

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 70. □

Ще въведем едно много полезно понятие, което се ползва в няколко доказателства за долни граници.

Определение 98: Граф на сравненията за уникални елементи

Нека са дадени числа $a_1, \ldots, a_n$, 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 $a_1, \ldots, a_n$, 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.

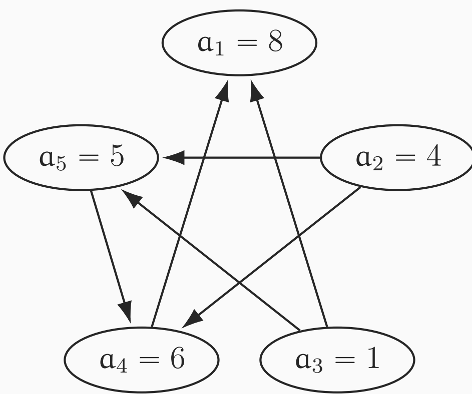
Наблюдение 69: Графът на сравненията е даг

Графът на сравненията е ацикличен, което следва веднага от факта, че релацията $<$ е антисиметрична.

Предвид това, оттук насетне 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 $a_3 \overset{?}{<} a_5$, $a_3 \overset{?}{<} a_1$, $a_1 \overset{?}{<} a_4$, $a_2 \overset{?}{<} a_5$, $a_2 \overset{?}{<} a_4$ и $a_4 \overset{?}{<} a_5$ с изходи съответно Y, Y, N, Y, Y и N.

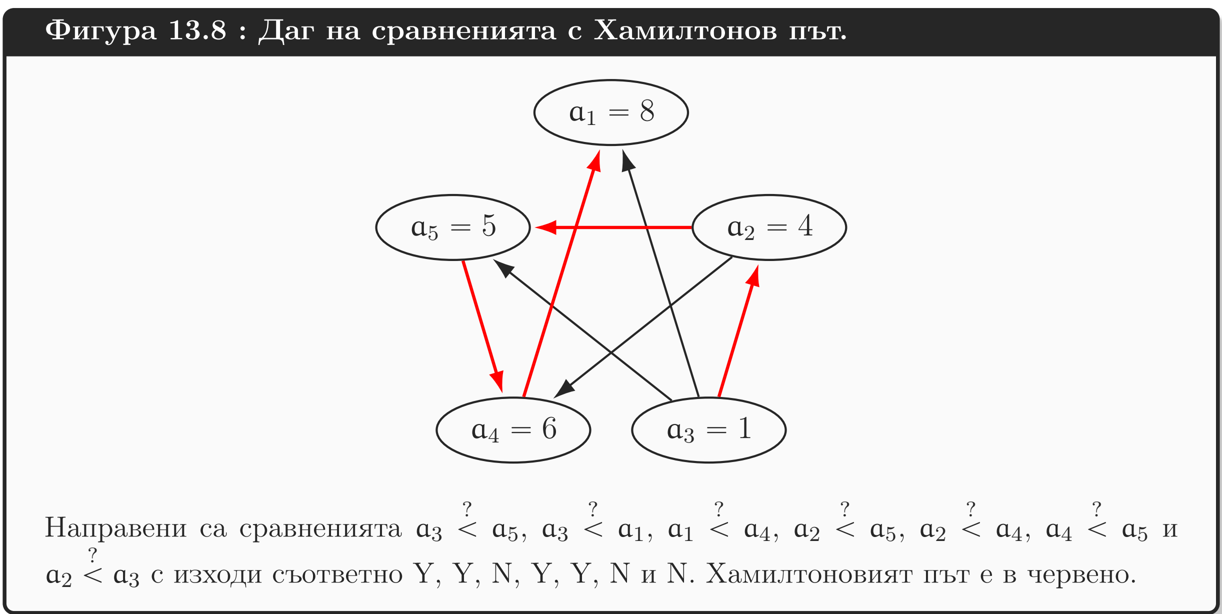
Теорема 70 влече, че след извършването на всички сравнения от сортиращо дърво за вземане на решение върху вход от уникални числа, полученият даг на сравненията съдържа Хамилтонов път, като Хамилтоновият път точно отговаря на уникалната пермутация-изход. В терминологията на Теорема 70, Хамилтоновият път е

$$p = a_{i_1}, a_{i_2}, \ldots, a_{i_n}$$

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 $a_3 \overset{?}{<} a_5$, $a_3 \overset{?}{<} a_1$, $a_1 \overset{?}{<} a_4$, $a_2 \overset{?}{<} a_5$, $a_2 \overset{?}{<} a_4$, $a_4 \overset{?}{<} a_5$ и $a_2 \overset{?}{<} a_3$ 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 15: дали дадени числа $a_1, \ldots, a_n$ са уникални, или има повторения. Задачата е задача за разпознаване. За краткост пишем "EU", от Element Uniqueness, 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 $a_i \overset{?}{<} a_j$. 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 $A[i] \overset{?}{=} A[j]$, но това може да се реализира чрез $A[i] \overset{?}{\nless} A[j] \wedge A[j] \overset{?}{\nless} A[i]$.

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 59 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

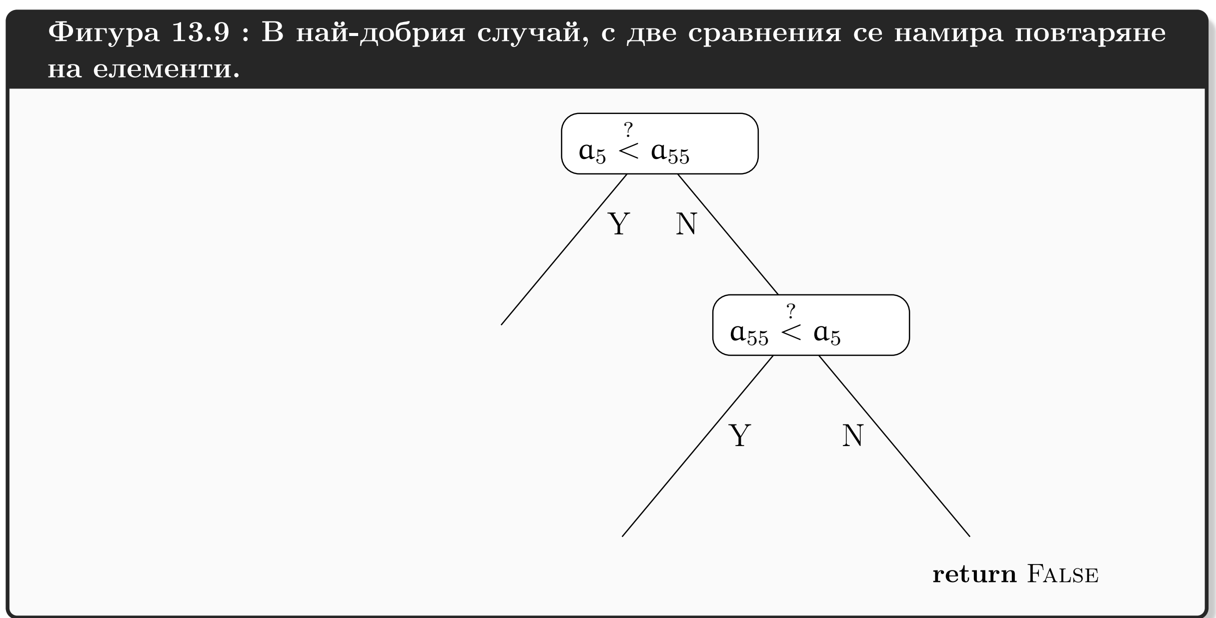
[†]А именно $P_n = \sum_{k=1}^{n} {n \brace k} k!$. Вижте Допълнение 13 за числата на Bell с наредба.

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 24 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 "дърво**то** 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 95, 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 69, 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 98). Сега ребро се слага само при положителен отговор, защото искаме ребрата да са само от по-малки към по-големи елементи.

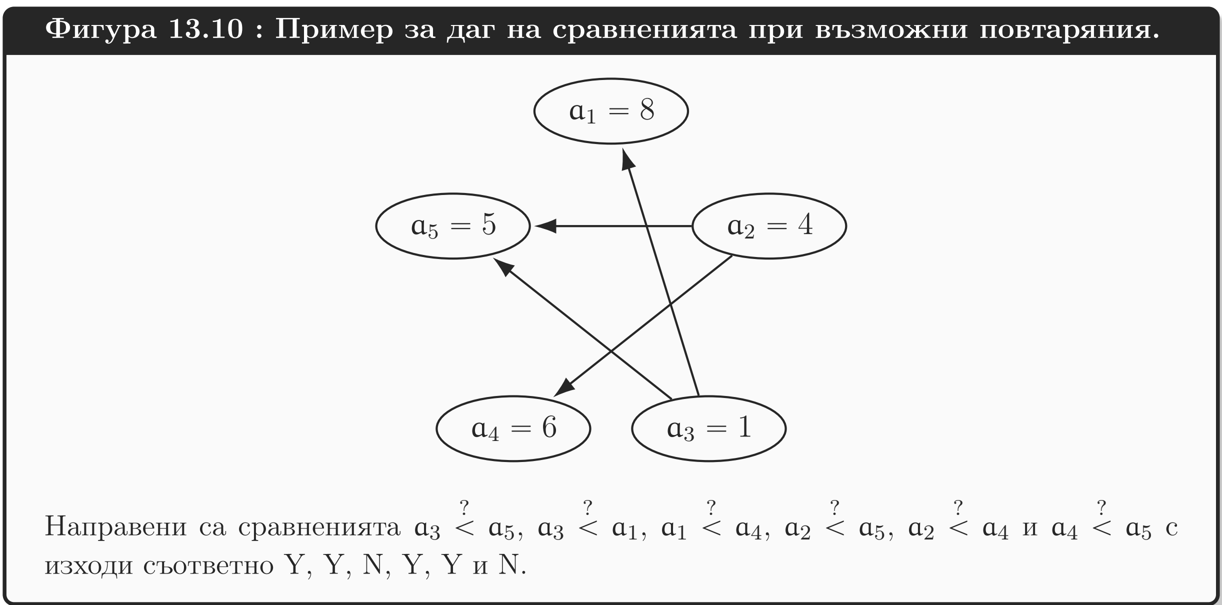
Определение 99: Граф на сравненията при възможни повтаряния

Нека са дадени числа $a_1, \ldots, a_n$, не непременно уникални. Нека са извършени сравнения от вида $a_i \overset{?}{<} a_j$, като $i \neq j$. *Графът на сравненията* е ориентиран граф с върхове $a_1, \ldots, a_n$, чиито ребра са генерирани по следния начин. За всяко сравнение $a_i \overset{?}{<} a_j$, 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 99. 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 пораждат ребра.

Фигура 13.10 : Пример за даг на сравненията при възможни повтаряния.



Направени са сравненията $a_3 \overset{?}{<} a_5$, $a_3 \overset{?}{<} a_1$, $a_1 \overset{?}{<} a_4$, $a_2 \overset{?}{<} a_5$, $a_2 \overset{?}{<} a_4$ и $a_4 \overset{?}{<} a_5$ 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 71: За дърветата на EU върху Да-екземплярите

Нека са дадени числата $a_1, \ldots, a_n$, две по две различни. Нека $A = \langle a_1, \ldots, a_n \rangle$. Нека

$$a_{i_1} < a_{i_2} < \cdots < a_{i_n}$$

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 70. 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 70 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 $k \in \{1, \ldots, n-1\}$, 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_5. 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 25: Установяването на уникалност генерира Хамилтонов път

В контекста на Теорема 71, нека дагът на сравненията (Определение 99), който $T(A)$ генерира, е G. Тогава G съдържа Хамилтонов път p. Нещо повече:

$$p = a_{i_1}, a_{i_2}, \ldots, a_{i_n}$$

Това, че сортиращите дървета различават пермутациите, е заложено в дефиницията им (Определение 96), така че просто го отбелязахме в Наблюдение 66. В контекста на EU обаче, изобщо не е очевидно, че дърветата (на EU) различават пермутациите. Но ние вече почти сме доказали това.

Теорема 72: Дърветата за вземане на решение на EU различават пермутациите

Всяко дърво за вземане на решение на EU различава пермутациите.

Доказателство: Нека имената от Теорема 71 и Следствие 25 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

$$\pi = a_{i_1} < a_{i_2} < \cdots < a_{i_n}$$

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 72, 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 на стр. 239, които очевидно имат сложност по време $\Theta(\lg n)$ в най-лошия случай.

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!

Естествено, задачата мълчаливо допуска, че ключовете са *позиционни*. Това означава, че всеки ключ има две външно различими 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

[†]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.

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 60: 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 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.

[†] 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 непревидено 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.

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 прибягва 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 гуугъл за *rubber hose crypto* за повече информация. В по-кротки и по-малко драматични сценарии, излизането извън модела по отношение на сигурността става не чрез прибягване към насилие, а чрез използване на, примерно, *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 свързването),
- 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 шикалкавене 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 $K = \{k_1, \ldots, k_n\}$, а множеството от лампите е $L = \{\ell_1, \ldots, \ell_n\}$. Разбиваме K на две равномощни множества, да кажем $K_1 = \{k_1, \ldots, k_{n/2}\}$ и $K_2 = \{k_{(n/2)+1}, \ldots, k_n\}$. 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.

Първо 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, ..., k_n, това, което се иска, е пермутация $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$, такава че k_i пали лампа $\ell_{\pi(i)}$,

за $1 \leqslant i \leqslant n$. 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

$$h(n) \geqslant \log_{2^n} n!$$

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.

Второ решение. Има и друго решение, пак с дърво за вземане на решение. Да установим кой ключ коя лампа пали е същото като да сортираме: и при ключовете, и при сортирането на числа искаме да намерим една пермутация измежду $n!$ възможни пермутации.

При сортирането в теоретично-информационния смисъл елементарното запитване (query) дали един елемент е по-малък от друг елемент, поради което всеки вътрешен връх на дървото за вземане на решение съдържа именно сравнение.

Кое е елементарното запитване при ключовете и лампите? То е следното: за дадено множество ключове и дадено множество от лампи, за всеки от ключовете дали пали лампа измежду тези лампи. Това обаче съответства на паралелен алгоритъм, а ние разглеждаме само секвенциални алгоритми, при които действа само един агент, който в даден момент прави само едно нещо. Ерго, ако искаме да приложим резултатите за дърветата за вземане на решение от Подсекция 13.2.2 и Подсекция 13.2.3, елементарното запитване трябва да е следното: за даден ключ и множество лампи, дали лампата, която той пали, е измежду тези лампи. За един ключ отговорът е булев, така че дървото е бинарно (в паралелния вариант, отговорът е булев вектор).

Това дърво за вземане на решение също сортира, в някакъв смисъл. Ако направим по-близка аналогия със сортирането на числа, ключовете са числата, лампите са позициите, и въпросите, които се задават (от секвенциалния алгоритъм) са от вида "За еди-кое число, дали правилната му позиция в сортираната редица е някоя от еди-кои позиции?". Листата са асоциирани с не повече от една пермутации, пермутациите са $n!$, дървото е бинарно, откъдето имаме долна граница $\Omega(n \lg n)$ за броя на въпросите в най-лошия случай. Отново: това е дърво, съответно на секвенциален алгоритъм.

Паралелният алгоритъм, ползващ n изчислителни агенти, работещи едновременно, може да е най-много n пъти по-бърз от секвенциалния, защото има секвенциален алгоритъм, който симулира паралелния със загуба на ефикасност точно n – очевидно никой паралелен алгоритъм с n паралелно работещи агенти не може да е повече от n пъти по-бърз от секвенциалния със само един работещ агент.

Оттук заключаваме, че всяка паралелна стратегия—елементарното действие е "разцъкване" на много ключове с последващо наблюдение—която намира свързването, прави $\Omega(\lg n)$ запитвания.

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 $a_1, a_2, \ldots, a_n$.
решение: Минимално $|a_i - a_j|$ за $1 \leqslant i < j \leqslant n$.

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*. А именно, ще редуцираме задачата EU до задачата Най-близки елементи. По този начин ще използваме наготово резултата за долна граница на EU. Редукцията става по следния начин.

Нека AlgCP е произволен алгоритъм за задачата Най-близки елементи (базиран на сравнения, естествено). Да разгледаме следния алгоритъм AlgX.

```
AlgX(Числа a1, a2, ..., an)
1  d ← AlgCP(a1, a2, ..., an)
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.

Едно забележка. В предния параграф се казва "в частност", защото негацията на съждението

> *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 61.

Допълнение 61: $f(n) \neq \Omega(g(n))$ **не влече** $f(n) = o(g(n))$

Какво следва от факта, че $f(n) \neq \Omega(g(n))$? С други думи, какво влече $f(n) \not\succeq g(n)$? Читателят може да се изкуши да помисли, че тогава непременно $f(n) \prec g(n)$, по аналогия с импликацията $x \not\geq y \rightarrow x < y$ върху реалните числа. Само че $\succeq$ и $\prec$ 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 12, има точно два случая, в $f(n) \not\succeq g(n)$ и $f(n) \not\prec g(n)$:

1. (2.39), тоест функциите са асимптотично несравними;
2. (2.40), тоест $f(n) \preceq g(n)$ 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 срещащите 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, срещащи се на практика функции е вярно, че $f(n) \not\succeq g(n)$ влече $f(n) \prec g(n)$.

Казваме, че задачата EU *се редуцира* до задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ в смисъл, че задачата EU има решение, което се състои в решение на задачата НАЙ-БЛИЗКИ ЕЛЕМЕНТИ, плюс пренебрежимо малко допълнителна работа. Засега ще се задоволим с неформално разбиране за "редукция". В Секция 14.6 ще разгледаме редукциите много по-подробно и прецизно.

Грубо казано, от факта, че Π_1 се свежда до Π_2 правим два извода:

- Ако Π_1 е трудна задача (каквото и да значи това), то Π_2 е трудна. Няма как някаква вътрешна, иманентна сложност да изчезне при редукцията.
- Ако Π_2 е лесна (каквото и точно да значи това), то Π_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 Π_1 е лесна, то Π_2 е лесна. **Такъв извод няма**. Винаги можем да направим нещата произволно сложни.
- Ако Π_2 е трудна, то Π_1 е трудна. **Такъв извод няма**. Причината отново е, че винаги можем да направим нещата сложни.

Наблюдение 70

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 52: Мода

екземпляр: Числа $a_1, a_2, \ldots, a_n$.
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 a1, a2, ..., an)
1  m ← AlgMode(a1, a2, ..., an)
2  c ← 0
3  for i ← 1 to n
4      if ai = m
5          c ← 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. Заключаваме, 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 весията е за броене.

Изч. Задача: 2SUM, версия за броене

екземпляр: Масив от числа $A[1..n]$.
решение: Броят на наредените двойки (i, j), такива че $1 \leqslant i < j \leqslant n$ и $A[i] + A[j] = 0$.

Нека Alg2SUMCount е произволен алгоритъм за задачата 2SUM в 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 $1 \leqslant i \leqslant n$. Тогава sum на ред 2 получава стойност n и AlgZ връща Да на ред 4.

Сега да допуснем, че съществуват $i_1, \ldots, i_k$, такива че $k > 1$ и $1 \leqslant i_1 < \cdots < i_k \leqslant n$ и $A[i_1] = \cdots = A[i_k]$. 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. Заключаваме, 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 40. 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 $a_1, \ldots, a_n$.

- Първо построяваме във време $\Theta(n)$ наредените двойки $(a_1, a_1 + \epsilon), \ldots, (a_n, a_n + \epsilon)$ 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 $a_1, \ldots, a_n$. 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 [41], 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 $O\left(n^2 \cdot \frac{\lg \lg n}{\lg n}\right)$ [54]. 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 100: 3SUM-hard

Нека Π е изчислителна задача за разпознаване. Казваме, че Π е 3SUM-*hard*, ако произволен екземпляр с размер n на 3SUM може да бъде решен чрез

1. решаване на Π върху константен брой екземпляри (на Π), 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, "Π е 3SUM-hard" казва "Π е поне толкова трудна, колкото 3SUM".

Забележете, че ако Π е 3SUM-hard, то 3SUM се свежда до Π, но "Π е 3SUM-hard" казва нещо доста конкретно и добре дефинирано, предвид Определение 100, за разлика от неформалното и не особено конкретно "3SUM се свежда до Π".

Изч. Задача 53: DEGENERACY TESTING

екземпляр: $p_1, p_2, \ldots, p_n$: точки в равнината
въпрос: Дали поне три от точките са колинеарни?

Ще докажем, че DEGENERACY TESTING е 3SUM-hard. Първо един помощен факт.

Лема 61

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 и a, и 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 $y = \alpha x + \beta$. □

Теорема 73: Задачата Degeneracy Testing е 3SUM-hard

Degeneracy Testing е 3SUM-hard.

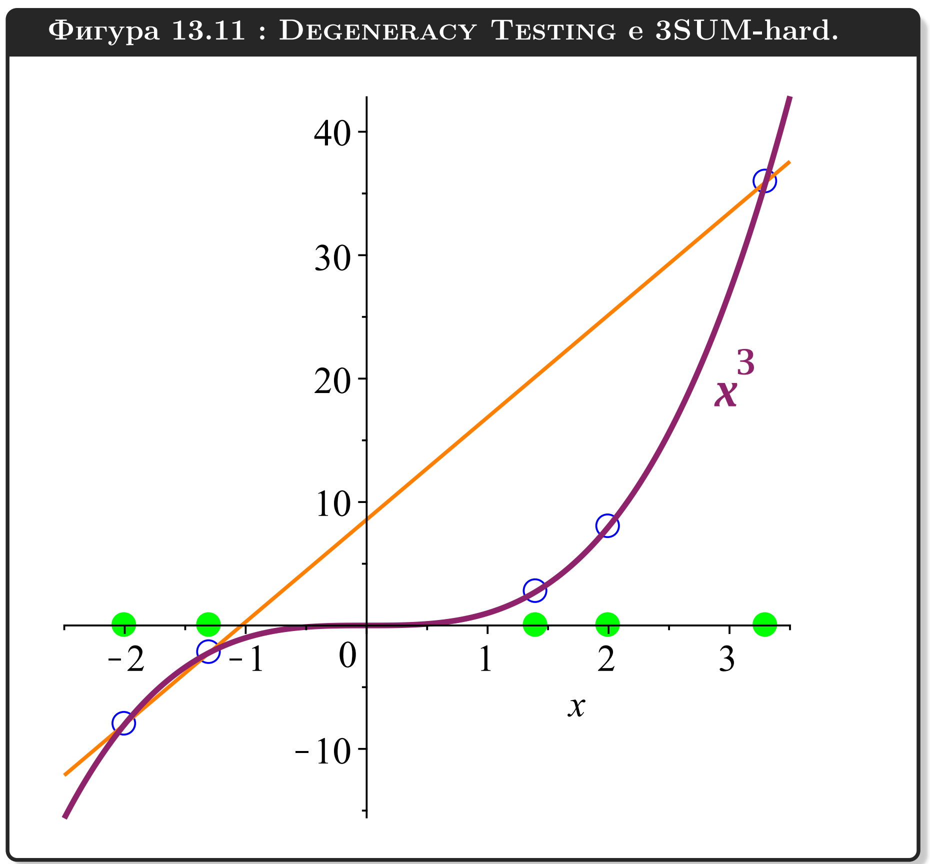
Доказателство: Нека е даден екземпляр на 3SUM: множество от реални числа $\{s_1, \ldots, s_n\}$. От него конструираме екземпляр $\{(s_1, s_1^3), \ldots, (s_n, s_n^3)\}$ на Degeneracy Testing. Съгласно Лема 61, отговорът на въпроса "Дали има поне три точки от $\{(s_1, s_1^3), \ldots, (s_n, s_n^3)\}$, които са колинеарни?" е същият като отговорът на въпроса "Дали има три елемента на $\{s_1, \ldots, s_n\}$, които се сумират до нула?". Конструкцията работи достатъчно бързо според изискването на Определение 100, 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. □

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).



Фигура 13.11 : Degeneracy Testing е 3SUM-hard.

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 Ви се налага да зададете оттук насетне 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, която следвате, при което той знае всеки следващ Ваш въпрос, но може и да не я 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 $a_5 \geqslant a_{10}$, нито може да даде отговори, от които следва, че $a_5 \geqslant a_{10}$. 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 $a_i \overset{?}{<} a_j$, за да изчислите пермутацията на входа, която представлява сортирана редица. 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 $a_i \overset{?}{<} a_j$, дяволът отговаря ДА или НЕ въз основа на един единствен критерий – при кой от тези отговори, броят на пермутациите, които остават възможни (тоест, които са консистентни с всички досегашни отговори и този отговор) е по-голям. Ако отговор ДА дава по-голям брой пермутации, дяволът ще каже ДА. Ако отговор НЕ дава по-голям брой пермутации, дяволът ще каже НЕ. Ако отговор ДА оставя същият брой пермутации като отговор НЕ, дяволът казва ДА или НЕ произволно. 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 въпроси и отговори, 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 $a_1 \overset{?}{<} a_2$ и $a_2 \overset{?}{<} a_3$. 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 $a_1 \overset{?}{<} a_2$, или $a_1 \overset{?}{<} a_3$, или $a_2 \overset{?}{<} a_3$. 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,

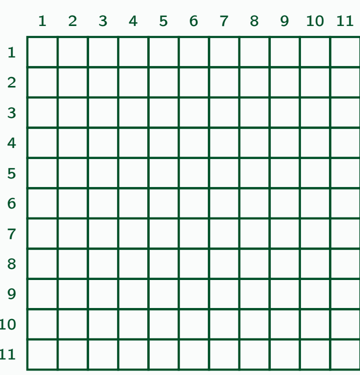
с отрицателния. Така че няма значение какво ще отговори на първото запитване; да кажем, че дяволът винаги отговаря с ДА на първия въпрос. На втория въпрос обаче дяволът не отговаря произволно! Тъй като след отговора на първия въпрос има 3 възможни пермутации, консистентни с отговора, то, какъвто и втори въпрос да зададете, съществува отговор—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 уличе́н 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 онагледи 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. Няма смисъл от дявол, който нарочно играе субоптимално, разчитайки на Вашата слабост, за да може играта да приключи най-скоро.

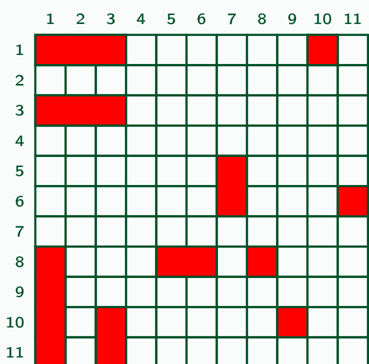
Допълнение 62: Морски шах с дявола

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.

Правилата са следните. Играят двама играчи. Всеки разполага с квадратна мрежа, да кажем 11×11, с номерирани редове и колони, наречена *море*:

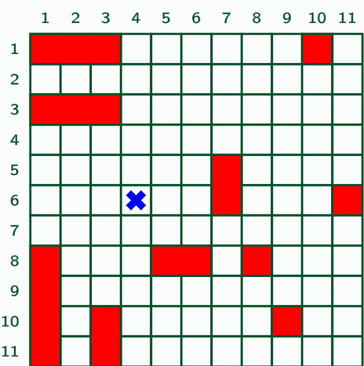


и няколко фигури, наречени *кораби*. Всеки кораб е правоъгълник, който може да бъде поставен, хоризонтално или вертикално, върху няколко съседни квадратчета от мрежата, запълвайки ги точно. Корабите са следните: един кораб 1×4, два кораба 1×3, три кораба 1×2 и четири кораба 1×1. Всеки играч крие своето 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 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:



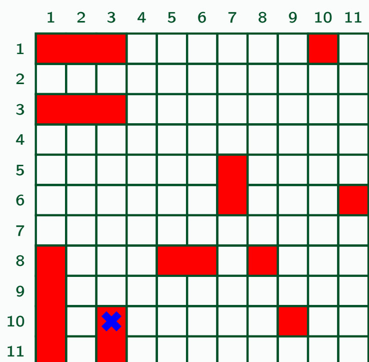
Самата игра се състои в редуващи се "изстрели" на всеки играч в морето на другия играч, като "изстрел" е просто казване на координатите на някое квадратче от морето на противника. Всеки играч записва изстрелите на противника си с хиксове върху своето море. Единият играч е първи. Той или тя казва на втория някаква наредена двойка (i, j), където $1 \leqslant i, j \leqslant 11$. 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:

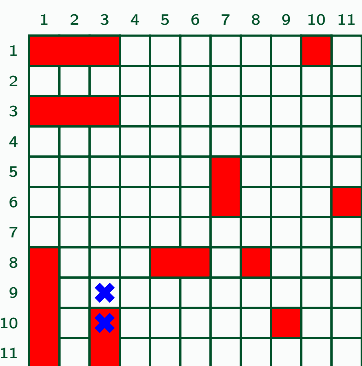
  

  и на свой ред играе, казвайки координатите на някакво квадратче от морето на първия играч.

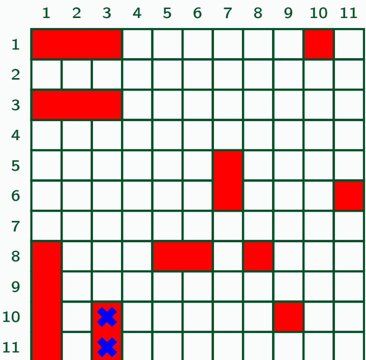
- Ако на това квадратче има кораб, който е по-голям от 1×1 и има поне едно неударено квадратче освен (i, j) (в нашия пример, нека $(i, j) = (10, 3)$), то вторият играч казва "удар" и отбелязва удара с хикс:

  След това отново играе първият играч – този, който току-що нанесе удара. Първият продължава да играе, докато или не уцели вода:

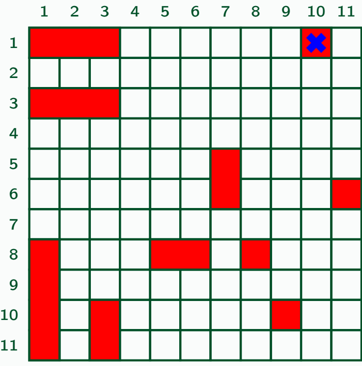


при което вече играе вторият играч, или не удари и последното досега неударено квадратче:

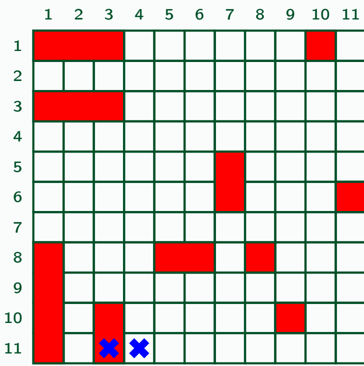


при което вторият играч казва "потопен" и първият играч продължава да играе.

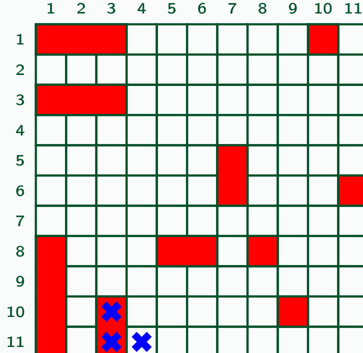
- Ако или на това квадратче има кораб 1×1—в нашия пример, нека $(i, j) = (1, 10)$:



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:

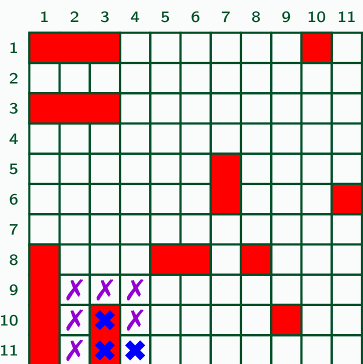


то вторият играч казва "потопен" и отбелязва мястото на удара с хикс.



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 улучват 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".

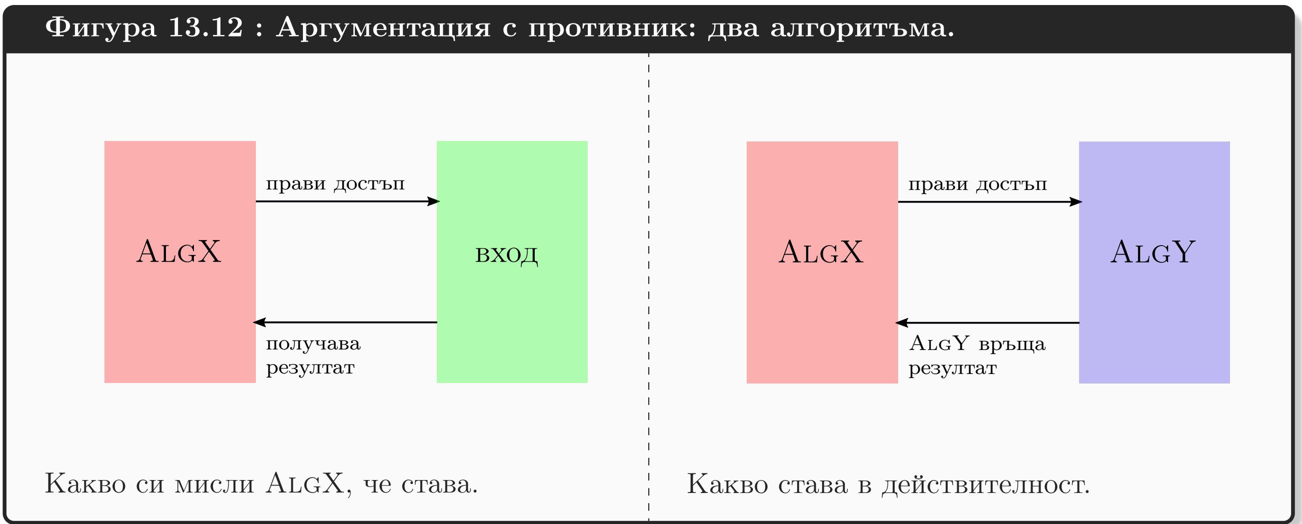
Примерът с морския шах не е перфектна илюстрация за даване на долна граница на изчислителна задача, понеже при решаване на изчислителна задача играете само Вие (само Вие задавате въпроси), а дяволът само отговаря; в морския шах Вие се редувате с дявола за задаване на въпроси. Въпреки тази разлика, би трябвало този пример да даде добра интуиция за стратегията на опонент, който е максимално злонамерен и има неограничена изчислителна мощ.

[a] Това 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 да 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.
- AlgY трябва да е консистентен. Редицата резултати $\mathcal{R}$, които той дава на AlgX, трябва да е такава, че да има поне един вход със съответната големина, за който редицата от резултатите е точно $\mathcal{R}$.

Както казахме, AlgY играе ролята на противника (дявола). Дизайнерите на AlgY обаче сме ние. Нашата цел е да докажем долна граница за работата на AlgX в най-лошия случай. За тази цел ние конструираме противник (AlgY), който е в състояние да създаде такъв вход, който да накара AlgX да извърши поне толкова достъпи, колкото е долната граница, която доказваме. Подчертаваме, че AlgY не просто генерира някакъв вход поначало, а прави нещо много по-изтънчено и перфидно: AlgY **създава входа в процеса на достъпи и връщане на резултати**, отчитайки конкретните достъпи, извършени от AlgX.

Да кажем, че AlgY кара AlgX 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 AlgY, който кара AlgX 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 AlgY$'$, който кара AlgX 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 71

Да допуснем, че сме доказали, че конструираният от нас противник кара AlgX 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 Π. Това е сравнително по-труден за конструиране и анализиране противник. Сега вече противникът AlgY не знае нищо за конкретния алгоритъм AlgX, който решава Π. От гледна точка на AlgY, AlgX е просто черна кутия, от която излизат достъпи до входа (queries). Фигура 13.12 илюстрира адекватно и тази ситуация, но сега имаме предвид, че

- AlgY не знае нищо за AlgX.
- "AlgX" не означава конкретен алгоритъм, а означава кой да е алгоритъм за изчислителната задача Π.

Допусканията за неограничена мощност, "злонамереност" и консистентност на AlgY 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 $\lceil \log_2 n \rceil$. 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 безкрайно 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 числата, 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

[†]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.

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 $d \in S \backslash \{c\}$. 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:

$$|S_0| = n$$
$$|S_{i+1}| \geqslant \left\lceil \frac{|S_i|}{2} \right\rceil, \text{ ако } i \geqslant 1$$

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 $a_1, a_2, \ldots, a_n$. 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, ..., a_n: реални числа)
1  max ← a_1
2  for i ← 2 to n
3      if a_i > max
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[†], то противникът прави a_i по-малко от всяко друго число от входа. При това положение няма как a_i да е максимално число. Не можем да уличим противника в лъжа, защото a_i не е било сравняван изобщо, така че след края на работата на ALGMAX, когато поискаме от противника да видим входа, той може да даде на a_i, каквото поиска. Отговорите, които противникът е дал, ще са консистентни с тази стойност на a_i.

- Ако ALGMAX върне a_j, където $j \neq i$, противникът прави a_i по-голям от всеки друг елемент. При това положение a_i е максималното число. Отново, не можем да уличим противника в лъжа, защото a_i не е било сравняван изобщо.

И в двата случая ALGMAX не работи коректно, така че той не е коректен алгоритъм и допускането е опровергано.

[†]Звучи безумно алгоритъмът да върне елемент, който изобщо не е бил разгледан, но тук правим теоретично доказателство и разглеждаме всички случаи.

Долната граница, която следва от съображението, че всеки елемент трябва да е бил сравнен поне веднъж с друг, е $\left\lceil \frac{n}{2} \right\rceil$. За да се убедим в това, да си представим графа на сравненията, но в неориентиран вариант. Ние вече въведохме "ориентиран граф на сравненията" (Определение 99). *Неориентиран граф на сравненията* е нещо подобно: върховете са числата a_1, a_2, ..., 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 $m \geqslant \left\lceil \frac{n}{2} \right\rceil$.

Но долната граница $\left\lceil \frac{n}{2} \right\rceil$ 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 74: 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, ..., 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, ..., 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 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, ..., a_n. Противникът не може да бъде уличе́н 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 големини 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 големини между два съседни по големина елементи от G'', и така нататък – всичко е възможно за относителните им големини.

Заключаваме, че неориентираният граф на сравненията е свързан. Добре известен факт от теорията на графите е, че всеки свързан граф е изпълнено $m \geqslant n-1$. Тогава броят на сравненията е поне $n-1$.

Второ доказателство: Сега ще използваме "даг на сравненията". БОО, нека числата са две по две различни, така че имаме предвид Определение 98. Нека AlgMax е произволен алгоритъм за намиране на максимално число. Нека AlgMax$(a_1, \ldots, a_n)$ връща като максимално число a_i. Нека дагът на сравненията, генериран от тези изчисления, е $\widetilde{G}$.

Наблюдение 72

В текущия контекст, за всяко $j \in \{1, \dots, n\}$ е вярно, че

- a_j е сифон в $\widetilde{G}$ 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 $\widetilde{G}$ тстк a_j не е спечелил нито едно сравнение; иначе казано, a_j е загубил всички сравнения, в които е участвал.

Лема 62

В текущия контекст, $\widetilde{G}$ има точно един сифон и той е максималното число измежду $a_1, \dots, a_n$.

Доказателство: Нека a_i е максималното число. Очевидно a_i не е губил сравнения. От Наблюдение 72 следва, че той е сифон в $\widetilde{G}$. Ще покажем, че в $\widetilde{G}$ други сифони няма.

Да допуснем противното. Тогава някой a_k, такъв че $k \neq i$, е сифон в $\widetilde{G}$, което е същото (Наблюдение 72) 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 същите като тези преди увеличаването на a_k. Полученото противоречие показва, че такъв a_k не съществува. □

Желаният резултат следва веднага от Лема 62, 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. □

Трето доказателство: Твърдим, че дагът на сравненията $\widetilde{G}$ е слабо свързан. Наистина, ако $\widetilde{G}$ 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 уличе́н в лъжа, защото стойностите, които ще генерира като вход, ще са консистентни с резултатите от сравненията.

Щом $\widetilde{G}$ е слабо свързан, той има поне $n-1$ ребра, което значи, че са направени поне $n-1$ сравнения. □

По същество третото доказателство е същото като първото доказателство.

13.4.5 Долна граница $\left\lceil \frac{3n}{2} \right\rceil - 2$ 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 $n \geqslant 2$. 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 може да сторим това и по-бързо ето така.

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 $\left\lfloor \frac{n}{2} \right\rfloor$ и едно число a_j остава извън двойките. Намираме $\left\lfloor \frac{n}{2} \right\rfloor$ максимума, по един от всяка двойка, и още $\left\lfloor \frac{n}{2} \right\rfloor$ минимума, по един от всяка двойка. После глобалният максимум се намира измежду тези максимуми и a_j с $(\left\lfloor \frac{n}{2} \right\rfloor + 1) - 1 = \left\lfloor \frac{n}{2} \right\rfloor$ сравнения. После глобалният минимум се намира от тези минимуми 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 $(\left\lfloor \frac{n}{2} \right\rfloor + 1) - 1 = \left\lfloor \frac{n}{2} \right\rfloor$ сравнения. Общият брой на сравненията е $3\left\lfloor \frac{n}{2} \right\rfloor$. 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 $\left\lceil \frac{3n}{2} \right\rceil - 2$ 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 .. n]: цели числа без повторения, n ⩾ 2)
 1  k ← ⌊n/2⌋
 2  създай масиви MAX[1, ..., k], MIN[1, ..., k]
 3  for i ← 1 to k
 4      if A[2i − 1] > A[2i]
 5          MAX[i] ← A[2i − 1], MIN[i] ← A[2i]
 6      else
 7          MAX[i] ← A[2i], MIN[i] ← A[2i − 1]
 8  tmpmax ← MAX[1], tmpmin ← MIN[1]
 9  for i ← 2 to k
10      if MAX[i] > tmpmax
11          tmpmax ← MAX[i]
12      if MIN[i] < tmpmin
13          tmpmin ← MIN[i]
14  if isodd(n)
15      if A[n] > tmpmax
16          tmpmax ← A[n]
17      if A[n] < tmpmin
```

18 tmpmin ← $A[n]$
19 **return** (tmpmax, tmpmin)

Ще докажем, че този алгоритъм е оптимален по отношение на броя на сравненията. 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.

Да си представим произволен алгоритъм AlgMaxMin за тази задача. Нека G е дагът на сравненията, генериран от работата на AlgMaxMin върху вход с n елемента. От Лема 62 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 73

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.

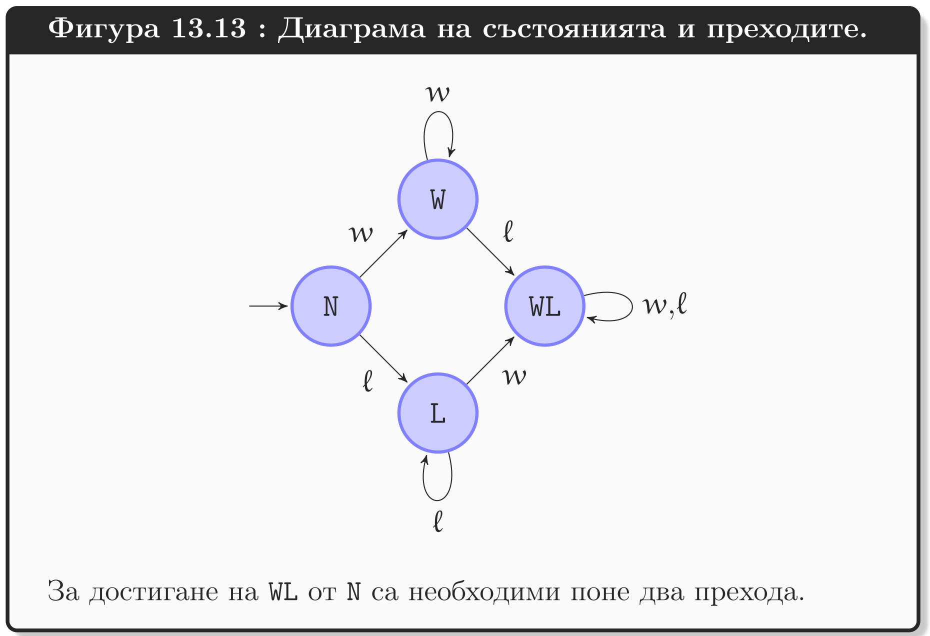
L, ако x е загубило поне едно сравнение и не е спечелило нито едно сравнение.

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 с 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 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 $a_i \overset{?}{<} a_j$. Всяко от a_i и a_j има състояние, което е елемент на $\{\mathtt{N}, \mathtt{W}, \mathtt{L}, \mathtt{WL}\}$. Противникът държи сметка за текущото състояние на всеки елемент. Да разгледаме всички възможни комбинации от състояния. Те са само $\binom{2+4-1}{2} = 10$, а не $4^2 = 16$, защото комбинациите са двуелементните мултимножества над опорното множество $\{\mathtt{N}, \mathtt{W}, \mathtt{L}, \mathtt{WL}\}$. 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 се 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 $\left\lfloor \frac{n}{2} \right\rfloor$ на брой. Във всеки друг случай (освен N срещу N), противникът отговоря 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 $\langle \mathtt{N}, \mathtt{N}, \ldots, \mathtt{N} \rangle$. 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.

Ключовото наблюдение е: това **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 74

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 $\left\lfloor \frac{n}{2} \right\rfloor$ 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

$$2n - 2 - \left\lfloor \frac{n}{2} \right\rfloor = 2n - 2 + \left\lceil -\frac{n}{2} \right\rceil = \left\lceil 2n + \left(-\frac{n}{2}\right) \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2 \qquad (13.7)$$

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 може да някое сравнение да се ползва 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 75

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 $\langle \mathtt{L}, \mathtt{WL} \rangle$, както е написано в таблицата; ако отговори $\langle \mathtt{W}, \mathtt{WL} \rangle$, 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 63: 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 $\langle \mathtt{WL}, \mathtt{WL} \rangle$. 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:

$$\overset{\mathtt{L}}{a_1} < \overset{\mathtt{WL}}{a_2} < \overset{\mathtt{W}}{a_3} \qquad\qquad \overset{\mathtt{L}}{a_4} < \overset{\mathtt{WL}}{a_5} < \overset{\mathtt{W}}{a_6}$$

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 трябват 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

$$a_4 < a_1 < a_2 < a_3 < a_5 < a_6$$
$$a_4 < a_1 < a_2 < a_5 < a_3 < a_6$$
$$a_4 < a_1 < a_2 < a_5 < a_6 < a_3$$
$$a_1 < a_4 < a_2 < a_3 < a_5 < a_6$$
$$a_1 < a_4 < a_2 < a_5 < a_3 < a_6$$
$$a_1 < a_4 < a_2 < a_5 < a_6 < a_3$$
$$a_1 < a_2 < a_3 < a_4 < a_5 < a_6$$
$$a_1 < a_2 < a_4 < a_3 < a_5 < a_6$$
$$a_1 < a_2 < a_4 < a_5 < a_3 < a_6$$
$$a_1 < a_2 < a_4 < a_5 < a_6 < a_3$$

Тогава има точно 10 пермутации, съвместими с отговора N. Излиза, че сравнението $a_2 \overset{?}{<} a_5$ води до перфектна дихотомия (разбиване на множеството на пермутациите на два равномощни дяла) и е съвсем смислено, ако искаме да сортираме.

Съгласно Наблюдение 74 и Наблюдение 75, $|X \cap Y| \leqslant \left\lfloor \frac{n}{2} \right\rfloor$. Тогава

$$|Z| \geqslant 2n - 2 - \left\lfloor \frac{n}{2} \right\rfloor$$

Ползваме (13.7) и получаваме

$$|Z| \geqslant \left\lceil \frac{3n}{2} \right\rceil - 2$$

13.4.6 Долна граница $\left\lfloor \frac{3n-2}{2} \right\rfloor$ за намиране на медиана

Фундамент. Става дума за Задача 16. Естествено, разглеждаме само намиране на медиана чрез сравнения. Алгоритъмът Compute Median на стр. 250 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 [22], [23] 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 [23] 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. Подобрения има и от двете страни. Доколкото е известно на автора на записките, най-ефикасното намиране на медиана става с $\leqslant 2.95n + o(n)$ сравнения [35], а най-високата долна граница за броя на сравненията е $2n + o(n)$ [13]. Всичко това е добре обяснено в [36].

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

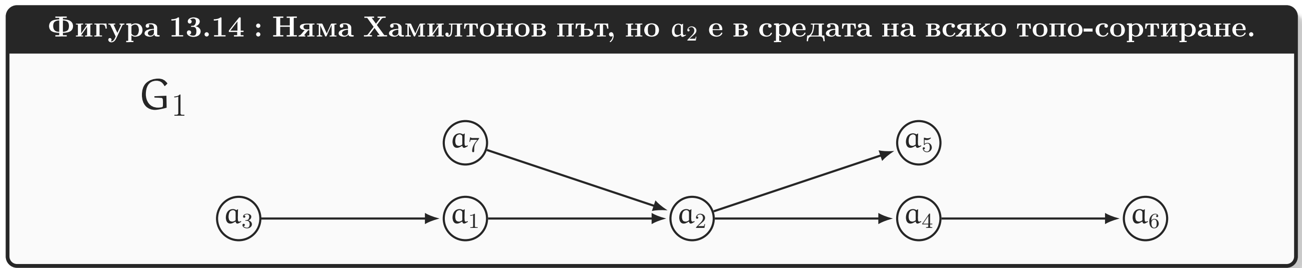
на долната и горната граница. Разминаването се получава само ако отчитаме мултипликативната константа.

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 98). Нека входът е $a_1, a_2, \ldots, a_n$. Нека $A = \{a_1, \ldots, a_n\}$. За простота допускаме, 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 екстремен 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 13.14 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 13.14 : 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 които 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 13.14, $S(a_3) = \emptyset$, $L(a_3) = \{a_1, a_2, a_4, a_5, a_6\}$, $S(a_1) = \{a_3\}$, $L(a_1) = \{a_2, a_4, a_5, a_6\}$, $S(a_2) = \{a_1, a_3, a_7\}$, $S(a_2) = \{a_4, a_5, a_6\}$ 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 13.14. 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 равномощни. Това не е случайно, както показва Лема 63. Първо една дефиниция.

Определение 101: 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 63

В текущия контекст, a_m е единствената възможна медиана тстк $|S(a_m)| = |L(a_m)| = \left\lfloor \frac{n}{2} \right\rfloor$.

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 $a_i \nparallel a_m$, 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 уличѐн 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 противното, веднага заключаваме, че за поне едно ребро $(u, v) \in E(G)$ е изпълнено $h(v) < h(u)$, което е невъзможно съгласно Определение 66.

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. □

Фигура 13.15 илюстрира Лема 63, показвайки два дага на сравненията.

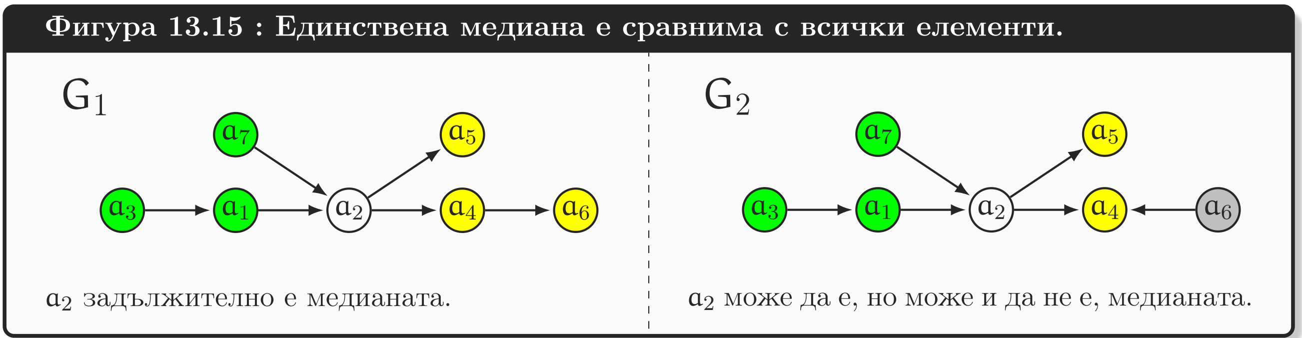
- G_1 е дагът от Фигура 13.14, в който всички елементи на сравними с a_2 и a_2 е единствената възможна медиана.
- G_2 е различен даг. В него a_6 не е сравним с a_2 и наистина a_2 не е единствената възможна медиана. Примерно, ако противникът даде стойности $a_3 = 1$, $a_1 = 2$, $a_7 = 3$, $a_2 = 10$, $a_4 = 11$, $a_5 = 12$ 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 $S(a_2) \uplus L(a_2)$, що се отнася до G_2.

Фигура 13.15 : Единствена медиана е сравнима с всички елементи.



a_2 задължително е медианата. | a_2 може да е, но може и да не е, медианата.

Следствие 26: 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 63, 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. □

Следствие 27

Ако дагът на сравненията определя медианата еднозначно, той има поне $n-1$ ребра.

Доказателство: Следва веднага от Следствие 26: всеки слабо свързан ориентиран граф има поне $n-1$ ребра по същите причини, по които всеки свързан неориентиран граф има поне $n-1$ ребра. □

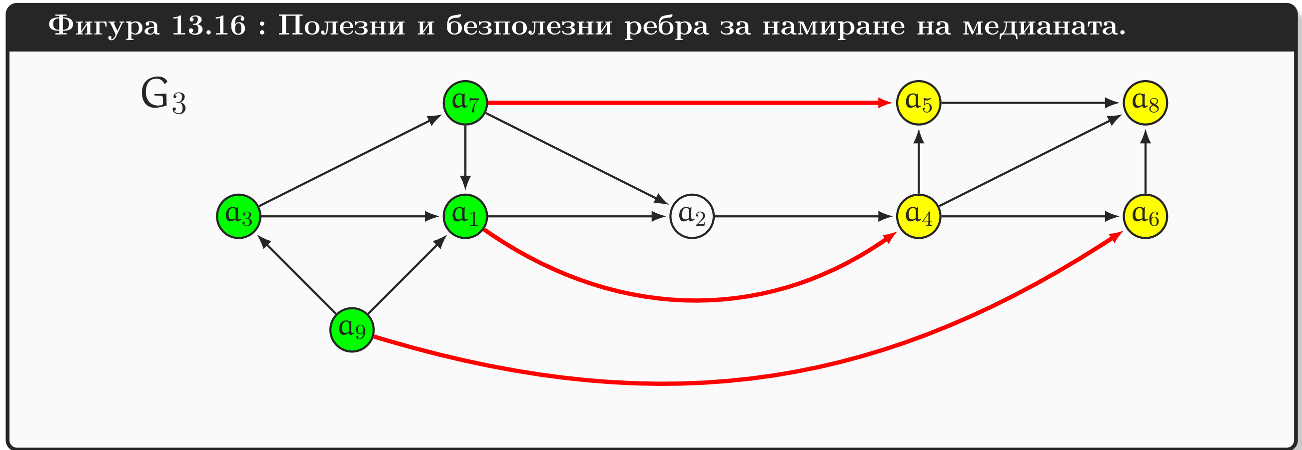
Долната граница за броя на сравненията $n-1$ обаче е твърде слаба. Сега ще я подобрим. Ще опишем стратегия на противника, която води до това, че всеки алгоритъм за намиране на медиана прави поне $\left\lfloor \frac{3n-2}{2} \right\rfloor$ сравнения.

Определение 102: 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 13.14 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 13.16. 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.

Фигура 13.16 : Полезни и безполезни ребра за намиране на медианата.



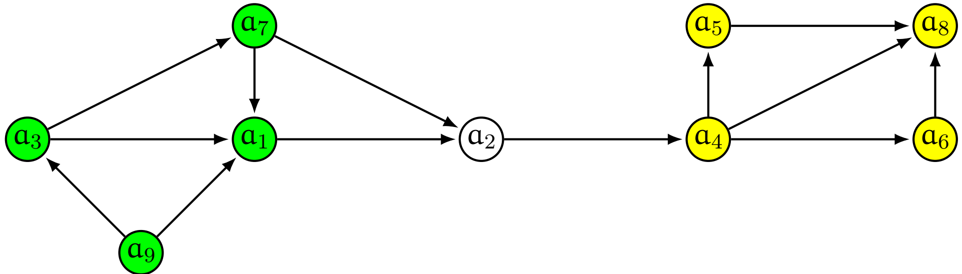
Лесно се вижда, че всяко ребро (u, v) е полезно тстк или $u, v \in S(a_m) \cup \{a_m\}$, или $u, v \in L(a_m) \cup \{a_m\}$; иначе казано, то е безполезно, ако $u \in S(a_m)$ и $v \in L(a_m)$. Образно казано, безполезните ребра са точно тези, които "прескачат над" a_m.

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 63 и си припомнете, че

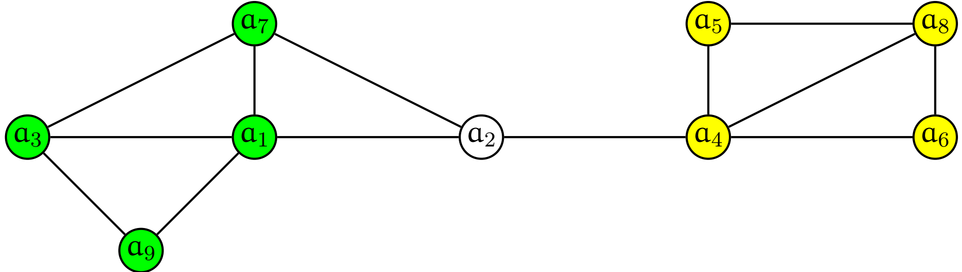
- $S(a_m)$ са върховете, от които a_m е достижим, а това става само чрез пътища, завършващи с a_m; тези нито един от тези пътища не може да съдържа безполезно ребро.
- $L(a_m)$ са върховете, достижими от a_m, а това става само чрез пътища, започващи с a_m; тези нито един от тези пътища не може да съдържа 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 13.14 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 13.16, 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 $a_2 \overset{?}{<} a_4$ 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 пречкат в разсъжденията. Графът от Фигура 13.16 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 ориентираното 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 $C[1..n]$ от числа. Знаем, че n е нечетно; нека $n = 2k+1$ за някое естествено k. Подмасивът $C[1..k]$ е за малките числа, а подмасивът $C[k+2..n]$ е за големите числа. $C[k+1]$ не е нито малко, нито голямо – то е за медианата. В самото начало C 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 C съгласно правило, което ще видим след малко. След приключването на сравненията, C 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 C. Най-лесно ще му бъде, на всяко число от C да даде стойност, равна на позицията му в C. По този начин най-малкото число във входа ще има стойност 1, най-голямото ще има стойност n, а медианата ще има стойност $k+1$.

Да кажем, че медианата е елемент a_m от входа. Ясно е, че малките числа са точно елементите на $S(a_m)$, а големите числа са точно елементите на $L(a_m)$.

Масивът C 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 C, 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:

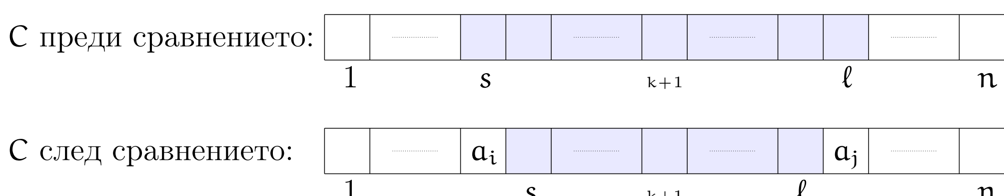
$$s \leftarrow 1$$
$$\ell \leftarrow n$$

Когато направим сравнение $a_i \stackrel{?}{<} a_j$, противникът слага a_i и a_j в C по следното правило.

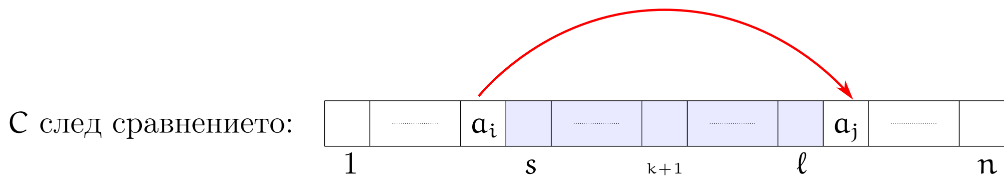
- Ако нито a_i, нито a_j са участвали в сравнение досега, то противникът отговаря Y и прави

  $$C[s] \leftarrow a_i,\ s{+}{+}$$
  $$C[\ell] \leftarrow a_j,\ \ell{-}{-}$$

  Нагледно, ето какво става. Със светлосин фон са маркирани свободните клетки в C.



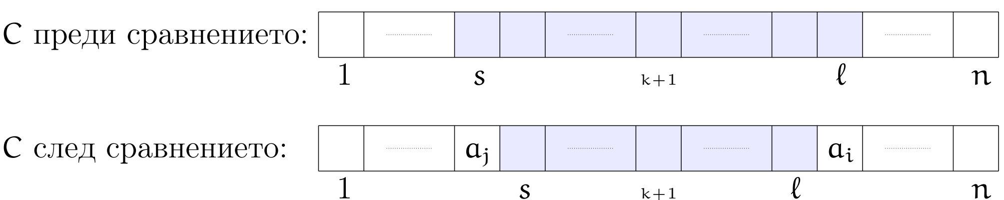
  По този начин сравнението става безполезно – очевидно в дага на сравненията това сравнение дава безполезно ребро, което "прескача над" медианата.



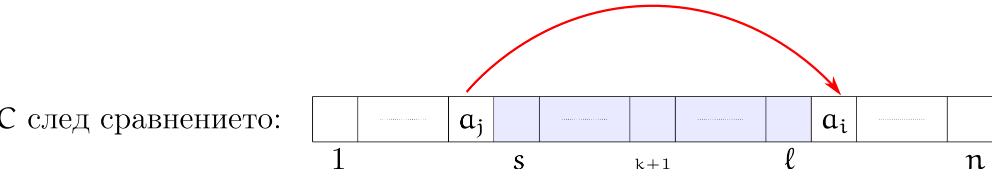
Естествено, решението a_i да стане малко число, а a_j да стане голямо число е произволно. Със същия успех противникът можеше да отговори N и да направи

$$C[s] \leftarrow a_j,\ s{+}{+}$$
$$C[\ell] \leftarrow a_i,\ \ell{-}{-}$$

Нагледно,



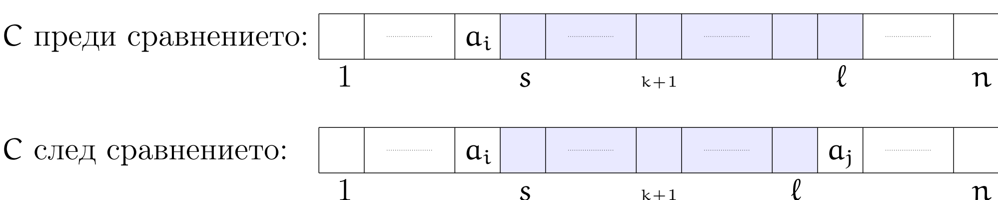
Сравнението пак е безполезно, защото реброто пак е безполезно.



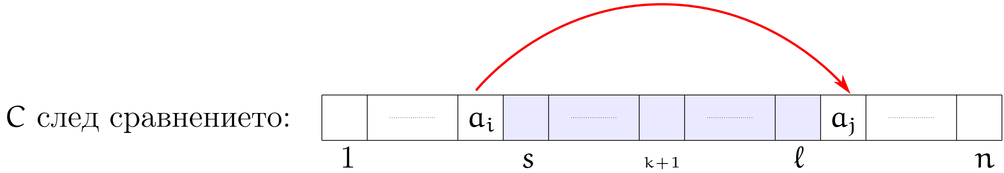
- Нека точно едното от a_i, a_j е участвало в сравнение досега. БОО, нека това е a_i.
  - Ако a_i е малко число, то противникът отговаря Y и прави a_j голямо число така

  $$C[\ell] \leftarrow a_j,\ \ell{-}{-}$$

  Нагледно:



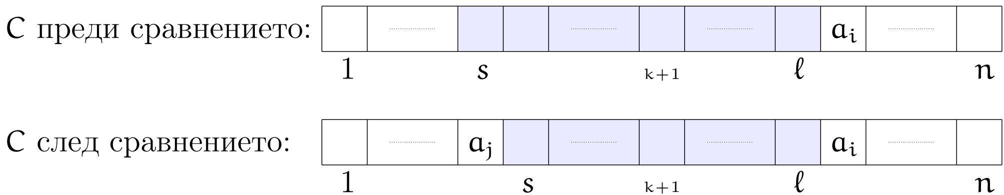
  По този начин сравнението става безполезно.



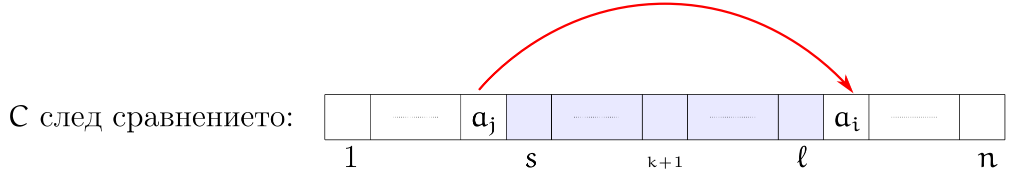
- Ако a_i е голямо число, то противникът отговаря N и прави a_j малко число така

  $$C[s] \leftarrow a_j, \; s{+}{+}$$

  Нагледно:



  По този начин сравнението става безполезно.

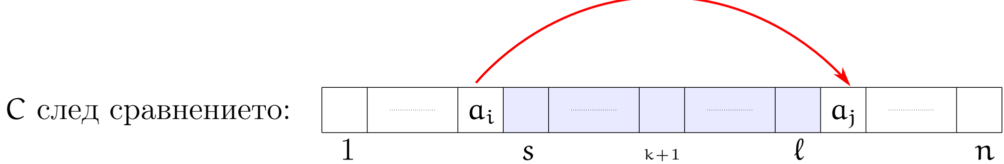


- Остава да видим възможността и a_i, и a_j да са участвали в сравнения досега[†]. Тогава противникът казва истината според съдържанието на C, без да променя нищо в C.

  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 C. Алгоритимът няма представа какви са статусите на a_i и a_j според противника.

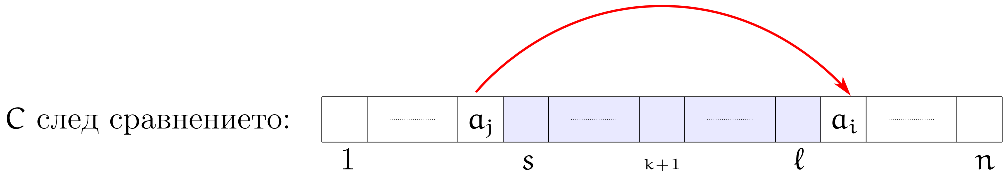
  - Да допуснем, че a_i и a_j имат различен статус.

    Ако a_i е малко, а a_j е голямо, противникът връща Y. Сравнението $a_i \stackrel{?}{<} a_j$ е очевидно безполезно.



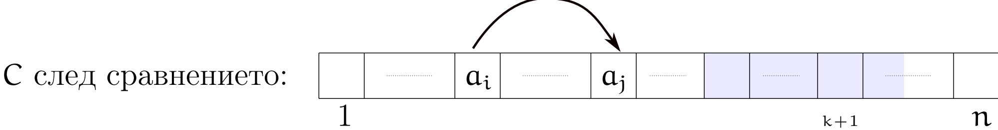
    Ако a_i е голямо, а a_j е малко, противникът връща N. Сравнението $a_i \stackrel{?}{<} a_j$ е очевидно безполезно.

[†]Естествено, те не са били сравнявани помежду си; алгоритъмът не е глупав и не би сравнил същите елементи от входа повторно.

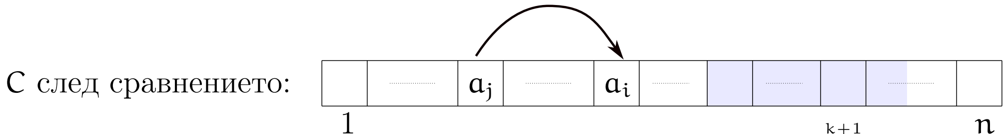


- Да допуснем, че a_i и a_j имат еднакъв статус. БОО, нека и двата са малки. Сега противникът казва истината съгласно взаимното им разположение в C.

  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.



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 ѝ, 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 $\left\lfloor \frac{n}{2} \right\rfloor$ от сравненията да се окажат безполезни, и алгоритъмът е безсилен да предотврати това. Алгоритъмът може да минимизира броя на безполезните сравнения, ако $\left\lfloor \frac{n}{2} \right\rfloor$ пъти сравнява числа, които досега не са били сравнявани. По този начин $\left\lfloor \frac{n}{2} \right\rfloor$ 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 C 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 $\left\lfloor \frac{n}{2} \right\rfloor$ безполезни сравнения; тоест, в дага на сравненията да има поне $\left\lfloor \frac{n}{2} \right\rfloor$ 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 27).

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.

13.4.7 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 двуелементни 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 противното: 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.

13.4.8 Долна граница $2n - 1$ за търсене в сортиран по редове и колони, $n \times n$ масив

Дадена е двумерен $n \times n$ масив M от числа. M е сортиран по редове и колони; тоест,

$$\forall i \in \{1, \ldots, n\} \; \forall j \in \{1, \ldots, n-1\} : M[i, j] \leqslant M[i, j+1]$$
$$\forall j \in \{1, \ldots, n\} \; \forall i \in \{1, \ldots, n-1\} : M[i, j] \leqslant M[i+1, j]$$

Дадено е и число key. Пита се дали key се среща в M.

Ето пример за такъв масив 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

Също като при задачата Търсене на стр. 239, 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 примерния масив 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 $i \in \{1,\ldots,n\}$. 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 подмасива. В два от тях, маркирани с червен фон, числото 52 не може да се среща; в другите два, маркирани със зелен фон, числото 52 може да се среща.

	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 подмасива. Спирачката на рекурсията е главен диагонал с дължина единица.

Тази идея не е съвсем тривиална за реализиране. В примера, който току-що видяхме, подмасивите, върху които ще се изпълнява рекурсивно алгоритъма, са с размери 3×3; тоест, това са пак квадратни масиви, но всеки със страна $n/2$. Но може n да е не е четно, а може и точката, която дели главния диагонал, да не го дели наполовина, при което подмасивите, в които търсенето продължава, са далече от квадратни. Примерно, нека числото, което търсим, е пак 52, но в този масив:

	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 подмасиви, 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 оказа точно в средата му, рекурентното уравнение, което описва сложността по време в най-лошия случай, е

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n \tag{13.8}$$

Това е само ако на **всяко** ниво на рекурсията точката-делител на главния диагонал се пада по средата му. Събираемото $\lg n$ отразява работата на двоичното търсене върху главния диагонал, а $2T\left(\frac{n}{2}\right)$ отразява работата на двете рекурсивни викания, всяко върху масив 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 $M[n, 1] < \text{key}$, то със сигурност $M[n-1, 1] < \text{key}, \ldots, M[1, 1] < \text{key}$.
2. Ако $M[n, 1] > \text{key}$, то със сигурност $M[n, 2] > \text{key}, \ldots, M[n, n] > \text{key}$.

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 масива, като поначало $p = n$ и $q = 1$. Ако $M[p, q] < \text{key}$, правим "ход надясно" $q{++}$. Ако $M[p, q] > \text{key}$, правим "ход нагоре" $p{-}{-}$. И правим това, докато не излезем извън масива; тоест, докато $p \geqslant 1$ и $q \leqslant n$. Когато излезем извън масива, извеждаме съобщение, че key не се среща в M.

Разбира се, в общия случай key може да се среща, така че на всяка стъпка първо проверяваме дали $M[p, q] = \text{key}$. Псевдокодът е следният.

```
Partially-ordered Matrix Search(M[1,...,n][1,...,n]: масив от числа, key: число)
1  p ← n, q ← 1
2  while p ⩾ 1 and q ⩽ n do
3      if M[p, q] = key
4          return (p, q)
5      else if M[p, q] < key
6          q++
7      else
8          p--
9  return −1
```

Приемаме, че коректността е очевидна. Тъй като "придвижването" става, в общия случай, по нещо като начупена линия, която започва от долу-ляво и върви само нагоре и надясно, максималният брой клетки, които ще проверим—което е същото като максималният брой изпълнения на тялото на **while**-цикъла—е $2n - 1$. Следователно, сложността по време на този алгоритъм е $\Theta(n)$ в най-лошия случай. 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 която 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:

	1	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 $M[n, 1], \ldots, M[n, n]$ или върху дясната страна $M[1, 1], \ldots, M[1, n]$. 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" наместо "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.

В тази задача символът "n" означаваше не големината на входа, а корен-квадратен от големината на входа. Забележете, че входът е с големина n^2 (плюс още 1 заради key), което е броят на елементите на масива. Поради това, алгоритъмът, който разгледахме, всъщност не е линеен, а със 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.

Конвенция 14: Големина на входа и 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, сложността да се изразява чрез големината на входа.

13.4.9 Задачата за яйцето

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 допускаме, 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. Иска 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 12.9.1. 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 $\forall (u, v) \in E : (u \in U$ 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 12.9.1 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.

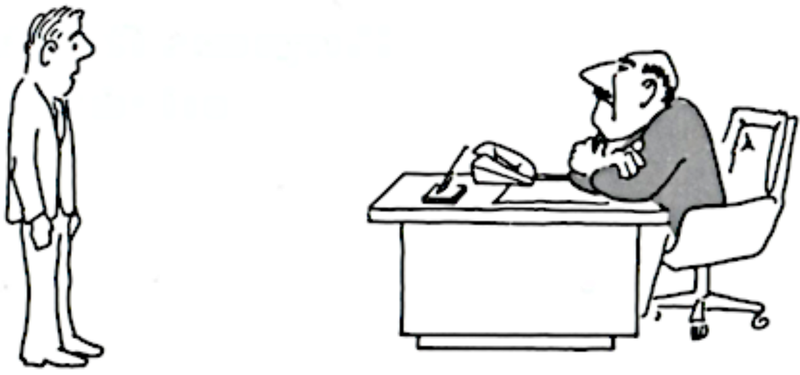
Колкото и да е странно, за тази просто звучаща, напълно естествена и очевидно полезна на практика задача най-ефикасният известен алгоритъм е експоненциален. Нещо повече: тази задача е твърде близка до хиляди други задачи, за всяка от които не е известен по-бърз от експоненциален алгоритъм. Нещо повече:

- ако за поне една от тези задачи се докаже някаква суперполиномиална долна граница, от това ще следва, че всяка от тях е нерешима в полиномиално време,
- а ако за една от тях се намери полиномиален алгоритъм, то за всяка от тях има полиномиален алгоритъм.

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 намиране на ефикасен алгоритъм, или доказване на суперполиномиална долна граница—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 [51]. 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 приповдигнат 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 като най-лоши случаи.

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, "алгоритъм X е ефикасен", ако сложността по време на X е $O(n^k)$, за някаква положителна константа k.

Определение 103: Неподатлива задача

Казваме, че изчислителна задача Π е *неподатлива*, ако за нея не съществува ефикасен алгоритъм.

Според Определение 103, 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. 83:

> *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 103 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 алгоритмичанта 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 [51, 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 $\{1, \ldots, n\}$ 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—решението е малко в размера на (кодирането на) екземпляра. За Hamiltonian Cycle, решението е с размер единица, защото се състои от един единствен бит; задачата е задача за разпознаване.

  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 "лошотията" идва от грамадния размер на решението. Ако задачата е "Генерирай всички подмножества на множество" или "Генерирай всички пермутации на $\{1, \ldots, n\}$ по дадено n", неизвестност поначало няма. Знае се колко са подмножествата. Знае се колко са пермутациите. Ако искаме, можем да дефинираме наредба (нарича се *ранкиране*, на английски *ranking*, за подробности вижте [124]) на подмножествата и пермутациите и много бързо да изчислим кои точно елементи се намират в i-тото подмножество, за $1 \leqslant i \leqslant 2^n$, или кое е j-тото число в i-тата пермутация, за $1 \leqslant i \leqslant n!$ и $1 \leqslant j \leqslant n$. С други думи, не се налага да имаме решението, за да научим тези неща, понеже те могат да се изчислят от общи съображения. И така, в тези задачи няма никаква концептуална неизвестност за решението; ние знаем всичко за него, дори да не разполагаме с него.

  Задачата "Генерирай всички покриващи дървета на даден граф" не е точно такава. Ако графът е пълният граф, примерно, тя също е задача с гигантско решение, за което неизвестност няма. Ако графът е произволен, предварителна неизвестност има, но като цяло все пак я слагаме във втората категория заради гигантския размер на решението в най-лошия случай.

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 64: 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 [53] 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. [140].
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 [33] 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 [139].

В Западния свят, Hartmanis и Stearns [61] 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 [28], 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 [37] разглежда редукции между задачи, които са неподатливи, но естествени и полезни на практика: VERTEX COVER и INDEPENDENT SET към SET COVER.

Meyer и Stockmeyer [104] 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 104

Нека Σ е крайна азбука. Тогава

$$\mathsf{RSQ}(\Sigma) = \{r \mid r \text{ е регулярен израз с повдигане на квадрат над } \Sigma \text{, такъв че } L(r) \neq \Sigma^*\}$$

Теорема 75: Експоненциална долна граница за $\mathsf{RSQ}(\Sigma)$ (Theorem 2.1, [104])

Съществува крайна азбука Σ, такава че, ако M е произволна машина на Turing, разпознаваща $\mathsf{RSQ}(\Sigma)$, 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 75 не касае по никакъв начин **NP**-пълнотата. Теоремата казва, че задачата $\mathsf{RSQ}(\Sigma)$ е пълна за класа на сложност **EXPSPACE**, като **NP** $\subset$ **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 [30]. Той акцентира ключовото значение на полиномиалните редукции, които "навързват" 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 **N**ondeterministically **P**olynomial 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 [81] 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, 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 [139], 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 Karp в САЩ през 1972 г. Чак през 1973 г. Левин публикува своята твърде лаконична статия [98], в която показва, че шест задачи, измежду които е и SAT, са универсални за множеството задачи, което днес наричаме **NP**. По мое мнение, статията на Левин [98] 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 85 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 Π, множеството от екземплярите ѝ се разбива на множеството от ДА-екземплярите Π_Y и множеството от НЕ-екземплярите Π_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. Биекцията 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 изполваме 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 Π и нейно кодиране $\mathcal{E} : \mathcal{I} \rightarrow \Sigma^*$, където

- "$\mathcal{E}$" идва от "encoding",
- $\mathcal{E}$ е биекция,
- $\mathcal{I}$ е множеството от екземплярите на Π,
- Σ е крайната азбука на кодирането, като най-често $\Sigma = \{0, 1\}$.

Наблюдение 76

В текущия контекст, Σ^* се разбива на следните три множества:

$$\{x \in \Sigma^* \mid x \text{ кодира ДА-екземпляр на } \Pi \text{ в } \mathcal{E}\}$$
$$\{x \in \Sigma^* \mid x \text{ кодира НЕ-екземпляр на } \Pi \text{ в } \mathcal{E}\}$$
$$\{x \in \Sigma^* \mid x \text{ не е валидно кодиране на екземпляр на } \Pi \text{ в } \mathcal{E}\}$$

Първото от тези множества е от особен интерес. В някакъв смисъл, ще идентифицираме Π с него.

Определение 105: Език, съответен на задача за разпознаване

Нека Π е задача за разпознаване и $\mathcal{E}$ е кодиране за нея с азбука Σ. Тогава *езикът, съответен на* Π *и* $\mathcal{E}$, е

$$L(\Pi, \mathcal{E}) = \{x \in \Sigma^* \mid x \text{ кодира ДА-екземпляр на } \Pi \text{ в } \mathcal{E}\}$$

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 54: ПАЛИНДРОМИ

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 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 тегави 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 биективно, 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 тегави 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 кодирания 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 един 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 кодирания 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 [51], разумните кодирания удовлетворяват две изисквания (без това да е дефиниция на "разумно кодиране").

- Първо, всеки две разумни кодирания са *полиномиално свързани*, на английски *polynomially related*. Ако Π е задача за разпознаване и $\mathcal{E}_1$ и $\mathcal{E}_2$ са нейни кодирания, 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 $\boldsymbol{x}$ на Π, ако $\boldsymbol{\sigma}_1$ е кодирането на $\boldsymbol{x}$ чрез $\mathcal{E}_1$, а $\boldsymbol{\sigma}_2$ е кодирането на $\boldsymbol{x}$ чрез $\mathcal{E}_2$, то $|\boldsymbol{\sigma}_1| \leqslant p_1(|\boldsymbol{\sigma}_2|)$ и $|\boldsymbol{\sigma}_2| \leqslant p_2(|\boldsymbol{\sigma}_1|)$. На прост български, разумните кодирания 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 $\boldsymbol{\sigma}$, който $\mathcal{E}$ дава, да може в полиномиално време да се реконструира обектът, който $\boldsymbol{\sigma}$ кодира. Това предотвратява кодирания, 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, което не е ефикасно декодируемо. Да разгледаме задачата СОРТИРАНЕ НА ЧИСЛА. Азбуката е $\{0, 1\}$. Числата се кодират по следния начин. За всеки кодиращ стринг $\boldsymbol{\sigma}$:

  - ако $\boldsymbol{\sigma} = 1\boldsymbol{\alpha}$, то числото, което $\boldsymbol{\sigma}$ кодира, е числото, което се записва с $\boldsymbol{\alpha}$ в двоична позиционна бройна система,
  - ако $\boldsymbol{\sigma} = 0\boldsymbol{\alpha}$, нека k е числото, което се записва с $\boldsymbol{\alpha}$ в двоична позиционна бройна система. Тогава числото, което $\boldsymbol{\sigma}$ кодира, е броят на единиците, които принтира машината на Turing BB-k. Тук BB-k е машина на Turing с точно k състояния, която принтира максимален брой (непрекъснати) единици и спира, измежду всички

машини на Turing с точно k състояния, които принтират само единици и спират. За да разберете защо това е неефикасно, вижте задачата 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 [140] от 1936 г., но широко приетите днес дефиниции се несъществено различни. Тук ще ползваме дефиницията от [51].

Определение 106: Детерминирана машина на Turing (МТ)

Детерминирана машина на Turing, накратко *машина на Turing* или МТ, 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 "␣", тоест шпация, е в $\Gamma \backslash \Sigma$,
- 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*,
- $q_N \in Q$, $q_N \neq q_0, q_Y$ е *отхвърлящото състояние*,
- $\delta : (Q \backslash \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow, \downarrow\}$ 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 $\ldots, -1, 0, 1, \ldots$, но машината не "вижда" тези номера и те са само за наше удобство.

Да въведем малко нотации. Обикновено бележим стрингове с "$\mathbf{x}$" или "$\boldsymbol{\sigma}$". 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, $\ldots$, x_n в клетка n, като $n = |\mathbf{x}|$. Всички останали клетки съдържат шпации ␣. Забележете, че ␣ $\notin \Sigma$, 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 ␣, 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 оставя 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 107: 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: $(\!| q, y, i |\!)$.
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 непрекъсната 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 само от входа и от функцията на преходите.

Въпреки че Turing използва безкрайна лента за своята машина в оригиналната статия [140], можем спокойно да минем без понятията безкрайна лента и шпации (съдържанието на клетките извън "съдържанието на лентата"). Вместо това, можем да дефинираме само *текущ*

стринг, който в началото е точно входът $\mathbf{x}$. Този текущ стринг може обаче да нараства неограничено – при всеки опит на машината да "излезе" от него чрез ↢ вляво от началото му или ↣ вдясно от края му, към него (текущия стринг) се присъединява, вляво или вдясно, буквата '␣' като нова буква, която машината прочита. Ако възприемем това виждане, то съдържанието на лентата е точно текущият стринг и толкова.

Нека машината е наречена M. От общи съображения е ясно, че за даден вход $\mathbf{x}$

- или M спира в $\mathsf{q_Y}$, което бележим с "$\mathsf{M}(\mathbf{x}) = \mathsf{q_Y}$",
- или M спира в $\mathsf{q_N}$, което бележим с "$\mathsf{M}(\mathbf{x}) = \mathsf{q_N}$",
- или M не спира никога, което бележим с "$\mathsf{M}(\mathbf{x}) = \nearrow$". 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 $\mathsf{q_0}$. От него с която и да е буква оставаме в него, местейки главата надясно и записвайки каквото и да е. В такъв случай $\mathsf{q_Y}$ и $\mathsf{q_N}$, 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 108: Езикът, разпознаван от машина на Turing

Нека M е МТ с входна азбука Σ. *Езикът, разпознаван от* M, е

$$\{\mathbf{x} \in \Sigma^* \mid \mathsf{M}(\mathbf{x}) = \mathsf{q_Y}\}$$

Този език означаваме с $\mathsf{L_M}$.

Забележете, че не се иска машината да спира върху всички входове; тоест, да достига състояние $\mathsf{q_N}$ за всеки $\mathbf{x} \in \Sigma^* \backslash \mathsf{L_M}$. Ако $\mathbf{x}$ не е в езика, M може да го отхвърли чрез $\mathsf{q_N}$, а може и да не спре никога. Но ако искаме M да съответства на алгоритъм, M трябва да спира върху всеки вход.

Определение 109: 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 стрингове от входната азбука и $\mathsf{L_M} = \mathsf{L}(\Pi, \mathcal{E})$. Ако кодирането се подразбира, това става просто $\mathsf{L_M} = \mathsf{L}(\Pi)$.

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	$\langle r_0, \sqcup, \rightarrowtail \rangle$	$\langle r_1, \sqcup, \rightarrowtail \rangle$	$\langle q_Y, \sqcup, \downarrow \rangle$
r_0	$\langle w_0, 0, \rightarrowtail \rangle$	$\langle w_0, 1, \rightarrowtail \rangle$	$\langle q_Y, \sqcup, \downarrow \rangle$
w_0	$\langle w_0, 0, \rightarrowtail \rangle$	$\langle w_0, 1, \rightarrowtail \rangle$	$\langle s_0, \sqcup, \leftarrowtail \rangle$
s_0	$\langle t, \sqcup, \leftarrowtail \rangle$	$\langle q_N, \sqcup, \downarrow \rangle$	$\langle s_0, \sqcup, \downarrow \rangle$
t	$\langle t, 0, \leftarrowtail \rangle$	$\langle t, 1, \leftarrowtail \rangle$	$\langle q_0, \sqcup, \rightarrowtail \rangle$
r_1	$\langle w_1, 0, \rightarrowtail \rangle$	$\langle w_1, 1, \rightarrowtail \rangle$	$\langle q_Y, \sqcup, \leftarrowtail \rangle$
w_1	$\langle w_1, 0, \rightarrowtail \rangle$	$\langle w_1, 1, \rightarrowtail \rangle$	$\langle s_1, \sqcup, \leftarrowtail \rangle$
s_1	$\langle q_N, \sqcup, \downarrow \rangle$	$\langle t, \sqcup, \leftarrowtail \rangle$	$\langle s_1, \sqcup, \downarrow \rangle$

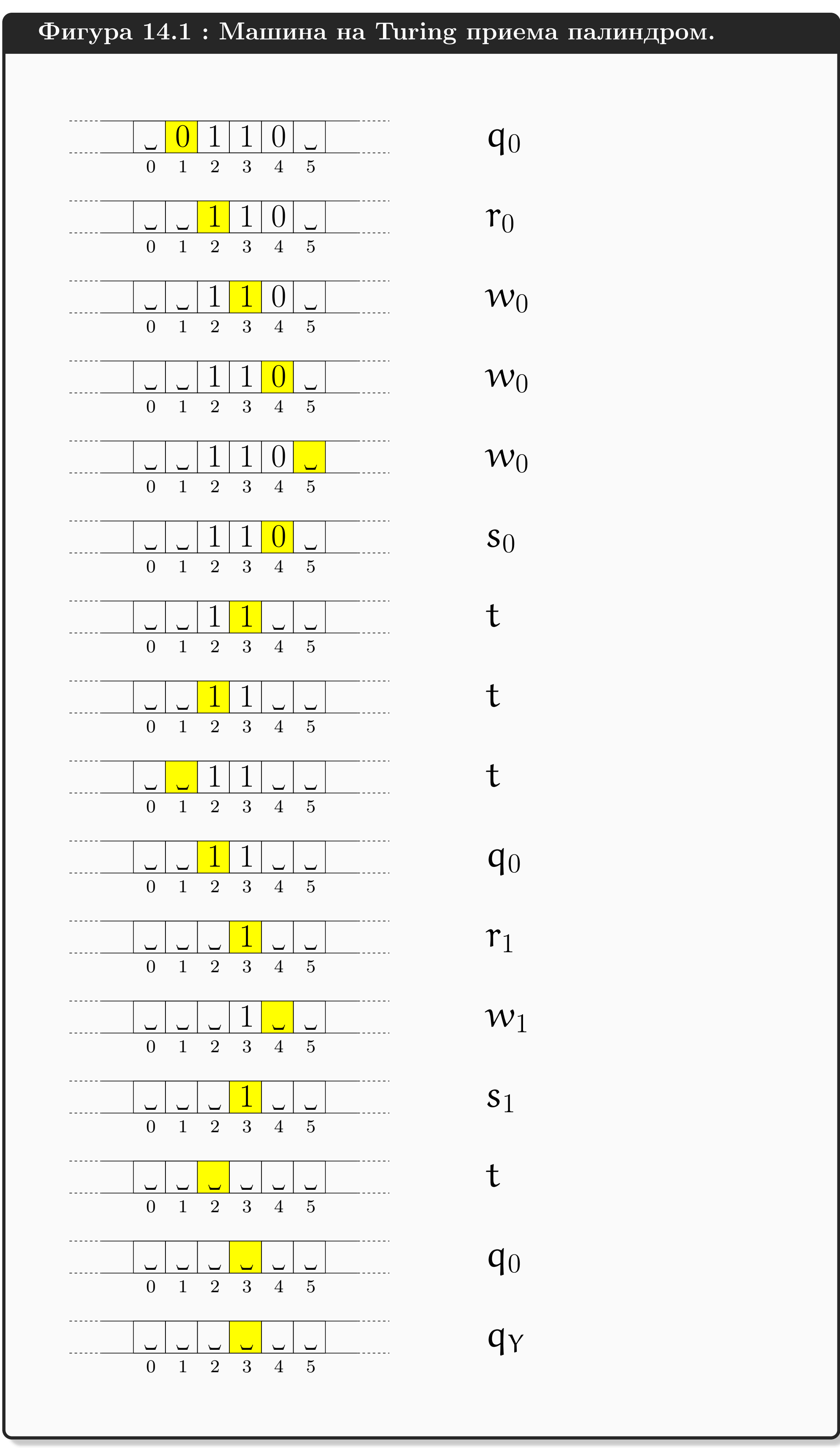
Таблица 14.1: Машина на Turing M_{pal}, която решава задачата Палиндроми.

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 76: Машината на Turing M_{pal} решава задачата Палиндроми върху $\{0, 1\}$.

За всеки стринг $x \in \{0, 1\}^*$, ако x е палиндром и главата на M_{pal} е върху първата буква на x[a], то M_{pal} с вход x приема, а ако x не е палиндром, то M_{pal} с вход x отхвърля.

[a]Както вече се разбрахме, ако $x = \epsilon$, то първата буква на x е $\textvisiblespace$.

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 110: Палиндром, индуктивна дефиниция

По отношение на азбуката $\{0, 1\}$,

- празният стринг ϵ е палиндром,
- стрингът 0 е палиндром и стрингът 1 е палиндром,
- ако x е палиндром, то $0x0$ е палиндром и $1x1$ е палиндром.
- нищо друго не е палиндром.

База: Базата на доказателството касае базовите случаи в Определение 110. Ако $x = \epsilon$, то наистина машината приема, понеже с $\delta(q_0, \textvisiblespace) = \langle q_Y, \textvisiblespace, \updownarrow \rangle$. Ако $x = 0$, то машината от q_0 отива в r_0 и после в q_Y. Ако $x = 1$, то машината от q_0 отива в r_1 и после в q_Y.

Индуктивно предположение: Да допуснем, че машината работи коректно върху всички стрингове с дължина не по-голяма от $n - 1$.

Индуктивна стъпка: Да разгледаме работата на M_{pal} върху стринг x с дължина $n \geqslant 2$. Машината запомня каква е буквата в началото чрез състоянията r_0 или r_1, но заменя началната буква с $\textvisiblespace$. 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 по очевидни причини. 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 остатъка 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 $n - 2$, така че индуктивното предположение е в сила и съгласно него машината коректно решава дали стрингът е палиндром. □

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.

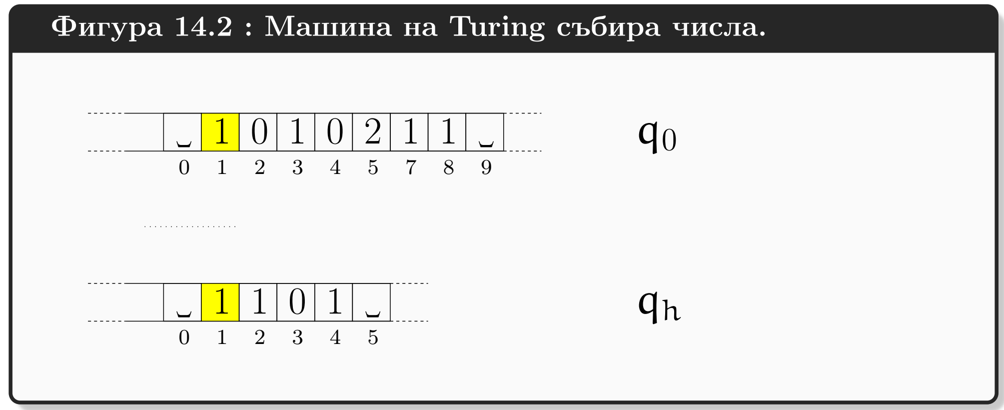
Сега ще видим, че с несъществени промени в дефинициите, машините на Turing могат да изчисляват функции, чиито кодомейн е **безкраен**; а именно, безкрайно множество от стрингове. Нека машината е M. Заменяме двете спиращи състояния q_Y и q_N със само едно спиращо състояние q_h. При влизането в q_h, M 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 $f_M : \Sigma^* \to (\Gamma \backslash \{␣\})^*$. 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 МТ съгласно Определение 106 използваме термина *машина-разпознавач* или само *разпознавач*; става дума за разпознавач на езици. Тъй можем да кодираме всеки мислим обект със стринг от $(\Gamma \backslash \{␣\})^*$, 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 $\Gamma = \{0, 1, 2, ␣\}$.

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.

Фигура 14.2 : Машина на Turing събира числа.



Забележете, че Определение 108 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

[†] В зависимост от това дали има входове, върху които не спира, или спира върху всеки вход.

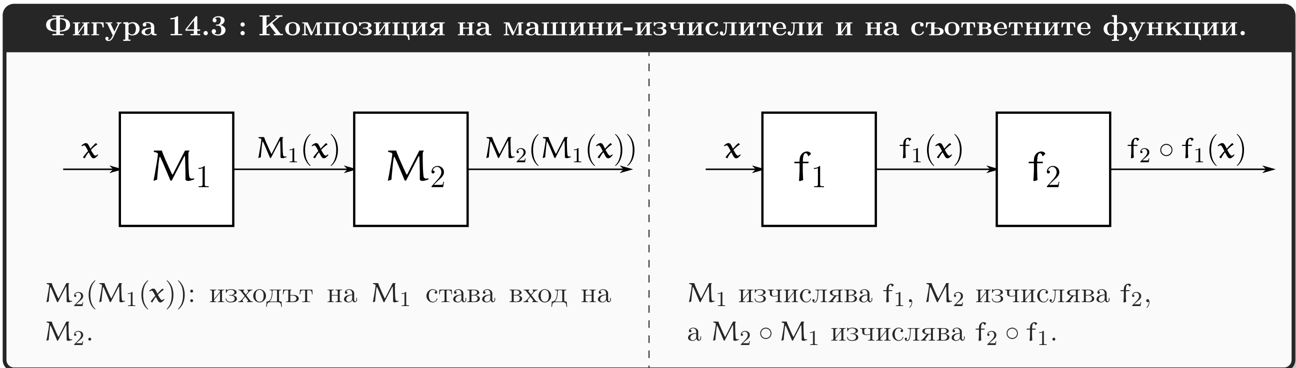
[‡] Всъщност, безброй много клетки.

чрез q_Y/q_N, или ако спира чрез q_h, множеството от изходите 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 $\Sigma \cup \{\textvisiblespace\}$, *композицията на* M_2 *след* M_1 *върху вход* $\mathbf{x}$ е нова машина, която първо пуска (или симулира, ако предпочитате) M_1 с вход x. Когато M_1 спре, тази нова машина веднага пуска 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 $M_1(\mathbf{x})$ бележи изхода на M_1 с вход $\mathbf{x}$, то $M_2(M_1(\mathbf{x}))$ бележи изхода на композицията на M_2 след M_1 върху вход $\mathbf{x}$. Само композицията, 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.



$M_2(M_1(\mathbf{x}))$: 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 111 е напълно аналогична на T_A от Определение 15 за сложността по време на алгоритми. Разликата е в това, че големина на входа на алгоритъм се дефинира с доста условност, докато при машините на Turing е кристално ясно какво е големина на входа.

Определение 111: 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 $T_M(n) : \mathbb{N}^+ \to \mathbb{N}^+$:

$$T_M(n) = \max_{x \in \Sigma^n} \{k \mid M(x) \text{ спира след точно } k \text{ стъпки }\}$$

M се нарича *полиномиална машина на Turing*, ако съществува полином $p(n)$, такъв че $\forall n \in \mathbb{N}^+ : T_M(n) \leqslant p(n)$.

Алтернативно, M е полиномиална машина на Turing, ако съществува константа $c \geqslant 0$, такава че $\forall n \in \mathbb{N}^+ : T_M(n) = O(n^c)$.

Забележете, че машината M може да е разпозвавач или изчислител. Това е без значение за Определение 111. Важното е за колко стъпки спира, а дали разпознава език или изчислява функция няма значение.

Определение 112: 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 113: Клас на сложност P

Класът на сложност **P**, по отношение на дадена азбука Σ, е

$$\mathbf{P} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \text{съществува полиномиална машина на Turing } M, \text{ такава че } L = L_M\}$$

В Определение 113, машината трябва да е разпознавач.

Задача за разпознаване Π се намира в **P**, ако съществува разумно кодиране $\mathcal{E}$ на Π, такова че $L(\Pi, \mathcal{E}) \in \mathbf{P}$, което означава да има полиномиална МТ M, която решава Π, 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 $L(\Pi, \mathcal{E}) = L_M$.

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.

[†]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 [114, Problem 2.8.5, стр. 52].

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 [114].

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 входа 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).

14.5 Недетерминизъм и недетерминирани машини на Turing

Докато обикновените (детерминирани) машини на Turing са физически реализируеми, изключвайки безкрайната лента, сега ще въведем способност на машините, наречена “недетерминизъм” която няма никакъв аналог в реалния свят. Недетерминираните машини на Turing са с чисто теоретично значение. Има два еквивалентни начина да се дефинира недетерминирата 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 106 с единствената разлика, че сега δ е релация на преходите:

$$\delta \subseteq \big((Q\backslash\{q_Y, q_N\}) \times \Gamma\big) \times \big(Q \times \Gamma \times \{\leftarrow, \rightarrow, \updownarrow\}\big)$$

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 107.

Как да интепретираме възможността машината да еволюира по няколко различни начина едновременно? Какво означава да има няколко нови състояния и изобщо няколко нови конфигурации? Една смислена интепретация 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 114: 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 $\widetilde{M}$ достигне отхвърляща конфигурация, то $\widetilde{M}$ спира, обаче останалите копия продължават, стига да не са в отхвърляща конфигурация или катастрофирали.

© Минко Марков, 2023г.

Наблюдение 77: 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. Недетерминираните машини с размножаване изчисляват дървовидно във времето, което е принципно различно от линейния във времето начин, по който изчисляват детерминираните машини[†].

Определение 115: 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 детерминираните 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 114. 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".

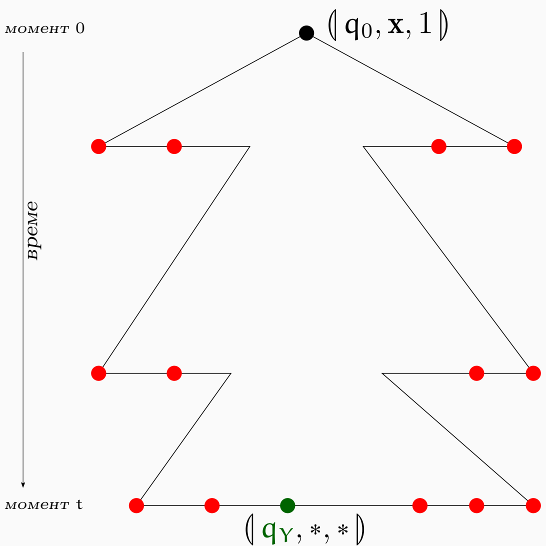
Да разглеждаме конкретен пример за работата на нетривиална НМТ, приемаща някакъв не съвсем тривиален вход и размножаваща се интензивно, би било прекалено. Фигура 14.1 показва пътя на изчислението на детерминирана МТ за сравнително проста задача и само 15 стъпки, и въпреки това е тегава за осмисляне. 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.

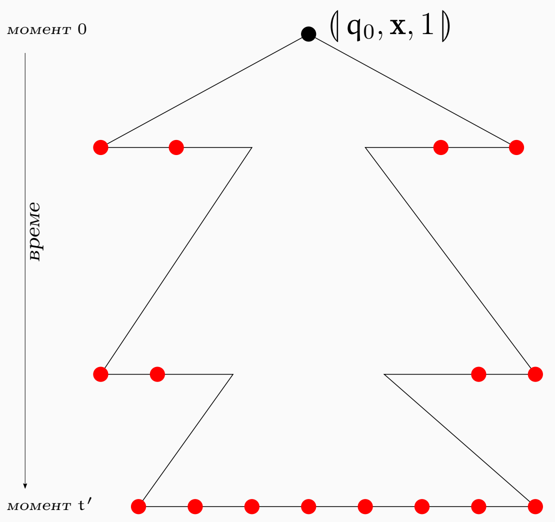
- На подфигурата вдясно, нито едно копие не е достигнало до приемаща конфигурация, а в момента t' всички копия вече или са вече достинали до отхвърлящи конфигурации, или са катастрофирали, и няма нито едно работещо копие. Поради това машината отхвърля входа x.

[†]Както и алгоритмите, които са основния предмет на тези лекционни записки.

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 65: 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 недетерминираните 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 [133, 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* $a = (a_1, a_2, \ldots, a_n)$ ...
>
> *At each step in the backtracking algorithm, we try to extend a given partial solution* $a = (a_1, a_2, \ldots, a_k)$ *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 doind 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[а]. 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[б] 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[в]. 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.

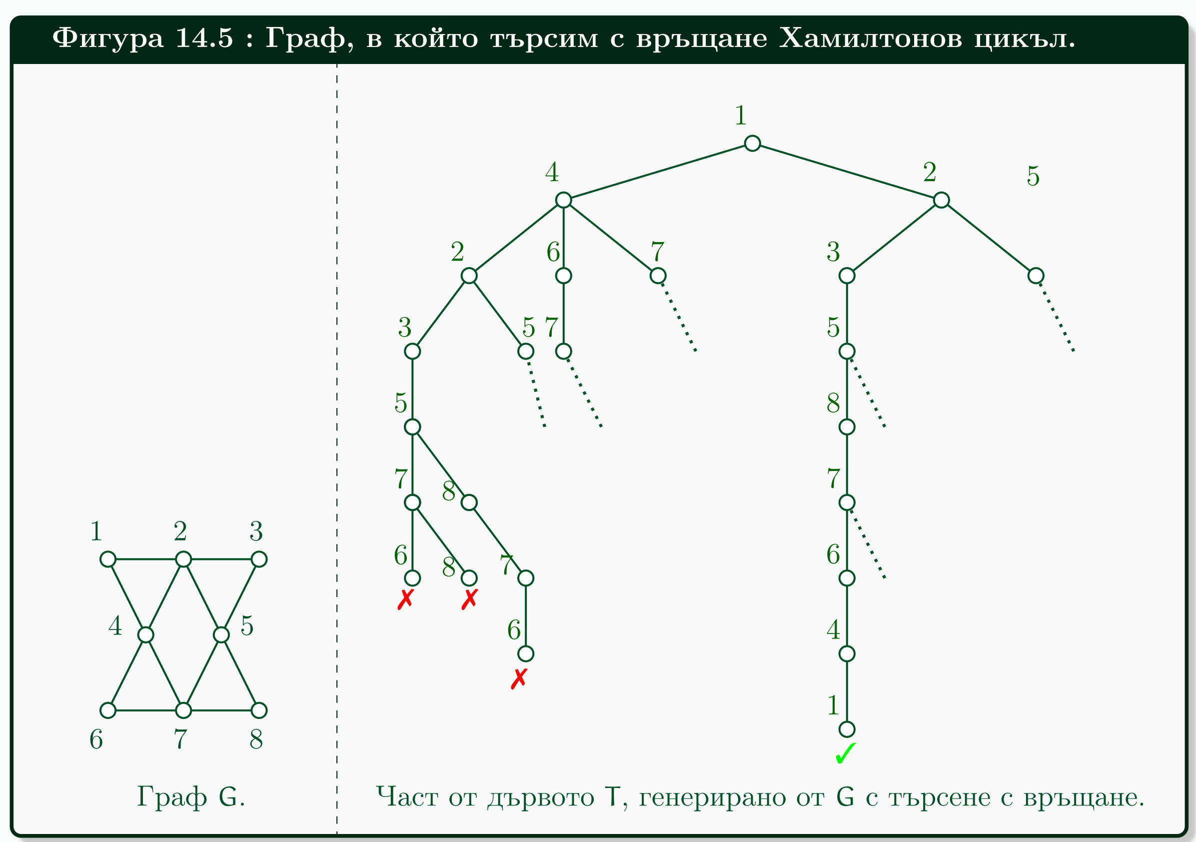
  Да използваме търсене с връщане. Частичното решение (partial solution), за което говори Skiena, е редица без повтаряне от върхове, такъв че за всеки два съседни във редицата върха, в графа има ребро; първият елемент на редицата е винаги u. Такова частично решение отговаря на прост път с един край u в 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 съседи са в частичното решение), се връщаме назад колкото може по-малко и опитваме отново. Съществената 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 съседи 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.

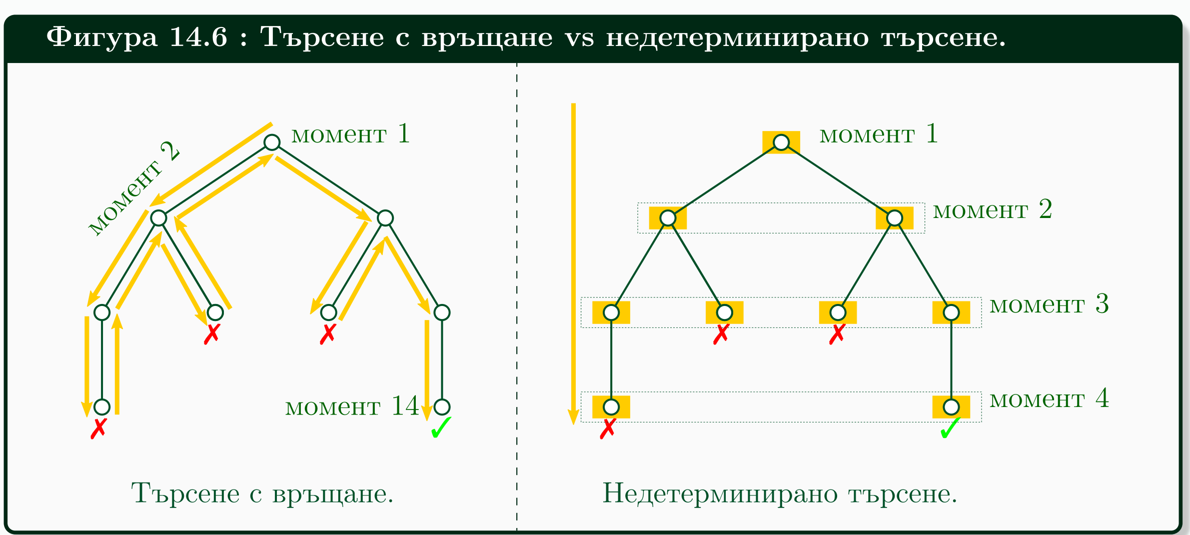
Фигура 14.5 : Граф, в който търсим с връщане Хамилтонов цикъл.



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 детерминираното търсене с връщане, реализиращо DFS в дървото на търсенето, и недетерминираното 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.



Търсене с връщане. 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 ✗, ако дървото е на търсенето, или q_N или катастрофа, ако дървото е на конфигурациите на НМТ) и приемащи листа (маркирани с ✓, ако дървото е на търсенето, или q_Y, ако дървото е на конфигурациите на НМТ).

[а]Споменахме това понятие в Секция 6.1, където въведохме "алгоритмична схема"

[б]**Потенциален** цикъл обаче! Може в графа този клас да не отговаря на цикъл поради липсващи ребра.

[в]И това понятие бе споменато в Секция 6.1.

Езици и задачи за разпознаване при НМТ. Определение 116 буквално повтаря Определение 108.

Определение 116: 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 117 буквално повтаря Определение 109.

Определение 117: Решаване на задача за разпознаване от недетерминирана машина на Turing

Казваме, че НМТ M решава задачата за разпознаване Π по отношение на кодирането $\mathcal{E}$, ако M спира върху всички стрингове от входната азбука и $L_M = L(\Pi, \mathcal{E})$. Ако кодирането се подразбира, това става просто $L_M = L(\Pi)$.

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 ѝ е x и нека x е Да-екземпляр на задачата за разпознаване, която машината решава. Това е важно: разглеждаме Да-екземпляр.

- Във Фаза 1, машината някак, магически, за само една стъпка, "отгатва" стринг $\sigma \in \Sigma^*$, който се нарича *сертификат*. Този стринг се оказва записан, и то само за единица време, в клетки $-1, -2, \ldots, -m$ на лентата, където $m = |\sigma|$. Тъй като входът x е записан от клетка 1 надясно, няма опасност сертификатът да презапише (част от) входа. Терминът "отгатва" е непрецизен, но се използва, за да се подчертае функцията

на $\boldsymbol{\sigma}$. Прецизният термин е "генерира". Машината генерира $\boldsymbol{\sigma}$ за една стъпка и я записва по указания начин.

- Във Фаза 2, машината работи напълно детерминирано. Машината M има не релация на преходите като в Подсекция 14.5.1, а функция на преходите, точно като в Определение 106. Работейки детерминирано, машината достига състояние q_Y, тоест приема входа (което е правилно, защото входът е ДА-екземпляр на задачата). Същественото е, че за детерминираното приемане на входа $\boldsymbol{x}$ машината ползва сертификата $\boldsymbol{\sigma}$. За разлика от това, обикновената детерминирана МТ приема входа без помощта на сертификат.

  Ясно е откъде идва името "сертификат": **$\boldsymbol{\sigma}$ сертифицира**, тоест удостоверява, факта, че $\boldsymbol{x}$ е ДА-екземпляр.

Ако това е неясно, да видим пример. Да разгледаме пак HAMILTONIAN CYCLE и НМТ със сертификат за нея. Входът $\boldsymbol{x}$ е кодиране на граф $G = (V, E)$. Сертификатът $\boldsymbol{\sigma}$ е кодиране на пермутация $\langle v_{i_1}, \ldots, v_{i_n} \rangle$ 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 читателат 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 $\boldsymbol{\sigma}$ на Хамилтонов цикъл във Фаза 1.

Във Фаза 2 машината само проверява, работейки като **детерминирана** машина, че $\boldsymbol{\sigma}$ наистина кодира Хамилтонов цикъл в графа, кодиран от $\boldsymbol{x}$. Проверката е лесна, поне на концептуално ниво: машината проверява, че всеки връх на графа се среща точно веднъж в $\boldsymbol{\sigma}$ (тоест, че $\{v_{i_1}, \ldots, v_{i_n}\} = V$) и че в графа има ребра (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), $\ldots$, $(v_{i_{n-1}}, v_{i_n})$, (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 $\boldsymbol{x}$ кодира ДА-екземпляр. Възниква въпросът, а ако $\boldsymbol{x}$ кодира НЕ-екземпляр? Ако задачата е HAMILTONIAN CYCLE, но входът не е (кодиране на) Хамилтонов граф, как ще бъде отхвърлен от машината? Отговорът е: **никак**. Верификаторите имат смисъл само за ДА-екземпляри, за които има сертификати. Колкото и да е контраинтуитивно, няма да разглеждаме работата на верификатор върху НЕ-екземпляри. Отново се сблъскваме с асиметрия между приемането и отхвърлянето при недетерминизма.

При НМТ с размножаване, асиметрията беше между приемането, което става при поне едно достигане до q_Y, и отхвърлянето, което става с това, че всяко достигане до край на изчислението е достигане до q_N или катастрофа. При НМТ-верификатор, асиметрията е в това, че разглеждаме само как машината **може** да приеме, но изобщо не се интересуваме от това, как машината решава задачата в общия случай, когато се налага и да отхвърля. 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.

Наблюдение 78: Верификаторите не отхвърлят

Когато разглеждаме недетерминирани машини на Turing-верификатори, ги разглеждаме именно като верификатори: разглеждаме работата им по приемането на ДА-екземплярите, игнорирайки НЕ-екземплярите.

Но това, че машина-верификатор не отхвърля, не означава, че може да приема всичко! Задължително верификаторът трябва да е така направен, че да не приема НЕ-екземпляри при никакви сертификати. Ако не съблюдаваме това, 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.

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 77: 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 118). Помним, че M' е нормална детерминирана машина с изключение на това, че може да отгатва сертификати.

Доказателство: Ще разглеждаме само ДА-екземпляри съгласно Наблюдение 78. 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 $b_1, \ldots, b_q$ 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 $\boldsymbol{\sigma}$, за да разбере коя от $b_1, \ldots, b_q$ е правилната конфигурация. "Правилната конфигурация" е тази b_i, която е след a по пътя p. А конфигурациите $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_q$ са без значение за M' и тя ги игнорира. Нейната цел е да достигне w и така да приеме $\mathbf{x}$.

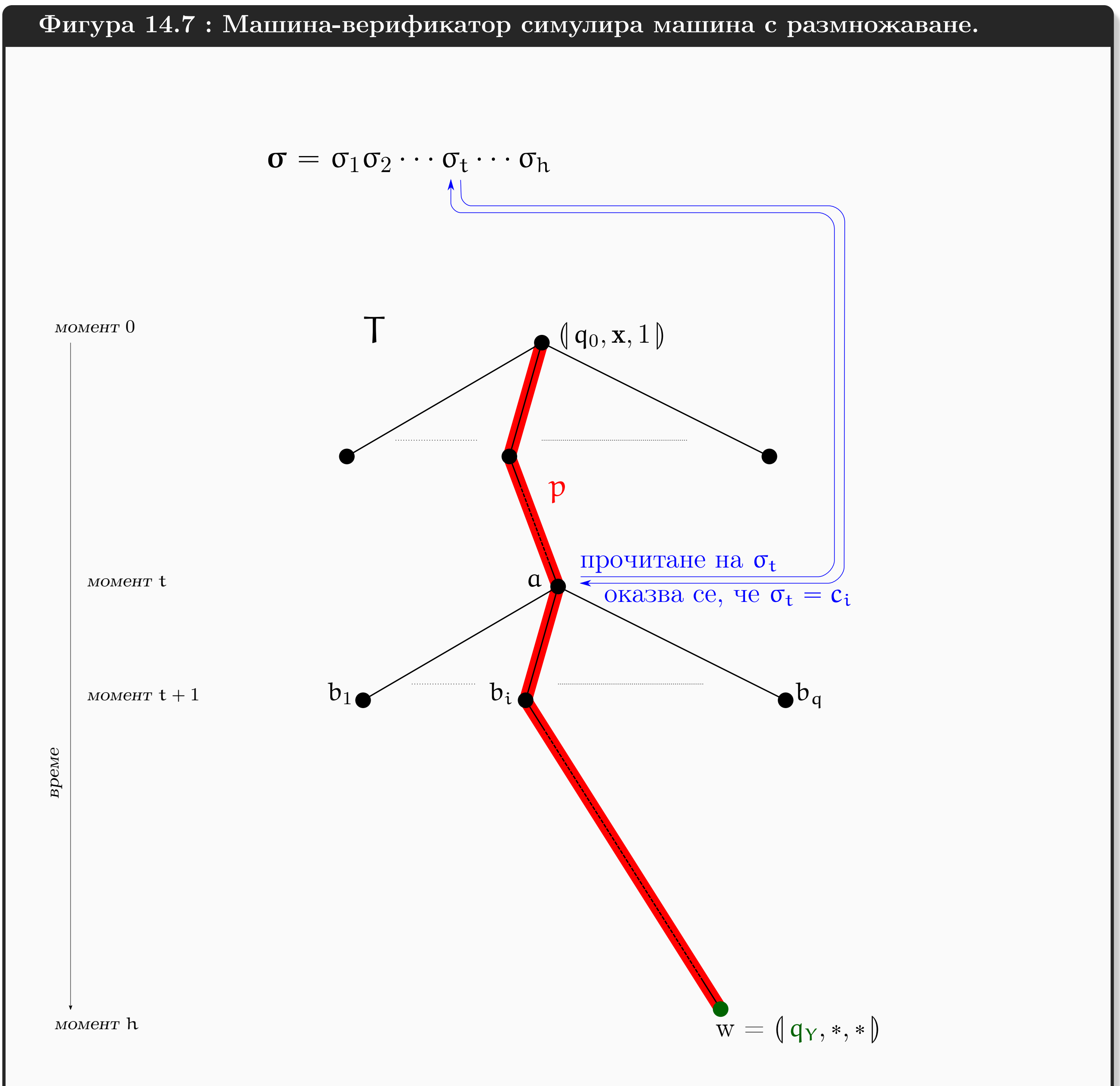
  Какво чете M' от $\boldsymbol{\sigma}$? Нека разклонеността на T е ℓ и $\ell \geqslant 2$. Нека $\widetilde{\Sigma} = \{c_1, \ldots, c_\ell\}$ е подмножество на Σ. Сертификатът $\boldsymbol{\sigma}$ е стринг над $\widetilde{\Sigma}$ с дължина h, като σ_i е номерът на правилната следваща конфигурация на M при прехода от момент j към момент $j+1$, за $1 \leqslant j < h$. Тогава гореспоменатото четене от сертификата е просто прочитане на σ_t. σ_t е някое c_i от $\widetilde{\Sigma}$. 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. □

Фигура 14.7 илюстрира идеята на доказателството на Теорема 77. 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 78: 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 $\boldsymbol{\sigma}$. Всяка буква на $\boldsymbol{\sigma}$ е от крайната азбука $\widetilde{\Sigma} \subset \Sigma$, така че за всяка буква на $\boldsymbol{\sigma}$ има $|\widetilde{\Sigma}|$ възможности, а M' може да симулира всички тези възможности едновременно, размножавайки се в $|\widetilde{\Sigma}|$ нови копия.

Ако M не чете сертификата изобщо или сертификатът е празен, то тя е напълно детерминирана машина, неизползваща 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 $|\widetilde{\Sigma}|$ 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 конфикурациите 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 $|\widetilde{\Sigma}|$ 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. □

14.5.4 Какви нови възможности ни дава недетерминизмът

Като ефективност на изчислението, недетерминизмът не ни дава нищо вповече. 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 118 буквално повтаря Определение 111.

Определение 118: 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 $T_M(n) : \mathbb{N}^+ \to \mathbb{N}^+$:

$$T_M(n) = \max_{x \in \Sigma^n} \{k \,|\, M(x) \text{ спира след точно } k \text{ стъпки }\}$$

M се нарича *полиномиална недетерминирана машина на Turing*, ако съществува полином $p(n)$, такъв че $\forall n \in \mathbb{N}^+ : T_M(n) \leqslant p(n)$.

Определение 119: 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 детерминираните машини, задача за разпознаване Π се намира в **NP**, ако съществува разумно кодиране $\mathcal{E}$ на Π, такова че $L(\Pi, \mathcal{E}) \in \mathbf{NP}$, което означава да има недетерминирана полиномиална машина M, която решава Π, кодирана чрез $\mathcal{E}$.

Следното твърдение е прекалено очевидно, за да бъде наричано “теорема”.

Наблюдение 79: 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 28: $\mathbf{P} \subseteq \mathbf{NP}$

Щом всяка задача от **P** е в **NP**, то $\mathbf{P} \subseteq \mathbf{NP}$.

Най-важният нерешен въпрос в теоретичната Компютърна наука е дали $\mathbf{NP} \subseteq \mathbf{P}$. Иначе казано, $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 66: 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 64, съществува отколешен 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 то е невярно.

Резултатите на Gödel [53] и Turing [140] 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 $\mathfrak{P}$ върху входа $\mathfrak{I}$. Ако в екземпляра на задачата има и естествено число k и въпросът е "Дали $\mathfrak{P}$ с вход $\mathfrak{I}$ терминира за не повече от k стъпки?", задачата няма да е нерешима: просто симулираме $\mathfrak{P}(\mathfrak{I})$ 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 опитаме да генерираме всички стрингове с дължина $\leqslant k$ 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 $\phi(n) \geqslant k{\cdot}n$. 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 $\phi(n) \geqslant k \cdot n$ 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 [33, 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 [140] 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 [53] 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 **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 **NP**. Gödel пита дали можем

бързо да създадем доказателство, което е късо. Въпросът $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, който въпрос е темата на това допълнение. Въпросът, повдигнат от Gödel до von Neumann, в някакъв смисъл е същият като $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. 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 $\mathbf{P} = \mathbf{NP}$, то хората наистина са машини, или поне може да се опишат адекватно алгоритмично, точно както твърдят материалистите. Ако обаче $\mathbf{P} \neq \mathbf{NP}$, то има нещо уникално и неповторимо в човешкия гений, което не може да бъде реплицирано в машина или поне в алгоритъм. И така, $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ 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 $\mathbf{P} \neq \mathbf{NP}$, 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ е статията на Russell Impagliazzo [74] от 1995 г. Той разглежда пет възможности, само най-простата (алгоритмично) от които е $\mathbf{P} = \mathbf{NP}$. Ако $\mathbf{P} \neq \mathbf{NP}$, 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 [51, стр. 34]:

> *If $\mathbf{P}$ differs from $\mathbf{NP}$, ... all problems in $\mathbf{NP} \setminus \mathbf{P}$ are intractable. ...*
>
> *Of course, until we can prove that $\mathbf{P} \neq \mathbf{NP}$, there is no hope of showing that any particular problem belongs to $\mathbf{NP} \setminus \mathbf{P}$. For this reason, the theory of $\mathbf{NP}$-completeness focuses on proving results of the weaker form "if $\mathbf{P} \neq \mathbf{NP}$, then $\Pi \in \mathbf{NP} \setminus \mathbf{P}$". ...*

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 120: Karp редукции между задачи

Нека Π_1 и Π_2 са задачи за разпознаване. *Karp редукция от* Π_1 *в* Π_2 е всяка функция $f : \Pi_1 \to \Pi_2$, такава че

- x е ДА-екземпляр на Π_1 тстк $f(x)$ е ДА-екземпляр на Π_2 и
- f е изчислима в полиномиално време.

f изобразява ДА-екземпляри на Π_1 в ДА-екземпляри на Π_2 и НЕ-екземпляри на Π_1 в НЕ-екземпляри на Π_2, и освен това работи бързо, тоест в полиномиално време. Мислим за f като за полиномиален алгоритъм, чието множеството от входовете е Π_1, а множеството от изходите е подмножество на Π_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 Π_1 в един екземпляр от Π_2. С други думи, може f да не е инекция[†]. Фактът, че има many-one reduction от Π_1 в Π_2 се бележи с "$\Pi_1 \leqslant_m \Pi_2$".

Many-one reduction, която е и изчислима в полиномиално време, е Karp редукция. Като синоним ще ползваме "полиномиална редукция" или дори само "редукция", защото се подразбира, че е полиномиална. Ако има полиномиална редукция от Π_1 до Π_2 ще казваме, че Π_1 *се свежда полиномиално до* Π_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 121: Karp редукции между езици

Нека Σ е азбука и $L_1, L_2 \in \Sigma^*$. *Karp редукция от* L_1 *в* L_2 е всяка функция $f : \Sigma^* \to \Sigma^*$, 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 80 далечно прилича на Наблюдение 10, но забележете, че Наблюдение 10 касае **работните променливи** на алгоритъм, докато Наблюдение 80 касае **изхода** на алгоритъм (или изхода на МТ).

[†]За да е така, трябва или всички тези екземпляри от Π_1 да са ДА-екземпляри и екземплярът от Π_2 да е ДА-екземпляр, или всички тези екземпляри от Π_1 да са НЕ-екземпляри екземплярът от Π_2 да е НЕ-екземпляр.

Наблюдение 80: Полиномиално ограничение по време влече полиномиално ограничение на размера на изхода.

В контекста на Определение 120 или Определение 121, резултатът от редукцията е полиномиално ограничен. По-точно казано,

- в контекста на Определение 120, 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 121, съществува полином p, такъв че за всеки $\sigma \in \Sigma^*$, $|f(\sigma)| \leqslant p(|\sigma|)$.

Обосновката е практически същата като обосновката на Наблюдение 10: за всеки записан символ от изхода, машината трябва да използва поне единица от дискретното си време.

Нотация 14: Бележим полиномиалните редукции с "$\leqslant_p$"

Фактът, че Π_1 се свежда полиномиално до Π_2, записваме с "$\Pi_1 \leqslant_p \Pi_2$".

Нотацията "$\leqslant_p$" задава релация над задачите за разпознаване, която ще означаваме със същия символ.

Определение 122: $\leqslant_p$ като релация

Нека $\mathcal{D}$ е множеството от задачите за разпознаване. Релацията $\leqslant_p \subseteq \mathcal{D} \times \mathcal{D}$ се дефинира така: за всеки $\Pi_1, \Pi_2 \in \mathcal{D}$, $\Pi_1 \leqslant_p \Pi_2$ 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 $L_1 \leqslant_p L_2$.

14.6.2 Основни свойства

За изложението върху редукции следните свойства на полиномите са ключово важни.

Наблюдение 81: 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 79: Полиномиалните редукции са транзитивни.

$\leqslant_p$ е транзитивна.

Доказателство: Доказателството е по-кратко, ако редукциите са между езици (Определение 121), а не между задачи (Определение 120). Нека L_1, L_2 и L_3 са езици над някаква азбука Σ. Нека $L_1 \leqslant_p L_2$ и $L_2 \leqslant_p L_3$. Ще докажем, че $L_1 \leqslant_p L_3$.

По определение,

- съществува функция $f_{1,2} : \Sigma^* \to \Sigma^*$, 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 $f_{2,3} : \Sigma^* \to \Sigma^*$, 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 80, размерът на $M_{1,2}(\sigma)$ е ограничен така:

$$|M_{1,2}(\sigma)| \leqslant p'(|\sigma|) \tag{14.1}$$

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 $|\tau| \leqslant p'(|\sigma|)$ съгласно (14.1).
- После пускаме $M_{2,3}$ върху τ. Времето за работа на $M_{2,3}(\tau)$ е ограничено от $p''(|\tau|)$. Заключаваме, 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 81, сумата и композицията на полиноми е полином. Заключаваме, че $p'(|\sigma|) + p''(p'(|\sigma|))$ е полином на $|\sigma|$. Тогава $M_{2,3} \circ M_{1,2}(\sigma)$ е полиномиална МТ, изчисляваща $f_{2,3} \circ f_{1,2}$. Което и трябваше да покажем. □

"Преднаредба" дефинирахме в Допълнение 17. 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 32: 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. Образно 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 32) е релация върху класовете на еквивалентност. Фактор-релацията задължително е антисиметрична; освен това е рефлексивна и транзитивна, което я прави частична наредба.

Следствие 29: Полиномиалните редукции задават преднаредба.

$\leqslant_p$ е преднаредба.

Доказателство: Следва направо от Теорема 79, понеже рефлексивността е очевидна. □

И така, полиномиалните редукции задават преднаредба върху множеството от задачите за разпознаване (или върху езиците, ако предпочитате).

Определение 123: Полиномиално еквивалентни задачи

За две задачи Π_1 и Π_2 казваме, че са *полиномиално еквивалентни*, ако $\Pi_1 \leqslant_p \Pi_2$ и $\Pi_2 \leqslant_p \Pi_1$. Ще отбелязваме този факт с "$\Pi_1 \equiv_p \Pi_2$".
Ако $\Pi_1 \leqslant_p \Pi_2$ и $\Pi_2 \nleqslant_p \Pi_1$, ще казваме, че Π_1 *е полиномиално строго по-малка от* Π_2, което ще записваме с "$\Pi_1 <_p \Pi_2$".
Ако $\Pi_1 \equiv_p \Pi_2$ или $\Pi_1 <_p \Pi_2$ или $\Pi_2 <_p \Pi_1$, казваме, че Π_1 *и* Π_2 *са полиномиално сравними*. В противен случай, те са *полиномиално несравними*.

В контекста на казаното горе за преднаредбите, Π_1 и Π_2 са полиномиално еквивалентни тстк те са от един и същи клас на еквивалентност на $\leqslant_p$. Ако обаче $\Pi_1 <_p \Pi_2$, то Π_1 и Π_2 са от различни класове на еквивалентност, които класове са сравними във фактор-релацията, като класът на Π_1 предшества класа на Π_2. Ние не сме показали съществуването на полиномиално несравними задачи, но **ако** има такива задачи, **то** те са в класове на еквивалентност на $\leqslant_p$, които класове са несравними във фактор-релацията.

Започнахме Секция 14.6 с това, че ако Π_1 се редуцира до Π_2, то Π_1 е поне толкова трудна, колкото Π_2. Теорема 80 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 80: Класът P и полиномиалните редукции

Нека L_1 и L_2 са езици над някаква азбука Σ. Нека $L_1 \leqslant_p L_2$. Тогава $L_2 \in \mathbf{P}$ влече $L_1 \in \mathbf{P}$. Алтернативно, $L_1 \notin \mathbf{P}$ влече $L_2 \notin \mathbf{P}$.

Доказателство: Нека $f : \Sigma^* \to \Sigma^*$ 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 $\mathbf{x} \in \Sigma^*$, $M_f(\mathbf{x})$ работи във време, ограничено от $p_f(|\mathbf{x}|)$.

Да допуснем, че $L_2 \in \mathbf{P}$. Нека M_2 е полиномиална МТ, разпознаваща L_2. Тогава съществува полином p_2, такъв че за всеки $\mathbf{x} \in \Sigma^*$, $M_2(\mathbf{x})$ работи във време, ограничено от $p_2(|\mathbf{x}|)$.

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 $\boldsymbol{\sigma} \in \Sigma^*$. Първо пускаме M_f с вход $\boldsymbol{\sigma}$ и после пускаме M_2 върху изхода на $M_f(\boldsymbol{\sigma})$. Като цяло, пускаме $M_2 \circ M_f(\boldsymbol{\sigma})$ и така симулираме $M_1(\boldsymbol{\sigma})$.

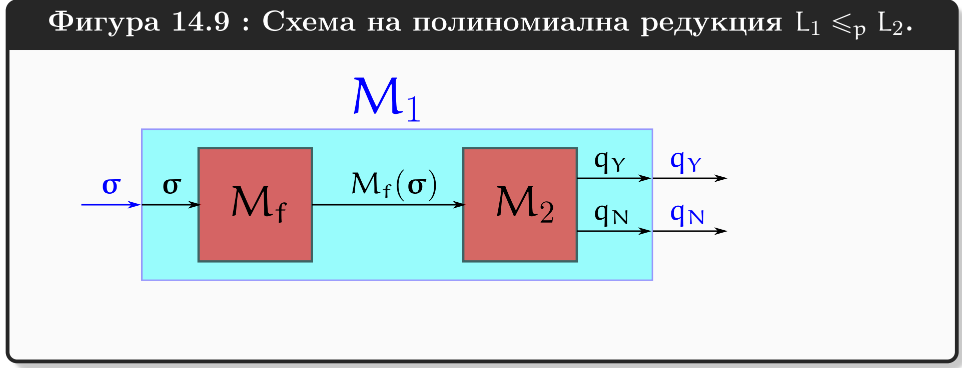
Но M_2 е машина-разпознавач, така че изходът от $M_2 \circ M_f(\boldsymbol{\sigma})$ е или q_Y, или q_N. По дефиниция, $\boldsymbol{\sigma} \in L_1$ тстк $f(\boldsymbol{\sigma}) \in L_2$, така че

- ако $M_2 \circ M_f(\boldsymbol{\sigma}) = q_Y$, то $\boldsymbol{\sigma} \in L_1$,
- ако $M_2 \circ M_f(\boldsymbol{\sigma}) = q_N$, то $\boldsymbol{\sigma} \notin L_1$.

Твърдим, че композицията $M_2 \circ M_f(\mathbf{x})$ работи във време, ограничено от $p_f(|\mathbf{x}|) + p_2(p_f(|\mathbf{x}|))$, което е някакъв полином. Това заключение правим по начин, напълно аналогичен на начина, по който направихме заключението в доказателството на Теорема 79. Показахме, че $L_2 \in \mathbf{P}$ влече $L_1 \in \mathbf{P}$. Но това е същото като $L_1 \notin \mathbf{P}$ влече $L_2 \notin \mathbf{P}$ – тези две съждения са взаимно контрапозитивни. □

Фигура 14.9 илюстрира нашата представа за полиномиалните редукции. $M_1(\boldsymbol{\sigma})$ се реализира чрез $M_2 \circ M_f(\boldsymbol{\sigma})$. 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 $L_1 \leqslant_p L_2$.



"Минималният клас на еквивалентност на $\leqslant_p$" в заглавието на Теорема 81 означава следното. Тъй като $\leqslant_p$ 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 $\mathbf{P}$.

Теорема 81: P е минималният клас на еквивалентност на $\leqslant_p$

Нека Π_1, Π_2 и Π_3 са задачи за разпознаване, като $\Pi_1, \Pi_2 \in \mathbf{P}$ и $\Pi_3 \notin$ P, като за Π_3 знаем поне един ДА-екземпляр с константна големина и поне един НЕ-екземпляр с константна големина. Тогава $\Pi_1 \equiv_p \Pi_2$ и $\Pi_1 <_p \Pi_3$.

Доказателство: Това, че $\Pi_1 \equiv_p \Pi_2$, следва директно от Теорема 80. Ерго, задачите от **P** са в един и същи клас на еквивалентност на $\leqslant_p$.

Това, че $\Pi_3 \nleqslant_p \Pi_1$, също следва директно от Теорема 80. Ерго, този клас на еквивалентност на $\leqslant_p$ се състои точно от задачите от **P**.

Да се убедим, че $\Pi_1 \leqslant \Pi_3$. Нека x е произволен екземпляр на Π_1. Щом $\Pi_1 \in \mathbf{P}$, задачата Π_1 е решима в полиномиално време. Редукцията от Π_1 в Π_3 работи така: в полиномиално време изчисляваме дали x е ДА-екземпляр или НЕ-екземпляр на Π_1. Ако x е ДА-екземпляр на Π_1, конструираме някакъв ДА-екземпляр y на Π_3. Ако x е НЕ-екземпляр на Π_1, конструираме някакъв НЕ-екземпляр z на Π_3. Забележете, че не се опитваме да правим алгоритъм за Π_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. □

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 55: TSP, оптимизационна версия

екземпляр: Наредена двойка $\langle C, \text{dist}\rangle$, където C е множество от градове $C = \{c_1, \ldots, c_n\}$, а dist е функция на разстоянията $d : C \times C \to \mathbb{N}^+$, такава че $\text{dist}(c_i, c_i) = 0$ за $1 \leqslant i \leqslant n$ и $\text{dist}(c_i, c_j) = \text{dist}(c_j, c_i)$ за $1 \leqslant i < j \leqslant n$.
въпрос: Пермутация π на градовете, такава че

$$\text{dist}(c_{\pi(1)}, c_{\pi(2)}) + \text{dist}(c_{\pi(2)}, c_{\pi(3)}) + \cdots \text{dist}(c_{\pi(n-1)}, c_{\pi(n)}) + \text{dist}(c_{\pi(1)}, c_{\pi(n)})$$

е минимално.

Изч. Задача 56: TSP, версия за разпознаване

екземпляр: Наредена тройка $\langle C, \text{dist}, k\rangle$, където C е множество от градове $C = \{c_1, \ldots, c_n\}$, dist е функция на разстоянията $d : C \times C \to \mathbb{N}^+$, такава че $\text{dist}(c_i, c_i) = 0$ за $1 \leqslant i \leqslant n$ и $\text{dist}(c_i, c_j) = \text{dist}(c_j, c_i)$ за $1 \leqslant i < j \leqslant n$, а k е цяло положително число.
въпрос: Дали има пермутация π на градовете, такава че

$$\text{dist}(c_{\pi(1)}, c_{\pi(2)}) + \text{dist}(c_{\pi(2)}, c_{\pi(3)}) + \cdots \text{dist}(c_{\pi(n-1)}, c_{\pi(n)}) + \text{dist}(c_{\pi(1)}, c_{\pi(n)}) \leqslant k$$

Допълнение 67: 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 $n \geqslant 3$, 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".
Трябва ли непременно да се избягва повторно минаване през градове? Ако мислим за житейската задача, очевидно не. Първо, има примери, в които на търговския пътник е по-евтино да минава многократно през даден град, за да извърши обиколката. Второ, 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 $n \geqslant 5$, така че при пет или повече града пътната мрежа трябва да има доста тунели, подлези и надлези, така че да реализира истински пълен граф. Ерго, на практика, ако градовете са от порядъка на десетки, търговският пътник най-вероятно ще ги обиколи по цикъл, който не е Хамилтонов, понеже повтаря върхове-градове.
Но приемаме, че задачата е дефинирана по този начин и толкова. Графът е пълен, така че при $n \geqslant 3$ 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 SALES**MAN** PROBLEM на TRAVELING SALES**PERSON** PROBLEM, по очевидни причини. За много от тези промени 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 някакъв неверен стереотип.

Редукция 1: HAMILTONIAN CYCLE $\leqslant_p$ TSP

Конструкция: Даден е екземпляр на HAMILTONIAN CYCLE, който е неориентиран граф $G = (V, E)$. Нека $V = \{v_1, \ldots, v_n\}$. От G трябва да "изработим" екземпляр на TSP във вариант за разпознаване (Задача 56): трябва да конструираме множество C от градове, да конструираме разстояние между всеки два различни града, и да конструираме число k. Алгоритъмът, който извършва конструкцията, трябва да е полиномиален в размера на G.

Има безброй начини да се направи редукцията, но сякаш най-простият е този. Първо, $C \leftarrow \{c_1, \ldots, c_n\}$, където $c_1, \ldots, c_n$ са някакви протоелементи. Второ, за $1 \leqslant i < j \leqslant n$,

- ако $(v_i, v_j) \in E$, правим $\mathrm{dist}(c_i, c_j) \leftarrow 1$,
- ако $(v_i, v_j) \notin E$, правим $\mathrm{dist}(c_i, c_j) \leftarrow 2$.

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 $s = v_{q_1}, \ldots, v_{q_n}$. Но очевидно $\text{dist}(c_{q_a}, c_{q_{a+1}}) = 1$ за $1 \leqslant a \leqslant n-1$ и освен това $\text{dist}(c_{q_1}, c_{q_n}) = 1$. Тогава има пермутация π на градовете, каквато се "иска" в Задача 56 предвид това, че $k = n$. Пермутацията е $\pi(a) = q_a$, за $1 \leqslant a \leqslant n$.

Сега допускаме, че $\langle C, \text{dist}, k \rangle$ е ДА-екземпляр на TSP. Разглеждаме пермутацията π според дефиницията на Задача 56. Да допуснем, че има два различни 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 констанен 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. □

Редукцията на HAMILTONIAN CYCLE към TSP е класика. Тя е първият пример за редукция в учебника на Garey и Johnson [51, 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 $\leqslant_p$ HALTING PROBLEM

Конструкция: Разглеждаме произволен екземпляр на ЧЕТНОСТ, който е едно естествено число n.

В константно време изчисляваме четността му.

- Ако $\mathfrak{n}$ 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 $\mathfrak{n}$ 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, ЧЕТНОСТ $\leqslant_p$ HALTING PROBLEM. Естествено, HALTING PROBLEM $\nleqslant_p$ ЧЕТНОСТ, така че е изпълнено ЧЕТНОСТ $<_p$ HALTING PROBLEM. □

В източници от Интернет, често редукциите към HALTING PROBLEM се правят по следния начин. Пак разглеждаме свеждане на ЧЕТНОСТ към HALTING PROBLEM.

Ако $\mathfrak{n}$ е четно, спри, в противен случай влез в безкраен цикъл.

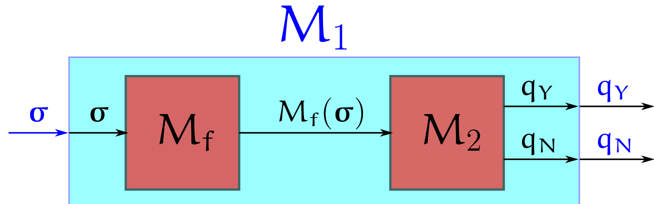
Да наречем тази програма E2H. Тя "улавя духа" на същината на нещата, но по формални причини не е редукция от ЧЕТНОСТ към HALTING PROBLEM. По определението, приложено в този контекст, редукцията е тотална функция, която изобразява четните числа в наредени двойки $(\mathfrak{P}, \mathfrak{I})$, където $\mathfrak{P}$ 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. E2H 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 $\boxed{(\mathfrak{P}, \mathfrak{I}), \text{ където } \mathfrak{P} \text{ спира върху } \mathfrak{I}}$ в никое разумно кодиране.

Според автора, причината да се ползват конструкции като E2H 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 Karp редукции срещу Turing редукции

Да разгледаме отново общата схема на Karp редукция[†], показана на Фигура 14.9. Тъй като това е важно, ето отново схемата тук:

[†]"Karp 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 Karp редукциите. При Karp редукциите, M_1 с вход $\boldsymbol{\sigma}$ първо генерира стринг $\boldsymbol{\tau} = M_f(\boldsymbol{\sigma})$ (като $|\boldsymbol{\tau}|$ е полиномиален в $|\boldsymbol{\sigma}|$), подходящ за вход на M_2, и вика M_2 само веднъж с вход $\boldsymbol{\tau}$ и просто повтаря изхода на $M_2(\boldsymbol{\tau})$. 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 Karp 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 характиристика 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** Karp редукции. В теорията, която ще разгледаме, понякога се искат непременно Karp редукции и е груба грешка в тези случаи да се ползват Turing редукции. Следната дефиниция на "Turing редукция" е от [51, 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 124: Turing редукции между задачи за търсене

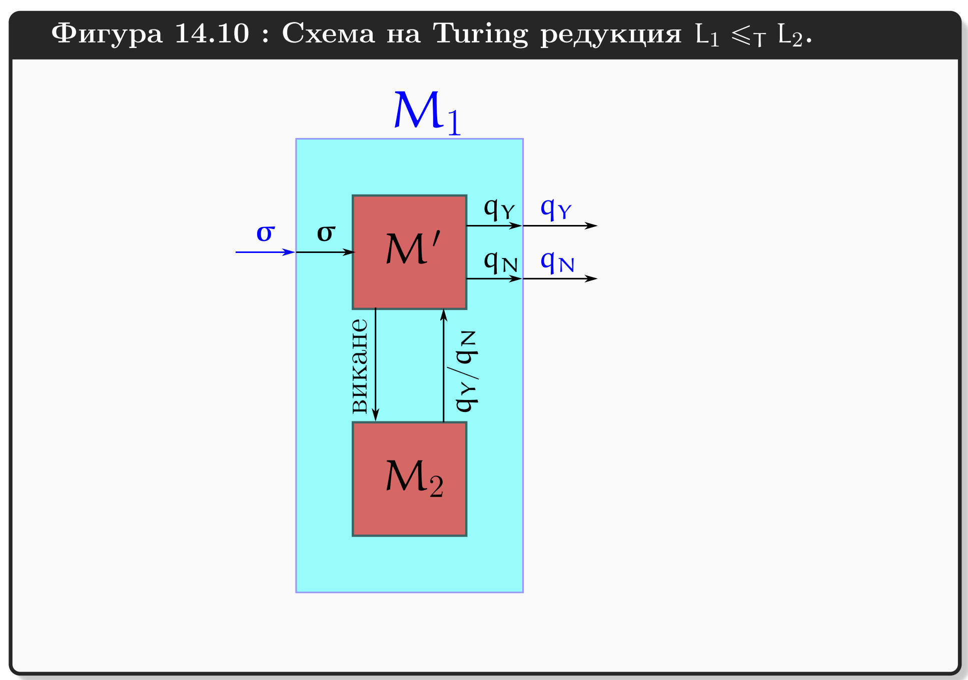
Нека Π_1 и Π_2 са задачи за търсене. *Полиномиална Turing редукция от* Π_1 *в* Π_2 е алгоритъм $\text{ALG}_{1,2}$, който решава Π_1 чрез викане на хипотетична процедура ALG_2 за решаване на Π_2, такива че, ако ALG_2 е полиномиален алгоритъм, то $\text{ALG}_{1,2}$ е полиномиален алгоритъм.
Фактът, че има такава редукция, бележим с "$\Pi_1 \leqslant_T \Pi_2$".

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 124 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 Karp редукциите. Turing редукциите са по-естествени и лесни за измисляне от Karp редукциите, и това важи особено силно за програмисти, които нямат опит в редукциите. Наистина, трябва известен опит, за да се избягва изкушението да се предлагат Turing редукции като решения на задачи, искащи Karp 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 $\boldsymbol{\sigma} \in L_1$—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 124, е в това, че и M_1, и M_2 са разпознавачи, а не изчислители.



Фигура 14.10 : Схема на Turing редукция $L_1 \leqslant_T L_2$.

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 редукцията: ако Π_1 е трудна, то и Π_2 трябва да е трудна. Но, отново да натъртим, това **не е** Karp редукция. Karp редукцията "изработва" един единствен вход на M_2, докато при Turing редукцията може да се "изработват" много такива входове, стига техният брой да не е суперполиномиален (за да имаме право да говорим за **полиномиална** Turing редукция). При Karp редукциите (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 редукциите крайният отговор се дава от M', която е "мозъкът на операцията".

За илюстрация на разликата между Karp и Turing редукциите, ще разгледаме Turing редукция и Karp редукция, и в двете посоки, между две много подобни задачи: HAMILTONIAN CYCLE (Задача 2) и HAMILTONIAN PATH.

Изч. Задача 57: 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 $n \geqslant 3$ и че става дума за свързани графи.

Hamiltonian Cycle $\leqslant_T$ 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 редукция: 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.

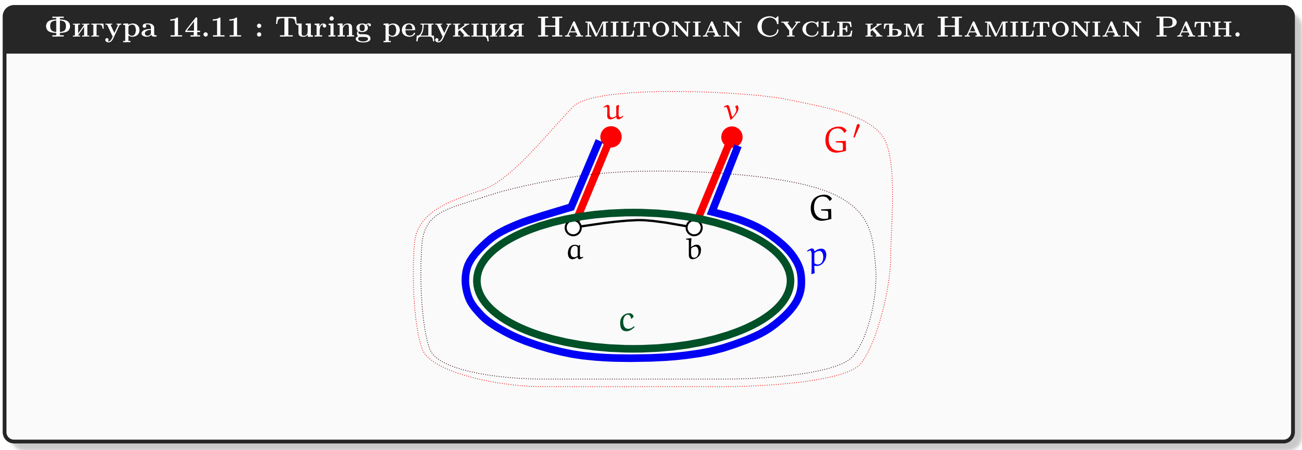
В едната посока: Да допуснем, че G има Хамилтонов цикъл c. Но този цикъл има n ребра. Нека $e = (a, b)$ е произволно ребро от c. Напълно очевидно е, че $G' = (V \cup \{u, v\}, E \cup \{(a, u), (b, v)\})$ има Хамилтонов път, който "тръгва" от u, минава през реброто (a, u), след това "върви" точно по c 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 $\widehat{p}$ в G. Добавяме (a, b) към $\widehat{p}$ и получаваме Хамилтонов цикъл c в G. □

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 представяте 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 Karp 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 Karp редукция. Мнението на автора на записките е, че тя е значително по-неочевидна и нетривиала от Turing редукцията.

Редукция 3: Hamiltonian Cycle $\leqslant_p$ Hamiltonian Path

Конструкция: Разглеждаме произволен екземпляр на Hamiltonian Cycle, който е граф $G = (V, E)$. БОО, $|V| \geqslant 3$. БОО, G няма срязващи върхове[†]. Разглеждаме произволен връх $x \in V$. Нека съседите на x са $y_1, \ldots, y_k$, като $k \geqslant 2$.

Добавяме един нов връх x' и го правим съсед точно на всеки от $y_1, \ldots, y_k$. 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.

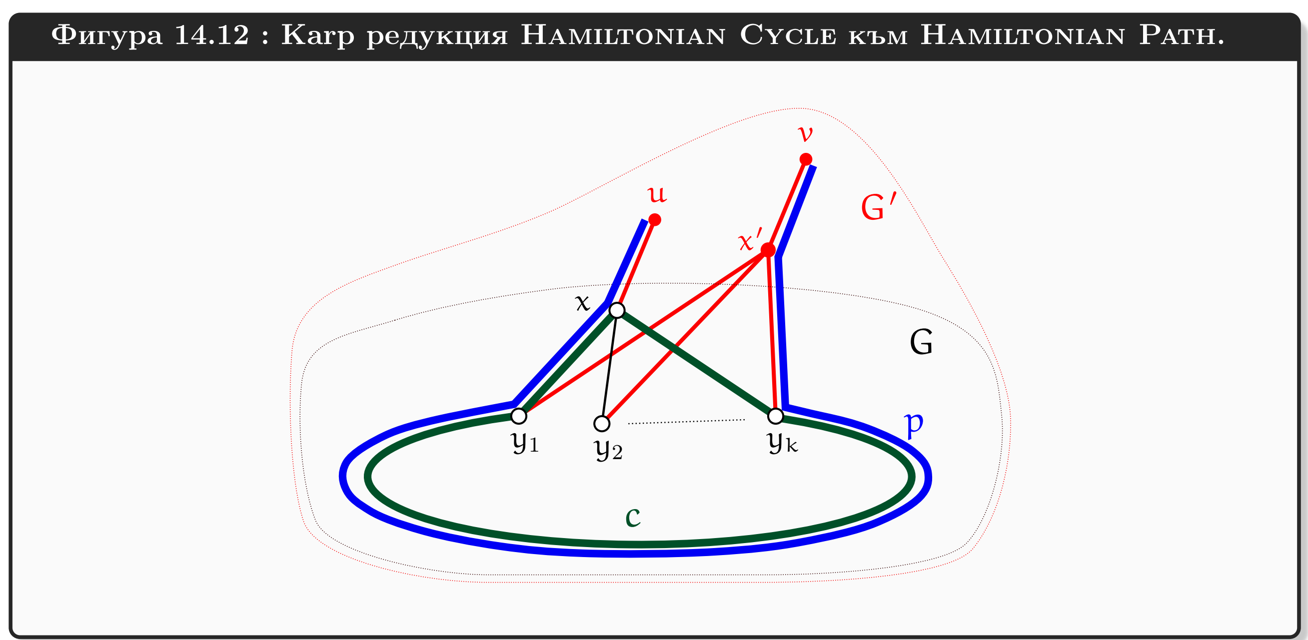
[†]Наистина е без ограничение на общността, защото в някакъв preprocessing, в полиномиално време може да се изчисли дали G има срязващи върхове. Ако има, то G е Не-екземпляр на Hamiltonian Cycle по очевидни причини и тогава нашата Karp редукция конструира най-прост граф, в който няма Хамилтонов път, 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 $y_1, \ldots, y_k$; БОО, нека е реброто (x, y_1). p съдържа точно едно ребро с краища x' и някой от $y_1, \ldots, y_k$, 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 $\widehat{p}$ в G. Забележете, че $\widehat{p}$ съдържа реброто (x, y_1), но не съдържа реброто (x', y_k), понеже x' е изтрит. Също така $\widehat{p}$ не съдържа реброто (x, y_k), но по друга причина – то не присъства в p. Добавяме (x, y_k) към $\widehat{p}$ и получаваме Хамилтонов цикъл c в G. □

Фигура 14.12 илюстрира доказателството.

Фигура 14.12 : Karp редукция Hamiltonian Cycle към Hamiltonian Path.



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 $e = (x, y)$, инак G би бил Хамилтонов. Очевидно е, че $c - e$ представлява Хамилтонов път в G.

Karp редукцията също е лесна.

Редукция 4: HAMILTONIAN PATH $\leqslant_p$ 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[†] 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'. □

Забележка 4: Karp редукциите и Turing редукциите не са едно и също нещо

Доста хора бъркат Karp редукциите и Turing редукциите. Истината е, че всяка Karp редукция е Turing редукция, което е очевидно, но не всяка Turing редукция е Karp редукция. Turing редукциите са строго по-мощни. Това е показано в Допълнение 75: ако Karp и Turing редукциите бяха равномощни, нямаше да има разлика между класовете на сложност **NP** и **coNP**.
Накратко, може да има Turing редукция $\Pi_1 \leqslant_T \Pi_2$, но да няма Karp редукция $\Pi_1 \leqslant_p \Pi_2$.

14.7 NP-пълнота

Съществува представа, че **NP** се състои само от трудни задачи, но това съвсем не е така съгласно Определение 119. **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 28, $\mathbf{P} \subseteq \mathbf{NP}$.

Забележка

Съждението "Задача Π е в **NP**" казва малко за трудността на Π: 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 Π не е в нито един от тях. Примерно, Π не е алгоритмично нерешима.
Но от гледна точка на практическа решимост, "Задача Π е в **NP**" не казва нищо съществено. Може Π 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 125: Клас на сложност NP-c

Нека Π е задача за разпознаване. Казваме, че Π е ***NP**-пълна*, ако

1. $\Pi \in \mathbf{NP}$ и
2. за всяка задача $\Pi' \in \mathbf{NP}$ е вярно, че $\Pi' \leqslant_p \Pi$.

Класът на сложност **NP-c** 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-c** може да е празното множество. Съществуването на **NP**-пълна задача ще докажем в Лекция 15.

Класът **NP-c**, 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

- Ако $\mathbf{P} = \mathbf{NP}$, то $\mathbf{P} = \mathbf{NP} = \mathbf{NP\text{-}c}$, защото всеки две задачи от **P** се свеждат полиномиално една до друга. При това положение няма смисъл да разглеждаме **NP-c**. Не че Определение 125 става некоректно, но обектът, който то определя, се тривиализира.
- Ако $\mathbf{P} \neq \mathbf{NP}$, то $\mathbf{NP\text{-}c} \subset \mathbf{NP}$. В този случай има смисъл да разглеждаме множеството от най-тежките задачи в **NP**, което е **NP-c**.

В отсъствието на отговор на $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, ние строим теории, които може да колабират в нищото, ако се окаже, че $\mathbf{P} = \mathbf{NP}$. Както ще видим дори в тези записки нататък, доста съществени резултати са относителни в смисъл, че са формулирани като "Ако $\mathbf{P} \neq \mathbf{NP}$, то ...". Наистина, мнозинството от изследователите вярват, че $\mathbf{P} \neq \mathbf{NP}$, но вярата не е достатъчна.

14.8 NP-трудност

Определение 126: Клас на сложност NP-h

Нека Π е задача за разпознаване. Казваме, че Π е ***NP**-трудна*, ако за всяка задача $\Pi' \in \mathbf{NP}$ е вярно, че $\Pi' \leqslant_p \Pi$. 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, $\mathbf{NP\text{-}c} = \mathbf{NP\text{-}h} \cap \mathbf{NP}$. 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 82: Halting Problem ∈ NP-h

Halting Problem е **NP**-трудна.

Доказателство: Да разгледаме произволна задача Π' от множеството **NP**. Ще покажем, че $\Pi' \leqslant_p$ Halting Problem.

Ние не знаем точно каква е Π', така че не може да използваме само неин екземпляр в редукцията. Щом $\Pi' \in$ **NP**, съществува полиномиална НМТ M', която решава Π'. Допускаме, че разполагаме с M', и то съвсем конкретно – с някакво подходящо описание (кодиране) на M'. Освен M' имаме и произволен екземпляр на Π', само че кодиран като стринг x с някакво подходящо кодиране.

Иска се от 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 извършва от МТ M, която **може** да симулира M' върху x. Тъй като $\Pi' \in$ **NP**, за някакъв брой стъпки M' или приема, или отхвърля x, като това става по правилата на недетерминираните 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[†].

Машината 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. □

В заключение, **NP**-трудността е нещо като долна граница. Съждението "Задача Π е **NP**-трудна" е същото като "Задача Π 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 $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Лекция 15

SAT и теоремата на Cook.

Резюме: Правим рекапитулация на булевите функции и булевите формули. Въвеждаме валюации 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 $\mathcal{X} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$. 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*: $\mathtt{x}_i$, който е *положителният литерал* и който е името на променливата, и $\overline{\mathtt{x}_i}$, който е *отрицателният литерал*. Визуалната разлика между 'x_i' с нормален математически шрифт и '$\mathtt{x}_i$' 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 'x_i' означава променлива или литерал.

Азбуката Σ_ν се състои от литералите:

$$\Sigma_\nu = \{\mathtt{x}_1, \overline{\mathtt{x}_1}, \mathtt{x}_2, \overline{\mathtt{x}_2}, \ldots\}$$

Нотация 15: $\mathsf{Var}(\phi)$

Ако ϕ е булева формула (Определение 127), то с "$\mathsf{Var}(\phi)$" означаваме множеството от нейните променливи.

[†]Ако сме още по-педантични, името на променливата е стринг над крайната азбука $\{\mathtt{x}, 0, 1, \ldots, 9\}$, но чак толкова педантични няма да бъдем.

Определение 127: Синтаксис на булевите формули

Азбуката на булевите формули е $\Sigma_\nu \cup \{(,), \neg, \wedge, \vee\}$.

❶ Всеки положителен литерал x_i е булева формула. $\mathsf{Var}(x_i) = \{x_i\}$.

❷ Всеки отрицателен литерал '$\overline{x_i}$' е булева формула. $\mathsf{Var}(\overline{x_i}) = \{x_i\}$.

❸ Ако ϕ е булева формула, то $\neg\phi$ е булева формула. $\mathsf{Var}(\neg\phi) = \mathsf{Var}(\phi)$.

❹ Нека ϕ и ψ. Тогава $(\phi \wedge \psi)$ е булева формула и $\mathsf{Var}((\phi \wedge \psi)) = \mathsf{Var}(\phi) \cup \mathsf{Var}(\psi)$.

❺ Нека ϕ и ψ. Тогава $(\phi \vee \psi)$ е булева формула и $\mathsf{Var}((\phi \vee \psi)) = \mathsf{Var}(\phi) \cup \mathsf{Var}(\psi)$.

Означаваме множеството от всички булеви формули с $\mathcal{A}$.

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 127:

$$\begin{array}{lll}
\phi_1 = x_{14}, & \phi_2 = \overline{x_{14}}, & \phi_3 = \neg x_{14}, \\
\phi_4 = \neg\overline{x_{14}}, & \phi_5 = \neg((x_2 \vee x_3) \vee x_{14}), & \phi_6 = \neg\neg\neg(x_1 \wedge x_2), \\
\phi_7 = (\overline{x_1} \wedge (\overline{x_1} \wedge \overline{x_1})), & \phi_8 = ((\overline{x_1} \wedge \overline{x_1}) \wedge \overline{x_1}), & \phi_9 = (((\overline{x_1} \wedge \overline{x_1}) \wedge \overline{x_1}) \wedge \neg\overline{x_1})
\end{array}$$

Формулите съгласно Определение 127 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 $f : \{0,1\}^n \to \{0,1\}$, 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 127 задава само синтаксиса на формулите. То казва кои стрингове над $\Sigma_\nu \cup \{(,), \neg, \wedge, \vee\}$ 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 128: Валюация

За всяко $\mathcal{Y} \subseteq \mathcal{X}$, *валюация на* $\mathcal{Y}$ е всяка функция $t : \mathcal{Y} \to \{0,1\}$.

На английски терминът най-често е *truth assignment*. На български понякога се казва "булева оценка" вместо "валюация".

Конвенция 17: Валюациите като булеви вектори

Нека $\mathcal{Y}$ е крайно и $|\mathcal{Y}| = n$. Тогава $\mathcal{Y} = \{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ за някое $\{i_1, i_2, \ldots, i_n\} \subset \mathbb{N}$. БОО, нека $i_1 < i_2 < \cdots < i_n$. 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 $1 \leqslant k \leqslant n$.
Предвид това, ще мислим за валюация на крайни множество от булеви променливи като за булев вектор, когато е удобно за изложението.

Нотация 16: $\mathsf{Val}(\mathcal{Y})$

Ако $\mathcal{Y}$ е множество от булеви променливи, множеството от валюациите на $\mathcal{Y}$ ще бележим с "$\mathsf{Val}(\mathcal{Y})$". Ако $|\mathcal{Y}| = n \in \mathbb{N}$, в светлината на Конвенция 17 имаме право да кажем, че $\mathsf{Val}(\mathcal{Y}) = \{0,1\}^n$. Тогава всяка функция $f : \mathsf{Val}(\mathcal{Y}) \to \{0,1\}$ е булева функция с арност n.

Допълнение 68: 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. Тогава "$\alpha = \beta$" е формула, която има булев смисъл – за всяка конкретно даване на стойности на нейните променливи, или е вярно, че стойността на α е равна на стойността на β, или това не е вярно.
Имаме право да говорим за удовлетворимост на формулата "$\alpha = \beta$". Тя е удовлетворима тстк съществува даване на стойности на променливите, за което стойността на α е равна на стойността на β. Като пример да разгледаме формулите $x + y = z$, $u + 1 \geqslant u$ и $v + 1 \leqslant v$ над реалните числа. Първата и втората са удовлетворими, а третата не е.
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. Тъй като всяка булева формула—дори да не съдържа релационни символи като "$=$" или "$\leqslant$"—има булев интерпретация, то има смисъл да говорим за удовлетворимост на всякакви булеви формули.

Определение 129: Семантика на булева формула под валюация. Удовлетворимост и фалшифицируемост.

Нека α е булева формула съгласно Определение 127. За всяка $t \in \mathsf{Val}(\mathsf{Var}(\alpha))$ дефинираме $\mathsf{TA}(\alpha, t) \in \{0, 1\}$, 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 $\mathsf{TA}(\alpha, t)$ е равно на $t(x_i)$.
- Ако α е получена чрез ❷, то α е отрицателен литерал $\overline{x_i}$. Тогава $\mathsf{TA}(\alpha, t)$ има обратната стойност на $t(x_i)$.
- Ако α е получена чрез ❸, то α е "$\neg\phi$" за някоя $\phi \in \mathcal{A}$. Тогава $\mathsf{TA}(\alpha, t)$ има обратната стойност на $\mathsf{TA}(\phi, t)$.
- Ако α е получена чрез ❹, то α е "$(\phi \wedge \psi)$" за някои $\phi, \psi \in \mathcal{A}$. Тогава $\mathsf{TA}(\alpha, t)$ има стойност 1 тстк и $\mathsf{TA}(\phi, t|_{\mathsf{Var}(\phi)})$, и $\mathsf{TA}(\psi, t|_{\mathsf{Var}(\psi)})$ имат стойност 1.
- Ако α е получена чрез ❺, то α е "$(\phi \vee \psi)$" за някои $\phi, \psi \in \mathcal{A}$. Тогава $\mathsf{TA}(\alpha, t)$ има стойност 1 тстк поне едната от $\mathsf{TA}(\phi, t|_{\mathsf{Var}(\phi)})$ и $\mathsf{TA}(\psi, t|_{\mathsf{Var}(\psi)})$ има стойност 1.

Казваме, че t *удовлетворява* α, ако $\mathsf{TA}(\alpha, t) = 1$. Казваме, че α е *удовлетворима*, ако съществува валюация на нейните променливи, която я удовлетворява.
Дуално, t *фалшифицира* α, ако $\mathsf{TA}(\alpha, t) = 0$. 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 129. 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 $t|_{\mathsf{Var}(\phi)}$ удовлетворява ϕ; тоест, дали $t|_{\mathsf{Var}(x_1)}$ 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 130: Семантика на булева формула.

Нека ϕ е произволна булева формула съгласно Определение 127. *Семантиката* на ϕ е булевата функция $f : \mathsf{Val}(\mathsf{Var}(\phi)) \rightarrow \{0, 1\}$, дефинирана така

$$\forall t \in \mathsf{Val}(\mathsf{Var}(\phi)) : f(t) = 1 \text{ тстк } t \models \phi$$

Ако ϕ е булева формула, такава че $\forall t \in \mathsf{Val}(\mathsf{Var}(\phi)) : t \models \phi$, 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 $\forall t \in \mathsf{Val}(\mathsf{Var}(\phi)) : t \not\models \phi$, 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 82

Нека φ е булева формула. φ е противоречие тстк $\neg\varphi$ 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 $\mathsf{TA}(\phi, t)$ за всяка $t \in \mathsf{Val}(\mathsf{Var}(\phi))$. 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 127 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

$x_1 x_3 x_4$,

$x_1 \overline{x_4}$,

$x_1 x_2 x_3 x_4 x_5 x_6$,

$\overline{x_3}$ 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 131: ДНФ

Дизюнктивна нормална форма, съкратено ДНФ, е формула, която се състои от една или повече различни конюнктивни клаузи, свързани със символа '$\vee$'.

Примери за ДНФ са

$$\begin{aligned}&x_2\overline{x_3}x_4 \text{ и}\\&x_1x_4 \vee x_2\overline{x_5}\,\overline{x_6} \vee \overline{x_2}\,\overline{x_3}\end{aligned}$$

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 127, 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 $\lambda_1, \ldots, \lambda_k$ 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 $\alpha_1, \ldots, \alpha_t$ са конюнктивните клаузи.

Поради това Нотация 15, Определение 129 и Нотация 17 са приложими към ДНФ.

Наблюдение 83: Удовлетворимост на ДНФ

Нека ϕ е ДНФ и $t \in \mathsf{Val}(\mathsf{Var}(\phi))$. $t \models \phi$ тстк съществува конюнктивна клауза α на ϕ, такава че $t|_{\mathsf{Var}(\alpha)} \models \alpha$. А $t|_{\mathsf{Var}(\alpha)} \models \alpha$ тстк за всеки литерал λ на α е вярно, че $t|_{\mathsf{Var}(\lambda)} \models \lambda$.

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

$$
\begin{aligned}
&x_1 \vee x_3 \vee x_4,\\
&x_1 \vee \overline{x_4},\\
&x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6,\\
&\overline{x_3} \;\text{ и}\\
&x_2 \vee \overline{x_5} \vee \overline{x_6}
\end{aligned}
$$

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

$$
\begin{aligned}
&\{x_1, x_3, x_4\}\\
&\{x_1, \overline{x_4}\},\\
&\{x_1, x_2, x_3, x_4, x_5, x_6\},\\
&\{\overline{x_3}\} \;\text{ и}\\
&\{x_2, \overline{x_5}, \overline{x_6}\}
\end{aligned}
\tag{15.3}
$$

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 132: КНФ

Конюнкивна нормална форма, съкратено КНФ, е формула, която се състои от конкатенация чрез '$\wedge$' 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

$$
\begin{aligned}
&(x_2 \vee \overline{x_3} \vee x_4) \;\text{ и}\\
&(x_1 \vee x_4 \vee x_2) \wedge (x_2 \vee \overline{x_5} \vee \overline{x_6}) \wedge (\overline{x_2} \vee \overline{x_3})
\end{aligned}
$$

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

$$\begin{aligned}&\{\{x_2, \overline{x_3}, x_4\}\} \text{ и}\\&\{\{x_1, x_4\}, \{x_2\overline{x_5}, \overline{x_6}\}, \{\overline{x_2}, \overline{x_3}\}\}\end{aligned} \tag{15.4}$$

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 127, ако

- мислим за всяка дизюнккивна 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 $\lambda_1, \ldots, \lambda_k$ 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 $\alpha_1, \ldots, \alpha_t$ са дизюнктивните клаузи.

Поради това Нотация 15, Определение 129 и Нотация 17 са приложими към КНФ.

Сравнете Наблюдение 83 с Наблюдение 84. Те са дуални.

Наблюдение 84: Удовлетворимост на КНФ

Нека ϕ е КНФ и $t \in \mathsf{Val}(\mathsf{Var}(\phi))$. $t \models \phi$ тстк за всяка дизюнктивна клауза α на ϕ е вярно, че $t|_{\mathsf{Var}(\alpha)} \models \alpha$. А $t|_{\mathsf{Var}(\alpha)} \models \alpha$ тстк съществува поне един литерал λ в α, такъв че $t|_{\mathsf{Var}(\lambda)} \models \lambda$.

15.4.3 Превръщане на формули в КНФ или ДНФ

Ето как можем да превърнем произволна булева формула α (съгласно Определение 127) в КНФ или ДНФ. Първо се премахваме всички негации '¬', прилагайки, докато можем,

1. законите на De Morgan

   $$\begin{aligned}\neg(\phi \wedge \psi) &\equiv (\neg\phi \vee \neg\psi)\\\neg(\phi \vee \psi) &\equiv (\neg\phi \wedge \neg\psi)\end{aligned}$$

   така че всички символи негация '¬' са само пред литерали, може би в дълги редици. Например, ако е дадена формулата

   $$\neg\neg(\overline{x_1} \vee (\neg x_2 \wedge \neg\overline{x_3})) \tag{15.5}$$

правим

$$\neg\neg(\overline{x_1} \vee (\neg x_2 \wedge \neg \overline{x_3})) \equiv$$
$$\neg(\neg\overline{x_1} \wedge \neg(\neg x_2 \wedge \neg \overline{x_3})) \equiv$$
$$\neg(\neg\overline{x_1} \wedge (\neg\neg x_2 \vee \neg\neg \overline{x_3})) \equiv$$
$$(\neg\neg\overline{x_1} \vee \neg(\neg\neg x_2 \vee \neg\neg \overline{x_3})) \equiv$$
$$(\neg\neg\overline{x_1} \vee (\neg\neg\neg x_2 \wedge \neg\neg\neg \overline{x_3}))$$

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

$$(\overline{x_1} \vee (\overline{x_2} \wedge x_3)) \tag{15.6}$$

След като се освободим от негациите, правим следното.

- Ако искаме КНФ, прилагаме, докато можем, отляво надясно дистрибутивността на дизюнкцията спрямо конюнкцията

$$(\phi \vee (\psi \wedge \xi)) \equiv ((\phi \vee \psi) \wedge (\phi \vee \xi))$$
$$((\psi \wedge \xi) \vee \phi) \equiv ((\psi \vee \phi) \wedge (\xi \vee \phi))$$

В нашия пример, прилагаме това върху формулата от (15.6) и получаваме

$$(\overline{x_1} \vee (\overline{x_2} \wedge x_3)) \equiv ((\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3)) \tag{15.7}$$

което наистина е КНФ. Незначителна разлика с изискванията на Определение 132 е наличието на много скоби и ползването на "$\wedge$"; съгласно Определение 132 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

$$(\phi \wedge (\psi \vee \xi)) \equiv ((\phi \wedge \psi) \vee (\phi \wedge \xi))$$
$$((\psi \vee \xi) \wedge \phi) \equiv ((\psi \wedge \phi) \vee (\xi \wedge \phi))$$

В нашия пример, формулата от (15.6) е почти в ДНФ, само че с прекалено много скоби и използване на "$\wedge$"; съгласно Определение 131 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 83: 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 83 казва, но може да доведе до експоненциално нарастване на дължината на формулата.

15.5 Satisfiablity (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 Karp редукции до една фундаментална задача от логиката, наречена Satisfiablity 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 129, удовлетворимост е понятие, приложимо към произволна булева формула.

Изч. Задача 58: Satisfiablity (SAT)

екземпляр: КНФ ϕ.
въпрос: Дали ϕ е удовлетворима?

Следният факт се ползва в много при моделиране чрез КНФ.

Лема 64

Нека ϕ и ψ са следните КНФ

$$\phi = \bigwedge_{1 \leqslant i < j \leqslant n} (\overline{x_i} \vee \overline{x_j}) \tag{15.8}$$

$$\psi = \left(\bigvee_{i=1}^{n} x_i \right) \wedge \bigwedge_{1 \leqslant i < j \leqslant n} (\overline{x_i} \vee \overline{x_j}) \tag{15.9}$$

ϕ е удовлетворима само от валюации с не повече от една единица. ψ е удовлетворима само от валюации с точно една единица.

Доказателство: Разглеждаме валюации на $\{x_1, \ldots, x_n\}$. Първо да разгледаме валюацията $t = 00 \cdots 0$. Очевидно t удовлетворява ϕ, но не удовлетворява ψ, понеже $\mathsf{TA}((\bigvee_{i=1}^{n} x_i), t) = 0$.

Сега да разгледаме произволна валюация t' с точно една единица. Това означава, че $t'(x_s) = 1$ за някое $s \in \{1, \ldots, n\}$ и $t'(x_k) = 0$ за всяко $k \neq s$. Тогава $\mathsf{TA}((\bigvee_{i=1}^{n} x_i), t') = 1$. Освен това, за всеки i, j, такива че $1 \leqslant i < j \leqslant n$, $\mathsf{TA}((\overline{x_i} \vee \overline{x_j}), t'|_{\{x_i, x_j\}}) = 1$. Ерго, за всяка дизюнктивна клауза α във ϕ или ψ е вярно, че $\mathsf{TA}(\alpha, t'|_{\mathsf{Var}(\alpha)}) = 1$.

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 $1 \leqslant k < s \leqslant n$. Но тогава $\mathsf{TA}((\overline{x_k} \vee \overline{x_s}), t''|_{\{x_k, x_s\}}) = 0$, а клаузата $(\overline{x_k} \vee \overline{x_s})$ се среща и във ϕ, и в ψ. □

15.5.1 Моделиране на судоку позиция чрез КНФ.

Судоку е популярна занимателна игра (пъзел), в която в квадратна таблица 9×9 някои

квадратчета са празни, а в други има число от $\{1, \ldots, 9\}$. 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 $\{1, \ldots, 9\}$ по такъв начин, че във всеки хоризонтал всяко число от $\{1, \ldots, 9\}$ да се среща точно веднъж, всеки вертикал всяко число от $\{1, \ldots, 9\}$ да се среща точно веднъж и всяко каре всяко число от $\{1, \ldots, 9\}$ 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. 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 $\{1, \ldots, n\}$. Иска се след запълването, във всеки хоризонтал, вертикал и каре, всяко число от $\{1, \ldots, n\}$ да се среща точно веднъж. Очевидно обикновеното судоку е 3-судоку. n-судоку, във вариант за разпознаване, е **NP**-пълна задача [62].

Да дефинираме судоку като изчислителна задача, и то във вариант за разпознаване. Става дума за класическото 3-судоку.

Изч. Задача 59: 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 $\{1, \ldots, 9\}$ 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 екземплярите 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 наместо n^2, е по-лесно за възприемане.

Редукция 5: СУДОКУ $\leqslant_p$ SAT

Конструкция: Даден е екземпляр на СУДОКУ – това е 9×9 судоку 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 судоку позиция чрез КНФ.

Решението е на две фази. В първата фаза само моделираме правилата на судоку 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 судоку се състои от числа от $\{1, \ldots, 9\}$, 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 **<u>и</u>** 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 $\{1, \ldots, 9\}$ 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 валюация 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 валюация 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 судокуто.

И така, въвеждаме общо $9^3 = 729$, променливи, които наричаме $p_{i,j,k}$, където $1 \leqslant i, j, k \leqslant 9$. 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 $\{1, \ldots, 9\}$ се среща точно веднъж,

2. във всяка колона, всяко число от $\{1, \ldots, 9\}$ 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:

$$
\begin{aligned}
\phi_1 = &(p_{1,1,1} \lor p_{1,2,1} \lor p_{1,3,1} \lor \cdots \lor p_{1,9,1}) \land \\
&(p_{1,1,2} \lor p_{1,2,2} \lor p_{1,3,2} \lor \cdots \lor p_{1,9,2}) \land \\
&(p_{1,1,3} \lor p_{1,2,3} \lor p_{1,3,3} \lor \cdots \lor p_{1,9,3}) \land \\
&\dots \qquad\qquad\qquad\qquad\qquad\qquad \land \\
&(p_{1,1,9} \lor p_{1,2,9} \lor p_{1,3,9} \lor \cdots \lor p_{1,9,9})
\end{aligned}
\tag{15.10}
$$

Да разгледаме само първия ред $(p_{1,1,1} \lor p_{1,2,1} \lor p_{1,3,1} \lor \cdots \lor p_{1,9,1})$. Нека t е произволна валюация, която удовлетворява крайната КНФ. За поне едно $j \in \{1, \dots, 9\}$, $t(p_{1,j,1}) = 1$. 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 аналогично конструираме

$$
\begin{aligned}
\phi_2 = &(p_{2,1,1} \lor p_{2,2,1} \lor p_{2,3,1} \lor \cdots \lor p_{2,9,1}) \land \\
&(p_{2,1,2} \lor p_{2,2,2} \lor p_{2,3,2} \lor \cdots \lor p_{2,9,2}) \land \\
&(p_{2,1,3} \lor p_{2,2,3} \lor p_{2,3,3} \lor \cdots \lor p_{2,9,3}) \land \\
&\dots \qquad\qquad\qquad\qquad\qquad\qquad \land \\
&(p_{2,1,9} \lor p_{2,2,9} \lor p_{2,3,9} \lor \cdots \lor p_{2,9,9})
\end{aligned}
$$

И така нататък за останалите редове. След това конструираме $\phi_1 \land \phi_2 \land \cdots \land \phi_9$. Накратко, заради ❶ конструираме формулата

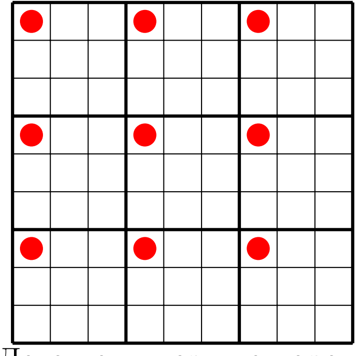
$$
\phi = \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \left(\bigvee_{j=1}^{9} p_{i,j,k} \right) \tag{15.11}
$$

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

$$\psi = \bigwedge_{j=1}^{9} \bigwedge_{k=1}^{9} \left(\bigvee_{i=1}^{9} p_{i,j,k} \right) \tag{15.12}$$

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 илюстрация 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

$$\xi = \bigwedge_{a \in \{1,4,7\}} \bigwedge_{b \in \{1,4,7\}} \bigwedge_{k=1}^{9} \left(\bigvee_{x \in \{0,1,2\}} \bigvee_{y \in \{0,1,2\}} p_{a+x,b+y,k} \right) \tag{15.13}$$

Заключаваме, 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 судоку позиция на ръка**.

Ако решаваме судоку 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 судокуто**. Примерно, може $\phi \wedge \psi \wedge \xi$ да бъде удовлетворена от валюация 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 валюация, 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

$$\gamma_{i,j} = \bigwedge_{1 \leqslant k < \ell \leqslant n} (\overline{p_{i,j,k}} \vee \overline{p_{i,j,\ell}}) \tag{15.14}$$

Съгласно Лема 64, $\gamma_{i,j}$ 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,

$$\gamma = \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \bigwedge_{1 \leqslant k < \ell \leqslant 9} (\overline{p_{i,j,k}} \vee \overline{p_{i,j,\ell}}) \tag{15.15}$$

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

$$\phi \wedge \psi \wedge \xi \wedge \gamma \tag{15.16}$$

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 $81 + 81 + 81 + 9 \times 9 \times 36 = 2\,925$ дизюнктивни клаузи. Тази КНФ точно описва правилата на судоку: 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 судоку.

Но $\phi \wedge \psi \wedge \xi \wedge \gamma$ моделира само правилата на судоку. 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 $\mathsf{TA}(\lambda, t|_{\mathsf{Var}(\lambda)}) = 1$. С което доказателството за коректността на Редукция 5 приключи.

За да бъде редукцията Karp 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-судоку, а не n-судоку). 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-судоку без проблеми, а там вече екземплярите не са с константна големина! Лесно се вижда, че ако екземплярът на n-судоку

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 краят на доказателството на Редукция 5. □

Като пример да разгледаме судоку позицията на стр. 758. 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:

$$\phi \wedge \psi \wedge \xi \wedge \gamma \wedge (p_{1,2,2}) \wedge (p_{1,4,5}) \wedge \cdots \wedge (p_{9,8,6})$$

Знаем, че позицията има решение (също показано на стр. 758, 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

$$\begin{aligned}
&t(p_{1,1,1}) = 0, t(p_{1,1,2}) = 0, t(p_{1,1,3}) = 0, t(p_{1,1,4}) = \mathbf{1}, t(p_{1,1,5}) = 0, t(p_{1,1,6}) = 0, t(p_{1,1,7}) = 0, t(p_{1,1,8}) = 0, t(p_{1,1,9}) = 0,\\
&t(p_{1,2,1}) = 0, t(p_{1,2,2}) = \mathbf{1}, t(p_{1,2,3}) = 0, t(p_{1,2,4}) = 0, t(p_{1,2,5}) = 0, t(p_{1,2,6}) = 0, t(p_{1,2,7}) = 0, t(p_{1,2,8}) = 0, t(p_{1,2,9}) = 0,\\
&t(p_{1,3,1}) = 0, t(p_{1,3,2}) = 0, t(p_{1,3,3}) = 0, t(p_{1,3,4}) = 0, t(p_{1,3,5}) = 0, t(p_{1,3,6}) = \mathbf{1}, t(p_{1,3,7}) = 0, t(p_{1,3,8}) = 0, t(p_{1,3,9}) = 0,\\
&\ldots\\
&t(p_{9,8,1}) = 0, t(p_{9,8,2}) = 0, t(p_{9,8,2}) = 0, t(p_{9,8,4}) = 0, t(p_{9,8,5}) = 0, t(p_{9,8,6}) = \mathbf{1}, t(p_{9,8,7}) = 0, t(p_{9,8,8}) = 0, t(p_{9,8,9}) = 0,\\
&t(p_{9,9,1}) = 0, t(p_{9,9,2}) = \mathbf{1}, t(p_{9,9,3}) = 0, t(p_{9,9,4}) = 0, t(p_{9,9,5}) = 0, t(p_{9,9,6}) = 0, t(p_{9,9,7}) = 0, t(p_{9,9,8}) = 0, t(p_{9,9,9}) = 0
\end{aligned}$$

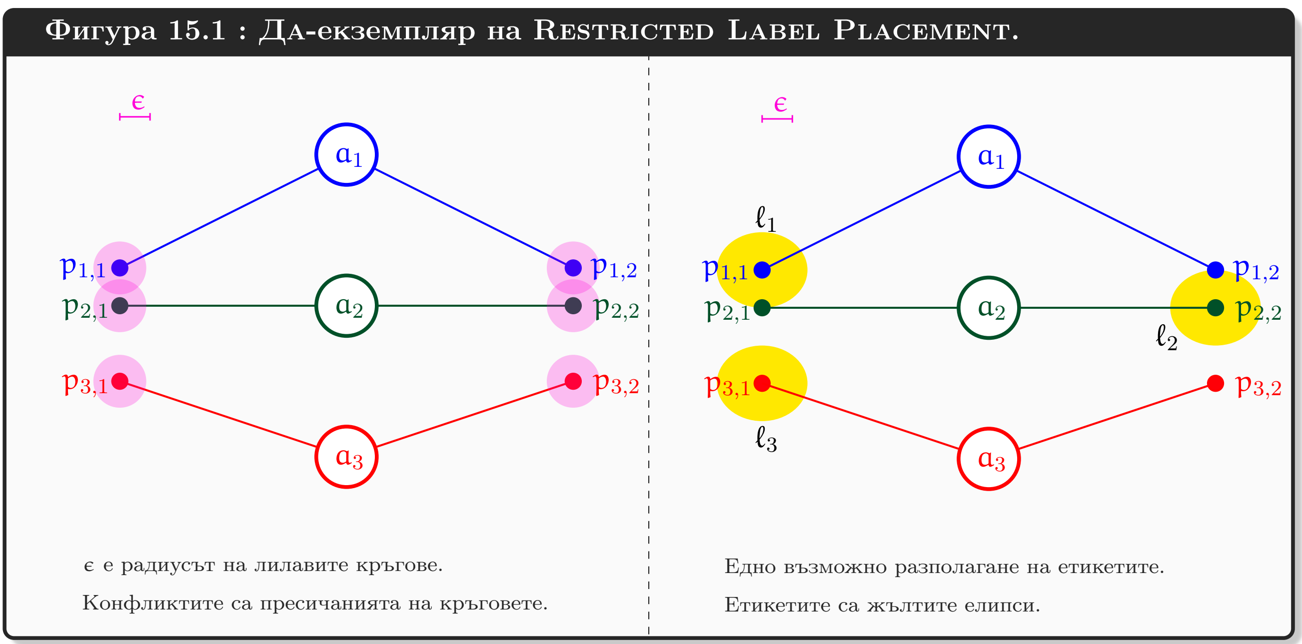
Единиците са написани в синьо, за да се отличават. Те са доста малко: само 81 на брой (за всяка клетка точно едно от съответните булеви променливи е единица). Останалите $729 - 81 = 648$ булеви променливи получават стойност нула от валюацията.

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 $a_1, \ldots, a_n$, 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 $p_{i,1}, \ldots, p_{i,t_i}$, 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.

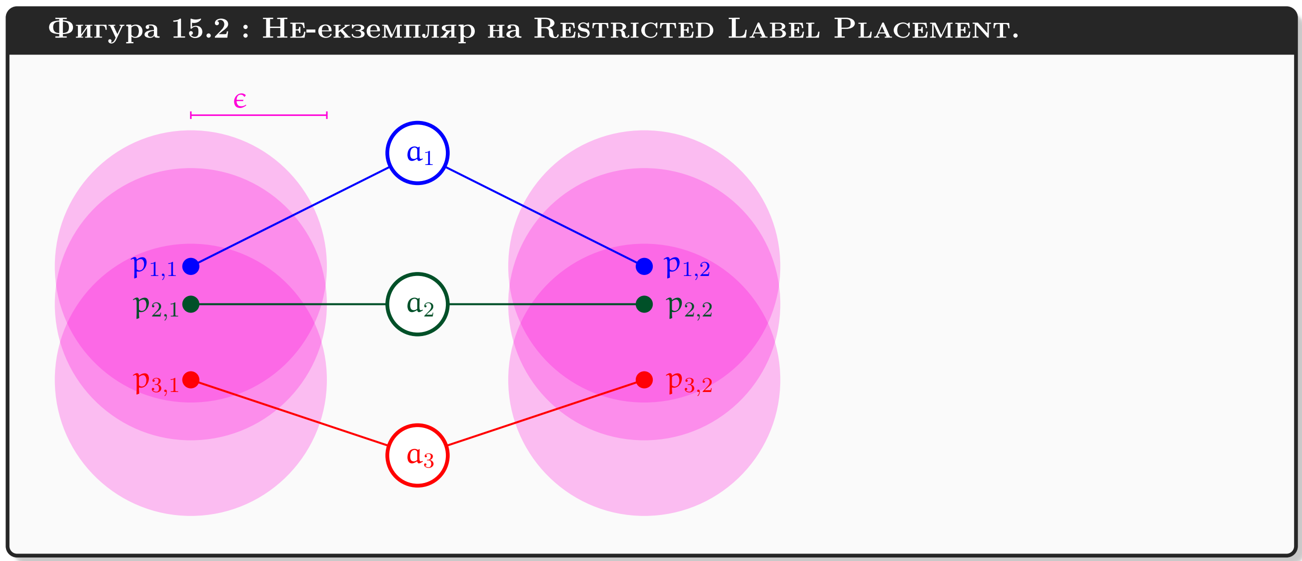
Фигура 15.1 илюстрира Да-екземпляр на Restricted Label Placement. Вляво е показан екземплярът, а вдясно има решение. Обектите са a_1, a_2 и a_3. Всеки обект a_i има точно две позиции $p_{i,1}$ и $p_{i,2}$, на точно една от които трябва да е неговият етикет ℓ_i. Съответствието между обекти и позиции е 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 $a_1, \ldots, a_n$, нито за етикетите $\ell_1, \ldots, \ell_n$, 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 $1 \leqslant i < j \leqslant n$, $a \in \{1, \ldots, t_i\}$ и $b \in \{1, \ldots, t_j\}$; 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 $t_i \geqslant 2$ за всяко $i \in \{1, \ldots, n\}$. 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 60: Restricted Label Placement

екземпляр: Непразни множества $P_1, \ldots, P_n$ без общи елементи, като за $1 \leqslant i \leqslant n$: $P_i = \{p_{i,1}, \ldots, p_{i,t_i}\}$ и $t_i \geqslant 2$. Множество $Q \subseteq \{\{p_{i,s}, p_{j,q}\} \mid 1 \leqslant i < j \leqslant n, 1 \leqslant s \leqslant t_i, 1 \leqslant q \leqslant t_j\}$.
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 $\leqslant_p$ SAT

Конструкция: Ето как моделираме екземпляр на Restricted Label Placement с КНФ. За всяко $i \in \{1, \ldots, n\}$ въвеждаме t_i булеви променливи $b_{i,1}, \ldots, b_{i,t_i}$, които отговарят на позициите $p_{i,1}, \ldots, p_{i,t_i}$, 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: $\overline{b_{i,s} \vee b_{j,q}}$ 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:

$$\phi = \bigwedge_{\{p_{i,s}, p_{j,q}\} \in Q} (\overline{b_{i,s}} \vee \overline{b_{j,q}}) \tag{15.17}$$

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 валюация, 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 валюацията $t(b_{i,j}) = 0$ 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, $1 \leqslant i \leqslant n$. Съгласно Лема 64, за всяко $i \in \{1, \ldots, n\}$, следната КНФ

ψ_i е удовлетворима само от валюации с точно една единица:

$$\psi_i = \left(\bigvee_{j=1}^{t_i} p_{i,j} \right) \wedge \bigwedge_{1 \leqslant k < s \leqslant n} (\overline{p_{i,k}} \vee \overline{p_{i,s}}) \tag{15.18}$$

Конструираме КНФ ψ от всички тези формули

$$\psi = \bigwedge_{1 \leqslant i \leqslant n} \psi_i \tag{15.19}$$

Това е и краят 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 $\leqslant_p$ 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 $V = \{v_1, \ldots, v_n\}$.

Булевите променливи са $b_{i,j}$ за $1 \leqslant i, j \leqslant n$. 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 $1 \leqslant j \leqslant n$, конструираме клауза

$$\phi_j = (b_{1,j} \vee b_{2,j} \vee \cdots \vee b_{n,j})$$

Но v_j не може да се появява повече от веднъж в пътя, за $1 \leqslant j \leqslant n$. Съгласно Лема 64, това се гарантира чрез формулата

$$\psi_j = \bigwedge_{1 \leqslant i < k \leqslant n} (\overline{b_{i,j}} \vee \overline{b_{k,j}})$$

като такава формула се прави за всяко $j \in \{1, \ldots, n\}$.

Второ, имаме клаузи, гарантиращи, че всяка позиция на пътя съдържа поне един връх. За $1 \leqslant i \leqslant n$, конструираме клауза

$$\xi_j = (b_{i,1} \vee b_{i,2} \vee \cdots \vee b_{i,n})$$

Но позиция i на пътя не може да съдържа повече от един връх, за $1 \leqslant i \leqslant n$. Съгласно Лема 64, това се гарантира чрез формулата

$$\tau_j = \bigwedge_{1 \leqslant j < k \leqslant n} (\overline{b_{i,j}} \vee \overline{b_{i,k}})$$

като такава формула се прави за всяко $j \in \{1, \ldots, n\}$.

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 съседи в G, правим формула, гарантираща, че v_i и v_j не са съседи в Хамилтоновия път. Конкатенираме тези формули с '$\wedge$' и получаваме:

$$\beta = \bigwedge_{\substack{1 \leqslant i < j \leqslant n \\ (v_i, v_j) \notin E}} \bigwedge_{k=1}^{n-1} (\overline{b_{k,i}} \vee \overline{b_{k+1,j}}) \tag{15.20}$$

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. □

15.6 SAT е NP-пълна

Определение 125 не гарантира съществуването на **NP**-пълни задачи. Сега ще видим първите образци от този интересен вид.

Допълнение 69: 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 **R** от теорията на изчислимостта съответства на **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 61: BOUNDED HALTING PROBLEM

екземпляр: НТМ M, вход $\mathbf{x}$ за M, естествено число k, написано унарно.
въпрос: Дали $M(\mathbf{x})$ спира за не повече от k стъпки?

Тук говорим за машина на Turing, а не за програма (сравнете със Задача 8). Иска се машината да е недетерминирана, защото, ако беше детерминирана, редукцията (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, редукцията, която включва изпълнението на k стъпки, щеше да е експоненциална, а не полиномиална.

Теорема 84: BOUNDED HALTING PROBLEM $\in$ NP-c

BOUNDED HALTING PROBLEM е **NP**-пълна.

Доказателство: Първо ще докажем, и то по два начина, че задачата е в **NP**. От Определение 119, ползващо Определение 116, 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 $\mathbf{x}$. 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 $(M, \mathbf{x}, k)$, такава че M с вход $\mathbf{x}$ 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 115 при вход $\mathbf{x}$. Коренът е стартовата конфигурация $(\!| q_0, \mathbf{x}, 1 |\!)$, останалите върхове са конфигурациите, които копията на машината генерират по време на работата, има поне едно листо-приемаща конфигурация $(\!| q_Y, *, * |\!)$ 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 77: 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 Bounded Halting Problem $\in$ **NP**. Фигура 14.7 добре илюстрира идеята на това доказателство.

Сега да се убедим, че Bounded Halting Problem е **NP**-трудна.

Редукция 8: $\forall \Pi \in \mathbf{NP} : \Pi \leqslant_p$ Bounded Halting Problem

Конструкция: Нека Π е произволна задача от **NP**. Ще докажем, че за всяка задача $\Pi \in \mathbf{NP}$ е вярно, че $\Pi \leqslant_p$ Bounded Halting Problem. Какво знаем за Π? Само това, че съществува НТМ M'' и съществува полином $p(n)$, такива че M'' приема всеки Да-екземпляр x на Π във време, ограничено от $p(|x|)$. По даден екземпляр x на Π, редукцията просто конструира $(M'', x, 1^{p(|x|)})$.
Коректността е очевидна: ако x е Да-екземпляр на Π и M'' е НМТ за Π с ограничаващ сложността полином $p(n)$, по дефиниция $(M'', x, 1^{p(|x|)})$ е Да-екземпляр на 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. □

Тази редукция е напълно тривиална за читател, интернализирал понятието полиномиална редукция и задачата Bounded Halting Problem. Обаче фактът, че Bounded Halting Problem $\in$ **NP-c** е напълно безполезен на практика за доказване за **NP**-пълнотата на други задачи. Задачата Bounded Halting Problem просто няма моделираща сила. Редукцията в теоремата на Cook (Теорема 85) е ценна, понеже задача SAT има грамадна моделираща сила и нейната **NP**-пълнота, предвид Теорема 86, се явява ключ за доказване на **NP**-пълнотата на много други задачи.

Една страни́чна 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 [114, глава 3]. Защо не ползваме диагоналния метод, за да докажем, аналогичното твърдение $\boxed{\mathbf{P} \neq \mathbf{NP}}$. За съжаление, или за щастие, диагоналният метод не е приложим (при определени допускания), за да различим **P** от **NP** в теорията на изчислителната сложност [46] – факт, който е далеч отвъд обхвата на тези лекционни записки.

[a] Всяка задача от **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 85: Теорема на Cook

SAT $\in$ **NP-c**.

Доказателство: Първо ще докажем, че SAT $\in$ **NP**. Нека е даден Да-екземпляр на SAT;

тоест, дадена е удовлетворима КНФ ϕ. НМТ-верификатор отгатва валюация 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 $t|_{\mathsf{Var}(\alpha)} \models \alpha$. 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 SAT $\in$ **NP-h**, което не е лесно.

Редукция 9: $\forall \Pi \in \mathbf{NP} : \Pi \leqslant_p \text{SAT}$

Конструкция: Разглеждаме произволна $\Pi \in \mathbf{NP}$. Мислим за Π 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 Π, е, че съществуват НМТ-верификатор M и полином $p(n)$, такива че M приема всеки ДА-екземпляр x на Π във време, ограничено от $p(|x|)$. Допускаме, че знаем $p(n)$ и че $p(n) \geqslant n, \forall n$. Допускаме, че имаме достъп до M, като знаем всичко за нея съгласно Определение 106: 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 Π, която свеждаме до SAT.

Въпреки че мислим за Π като за език, редукцията ще изобразява елементите на Π 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 $x \in \Pi$. 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 107 казва, че конфигурация на МТ за даден момент е $(\!| q, y, i |\!)$, 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 непрекъсната последователност 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 $\ldots, -1, 0, 1, \ldots$ 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 $-p(n) \leqslant i \leqslant p(n)+1$.

Заключаваме, 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 $n+1, \ldots, p(n)+1$ съдържат шпации,
- клекта 0 съдържа шпация, която служи за разделител между входа и сертификата,
- а съдържанието на клетки $-1, \ldots, -p(n)$ 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 $-1, \ldots, -p(n)$ са със шпации); със сигурност $|\sigma| \leqslant p(n)$, в противен случай M не би имала възможност да прочете целия сертификат, понеже няма да ѝ стигне времето.

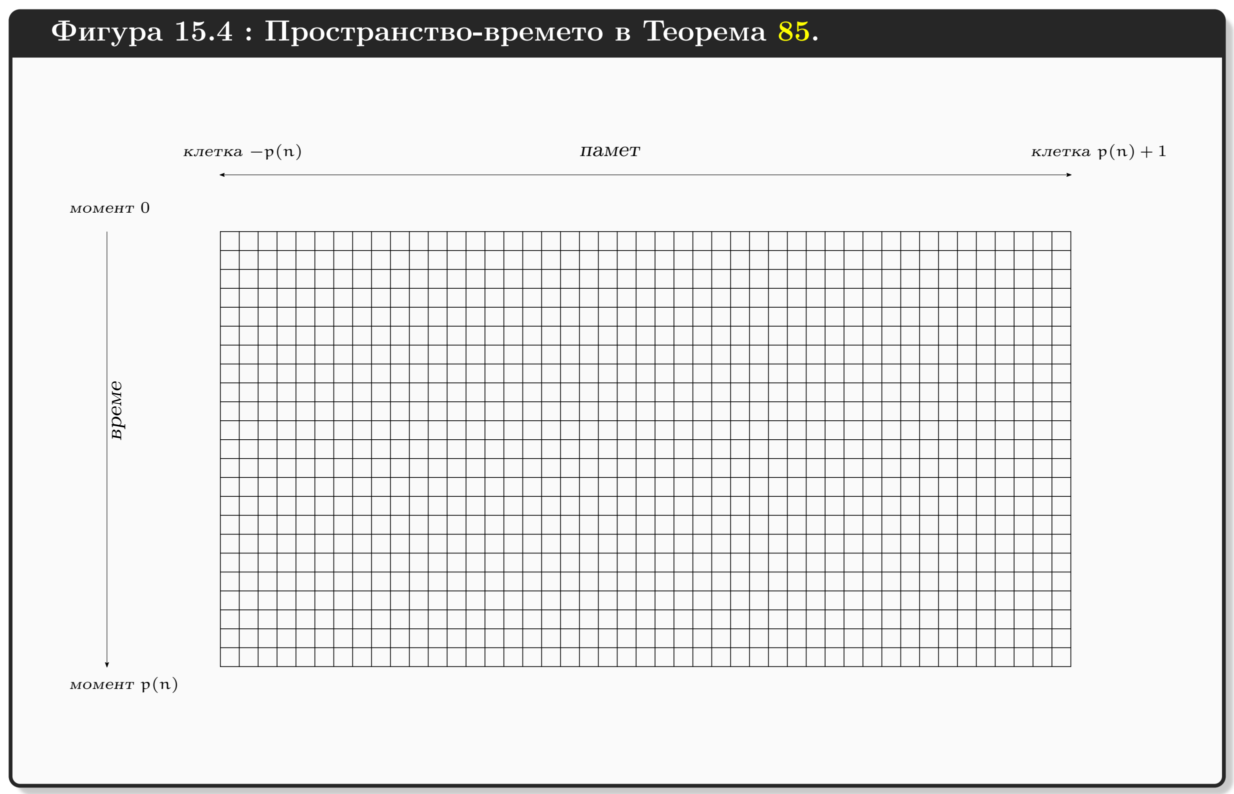
Фигурата освен това илюстрира факта, че подлентите вляво от $-p(n)$ и вдясно от $p(n)+1$ остават недостъпни за M и можем спокойно да мислим, че ги няма.

Фигура 15.3 : Началното съдържание на лентата на $M(x)$ в Теорема 85.

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.

Фигура 15.4 : Пространство-времето в Теорема 85.



Започваме да описваме редукцията. Нека множеството от състоянията на M е

$$Q = \{q_0, q_1, q_2, q_3, \ldots, q_r\}$$

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

$$\Gamma = \{\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_\nu\}$$

където γ_0 е шпацията '␣', а останалите са някакви работни букви. Очевидно $|\Gamma| = \nu + 1$. Ето какви променливи ще ползва редукцията. Те са доста и за удобство са групирани в три групи съобразно това, което моделират.

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

[†]Нещо подобно, но несравнимо по-сложно, се прави във физиката.

Група променливи	Как се менят индексите	Какво моделират
$S_{i,j}$	$i \in \{0, \ldots, p(n)\}$ $j \in \{0, \ldots, r\}$	В момент i, M е в състояние j
$H_{i,j}$	$i \in \{0, \ldots, p(n)\}$ $j \in \{-p(n), \ldots, p(n)+1\}$	В момент i, текущата клетка е j
$T_{i,j,k}$	$i \in \{0, \ldots, p(n)\}$ $j \in \{-p(n), \ldots, p(n)+1\}$ $k \in \{0, \ldots, \nu\}$	В момент i, клетка j съдържа буква k

Таблица 15.1: Променливите в Редукция 9.

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 64).

$$(S_{i,0} \vee S_{i,1} \vee \cdots \vee S_{i,r}), \text{ за } 0 \leqslant i \leqslant p(n) \tag{15.21}$$

$$(\overline{S_{i,j}} \vee \overline{S_{i,k}}) \quad, \text{ за } 0 \leqslant i \leqslant p(n) \text{ и } 0 \leqslant j < k \leqslant r \tag{15.22}$$

Група 2: Клаузите от тази група гарантират, че във всеки момент от изчислението на $M(x)$, точно една клетка е текущата (вижте Лема 64).

$$(H_{i,-p(n)} \vee H_{i,-p(n)+1} \vee \cdots \vee H_{i,p(n)+1}), \text{ за } 0 \leqslant i \leqslant p(n) \tag{15.23}$$

$$(\overline{H_{i,j}} \vee \overline{H_{i,k}}) \quad, \text{ за } 0 \leqslant i \leqslant p(n), \text{ и } -p(n) \leqslant j < k \leqslant p(n)+1 \tag{15.24}$$

Група 3: Клаузите от тази група гарантират, че във всеки момент от изчислението на $M(x)$, всяка клетка съдържа точно една буква (вижте Лема 64).

$$(T_{i,j,0} \vee T_{i,j,1} \vee \cdots \vee T_{i,j,\nu}), \text{ за } 0 \leqslant i \leqslant p(n) \text{ и } -p(n) \leqslant j \leqslant p(n)+1 \tag{15.25}$$

$$(\overline{T_{i,j,k}} \vee \overline{T_{i,j,\ell}}) \quad, \text{ за } 0 \leqslant i \leqslant p(n) \text{ и } -p(n) \leqslant j < k \leqslant p(n)+1 \text{ и } 0 \leqslant k < \ell \leqslant \nu \tag{15.26}$$

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 валюация t, такава че за всеки от тези литерали, да го наречем λ, е изпълнено $\mathsf{TA}(\lambda, t|_{\mathsf{Var}(\lambda)}) = 1$.

- Машината е в начално състояние $q_0 = 0$. Клаузата, която налага това, е

$$(S_{0,0}) \tag{15.27}$$

- Текущата клетка е 1. Клаузата, която налага това, е

$$(H_{0,1}) \tag{15.28}$$

- Входът е записан в клетки от 1 до n включително. На Фигура 15.3 отбелязахме буквите на входа като 'x_1', ..., 'x_n', но това е нотацията за елементите на $\mathbf{x}$, която не казва нищо за букви от Γ. Нека $x_1 = \gamma_{j_1}$, ..., $x_n = \gamma_{j_n}$, където $j_1, \ldots, j_n \in \{1, \ldots, \nu\}$ (нулата е изключена, защото входът не съдържа шпации). КНФ, която налага наличието на стринга $\gamma_{j_1}, \ldots, \gamma_{j_n}$ в клетки 1, ..., n, е

$$(T_{0,1,j_1}) \wedge (T_{0,2,j_2}) \wedge \cdots \wedge (T_{0,n,j_n}) \tag{15.29}$$

- Клетки $n+1$, ..., $p(n)+1$ съдържат шпации. КНФ, която налага това, е

$$(T_{0,n+1,0}) \wedge (T_{0,n+2,0}) \wedge \cdots \wedge (T_{0,p(n)+1,0}) \tag{15.30}$$

Забележете нещо важно. Никоя 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 валюация t е вярно, че t "раздава" някакви стойности на променливите $T_{0,-p(n),k}$, $T_{0,-p(n)+1,k}$, $\ldots T_{0,-1,k}$, където $k \in \{0, \ldots, \nu\}$, 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 $\boldsymbol{\sigma}$, това означава той да е разположен в клетки -1, -2, ..., $-|\boldsymbol{\sigma}|$ в нулевия момент, така че в нулевия момент там няма шпации, а в нулевия момент от клетки $-|\boldsymbol{\sigma}|-1$ 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.

$$(S_{p(n),1}) \tag{15.31}$$

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 валюция 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 ѝ 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

$$(H_{i,j} \vee \overline{T_{i,j,k}} \vee T_{i+1,j,k}), \text{ за } 0 \leqslant i \leqslant p(n) - 1 \quad \text{и } -p(n) \leqslant j < k \leqslant p(n) + 1 \quad \text{и } 0 \leqslant k \leqslant \nu \tag{15.32}$$

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 е ℓ правим следното. Нека

$$\delta(q_k, \gamma_\ell) = (q_{k'}, \gamma_{\ell'}, d)$$

където k' е новото състояние, ℓ' е новата буква, а $d \in \{\leftarrowtail, \rightarrowtail, \updownarrow\}$. Понеже моделираме позицията на главата, тоест, текущата клетка, с число, на '$\leftarrowtail$' съответства -1, на '$\rightarrowtail$' съответства 1 и на '$\updownarrow$' 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

$$
\begin{aligned}
&(Q_{i,k} \wedge H_{i,j} \wedge T_{i,j,\ell}) \rightarrow Q_{i+1,k'} &\equiv\\
&\overline{(Q_{i,k} \wedge H_{i,j} \wedge T_{i,j,\ell})} \vee Q_{i+1,k'} &\equiv\\
&\overline{Q_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee Q_{i+1,k'}
\end{aligned}
$$

И така, ето клаузите от седмата група, касаещи новото състояние.

$$
\begin{aligned}
(\overline{Q_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee Q_{i+1,k'}), \text{ за } & 0 \leqslant i \leqslant p(n) - 1\\
\text{и } & -p(n) \leqslant j < k \leqslant p(n) + 1\\
\text{и } & 0 \leqslant k \leqslant r\\
\text{и } & 0 \leqslant \ell \leqslant \nu
\end{aligned}
\tag{15.33}
$$

Аналогично конструираме и клаузите от седмата група, касаещи новата буква и клаузите, касаещи новата текуща клетка.

$$
\begin{aligned}
(\overline{Q_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee H_{i+1,j+d}), \text{ за } & 0 \leqslant i \leqslant p(n) - 1\\
\text{и } & -p(n) \leqslant j < k \leqslant p(n) + 1\\
\text{и } & 0 \leqslant k \leqslant r\\
\text{и } & 0 \leqslant \ell \leqslant \nu
\end{aligned}
\tag{15.34}
$$

$$
\begin{aligned}
(\overline{Q_{i,k}} \vee \overline{H_{i,j}} \vee \overline{T_{i,j,\ell}} \vee T_{i+1,j,\ell'}), \text{ за } & 0 \leqslant i \leqslant p(n) - 1\\
\text{и } & -p(n) \leqslant j < k \leqslant p(n) + 1\\
\text{и } & 0 \leqslant k \leqslant r\\
\text{и } & 0 \leqslant \ell \leqslant \nu
\end{aligned}
\tag{15.35}
$$

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 85. □

Ще приключим секцията с едно наблюдение. Теорема 85 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 85: Всяка ДНФ е удовлетворима

Всяка ДНФ е удовлетворима. По дефиниция (Определение 131), 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 $\phi = x_1\overline{x_2} \vee x_2x_3x_9 \vee \cdots$, 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, $\varphi = (x_1 \vee \overline{x_2})(x_2 \vee x_3 \vee x_9)\cdots$ 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 Π е произволна задача от **NP**. Искаме да докажем, че Π е **NP**-пълна. Ако доказателството е съгласно Определение 125, то е доста тегаво! Съгласно Определение 125, необходимо е да докажем, че всяка задача от **NP** се свежда до Π с Karp редукция. Доказателството на Теорема 85, което току-що видяхме, беше точно такова и наистина беше тегаво. Но, ако **вече** сме доказали **NP**-пълнотата на някаква задача Π' и ако докажем, че $\Pi' \leqslant_p \Pi$, то сме доказали, че **NP**-трудността на Π. Теорема 86 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 Π с Karp редукция.

Теорема 86: $\Pi \in \mathbf{NP} \wedge \Pi' \in \mathbf{NP\text{-}c} \wedge \Pi' \leqslant_p \Pi \rightarrow \Pi \in \mathbf{NP\text{-}c}$

Нека $\Pi, \Pi' \in \mathbf{NP}$. Ако $\Pi' \leqslant_p \Pi$ и $\Pi' \in \mathbf{NP\text{-}c}$, то $\Pi \in \mathbf{NP\text{-}c}$.

Доказателство: Тъй като по условие $\Pi \in \mathbf{NP}$, това, което остава да докажем, е, че за всяка задача $\Pi'' \in \mathbf{NP}$ е вярно, че $\Pi'' \leqslant_p \Pi$. Знаем, че $\Pi' \in \mathbf{NP\text{-}c}$. Това означава, че за всяка задача $\Pi'' \in \mathbf{NP}$ е вярно, че $\Pi'' \leqslant_p \Pi'$. Допускаме, че $\Pi' \leqslant_p \Pi$ и прилагаме Теорема 79. Заключаваме, че за всяка задача $\Pi'' \in \mathbf{NP}$ е вярно, че $\Pi'' \leqslant_p \Pi$. □

Следствие 30: Редици от доказателства на NP-пълнота, започващи със SAT

Теорема 85 и Теорема 86 позволяват следната схема за доказателства на **NP**-пълнота. За някаква задача Π_1 доказваме, че $\Pi_1 \in \mathbf{NP}$ и SAT $\leqslant_p \Pi_1$, от което следва, че Π_1 е **NP**-пълна. За друга задача Π_2 доказваме, че $\Pi_2 \in \mathbf{NP}$ и $\Pi_1 \leqslant_p \Pi_2$, от което следва, че Π_2 е **NP**-пълна. За трета задача Π_3 доказваме, че $\Pi_3 \in \mathbf{NP}$ и $\Pi_2 \leqslant_p \Pi_3$, от което следва, че Π_3 е **NP**-пълна. И така нататък.
Всяка такава редица SAT, Π_1, Π_2, Π_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 Π, чиято **NP**-пълнота трябва да докажем, някои задачи са много по удобни от други задачи в смисъл, че се свеждат до Π със

значително по-проста и лесна за разбиране редукция. Ясно е, че всяка задача от **NP-c** се свежда до Π с Karp редукция (стига да е вярно, че $\Pi \in$ **NP-c**), 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 86, 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 5: Внимание с посоката на редукцията!

В контекста на Теорема 86, ако докажем, че $\Pi \leqslant_p \Pi'$, не сме доказали нищо за **NP**-пълнотата на Π. Ама наистина **НИЩО**.
Трябва да свеждаме известната задача Π′ към новата задача Π, 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 $\mathbf{P} \neq \mathbf{NP}$. Да допуснем за целите на тази секция, че това наистина е така. Очевидно тогава е вярно $\mathbf{P} \cap \mathbf{NP\text{-}c} = \varnothing$; обратното влече $\mathbf{P} = \mathbf{NP}$. Но **NP** не се разбива на **P** и **NP-c**! Ако $\mathbf{P} \neq \mathbf{NP}$, в **NP** има задачи, които не са нито в **P**, нито в **NP-c**. Можем да кажем, че те са прекалено трудни, за да са в **P**, но не са достатъчно трудни, за да са в **NP-c**. Теоремата на Ladner доказва, че такива задачи има. В оригиналната статия тази теорема е изразена чрез Karp редукции и Turing редукции (вижте [91, 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 87: Теорема на Ladner

Да допуснем, че $\mathbf{P} \neq \mathbf{NP}$. Тогава има език $L \in \mathbf{NP} \backslash \mathbf{P}$, който не е **NP**-пълен. □

Доказателството е много сложно и е далеч отвъд обхвата на тези лекции. По същество, това доказателство е различно от оригиналното доказателство на Ladner [91, Theorem 1, стр. 158]; това доказателство първоначално е предложено от известния учен Russell Impaglaizzo в непубликувана статия. Задачата, която се разглежда, е SAT_H, дефинирана по отношение на произволна функция $H : \mathbb{N} \to \mathbb{N}$. 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 $\mathbf{P} \neq \mathbf{NP}$, то $\text{SAT}_H \notin \mathbf{P}$ и $\text{SAT}_H \notin \mathbf{NP\text{-}c}$.

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 133: Клас на сложност NP-i

Нека Π е задача за разпознаване. Казваме, че Π е ***NP**-междинна*, ако

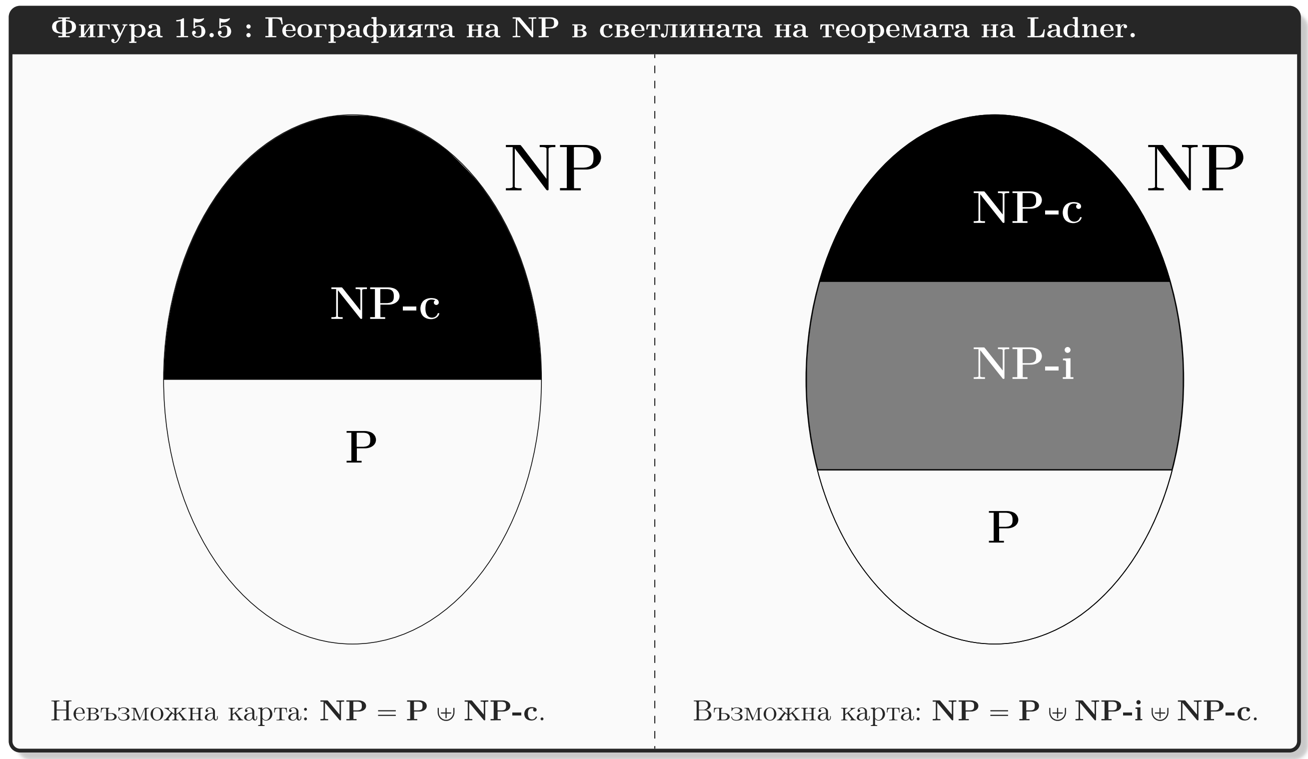
1. $\Pi \in \mathbf{NP}$ и
2. $\Pi \notin \mathbf{P}$ и $\Pi \notin \mathbf{NP\text{-}c}$.

Класът на сложност **NP-i** се състои точно от **NP**-междинните задачи.

На английски терминът **NP**-междинна е ***NP**-intermediate*. В теоретико-множествена нотация, $\mathbf{NP\text{-}i} = \mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NP\text{-}c})$. Теоремата на Ladner казва, че ако $\mathbf{P} \neq \mathbf{NP}$, то $\mathbf{NP\text{-}i} \neq \emptyset$.

Фигура 15.5 показва вляво "географска карта" на **NP**, която е възможна от общи съображения, при допускането, че $\mathbf{P} \neq \mathbf{NP}$: **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.



Невъзможна карта: $\mathbf{NP} = \mathbf{P} \uplus \mathbf{NP\text{-}c}$. Възможна карта: $\mathbf{NP} = \mathbf{P} \uplus \mathbf{NP\text{-}i} \uplus \mathbf{NP\text{-}c}$.

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* $\mathbf{NP} \setminus \mathbf{P}$

*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 [51, 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 theimage 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 perhapsit 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 62: GRAPH ISOMORPHISM

екземпляр: Неориентирани графи G_1 и G_2.
въпрос: Дали G_1 и G_2 са изоморфни?

Изч. Задача 63: 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 64: 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 **NP**, чийто минимален елемент е **P** и чийто максимален елемент е **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 [127].

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 **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 125. Следващите доказателствата за **NP**-пълнотата ще се базират на Теорема 86. Тези доказателства ще бъдат в редици, започващи със SAT, съгласно Следствие 30. Съвкупността от тези редици е дърво с корен SAT. Фигура 16.1 показва това дърво. Горната част на дървото е от [51, Глава 3.1, стр. 46–47].

Фигура 16.1 : Дървовидната структура на доказателствата за NP-трудност.

SAT

3SAT

Vertex Cover

Dominating Set

Independent Set

Clique

Art Gallery

Hamiltonian Cycle

Hamiltonian Path

Longest Path

3Colorability

Planar 3Colorability

0-1 Integer Programming

Max2SAT

NAESAT

MaxCut

3DM

2-Partition

Knapsack

Bin Packing

Multiprocessor Scheduling

$\star$-Free Regexp Ineq

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 ∈ NP-c. 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 65: 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.

$$\phi_1 = (x \vee y \vee z) \qquad \phi_2 = (x_1 \vee \overline{x_2} \vee x_3)(\overline{x_1} \vee \overline{x_2} \vee x_4)(x_1 \vee x_3 \vee x_4)$$

Забележете, че литералите трябва да са **точно** три, а не най-много три.

Редукция 10: SAT $\leqslant_p$ 3SAT

Конструкция: Нека е даден екземпляр ϕ на SAT. ϕ е КНФ над някакви променливи x_1, ..., x_n и с някакви клаузи α_1, ..., α_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 $\psi = \beta_1 \wedge \cdots \wedge \beta_m$. 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 никои 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, ..., 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 $\mathsf{TA}(\alpha_i, t|_{\mathsf{Var}(\lambda)}) = 1$. Но тогава за всяка валюация $t' \in \mathsf{Val}(\mathsf{Var}(\beta_i))$, такава че $\mathsf{TA}(\lambda, t'|_{\mathsf{Var}(\lambda)}) = 1$ 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 $\mathsf{Var}(\beta_i)$. Тогава във всяка от четирите клаузи на β_i има поне един литерал μ, такъв че $\mathsf{TA}(\mu, t'|_{\mathsf{Var}(\mu)}) = 1$. 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 $\mathsf{TA}(\mu_1, t'|_{\mathsf{Var}(\mu_1)}) = \mathsf{TA}(\mu_2, t'|_{\mu_2}) = 0$, 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

$$t' \models \beta_i \;\rightarrow\; t'|_{\mathsf{Var}(\lambda)} \models \alpha_i$$

Случай 2: α_i има точно два литерала. Тогава $\alpha_i = (\lambda \vee \mu)$ за някакви литерали λ и μ над различни променливи от $x_1, \ldots, x_n$. 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 $\alpha_i = (\lambda_1 \vee \cdots \vee \lambda_k)$, където $k \geqslant 4$. Въвеждаме $k-3$ нови променливи $y_{i,1}, \ldots, y_{i,k-3}$, които няма да ползваме за никоя друга α_j. Конструираме

$$\begin{aligned}
\beta_i = &(\lambda_1 \vee \lambda_2 \vee y_{i,1}) \wedge \\
&(\overline{y_{i,1}} \vee \lambda_3 \vee y_{i,2}) \wedge \\
&(\overline{y_{i,2}} \vee \lambda_4 \vee y_{i,3}) \wedge \\
&\ldots \\
&(\overline{y_{i,j-1}} \vee \lambda_{j+1} \vee y_{i,j}) \wedge \\
&(\overline{y_{i,j}} \vee \lambda_{j+2} \vee y_{i,j+1}) \wedge \\
&(\overline{y_{i,j+1}} \vee \lambda_{j+3} \vee y_{i,j+2}) \wedge \\
&\ldots \\
&(\overline{y_{i,k-4}} \vee \lambda_{k-2} \vee y_{i,k-3}) \wedge \\
&(\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k)
\end{aligned}$$

Твърдим, че α_i е удовлетворима тстк β_i е удовлетворима.

- В едната посока, нека α_i е удовлетворима. Тогава съществува $t \in \mathsf{Val}(\mathsf{Var}(\alpha_i))$, такава че $t \models \alpha_i$, което е същото като за поне един литерал λ_ℓ на α_i да е вярно $\mathsf{TA}(\lambda_\ell, t|_{\mathsf{Var}(\lambda_\ell)}) = 1$. Разглеждаме три подслучая.

  Случай 4.1: $\ell \in \{1, 2\}$. Тогава β_i се удовлетворява от валюацията

  $$t' \in \mathsf{Val}(\mathsf{Var}(\{\lambda_1, \ldots, \lambda_k\}) \cup \{y_{i,1}, \ldots, y_{i,k-3}\})$$

[†] Ако трябва да сме прецизни, **точно** една клауза на β_i, но това е без значение.

такава че

$$t'|_{\mathsf{Var}(\{\lambda_1,\ldots,\lambda_k\})} = t$$
$$t'(y_{i,j}) = 0, \text{ за } 1 \leqslant j \leqslant k-3$$

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: $\ell \in \{3, \ldots, k-2\}$. Тогава β_i се удовлетворява от валюацията

$$t' \in \mathsf{Val}(\mathsf{Var}(\{\lambda_1, \ldots, \lambda_k\}) \cup \{y_{i,1}, \ldots, y_{i,k-3}\})$$

такава че

$$t'|_{\mathsf{Var}(\{\lambda_1,\ldots,\lambda_k\})} = t$$
$$t'(y_{i,j}) = 1, \text{ за } 1 \leqslant j \leqslant \ell-2$$
$$t'(y_{i,j}) = 0, \text{ за } \ell-1 \leqslant j \leqslant k-3$$

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 $\mathsf{TA}(\mu, t'|_{\mathsf{Var}(\mu)}) = 1$, са в зелен цвят, а литералите μ, за които $\mathsf{TA}(\mu, t'|_{\mathsf{Var}(\mu)}) = 0$ 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.

$$\begin{aligned}
\beta_i = &(\lambda_1 \vee \lambda_2 \vee \mathbf{y_{i,1}}) \wedge \\
&(\overline{\mathbf{y_{i,1}}} \vee \lambda_3 \vee \mathbf{y_{i,2}}) \wedge \\
&(\overline{\mathbf{y_{i,2}}} \vee \lambda_4 \vee \mathbf{y_{i,3}}) \wedge \\
&\ldots \\
&(\overline{\mathbf{y_{i,\ell-3}}} \vee \lambda_{\ell-1} \vee \mathbf{y_{i,\ell-2}}) \wedge \\
&(\overline{\mathbf{y_{i,\ell-2}}} \vee \lambda_\ell \vee \mathbf{y_{i,\ell-1}}) \wedge \\
&(\overline{\mathbf{y_{i,\ell-1}}} \vee \lambda_{\ell+1} \vee \mathbf{y_{i,\ell}}) \wedge \\
&\ldots \\
&(\overline{\mathbf{y_{i,k-4}}} \vee \lambda_{k-2} \vee \mathbf{y_{i,k-3}}) \wedge \\
&(\overline{\mathbf{y_{i,k-3}}} \vee \lambda_{k-1} \vee \lambda_k)
\end{aligned}$$

Случай 4.3: $\ell \in \{k-1, k\}$. Тогава β_i се удовлетворява от валюацията

$$t' \in \mathsf{Val}(\mathsf{Var}(\{\lambda_1, \ldots, \lambda_k\}) \cup \{y_{i,1}, \ldots, y_{i,k-3}\})$$

такава че

$$t'|_{\mathsf{Var}(\{\lambda_1,\ldots,\lambda_k\})} = t$$
$$t'(y_{i,j}) = 1, \text{ за } 1 \leqslant j \leqslant k-3$$

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 валюация t'. Ще покажем, че тогава за поне един литерал λ_j е вярно, че $\mathsf{TA}(\lambda_j, t'|_{\mathsf{Var}(\lambda_j)}) = 1$. Да допуснем противното: за всяко $j \in \{0, \ldots, k\}$ е вярно, че $\mathsf{TA}(\lambda_j, t'|_{\mathsf{Var}(\lambda_j)}) = 0$.

  Разглеждаме първата клауза на β_i

  $$(\lambda_1 \vee \lambda_2 \vee y_{i,1})$$

  Щом е $\mathsf{TA}(\lambda_1, t'|_{\mathsf{Var}(\lambda_1)}) = 0$ и $\mathsf{TA}(\lambda_2, t'|_{\mathsf{Var}(\lambda_2)}) = 0$, трябва да е вярно, че $\mathsf{TA}(y_{i,1}, t'|_{\{y_{i,1}\}}) = 1$, 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 $\mathsf{TA}(\overline{y_{i,1}}, t'|_{y_{i,1}}) = 0$. Освен това $\mathsf{TA}(\lambda_3, t'|_{\mathsf{Var}(\lambda_3)}) = 0$. Тогава трябва да е вярно, че $\mathsf{TA}(y_{i,2}, t'|_{\mathsf{Var}(y_{i,2})}) = 1$, така че $t'(y_{1,2}) = 1$.

  Разглеждаме третата клауза на β_i

  $$(\overline{y_{i,2}} \vee \lambda_4 \vee y_{i,3})$$

  Щом $t'(y_{i,2}) = 1$, то $\mathsf{TA}(\overline{y_{i,2}}, t'|_{\{y_{1,2}\}}) = 0$. Освен това $\mathsf{TA}(\lambda_4, t'|_{\mathsf{Var}(\lambda_4)}) = 0$. Тогава трябва да е вярно, че $\mathsf{TA}(y_{i,3}, t'|_{\{y_{1,3}\}}) = 1$, така че $t'(y_{1,3}) = 1$.

  И така нататък – показваме, че за $j \in \{1, \ldots, k-3\}$, $t'(y_{i,j}) = 1$. Да разгледаме последната клауза на β_i

  $$(\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k)$$

  Забелязваме, че $\mathsf{TA}(\overline{y_{i,k-3}}, t'|_{\{y_{i,k-3}\}}) = \mathsf{TA}(\lambda_{k-1}, t'|_{\mathsf{Var}(\lambda_{k-1})}) = \mathsf{TA}(\lambda_k, t'|_{\mathsf{Var}(\lambda_k)}) = 0$, така че

  $$\mathsf{TA}((\overline{y_{i,k-3}} \vee \lambda_{k-1} \vee \lambda_k), t'|_{\{y_{i,k-3}\} \cup \mathsf{Var}(\lambda_{k-1}) \cup \mathsf{Var}(\lambda_k)}) = 0$$

  Тогава $t' \not\models \beta_i$. ↯

  Показахме, че за поне един литерал λ_j е вярно, че $\mathsf{TA}(\lambda_j, t'|_{\mathsf{Var}(\lambda_j)}) = 1$. Тогава $t'|_{\mathsf{Var}(\{\lambda_1, \ldots, \lambda_k\})} \models \alpha_i$. 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:

$$(x_1 \vee y_1 \vee y_2)(x_1 \vee y_1 \vee \overline{y_2})(x_1 \vee \overline{y_1} \vee y_2)(x_1 \vee \overline{y_1} \vee \overline{y_2})$$
$$(x_2 \vee \overline{x_3} \vee z)(x_2 \vee \overline{x_3} \vee \overline{z})$$
$$(x_4 \vee \overline{x_5} \vee x_6)$$
$$(\overline{x_1} \vee \overline{x_2} \vee w_1)(\overline{w_1} \vee x_7 \vee w_2)(\overline{w_2} \vee x_8 \vee \overline{x_9})$$

И двете КНФ са удовлетворими.

Допълнение 70: 2SAT $\in$ P

Задачата 2SAT се дефинира по очевидния начин.

Изч. Задача 66: 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 друга**та** 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, инак 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 67) 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 134: Implication graph

Нека е дадена формула ϕ, която е екземпляр на 2SAT. Нека $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$. *Импликационният граф*, на английски *the implication graph*, на ϕ е ориентираният граф $G = (V, E)$, където

$$\begin{aligned} V &= \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\} \\ E &= \{(u, v) \mid \phi \text{ има клауза, еквивалентна на } (\neg u, v) \text{ или } (u, \neg v)\} \end{aligned}$$

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.

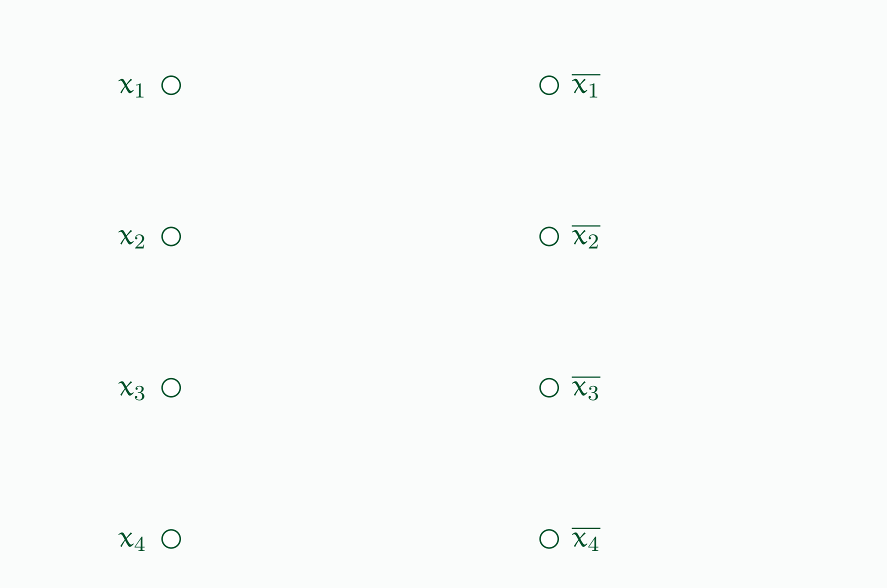
- Нека $\alpha = (x_i \vee x_j)$ е клауза на ϕ. Но $\alpha \equiv (\neg\overline{x_i} \vee x_j)$ и $\alpha \equiv (\neg\overline{x_j} \vee x_i)$. Виждаме, че α поражда ребрата $(\overline{x_i}, x_j)$ и $(\overline{x_j}, x_i)$ в графа.
- Нека $\alpha = (\overline{x_i} \vee x_j)$ е клауза на ϕ. Но $\alpha \equiv (\neg\overline{x_j} \vee \overline{x_i})$. Виждаме, че α поражда ребрата (x_i, x_j) и $(\overline{x_j}, \overline{x_i})$ в графа.
- Нека $\alpha = (\overline{x_i} \vee \overline{x_j})$ е клауза на ϕ. Виждаме, че α поражда ребрата $(x_i, \overline{x_j})$ и $(x_j, \overline{x_i})$ в графа.

Името "implication graph" идва оттам, че всяка дизюнкция $p \vee q$ е еквивалентна на импликацията $\neg p \to q$, но също така и на импликацията $\neg q \to p$. Тези две импликации са еквивалентни помежду си, естествено. Ерго, импликационният граф съдържа, под фóрмата 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 $(x \vee \overline{x})$, в импликационния граф щяхме да имаме съответни ребра-примки $(\overline{x}, \overline{x})$ и (x, x). Ако допускахме клаузи от вида $(x \vee x)$, в импликационния граф щяхме да имаме две еднакви съответни ребра $(\overline{x}, x)$[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 70 ще наричаме такъв граф *антисиметричен*.

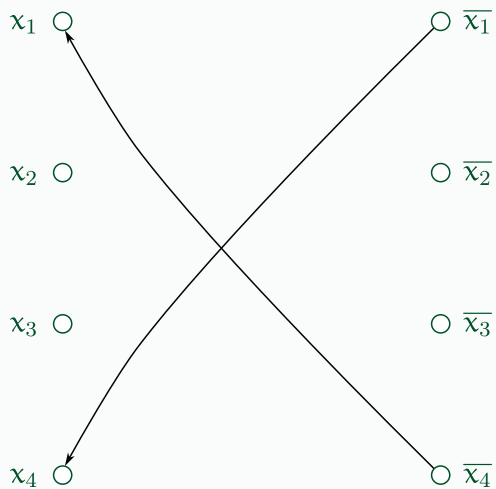
Да видим пример. Нека

$$\phi_1 = (x_1 \vee x_4)(\overline{x_2} \vee x_4)(x_2 \vee x_3)(\overline{x_1} \vee \overline{x_4})(\overline{x_2} \vee \overline{x_3})(x_2 \vee \overline{x_4}) \tag{16.1}$$

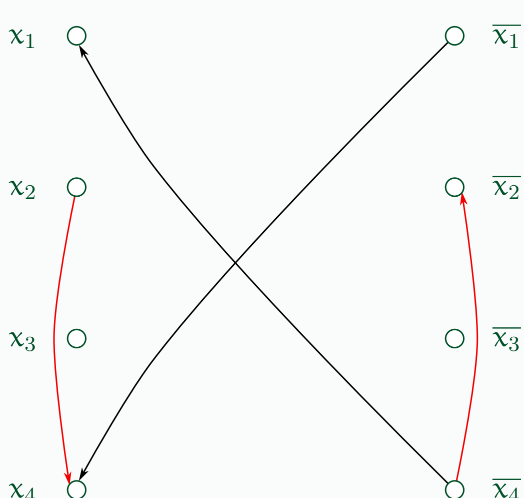
Променливите са x_1, ..., x_4, което означава, че импликационният граф има следните осем върха.



Да разгледаме клаузите. Има клауза $(x_1 \vee x_4)$ и заради нея слагаме ребра $(\overline{x_1}, x_4)$ и $(\overline{x_4}, x_1)$.



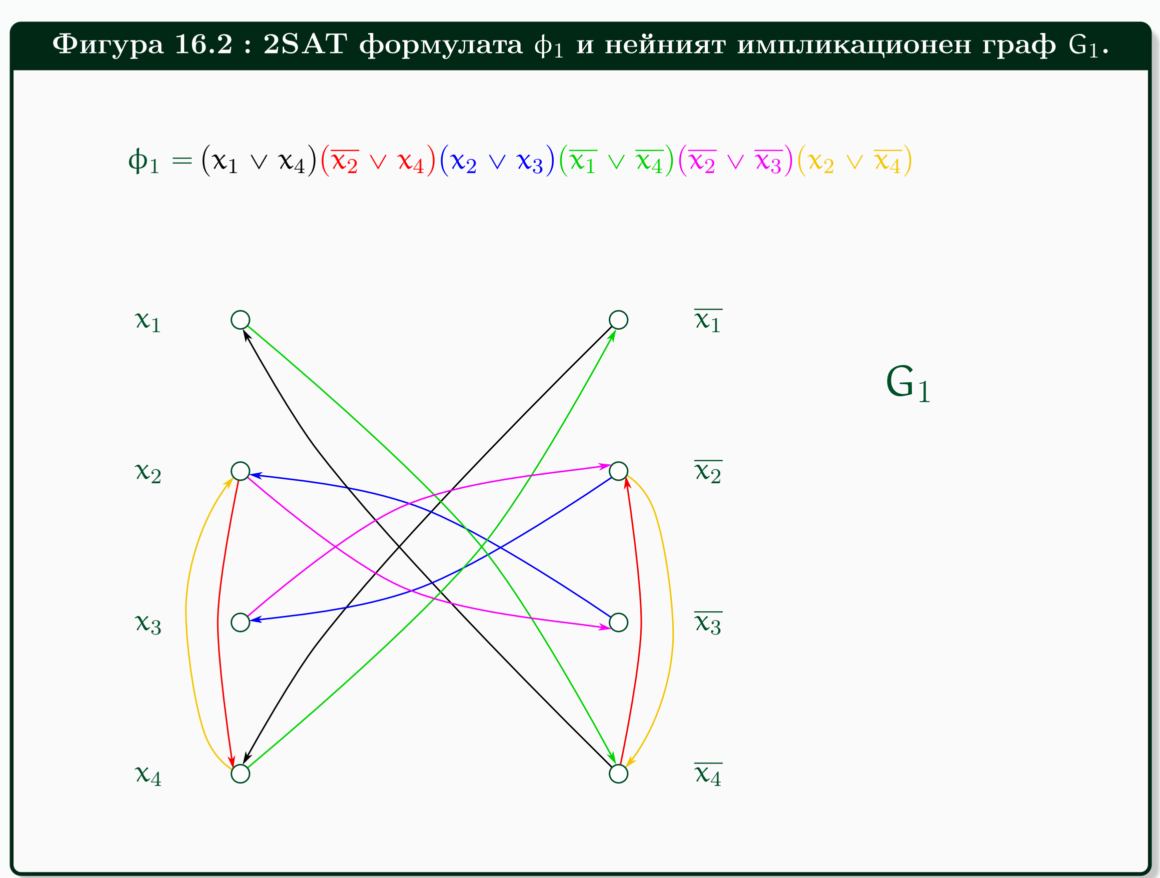
Има клауза $(\overline{x_2} \vee x_4)$ и заради нея слагаме ребра (x_2, x_4) и $(\overline{x_4}, \overline{x_2})$.



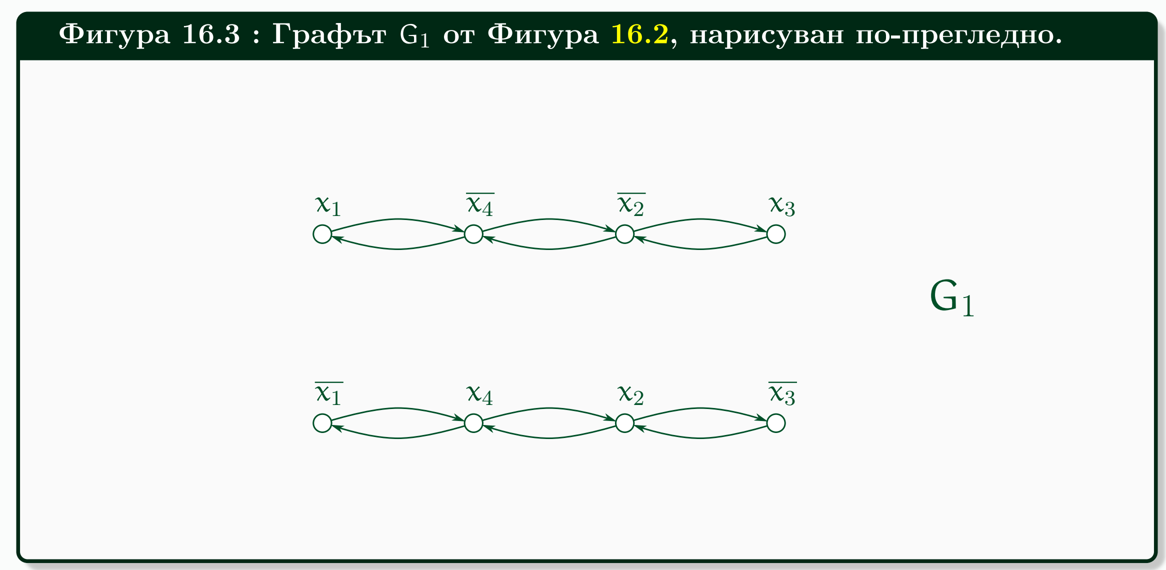
И така нататък. Целият граф G_1 е показан на Фигура 16.2, заедно с формулата.

Фигура 16.2 : 2SAT формулата ϕ_1 и нейният импликационен граф G_1.

$$\phi_1 = (x_1 \vee x_4)(\overline{x_2} \vee x_4)(x_2 \vee x_3)(\overline{x_1} \vee \overline{x_4})(\overline{x_2} \vee \overline{x_3})(x_2 \vee \overline{x_4})$$

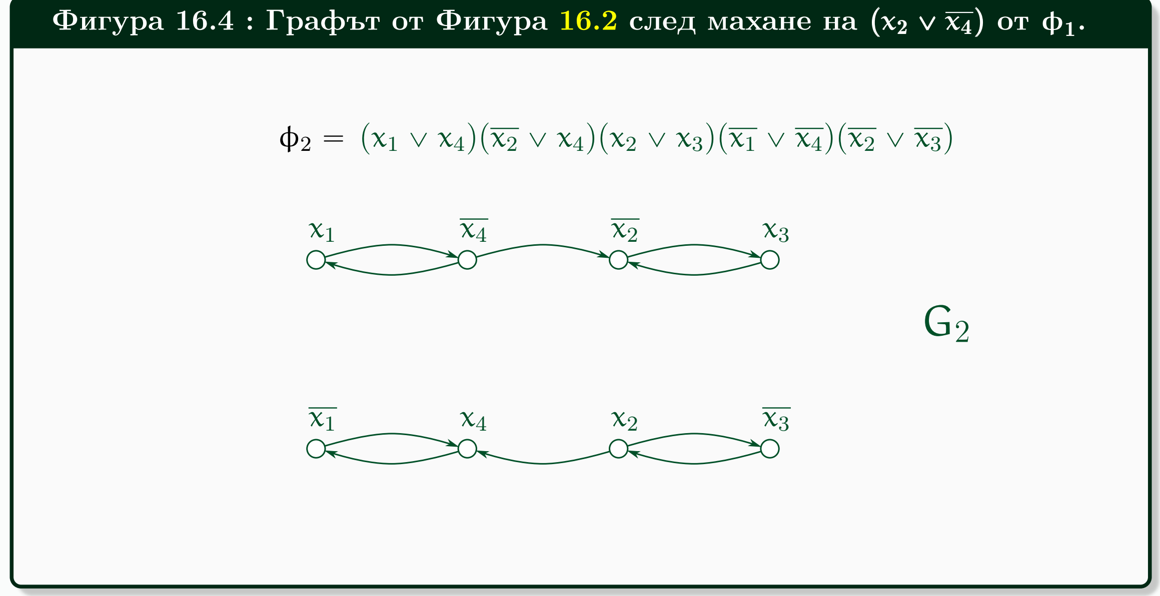


С цветове са подчертани двойките ребра, отговарящи на една и съща клауза. Фигура 16.3 показва графа от Фигура 16.2, но нарисуван по-прегледно.



Фигура 16.3 : Графът G_1 от Фигура 16.2, нарисуван по-прегледно.

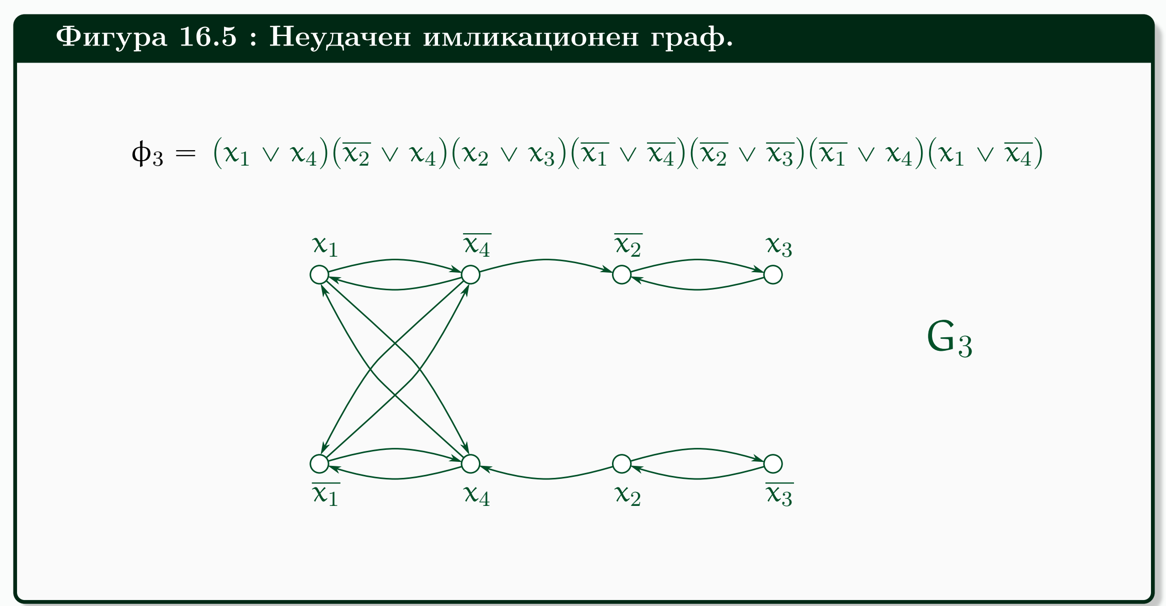
Графът G_1 от Фигура 16.3 не е добра илюстрация на антисиметричността на графа, понеже при него само обръщането на поляритета на литералите води до същия граф G_1, както и само обръщане на посоката на ребрата води до същия граф G_1. Но ако премахнем една клауза от ϕ_1 от (16.1), да кажем $(x_2 \vee \overline{x_4})$, 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 $(x_2 \vee \overline{x_4})$ от ϕ_1.

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 $\phi_3 = \phi_2(\overline{x_1} \vee x_4)(x_1 \vee \overline{x_4})$.

Фигура 16.5 : Неудачен импликационен граф.



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 $\mathsf{SC}()$:

$$\mathsf{SCC}(G_1) = \{\underbrace{\{x_1, \overline{x_2}, x_3, \overline{x_4}\}, \{\overline{x_1}, x_2, \overline{x_3}, x_4\}}_{\text{една двойка}}\}$$

$$\mathsf{SCC}(G_2) = \{\underbrace{\{x_1, \overline{x_4}\}, \{\overline{x_1}, x_4\}}_{\text{една двойка}}, \underbrace{\{\overline{x_2}, x_3\}, \{x_2, \overline{x_3}\}}_{\text{друга двойка}}\}$$

В контраст на това,

$$\mathsf{SCC}(G_3) = \{\{x_1, \overline{x_1}, x_4, \overline{x_4}\}, \{\overline{x_2}, x_3\}, \{x_2, \overline{x_3}\}\}$$

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 "$(x_2 \vee \overline{x_3})(\overline{x_2} \vee x_3)$", съответният импликационен граф ще има точно две силно свързани компоненти, като класовете на еквивалентност ще са

$$\{\{x_1, \overline{x_1}, x_4, \overline{x_4}\}, \{x_2, \overline{x_2}, x_3, \overline{x_3}\}\}$$

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 86

Импликационен граф G е удачен тстк съществува групиране на елементите на $\mathsf{SCC}(G)$ 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 $\mathsf{SCC}(G)$ неговият съответен елемент; респективно, за всяка силно свързана компонента на G, нейната съответна силно свързана компонента.

Примерно, на класа на еквивалентност $\{x_1, \overline{x_2}, x_3, \overline{x_4}\}$ от $\mathsf{SCC}(G_1)$, съответният клас е $\{\overline{x_1}, x_2, \overline{x_3}, x_4\}$; в контекста на силно свързаните компоненти на G_1, едната и другата са взаимно съответни. По отношение на G_2, класовете $\{x_1, \overline{x_4}\}$ и $\{\overline{x_1}, x_4\}$ са взаимно съответни; в контекста на силно свързаните компоненти на G_2, силно свързаните компоненти, индуцирати от тези класове, са взаимно съответни.

Лема 65

Нека $X = \{x_1, \ldots, x_n\}$ 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 $t \not\models \phi$ тстк съществува ребро $(a, b) \in E$, такова че $\mathsf{TA}(a, t|_{\mathsf{Var}(a)}) = 1$ и $\mathsf{TA}(b, t|_{\mathsf{Var}(b)}) = 0$.

Доказателство: Първо забелязваме, че в графа има ребро (a, b) тстк ϕ съдържа клауза α, еквивалентна на $(\neg a \vee b)$.
Странична забележка:е не е коректно да се каже "ϕ съдържа клаузата $(\overline{a} \vee b)$", понеже a и b са литерали, а не променливи. Ако a е отрицателен литерал $\overline{x_i}$ за някое $x_i \in X$, то ϕ съдържа не $(\overline{\overline{x_i}}, b)$, 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 $\mathsf{TA}(a, t|_{\mathsf{Var}(a)}) = 1$ и $\mathsf{TA}(b, t|_{\mathsf{Var}(b)}) = 0$. Тогава за клаузата α, еквивалентна на $(\neg a \vee b)$, е вярно, че $\mathsf{TA}(\alpha, t|_{\mathsf{Var}(\alpha)}) = 0$. 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, $\alpha \equiv (\neg a \vee b)$.

- $\mathsf{TA}(a, t|_{\mathsf{Var}(a)}) = 0$ и $\mathsf{TA}(b, t|_{\mathsf{Var}(b)}) = 0$. Тогава $\mathsf{TA}(\alpha, t|_{\mathsf{Var}(\alpha)}) = 1$ и $t \models \phi$.
- $\mathsf{TA}(a, t|_{\mathsf{Var}(a)}) = 0$ и $\mathsf{TA}(b, t|_{\mathsf{Var}(b)}) = 1$. Тогава $\mathsf{TA}(\alpha, t|_{\mathsf{Var}(\alpha)}) = 1$ и $t \models \phi$.
- $\mathsf{TA}(a, t|_{\mathsf{Var}(a)}) = 1$ и $\mathsf{TA}(b, t|_{\mathsf{Var}(b)}) = 1$. Тогава $\mathsf{TA}(\alpha, t|_{\mathsf{Var}(\alpha)}) = 1$ и $t \models \phi$. □

Теорема 88: 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 $x \in \mathsf{Var}(\phi)$, такава че в G съществуват пътища p и q, такива че $x \overset{p}{\rightsquigarrow} \overline{x}$ и $\overline{x} \overset{q}{\rightsquigarrow} x$. Щом ϕ е удовлетворима, съществува удовлетворяваща валюация t на $\mathsf{Var}(\phi)$. Да разгледаме възможностите за $t(x)$ поотделно.

- Да допуснем, че $t(x) = 1$. Тогава $\mathsf{TA}(x, t|_{\{x\}}) = 1$ и $\mathsf{TA}(\overline{x}, t|_{\{x\}}) = 0$. 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 $\overline{x}$ има стойност 0. Очевидно съществува ребро (u, v) в p, такова че $\mathsf{TA}(u, t|_{\mathsf{Var}(u)}) = 1$, а $\mathsf{TA}(v, t|_{\mathsf{Var}(v)}) = 0$. Съгласно Лема 65, $t \not\models \phi$. ↯
- Да допуснем, че $t(x) = 0$. Разглеждаме пътя q от $\overline{x}$ до x. Аналогично на предния случай, заключаваме, че в q има ребро (u, v), такова че $\mathsf{TA}(u, t|_{\mathsf{Var}(u)}) = 1$ и $\mathsf{TA}(v, t|_{\mathsf{Var}(v)}) = 0$. Съгласно Лема 65, $t \not\models \phi$. ↯

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 $\mathsf{Var}(\phi)$.

```
Alg2SAT(Удачен импликационен граф G на 2SAT формула φ)
 1  построяваме G/SC(G)
 2  сортираме G/SC(G) топологически
 3  foreach H ∈ SCC(G) в реда на топологическото сортиране
 4      foreach λ ∈ H
 5          (* λ е литерал на φ *)
 6          нека x е променливата на λ
 7          if t(x) не е дефинирана
 8              if λ е положителен литерал
 9                  t(x) ← 0
10              else
11                  t(x) ← 1
12  return t
```

Ще докажем коректността на алгоритъма Alg2SAT. Както знаем от Наблюдение 45, $G/SC(G)$ 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-връх 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 86). След топо-сортирането на фактор-графа, за всяка двойка взаимно съответни елементи на $SCC(G)$ 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, $TA(u, t|_{Var(u)}) = 1$ и $TA(v, t|_{Var(v)}) = 0$. 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 $TA(v, t|_{Var(v)})$ 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.

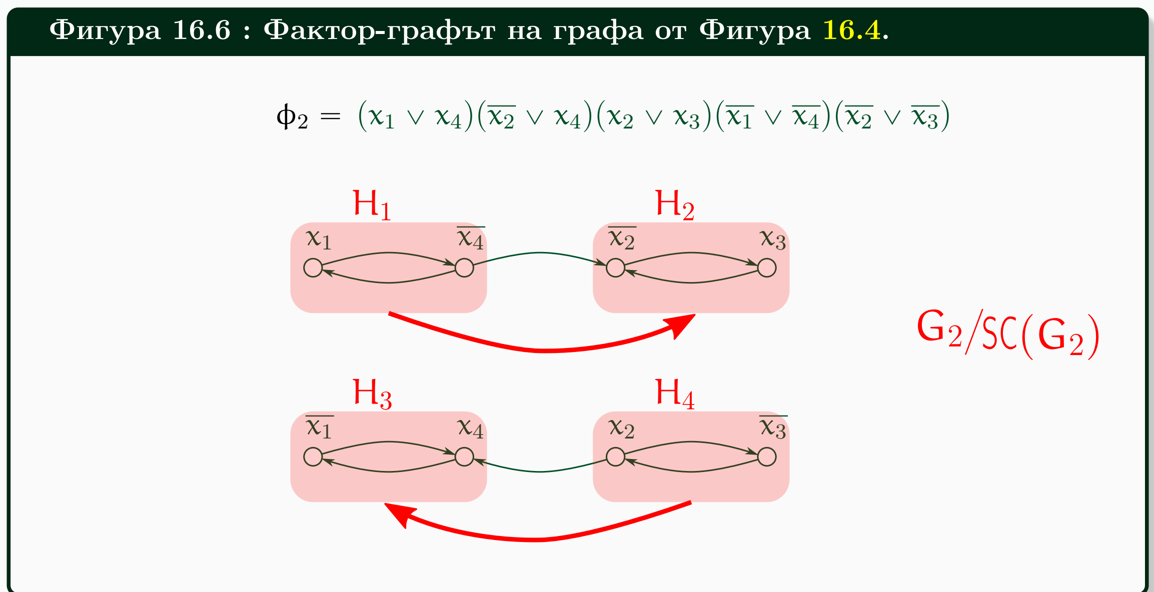
  И в двата случая няма как стигнем до ситуацията $TA(u, t|_{Var(u)}) = 1$ и $TA(v, t|_{Var(v)}) = 0$, която е "гибелна" за удовлетворяването съгласно Лема 65.

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 $\phi_2 = (x_1 \vee x_4)(\overline{x_2} \vee x_4)(x_2 \vee x_3)(\overline{x_1} \vee \overline{x_4})(\overline{x_2} \vee \overline{x_3})$. На Фигура 16.6 е показан фактор-графът му

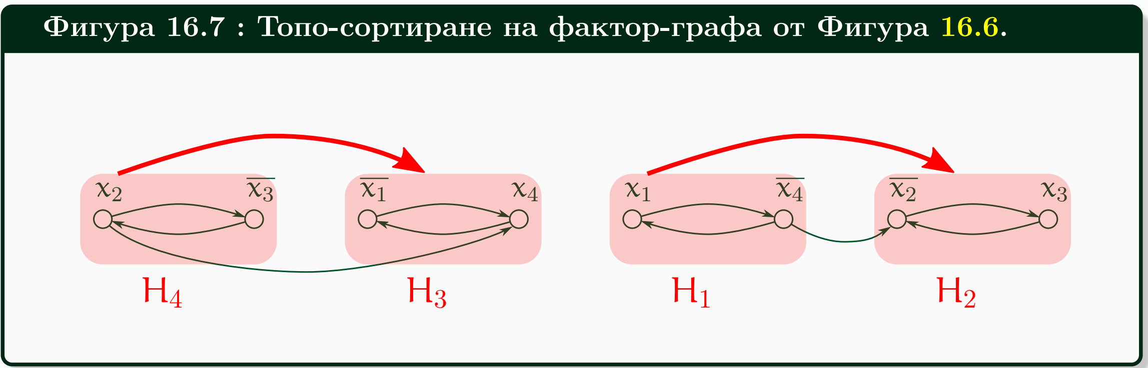
$$G_2/\mathsf{SC}(G_2) = (\{H_1, H_2, H_3, H_4\}, \{(H_1, H_2), (H_4, H_3)\})$$

Фигура 16.6 : Фактор-графът на графа от Фигура 16.4.

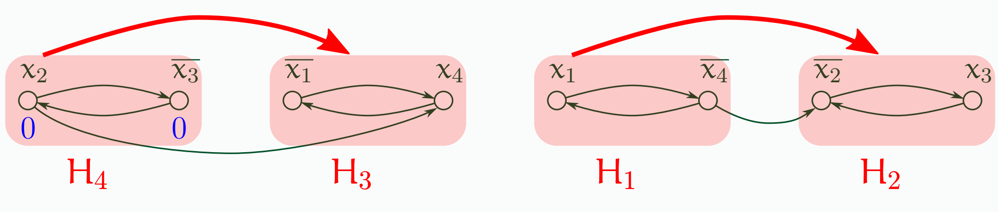


Да топо-сортираме фактор-графа. Фигура 16.7 показва възможен резултат.

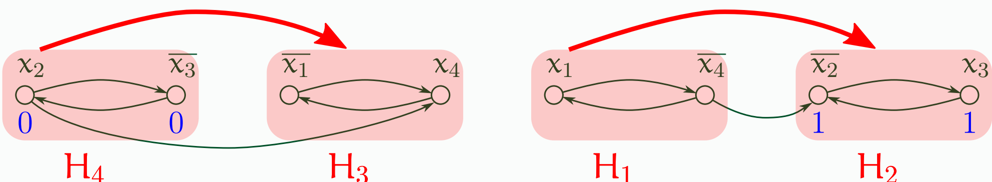
Фигура 16.7 : Топо-сортиране на фактор-графа от Фигура 16.6.



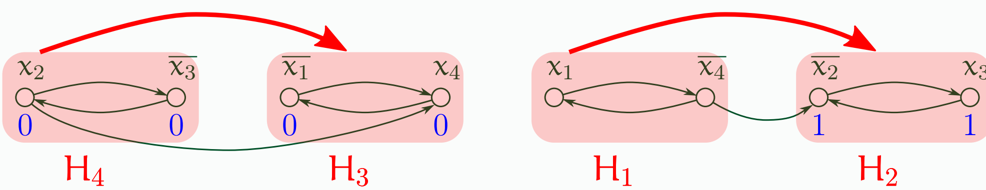
При първата итерация на главния цикъл (редове 3–11), променливата H получава стойност H_1. Алгоритъмът "нулира" и двата литерала x_2 и $\overline{x_3}$:



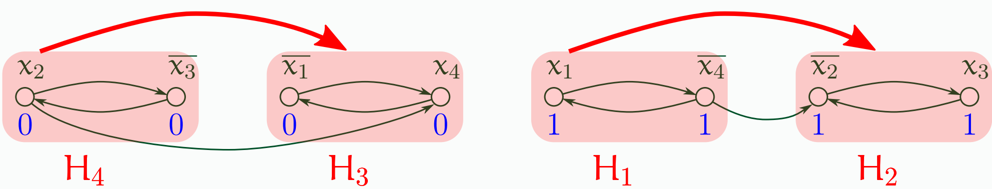
Това става с $t(x_2) \leftarrow 0$ и $t(x_3) \leftarrow 1$. Забележете, че H_2 е силно свързаната компонента, съответна на H_4, така че даването на стойности нула на литералите на H_4 дава стойности единица на литералите в H_2:



При втората итерация на главния цикъл (редове 3–11), променливата H получава стойност H_3. Алгоритъмът "нулира" и двата литерала $\overline{x_1}$ 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.

$$\phi_2 = (x_1 \vee x_4)(\overline{x_2} \vee x_4)(x_2 \vee x_3)(\overline{x_1} \vee \overline{x_4})(\overline{x_2} \vee \overline{x_3})$$

И накрая отбелязваме, че конструктивното доказателство на Теорема 88 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.

[a]Ако 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 ∈ NP-c. 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.

Изч. Задача 67: 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.

Изч. Задача 68: 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 $\leqslant_p$ 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 $(x \vee z)(x \vee \overline{z})$, 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:

$$\psi_i \equiv (\lambda_{1,i})(\lambda_{2,i})(\lambda_{3,i})(z_i)(\neg\lambda_{1,i} \vee \neg\lambda_{2,i})(\neg\lambda_{1,i} \vee \neg\lambda_{3,i})(\neg\lambda_{2,i} \vee \neg\lambda_{3,i})(\lambda_{1,i} \vee \overline{z_i})(\lambda_{2,i} \vee \overline{z_i})(\lambda_{3,i} \vee \overline{z_i})$$

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 66

Нека β е следната КНФ с десет клаузи:

$$\beta = \underbrace{(x_1)(x_2)(x_3)(z)}_{\text{група 1}} \underbrace{(\overline{x_1} \vee \overline{x_2})(\overline{x_1} \vee \overline{x_3})(\overline{x_2} \vee \overline{x_3})}_{\text{група 2}} \underbrace{(x_1 \vee \overline{z})(x_2 \vee \overline{z})(x_3 \vee \overline{z})}_{\text{група 3}}$$

Не съществува $t \in \mathsf{Val}(\mathsf{Var}(\beta))$, 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 удовлетворимите 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 група 1 и група 3 са удовлетворени. ✓
   - Ако направим $t(z) = 0$, клаузата (z) от група 1 не е удовлетворена, а група 3 е удовлетворена, така че удовлетворяваме точно шест клаузи.
2. Ако направим $t(x_1) = t(x_2) = 1$ и $t(x_3) = 0$, губим клаузата (x_3) от група 1 и $(\overline{x_1} \vee \overline{x_2})$ от група 2, така че дотук имаме пет удовлетворени клаузи.
   - Ако направим $t(z) = 1$, не губим повече от група 1, но губим $(x_3 \vee \overline{z})$ 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 $(x_2 \vee \overline{z})$ и $(x_3 \vee \overline{z})$ 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. □

Следствие 31

Нека x_1, x_2, x_3 и z са булеви променливи, а λ_i е литерал над x_i, за $1 \leqslant i \leqslant 3$. 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 $t \in \mathsf{Val}(\mathsf{Var}(\beta))$, за която съществува $i \in \{1, 2, 3\}$, такова че $\mathsf{TA}(\lambda_i, t|_{\mathsf{Var}(\lambda_i)}) = 1$. □

Ето обосновката на коректността.

- Да допуснем, че ϕ е удовлетворима. Разглеждаме произволна удовлетворяваща $t \in \mathsf{Val}(\mathsf{Var}(\phi))$.

  Разглеждаме произволно $i \in \{1, \ldots, m\}$. Нека $t_i = t|_{\mathsf{Var}(\phi_i)}$. 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 валюация удовлетворява точно някакво непразно подмножество X на нейните литерали. От Следствие 31 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 $t'_i \in \mathsf{Val}(\mathsf{Var}(\psi_i))$, че $t'_i|_{\mathsf{Var}(\phi_i)} = t_i$, 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 66, 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 $t' \in \mathsf{Val}(\mathsf{Var}(\psi))$, която се образува от комбинирането на $t'_1, \ldots, t'_m$. По-формално казано, $t'|_{\mathsf{Var}(\psi_i)} = t'_i$, за $1 \leqslant i \leqslant m$. 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 $t \in \mathsf{Val}(\mathsf{Var}(\phi))$. Съществува клауза ϕ_i, която за всеки литерал $\lambda_{j,i}$ е вярно, че $\mathsf{TA}(\lambda_{j,i}, t|_{\mathsf{Var}(\lambda_{j,i})}) = 0$. Съгласно Следствие 31, за всяка валюация на $\mathsf{Var}(\psi_i)$ е вярно, че тя удовлетворява не повече от шест клаузи на ψ_i. Заключаваме, че както валюацията $t|_{\mathsf{Var}(\phi_i)} \cup \{(z_i, 0)\}$, така и валюацията $t|_{\mathsf{Var}(\phi_i)} \cup \{(z_i, 1)\}$ удовлетворяват не повече от шест клаузи на ψ_i.

  Пак съгласно Следствие 31, за всяко $j \in \{1, \ldots, m\} \setminus \{i\}$, всяка валюация $t_j \in \mathsf{Val}(\mathsf{Var}(\psi_j))$ удовлетворява най-много седем от десетте клаузи на ψ_i. Заключаваме, 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. □

NAESAT $\in$ NP-c. 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 не неподатливост на други задачи.

Определение 135: NAE-удовлетворимост

Нека ϕ е КНФ с поне два литерала във всяка клауза. Казваме, че ϕ *е NAE-удовлетворима*, ако съществува $t \in \mathsf{Val}(\mathsf{Var}(\phi))$, такава че за всяка клауза α на ϕ е вярно, че

- α съдържа литерал λ, такъв че $\mathsf{TA}(\lambda, t|_{\mathsf{Var}(\lambda)}) = 1$ и
- α съдържа литерал μ, такъв че $\mathsf{TA}(\mu, t|_{\mathsf{Var}(\mu)}) = 0$.

На английски терминът е NAE-*satisfiable*. "NAE" идва от "**n**ot **a**ll **e**qual". 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 $\mathsf{TA}(\lambda, t|_{\{x_2\}}) = 1$, 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 $\mathsf{TA}(\lambda, t|_{\{x_2\}}) = 0$, което е невъзможно.

Съответната изчислителна задача е неподатлива, ако всяка клауза има поне три литерала. БОО, нека са точно три.

Изч. Задача 69: 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: 3SAT $\leqslant_p$ 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 $t \in \mathsf{Val}(\mathsf{Var}(\phi))$, която удовлетворява ϕ. Разглеждаме произволно $i \in \{1, \ldots, m\}$. Предвид симетричното участие на $\lambda_{1,i}$ и $\lambda_{2,i}$ в ψ_i, следните възможности са изчерпателни.

- $\mathsf{TA}(\lambda_{1,i}, t|_{\mathsf{Var}(\lambda_{1,i})}) = \mathsf{TA}(\lambda_{2,i}, t|_{\mathsf{Var}(\lambda_{2,i})}) = 1$. Всяка валюация $t'_i \in \mathsf{Val}(\mathsf{Var}(\psi_i))$, където $t(x_i) = t(y_i) = 0$ NAE-удовлетворява ψ_i. Казваме "всяка", понеже стойностите на $t(\mathsf{Var}(\lambda_{3,i}))$ 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.

  $$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

- $\mathsf{TA}(\lambda_{1,i}, t|_{\mathsf{Var}(\lambda_{1,i})}) = 1$ и $\mathsf{TA}(\lambda_{2,i}, t|_{\mathsf{Var}(\lambda_{2,i})}) = 0$. Правим $t(x_i) = t(y_i) = 0$. Отново ψ_i може да бъде NAE-удовлетворена.

  $$(\lambda_{1,i} \vee \lambda_{2,i} \vee x_i)(\overline{x_i} \vee \lambda_{3,i} \vee y_i)(x_i \vee \overline{y_i} \vee z_i)$$

- $\mathsf{TA}(\lambda_{1,i}, t|_{\mathsf{Var}(\lambda_{1,i})}) = \mathsf{TA}(\lambda_{2,i}, t|_{\mathsf{Var}(\lambda_{2,i})}) = 0$. 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 $t' \in \mathsf{Val}(\mathsf{Var}(\psi))$, която NAE-удовлетворява ψ. Разглеждаме произволно $i \in \{1, \ldots, m\}$.

- Ако за поне един литерал $\lambda_{j,i}$, където $j \in \{1, 2, 3\}$, е вярно, че $\mathsf{TA}(\lambda_{j,i}, t|_{\mathsf{Var}(\lambda_{j,i})}) = 1$, очевидно ϕ_i е удовлетворена.
- Да допуснем, че $\mathsf{TA}(\lambda_{1,i}, t|_{\mathsf{Var}(\lambda_{1,i})}) = \mathsf{TA}(\lambda_{2,i}, t|_{\mathsf{Var}(\lambda_{2,i})}) = \mathsf{TA}(\lambda_{3,i}, t|_{\mathsf{Var}(\lambda_{3,i})}) = 0$. 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':

  $$\forall w \in \mathsf{Var}(\psi) : \overline{t'}(w) = \neg t'(w)$$

  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 $\overline{t'}|_{\mathsf{Var}(\phi)} \models \phi$.

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. □

16.2.2 MaxCut

Тегло на срез в граф е броят на ребрата, прекосяващи среза (Определение 82). 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 [9]. 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 [59].

Изч. Задача 70: MaxCut, версия за разпознаване

екземпляр: Неориентиран мултиграф G, естествено число k.
въпрос: Дали в G има срез с тегло $\geqslant k$?

Редукция 13: NAESAT $\leqslant_p$ 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 $\geqslant k$. Нека $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$ 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

$$V = \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$$

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 $\overline{x_i}$ във ϕ, за всяко $i \in \{1, \ldots, n\}$. 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 $\overline{x_2}$ в клауза на ϕ води до поява на **ново** ребро с краища x_1 и $\overline{x_2}$ 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 $\overline{x_i}$.

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 $\geqslant 5m$. Ще докажем, че ϕ е NAE-удовлетворима.

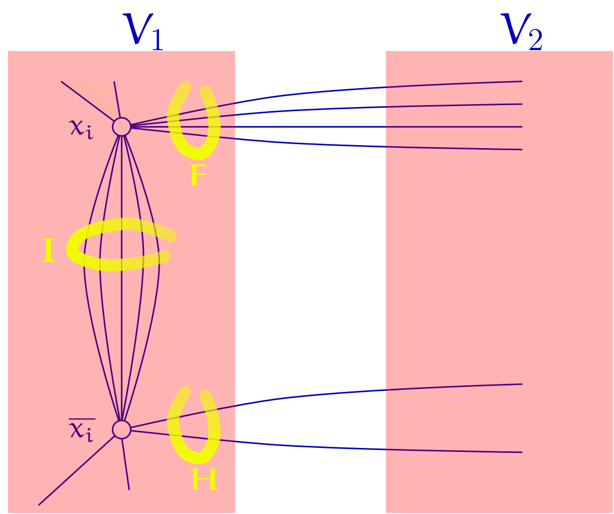
Лема 67

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 $\overline{x_i}$ са в единия дял; БОО, нека са във V_1. Нека

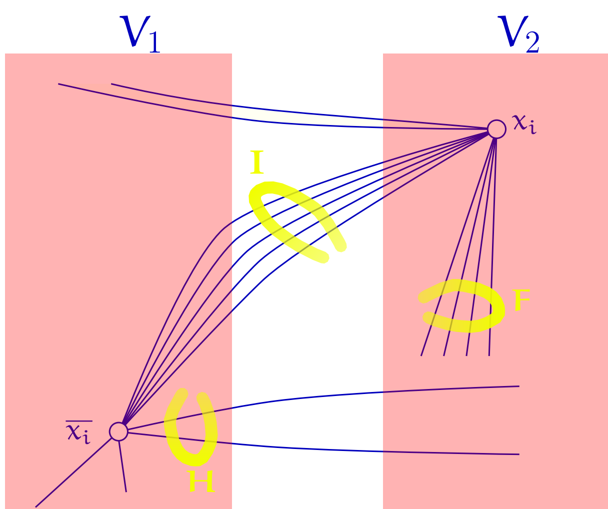
$$I = \{e \in E \mid f(e) = \{x_i, \overline{x_i}\}\}$$
$$F = \{e \in E \mid \exists \mu \in V_2 : (f(e) = \{x_i, \mu\})\}$$
$$H = \{e \in E \mid \exists \mu \in V_2 : (f(e) = \{\overline{x_i}, \mu\})\}$$

Щом x_i и $\overline{x_i}$ са във V_1, ребрата от I не допринасят за теглото на среза. Ребрата, инцидентни с x_i или $\overline{x_i}$ 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 $\overline{x_i}$ се среща в q клаузи на ϕ. По дефиницията на КНФ, x_i и $\overline{x_i}$ 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 $\overline{x_i}$ във ϕ поражда в графа две ребра, инцидентни с $\overline{x_i}$. Имайки предвид това, че x_i и $\overline{x_i}$ не се срещат заедно в клауза, заключаваме, че в графа има $2p + 2q$ ребра, инцидентни с точно единия от x_i и $\overline{x_i}$.

Ясно е, че $|F| \leqslant 2p$, понеже е възможно ребра, инцидентни с x_i, да не са във F. Аналогично, $|H| \leqslant 2q$, понеже е възможно ребра, инцидентни с $\overline{x_i}$, да не са в H. Заключаваме, че $|F| + |H| \leqslant 2\ell_i$, така че сумарният принос на x_i и $\overline{x_i}$ 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 $|F| \leqslant \ell_i$, 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 $\geqslant 5m$, 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 $\geqslant 5m$. □

Съгласно Лема 67, БОО допускаме, че за всяка двойка противоположни литерали-върхове, единият е в единия дял, а другият е в другия дял. Всяка $\{x_i, \overline{x_i}\}$ е свързана със сноп от ℓ_i ребра, които са прекосяващи среза ребра при текущите допускания. Тогава имаме $\sum_{i=1}^{n} \ell_i$ прекосяващи среза, свързващи противоположни литерали, ребра, а $\sum_{i=1}^{n} \ell_i = 3m$ (16.2).

Останалите $\geqslant 2m$ прекосяващи среза 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 среза, 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. Заключаваме, 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 слдната $t \in \mathsf{Val}(\mathsf{Var}(\phi))$. 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 $\overline{x_i}$ е във V_1, то $t(x_i) = 0$.

Помним, че в нито един дял не се съдържа противоложни 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 литирал 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 $\geqslant 5m$, 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. □

Ето пример. Да разгледаме следната КНФ ϕ, която често ще ползваме за пример:

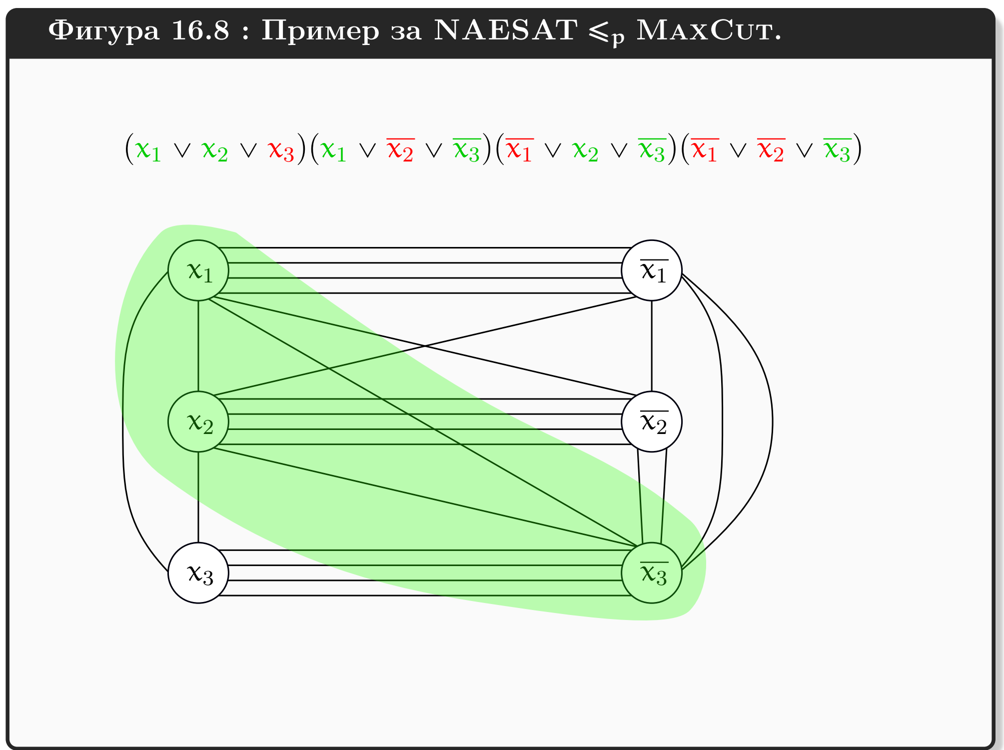
$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2} \underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3} \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4} \tag{16.3}$$

ϕ е NAE-удовлетворена от валюацията $\langle 1, 1, 0 \rangle$:

$$(x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

Фигура 16.8 показва съответния мултиграф и срез с тегло $5 \times 4 = 20$. Единият дял на среза 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 NAESAT $\leqslant_p$ MaxCut.



16.2.3 Vertex Cover

В Лекция 12 видяхме задачата Vertex Cover в оптимизационна версия: Задача 45. Във версия за разпознаване, задачата е следната.

Изч. Задача 71: Vertex Cover, версия за разпознаване

екземпляр: Неориентиран граф G, естествено число k.
въпрос: Дали G има върхово покриване U, такова че $|U| \leqslant k$?

В тази лекция имаме предвид задачата само във версия за разпознаване. В тази версия, задачата е **NP**-пълна.

Редукция 14: 3SAT $\leqslant_p$ Vertex Cover

Конструкция: Нека е даден екземпляр ϕ на 3SAT, като клаузите на ϕ са $\phi_1, \ldots, \phi_m$ и всяка клауза има точно три литерала. Нека $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$. Да кажем, че за $j \in \{1, \ldots, m\}$, $\phi_j = (\lambda_{1,j} \vee \lambda_{2,j} \vee \lambda_{3,j})$. 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 $\leqslant k$ тстк ϕ е удовлетворима.

Множеството от върховете е

$$V = \{u_i^+ \,|\, 1 \leqslant i \leqslant n\} \cup \{u_i^- \,|\, 1 \leqslant i \leqslant n\} \cup \{z_{i,j} \,|\, i \in \{1, 2, 3\}, j \in \{1, \ldots, m\}\}$$

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

$$E = \left\{ (u_i^+, u_i^-) \,|\, 1 \leqslant i \leqslant n \right\} \cup \left\{ (z_{i,j}, z_{k,j}) \,|\, 1 \leqslant i < \ell \leqslant 3, 1 \leqslant j \leqslant m \right\} \cup \bigcup_{i=1}^{m} E'_j$$

където, за $j \in \{1, \ldots, m\}$,

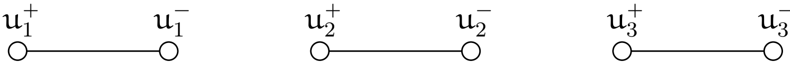
$$E'_j = \{(z_{1,j}, a_j), (z_{2,j}, b_j), (z_{3,j}, c_j)\}$$

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 $a_j = u_s^-$.

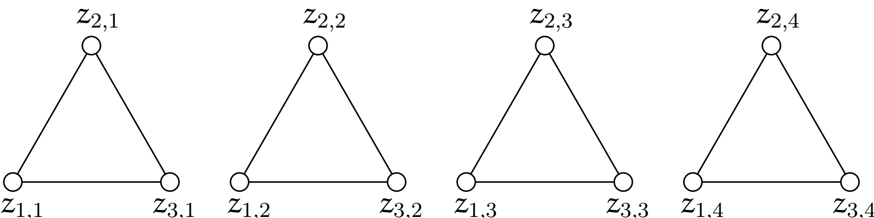
За удобство казваме, че ребрата от $\bigcup_{i=1}^{m} E'_j$ са *сините ребра*, а останалите ребра са *черните ребра*.

Преди да довършим конструкцията (все още не сме определили k), да илюстрираме с пример нещата дотук. Да разгледаме отново ϕ от (16.3):



$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2} \underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3} \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4}$$

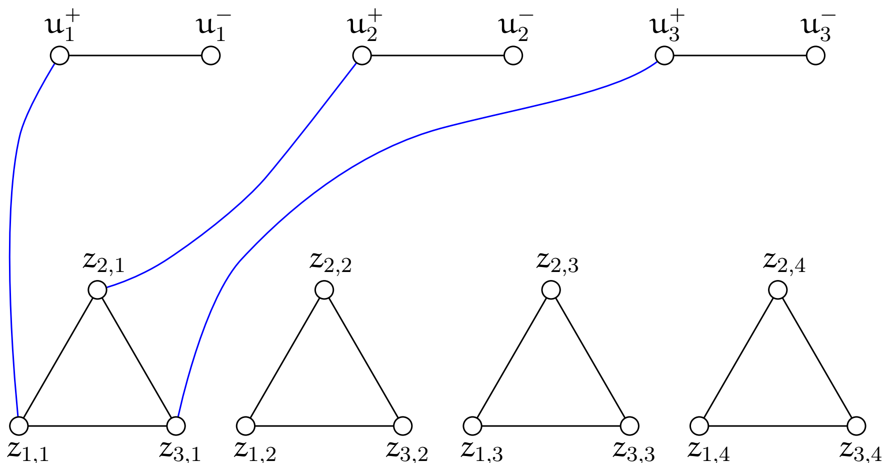
Променливите са x_1, x_2 и x_3 и има четири клаузи ϕ_1, ..., ϕ_4. Конструираме върхове u_1^+, u_1^-, u_2^+, u_2^-, u_3^+ и u_3^-, съответни на променливите, като конструираме черни ребра (u_1^+, u_1^-), (u_2^+, u_2^-) и (u_3^+, u_3^-). Освен това, за $1 \leqslant j \leqslant 4$, конструираме върхове $z_{1,j}$, $z_{2,j}$, $z_{3,j}$, с черно ребро между всеки два от тях. Това са 3-клики, съответни на клаузите. Игнорирайки ребрата от $\bigcup_{i=1}^{4} E'_j$ (сините ребра), картината е такава:



Да конструираме ребрата от $\bigcup_{i=1}^{4} E'_j$, 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.

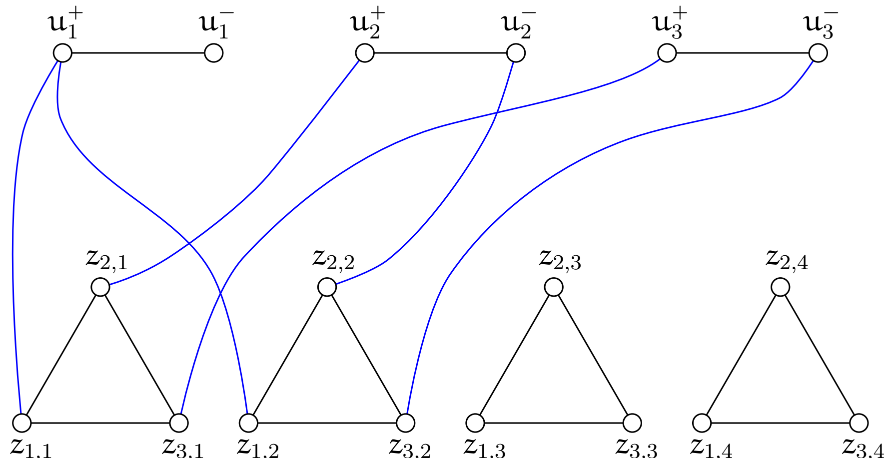
Добавяме ребрата от E_1' към предната илюстрация.



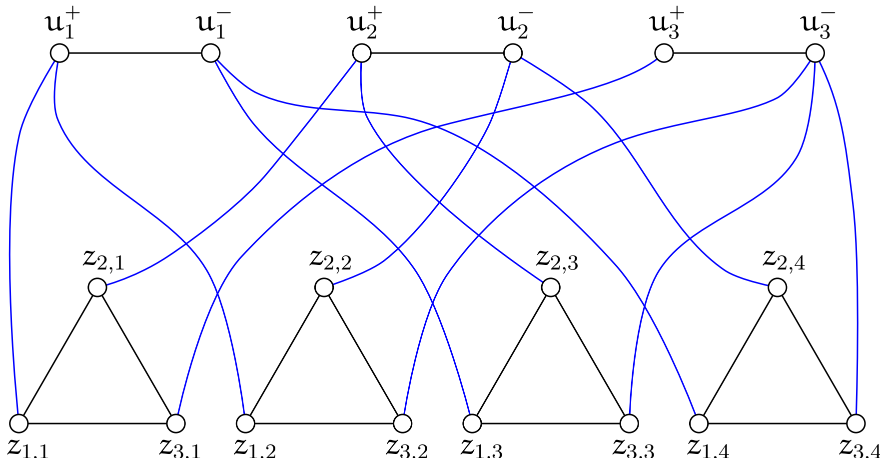
Да конструираме E_2'. То се състои от три ребра:

- 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 $\overline{x_2}$,
- и ребро $(z_{3,2}, u_3^-)$ заради третия литерал във ϕ_1, а именно $\overline{x_3}$.

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 конструкцията, 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 87: Заради сините ребра трябва координация между "горе" и "долу"

За всяко $j \in \{1, \ldots, m\}$, 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 U е произволно върхово покриване на G с мощност $2m + n$. Нека

$$\hat{U} = U \cap (\{u_i^+ \mid 1 \leqslant i \leqslant n\} \cup \{u_i^- \mid 1 \leqslant i \leqslant n\})$$

На прост български, $\hat{U}$ са "горните" върхове в покриването. Както вече забелязахме, $|\hat{U}| = n$ и всеки връх в $\hat{U}$ е точно единият от u_i^+ и u_i^-, по всички $i \in \{1, \ldots, n\}$.

Във всяка 3-клика "долу" има точно един връх, който не е в U. Да наречем съвкупността от тези върхове "долу", *важните върхове*. Съгласно Наблюдение 87, всеки важен връх е съсед на точно един връх от $\widehat{U}$. По този начин за всяка клика "долу" има точно един съответен връх "горе", който е в $\widehat{U}$. Това съответствие задава функция h с домейн множеството от 3-кликите и кодомейн $\widehat{U}$. В общия случай, h очевидно не е инекция, но също така не е и сюрекция, понеже е възможно върхове от $\widehat{U}$ да не са съседи на важни върхове[†].

Да разгледаме следната валюация t на $\mathsf{Var}(\phi)$: за всяко $i \in \{1, \ldots, n\}$, ако $u_i^+ \in \widehat{U}$, то $t(x_i) = 1$, а ако $u_i^- \in \widehat{U}$, то $t(x_i) = 0$. Тъй като точно единият от $\{u_i^+, u_i^-\}$ е в $\widehat{U}$, 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 $\widehat{U}$. За всяко $j \in \{1, \ldots, m\}$, $h(\{z_{1,j}, z_{2,j}, z_{3,j}\})$ съответства на литерала, от който клаузата ϕ_j бива удовлетворена:

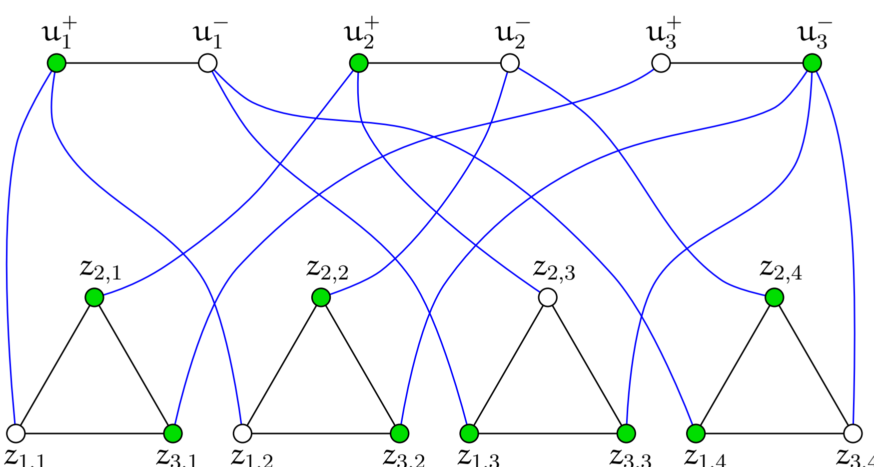
- ако $h(\{z_{1,j}, z_{2,j}, z_{3,j}\}) = u_i^+$, за някое $i \in \{1, \ldots, n\}$, то ϕ_j съдържа положителния литерал x_i и, тъй като $t(x_i) = 1$, вярно е, че $\mathsf{TA}(\phi_j, t) = 1$.

- ако $h(\{z_{1,j}, z_{2,j}, z_{3,j}\}) = u_i^-$, за някое $i \in \{1, \ldots, n\}$, то ϕ_j съдържа отрицателния литерал $\overline{x_i}$ и, тъй като $t(x_i) = 0$, вярно е, че $\mathsf{TA}(\phi_j, t) = 1$.

Доказахме, 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.

$$\phi = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

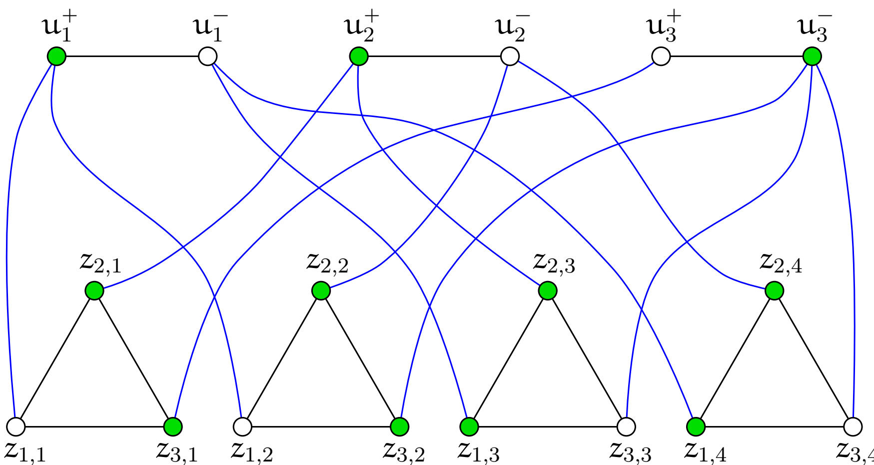


Забележете, че клауза може да бъде удовлетворена чрез повече от един литерал при дадена валюация. Примерно, щом $t(x_2) = 1$ и $t(x_3) = 0$, третата клауза се удовлетворява както чрез положителния литерал x_2, така и чрез отрицателния литерал $\overline{x_3}$. Но в графа точно един връх

[†]Представете си КНФ, в която един и същи литерал, да кажем x_1, се намира във всяка клауза. Тогава за $j \in \{1, \ldots, m\}$, 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 потриването; 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 $\overline{x_3}$. Отрицателният $\overline{x_3}$ щеше да е зеленият литерал в третата клауза, ако бяхме избрали $z_{3,3}$ да не е в покриването. Можехме ли да направим това? Да, разбира се, ето.

$$\phi = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$



Показахме 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 88

В контекста на редукцията 3SAT $\leqslant_p$ 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 показахме 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 46 и Задача 49. Ето трите задачи във версия за разпознаване.

Изч. Задача 72: INDEPENDENT SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G, естествено число k.
въпрос: Дали G има антиклика U, такава че $|U| \geqslant k$?

Изч. Задача 73: CLIQUE, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G, естествено число k.
въпрос: Дали G има клика U, такава че $|U| \geqslant k$?

Изч. Задача 74: DOMINATING SET, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Неориентиран граф G, естествено число k.
въпрос: Дали G има доминиращо множество U, такова че $|U| \leqslant k$?

Редукция 15: VERTEX COVER $\leqslant_p$ 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 67. □

Редукция 16: INDEPENDENT SET $\leqslant_p$ CLIQUE

Конструкция: Даден е произволен екземпляр $\langle G, k \rangle$ на INDEPENDENT SET. Построяваме екземпляр $\langle \overline{G}, k \rangle$ на CLIQUE. Твърдим, че $\langle G, k \rangle$ е ДА-екземпляр на INDEPENDENT SET тстк $\langle \overline{G}, k \rangle$ 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 $\overline{G}$. □

Редукция 17: VERTEX COVER $\leqslant_p$ 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 $\leqslant k$ тстк G' има доминиращо множество с мощност $\leqslant k$.

Нека $V = \{v_1, \ldots, v_n\}$ и $E = \{e_1, \ldots, e_m\}$. 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 $z_1, \ldots, z_m$: такива, които не присъстват във V. Конструираме

$$V' = V \cup \{z_1, \ldots, z_m\}$$
$$\forall j \in \{1, \ldots, m\} : E'_j = \{(u, z_j), (v, z_j)\}, \text{ където } f(e_j) = \{u, v\}$$
$$E' = \bigcup_{j=1}^{m} E'_j$$

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 G 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 $\{z_1, \ldots, z_m\}$.

- 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 68.

- За всеки връх z_j, $1 \leqslant j \leqslant m$, 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 $U \subseteq \{z_1, \ldots, z_m\}$; ерго, $U \cap V = \emptyset$, а $\emptyset$ няма как да е покриващо множество на G, който няма изолирани върхове. Да разгледаме множеството $\widetilde{E} = \{(u, v) \in E \,|\, u \notin U \text{ и } v \notin U\}$.

Наблюдение 89

В текущия контекст е вярно, че за всяко ребро $e_j \in \widetilde{E}$, 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 $\widetilde{E}$ са причината $U \cap V$ да не е върхово покриване на G: всяко от тях е "непокрито" и от двата края. Ако $\widetilde{E} = \emptyset$, $U \cap V$ е върхово покриване на G и доказателството е готово. Да си представим изпълнението на следния алгоритъм в текущия контекст.

```
ALG-DS-VC()
1  while Ẽ ≠ ∅ do
2      e_j е произволно ребро от Ẽ
3      Ẽ ← Ẽ\{e}
4      U ← U\{z_j}
5      if f(e_j) ∩ U = ∅
6          x е произволен връх от f(e_j)
7          U ← U ∪ {x}
8  return
```

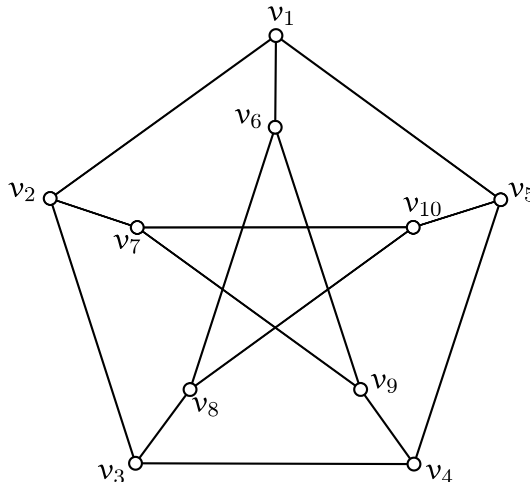
Инвариант за цикъла (редове 1–7) е:

> При всяко достигане на ред 1, U е покриващо множество за $G - \widetilde{E}$ и $|U| \leqslant k$.

Доказателството е тривиално и остава за читателя. Доказателството използва Наблюдение 89, което обосновава присъствието на z_j в U преди изпълнението на ред 4. Забележете, че проверката дали $f(e_j) \cap U = \emptyset$ на ред 5 има смисъл! Наистина, поначало $\widetilde{E}$ са ребра, чиито краища не са в U, обаче е възможно при предишна итерация на цикъла на алгоритъма, единият край на e_j да е бил сложен в U.

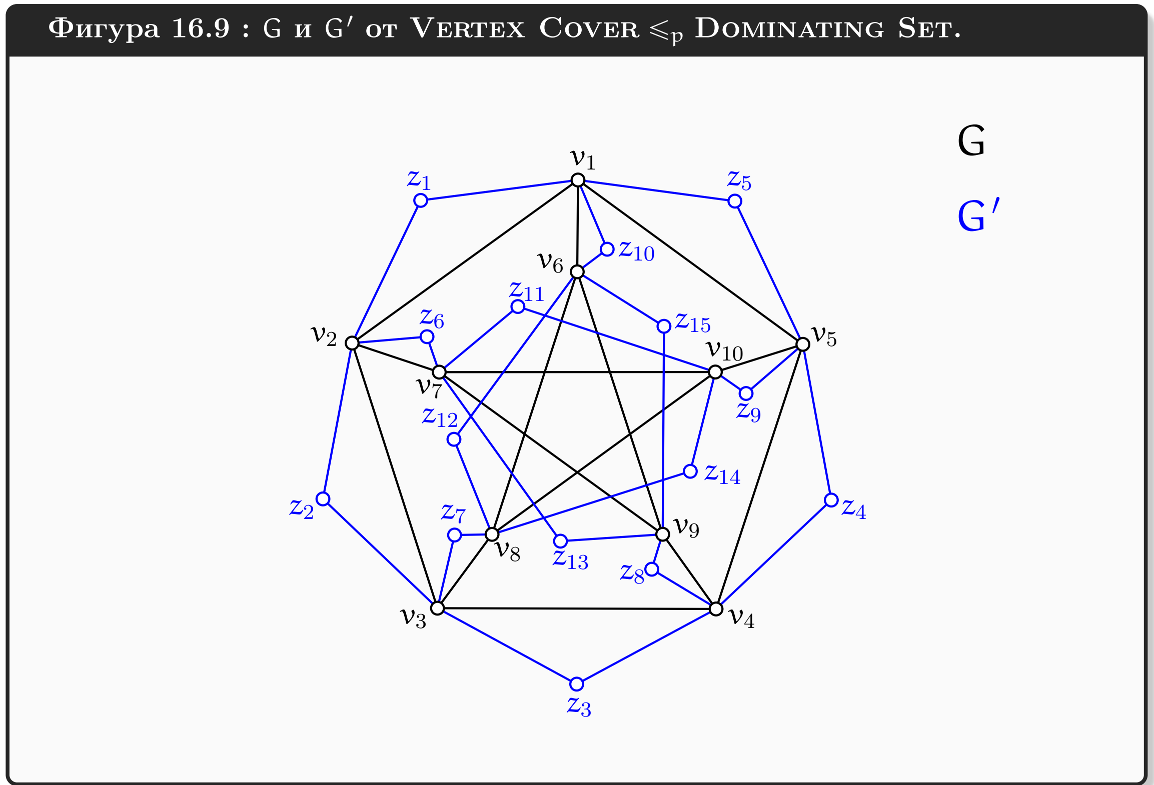
При терминирането на алгоритъма е изпълнено $\widetilde{E} = \emptyset$. Съгласно инварианта, U е покриващо множество на G, и освен това $|U| \leqslant k$. □

За илюстрация ще използваме графа на Petersen:

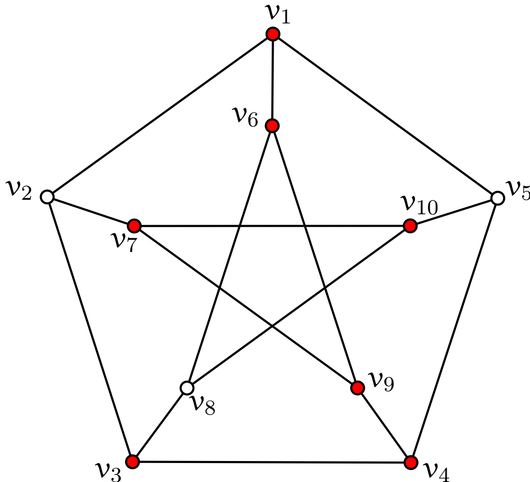


Фигура 16.9 показва конструкцията върху графа на Petersen. G от екземпляра на Vertex Cover е в черно, а G' от екземпляра на Dominating Set се получава от G с добавяне на сините върхове и ребра.

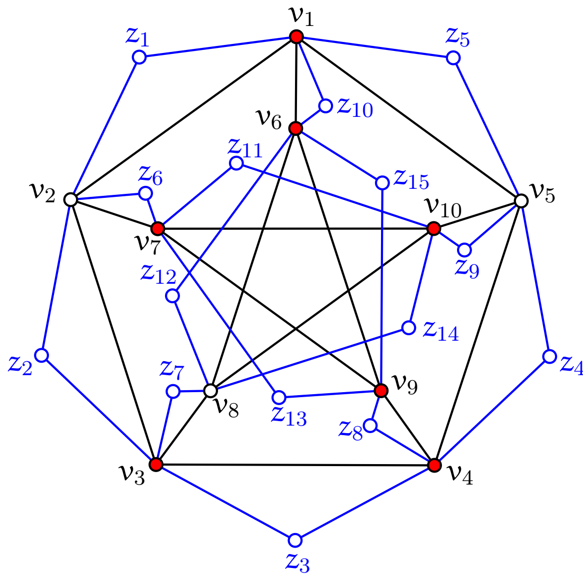
Фигура 16.9 : G и G′ от Vertex Cover $\leqslant_p$ Dominating Set.



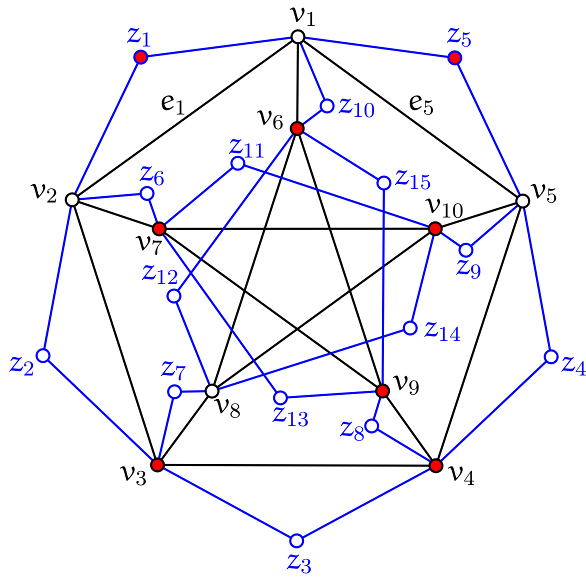
Ето върхово покриване на G със седем върха (нарисувани са в червено). Както знаем (вижте разсъжденията на стр. 588), това не е оптимално, понеже $\tau(G) = 6$. Но няма изискване екземплярът на задачата, от която свеждаме, да е оптимален. Важното е, че $\langle G, 7 \rangle$, където G е графът на Petersen, е Да-екземпляр на Vertex Cover.



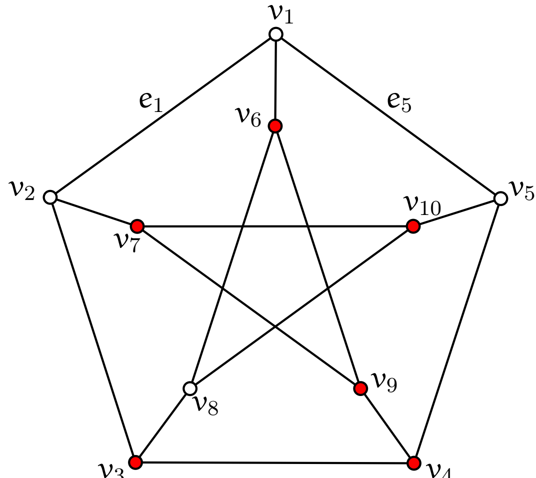
В графа G′, същите върхове са доминиращо множество:



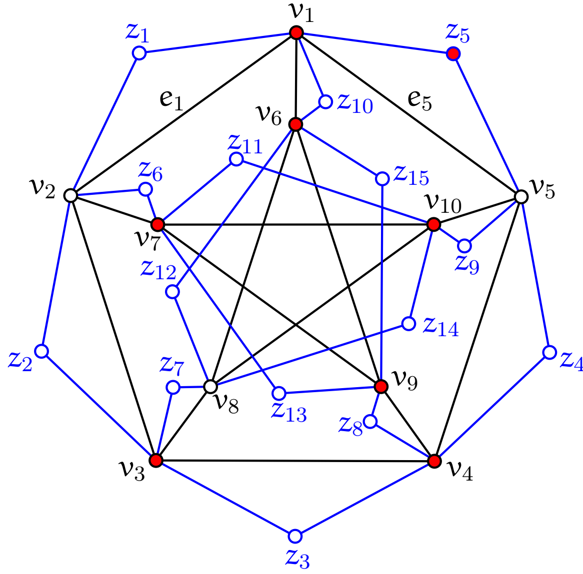
В обратната посока, да разгледаме доминиране на G' от осем върха:



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 $\widetilde{E}$ съдържа точно тях. Това се вижда по-ясно, ако разгледаме само G плюс върховете от доминиращото множество, които са и върхове на G:

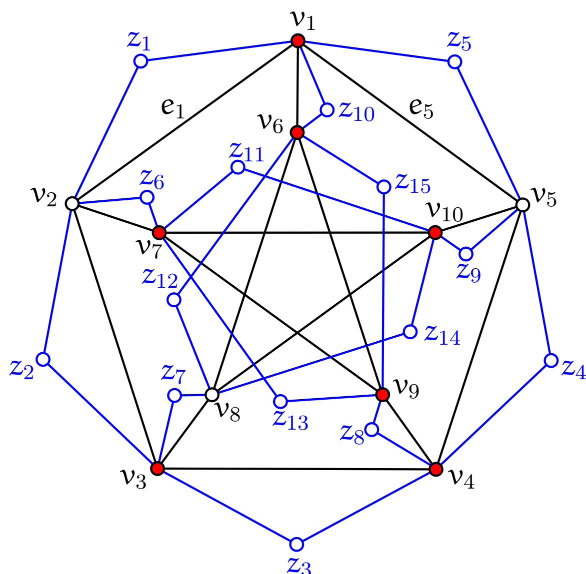


За да се справим с проблема, изпълняваме алгоритъма. Нека първо разгледаме e_1 на ред 2. Махаме e_1 от $\widetilde{E}$, махаме z_1 от U и вкарваме в U единият от съседите му; да кажем, v_1:



Странична забележка: след всяко такова "прехвърляне" на връх, U остава доминиращо множество на G'. В инварианта на алгоритъма не включихме това съждение, понеже там само искахме да покажем, че U след терминирането е покриване на G.

Реброто e_5 вече не е проблемно, но, ако следваме стриктно алгоритъма, трябва да изпълним още една итерация, понеже $\widetilde{E} = \{e_5\}$ е непразно. Махаме e_5 от $\widetilde{E}$ и махаме z_5 от U. Но $f(e_5) \cap U$ в този момент е $\{v_1\}$, така че редове 6 и 7 не се изпълняват. Алгоритъмът приключва, като 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 68: 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 $\leqslant k$, то G задължително има доминиращо множество с мощност $\leqslant k$.

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 $\leqslant k$, 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 $c_1, \ldots, c_n$ са реални константи, а $x_1, \ldots, x_n$ са реалните променливи.

Дадени са m ограничения

$$\sum_{k=1}^{n} a_{1,k} x_k \leqslant b_1$$
$$\sum_{k=1}^{n} a_{2,k} x_k \leqslant b_2$$
$$\dots$$
$$\sum_{k=1}^{n} a_{m,k} x_k \leqslant b_m$$

където $a_{i,k}$ са реални константи за $1 \leqslant i \leqslant m$ и $1 \leqslant k \leqslant n$, а $b_1, \ldots, b_m$ също са реални константи. Има и още ограничения: $x_k \geqslant 0$ за $1 \leqslant k \leqslant n$. 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 [115, стр. 5–6].

Огромен брой житейски и математически задачи може да се формулират като задачи за линейно програмиране [115], [57]. 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 изполване 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 $\mathbb{Z}^n$.

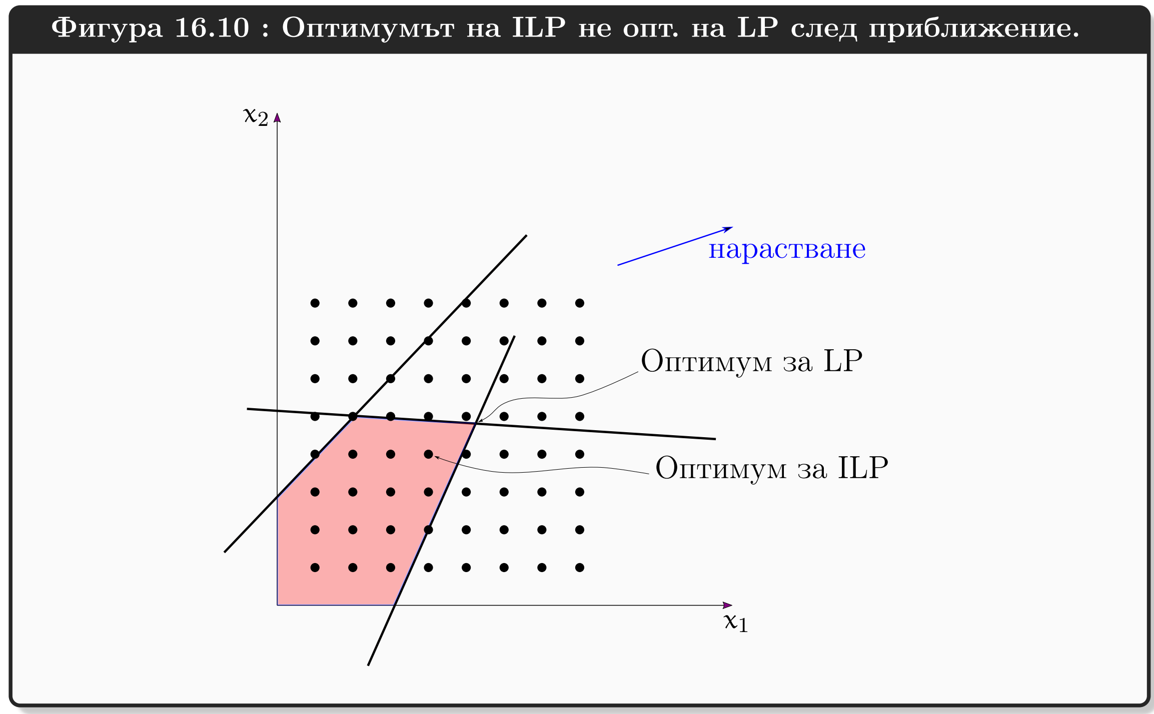
На пръв поглед може да използваме подходящ алгоритъм за линейно програмиране и после да закръглим $x_1, \ldots, x_n$ до най-близките цели числа. Тази идея се разисква подробно в [115, стр. 307–308]. Понякога тя работи, особено ако става дума за големи дробни числа, които са "не особено чувствителни" 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 [115, 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 перпендикулярният вектор на $c_1x_1 + c_2x_2 = \alpha$, $\alpha = \text{const}$.

Фигура 16.10 : Оптимумът на ILP не 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 контитуалната 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, можа 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 75: 0-1 Integer Programming

екземпляр: Нека $x_1, \ldots, x_n$ са булеви променливи. Дадени са m неравенства

$$\sum_{k=1}^{n} a_{1,k}x_k \leqslant b_1$$
$$\sum_{k=1}^{n} a_{2,k}x_k \leqslant b_2$$
$$\ldots$$
$$\sum_{k=1}^{n} a_{m,k}x_k \leqslant b_m$$

където $a_{i,k}$ са цели константи за $1 \leqslant i \leqslant m$ и $1 \leqslant k \leqslant n$ и $b_1, \ldots, b_m$ също са цели константи.

въпрос: Дали съществува булева функция $h : \{x_1, \ldots, x_n\} \rightarrow \{0, 1\}$, такава че всички ограничения са удовлетворени при нейните стойности на променливите?

Редукция 18: 3SAT $\leqslant_p$ 0-1 Integer Programming

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

Нека $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$. Конструираме съответен екземпляр ψ на 0-1 Integer Programming. Множеството от променливите на ψ е същото, а именно $\{x_1, \ldots, x_n\}$. Конструираме $m \times n$ булеви константи $a_{i,k}$ за $i \in \{1, \ldots, n\}$ и $k \in \{1, \ldots, m\}$ така:

$$a_{i,k} = \begin{cases} 0, & \text{ако литерал на променливата } x_i \text{ не се среща в клаузата } \phi_k \\ 1, & \text{ако литералът } \overline{x_i} \text{ се среща в клаузата } \phi_k \\ -1, & \text{ако литералът } x_i \text{ се среща в клаузата } \phi_k \end{cases}$$

За $k \in \{1, \ldots, m\}$ 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):

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1}\underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2}\underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3}\underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4}$$

Променливите са x_1, x_2 и x_3 и има четири клаузи $\phi_1, \ldots, \phi_4$. Генерираме следната система

от неравенства:

$$
\begin{aligned}
-x_1 - x_2 - x_3 &\leqslant -1 + 0\\
-x_1 + x_2 + x_3 &\leqslant -1 + 2\\
+x_1 - x_2 + x_3 &\leqslant -1 + 2\\
+x_1 + x_2 + x_3 &\leqslant -1 + 3
\end{aligned}
$$

Да умножим по -1 неравенствата:

$$
\begin{aligned}
+x_1 + x_2 + x_3 &\geqslant +1 - 0\\
+x_1 - x_2 - x_3 &\geqslant +1 - 2\\
-x_1 + x_2 - x_3 &\geqslant +1 - 2\\
-x_1 - x_2 - x_3 &\geqslant +1 - 3
\end{aligned}
$$

Да прехвърлим във всяко неравенство минус-единиците вдясно в лявата страна с обратен знак:

$$
\begin{aligned}
&x_1 &&+\ x_2 &&+\ x_3 &&\geqslant 1\\
&x_1 &&+\ (1-x_2) &&+\ (1-x_3) &&\geqslant 1\\
&(1-x_1) &&+\ x_2 &&+\ (1-x_3) &&\geqslant 1\\
&(1-x_1) &&+\ (1-x_2) &&+\ (1-x_3) &&\geqslant 1
\end{aligned}
\tag{16.4}
$$

ϕ е удовлетворима, например от валюацията $t = \langle 1, 1, 0 \rangle$. Съответно, неравенствата се удовлетворяват от тези стойности на променливите:

$$
\begin{aligned}
&1 &&+\ 1 &&+\ 0 &&\geqslant 1 \quad \checkmark\\
&1 &&+\ (1-1) &&+\ (1-0) &&\geqslant 1 \quad \checkmark\\
&(1-1) &&+\ 1 &&+\ (1-0) &&\geqslant 1 \quad \checkmark\\
&(1-1) &&+\ (1-1) &&+\ (1-0) &&\geqslant 1 \quad \checkmark
\end{aligned}
$$

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 $t \in \mathsf{Val}(\mathsf{Var}(\phi))$, такава че във всяка клауза α на ϕ има поне един литерал λ, такъв че $\mathsf{TA}(\lambda, t) = 1$. 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 $t \in \mathsf{Val}(\mathsf{Var}(\phi))$ съществува клауза α на ϕ, такава че за всеки литерал $\lambda \in \alpha$ е вярно, че $\mathsf{TA}(\lambda, t) = 0$.

[†] 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 неравеството, 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. □

16.2.6 3-Colorability, k-Colorability и Planar 3-Colorability

3-Colorability ∈ NP-c. Да си припомним 3-Colorability.

Изч. Задача 76: 3-Colorability

екземпляр: Неориентиран граф G.
въпрос: Дали G е върхово оцветим с $\leqslant 3$ цвята?

Редукция 19: 3SAT $\leqslant_p$ 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 $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$. 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 $V_1 = \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$. Чрез тях конструираме множество E_1 от n ребра, нито две от които нямат общ край:

$$E_1 = \{(x_i, \overline{x_i}) \,|\, 1 \leqslant i \leqslant n\}$$

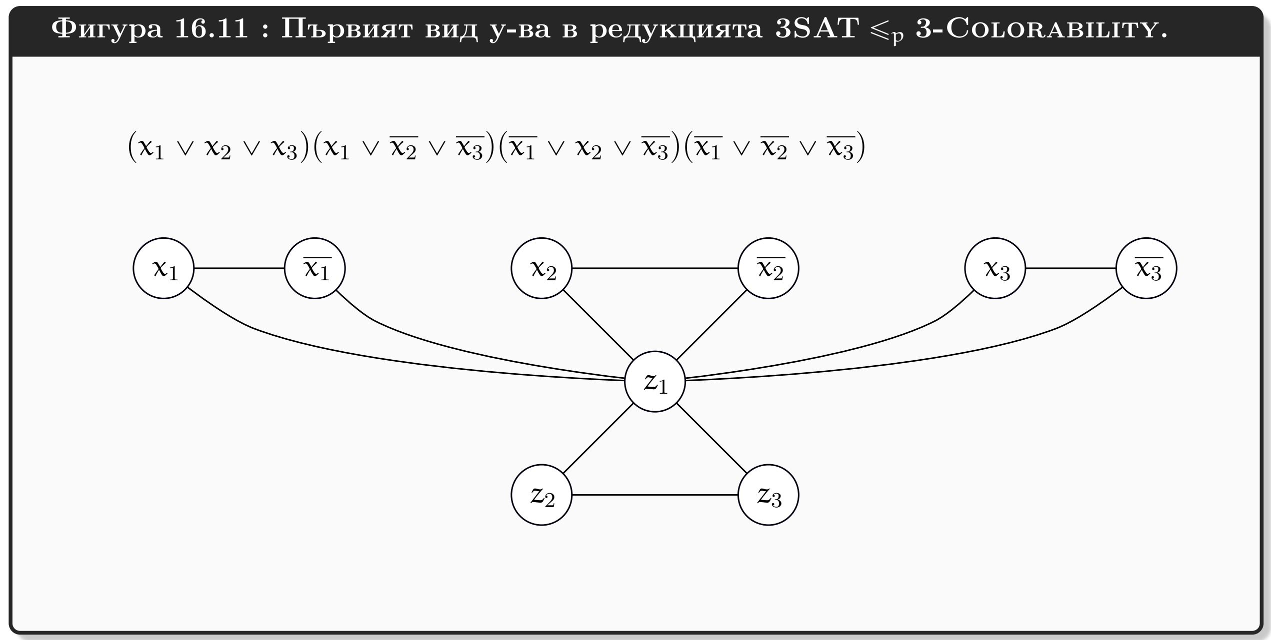
Добавяме три нови върха z_1, z_2 и z_3 и конструираме множество от ребра E_2:

$$E_2 = \{(x_i, z_1) \,|\, 1 \leqslant i \leqslant n\} \cup \{(\overline{x_i}, z_1) \,|\, 1 \leqslant i \leqslant n\} \cup \{(z_1, z_2), (z_1, z_3), (z_2, z_3)\}$$

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:

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1} \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2} \underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3} \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4}$$

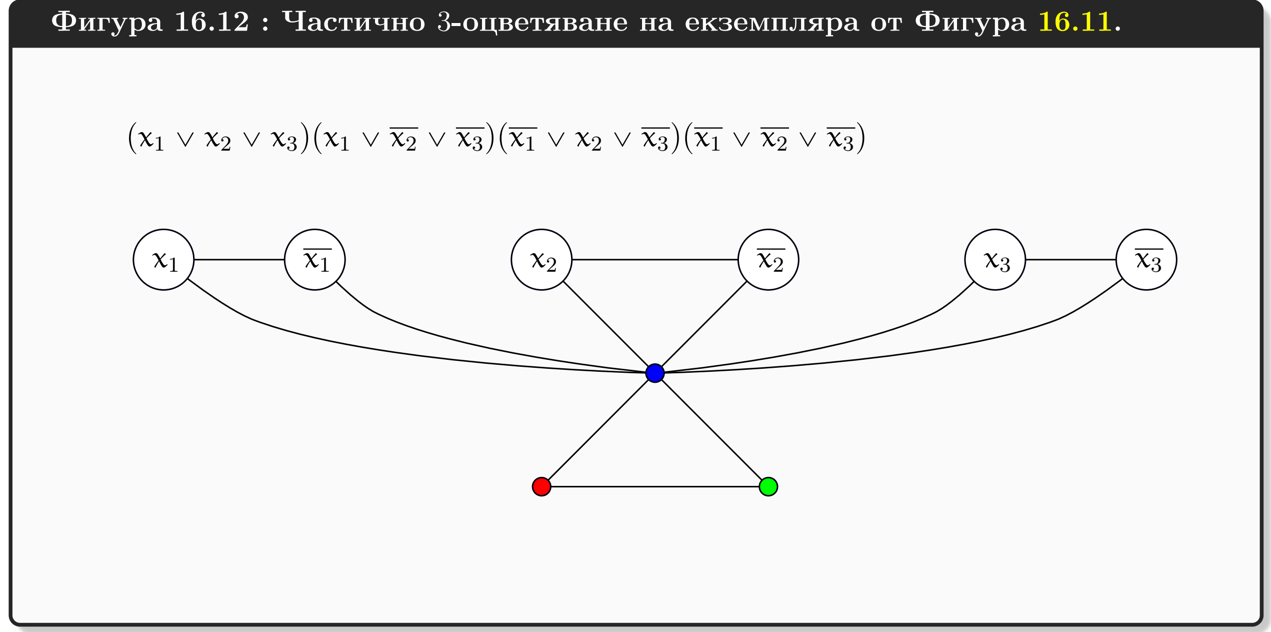
Фигура 16.11 показва четирите приспособления-триъгълници за този екземпляр.



Фигура 16.11 : Първият вид у-ва в редукцията 3SAT $\leqslant_p$ 3-Colorability.

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.

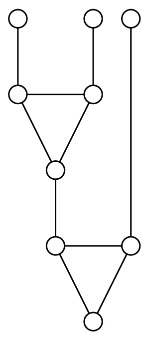
Читателят може би се досеща, че цветовете Истина и Лъжа са за върховете от V_1, понеже те отговарят на литерали, а литералите имат стойности 1 или 0 при дадена валюация. Тогава връх z_1, който е съсед на всички върхове от V_1, е Неутрален, което влече, че точно единият от z_2 и z_3 е Истина, а другият 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.

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.

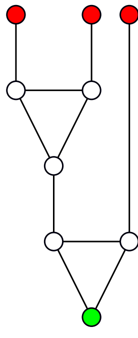


Тези приспособления се наричат *OR-gates*. Върхът от степен 2 (най-долу) се нарича *изходът*, а трите висящи върхове (най-горе) се наричат *входовете*. Сега ще стане ясно защо. Дефинираме, че оцветяване на OR-gate е *подходящо*, ако входовете и изходът получават само Истина или Лъжа; тоест, никога Неутрален. Очевидно това има смисъл, ако OR-gate-ът трябва да симулира логически елемент в двузначна логика.

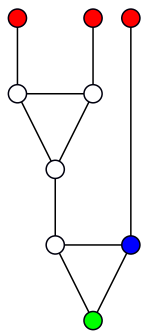
Лема 68

В текущия контекст, за всяко подходящо оцветяване на OR-gate, изхода е оцветен в Истина тстк поне единият вход е оцветен в Истина.

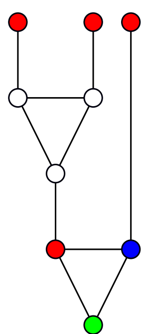
Доказателство: В едната посока, ще покажем, че е невъзможно изходът да е истина, а и трите входа да са лъжа. Да допуснем, че това е възможно.



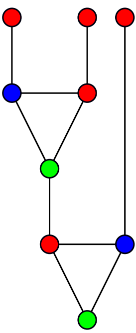
Върхът, който е между истината и лъжата, трябва да е неутрален:



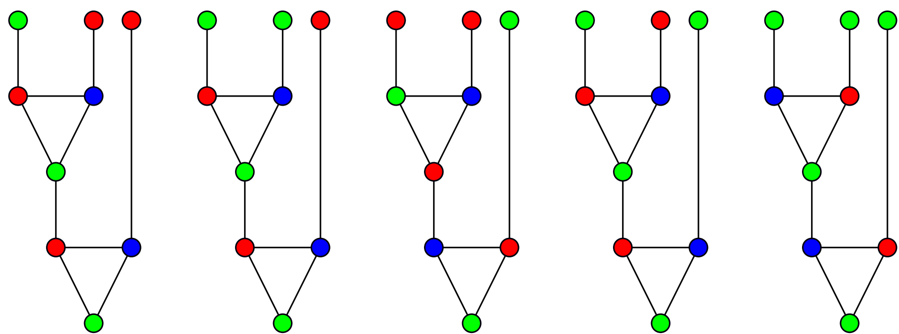
Тогава третият връх от долният триъгълник трябва да е лъжа:



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-а, а другите два края са z_1 и z_2.

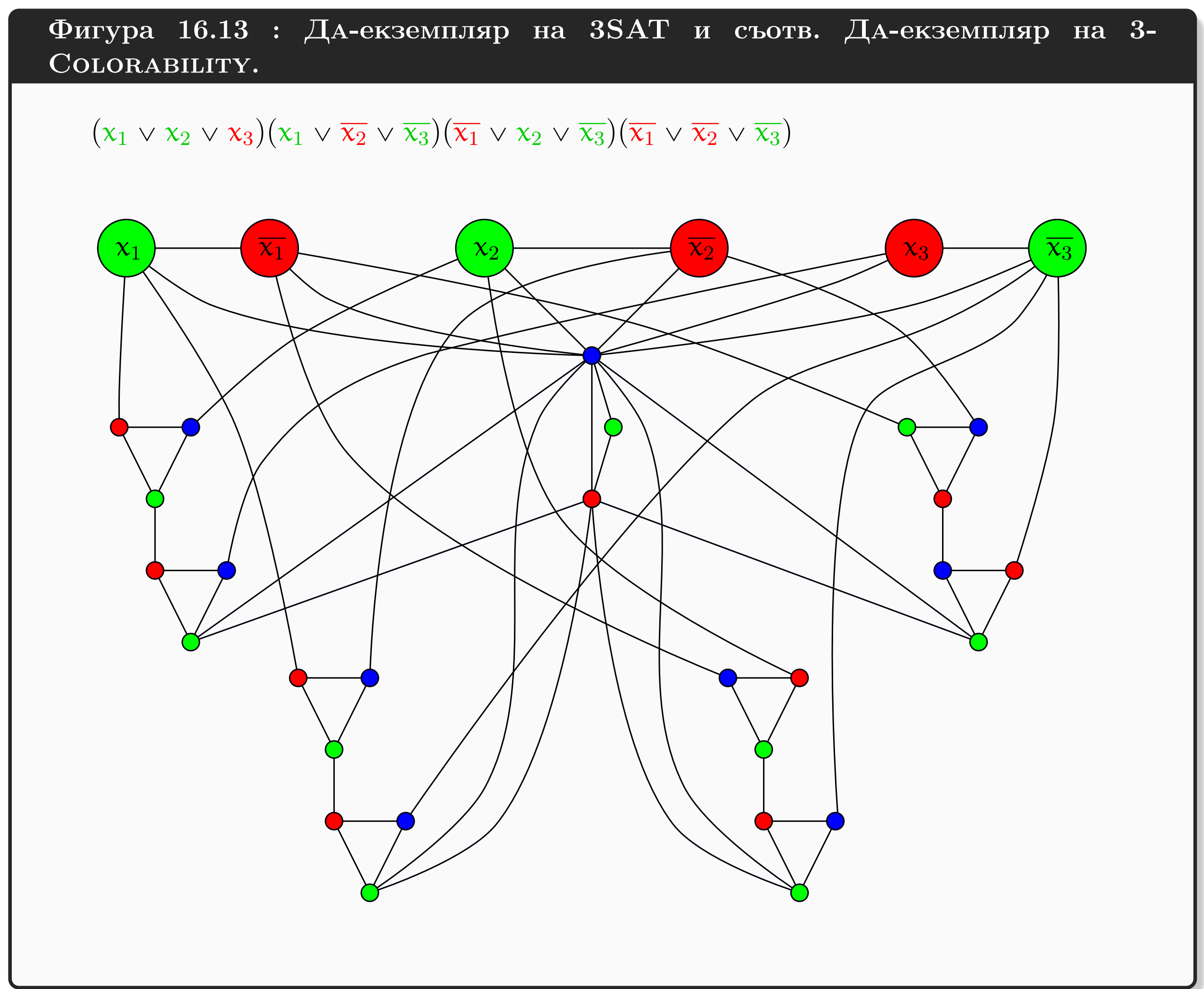
Обосновката на коректността е много лесна, ако си спомним, че z_1 е в цвят НЕУТРАЛЕН , а z_2 е в цвят ЛЪЖА. Това налага изходът на всеки OR-gate да е в цвят ИСТИНА. Ясно е, че всеки връх от V_1 е или истина, или лъжа, заради триъгълниците с два върха от V_1 и трети връх z_1, който е неутрален. Съгласно Лема 68, 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. □

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. Което, 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 валюацията $t = \langle 1, 1, 0 \rangle$ и на фигурата е показано съответно 3-оцветяване.

Фигура 16.13 : ДА-екземпляр на 3SAT и съотв. ДА-екземпляр на 3-COLORABILITY.



k-Colorability ∈ NP-c. Да си припомним k-Colorability.

Изч. Задача 77: k-Colorability

екземпляр: Неориентиран граф G, естествено число k.
въпрос: Дали G е върхово оцветим с $\leqslant k$ цвята?

Редукция 20: 3-Colorability $\leqslant_p$ 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. □

Аналогично, всеки екземпляр на 3SAT е и екземпляр на SAT. Буквално. Без "почти". Обаче редукцията SAT $\leqslant_p$ 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 85, че 3SAT, а не SAT, е **NP**-пълна, **то** доказателството за **NP**-пълнотата на SAT щеше да е много лесно, понеже редукцията 3SAT $\leqslant_p$ SAT е толкова тривиална, колкото 3-Colorability $\leqslant_p$ k-Colorability.

Planar 3-Colorability ∈ NP-c. Задачата е 3-Colorability, но върху планарни графи. Тъй като планарните графи са строго подмножество на графите, Редукция 19 не ни дава директно желания 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 78: Planar 3-Colorability

екземпляр: Неориентиран планарен граф G.
въпрос: Дали G е върхово оцветим с $\leqslant 3$ цвята?

Редукция 21: 3-Colorability $\leqslant_p$ 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 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 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 $p_1, \ldots, p_t$. Нека $P = \{p_1, \ldots, p_t\}$. Добре известно е, че $t \leqslant \binom{n}{4}$ и тази граница е точна. Съществено 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 $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$.

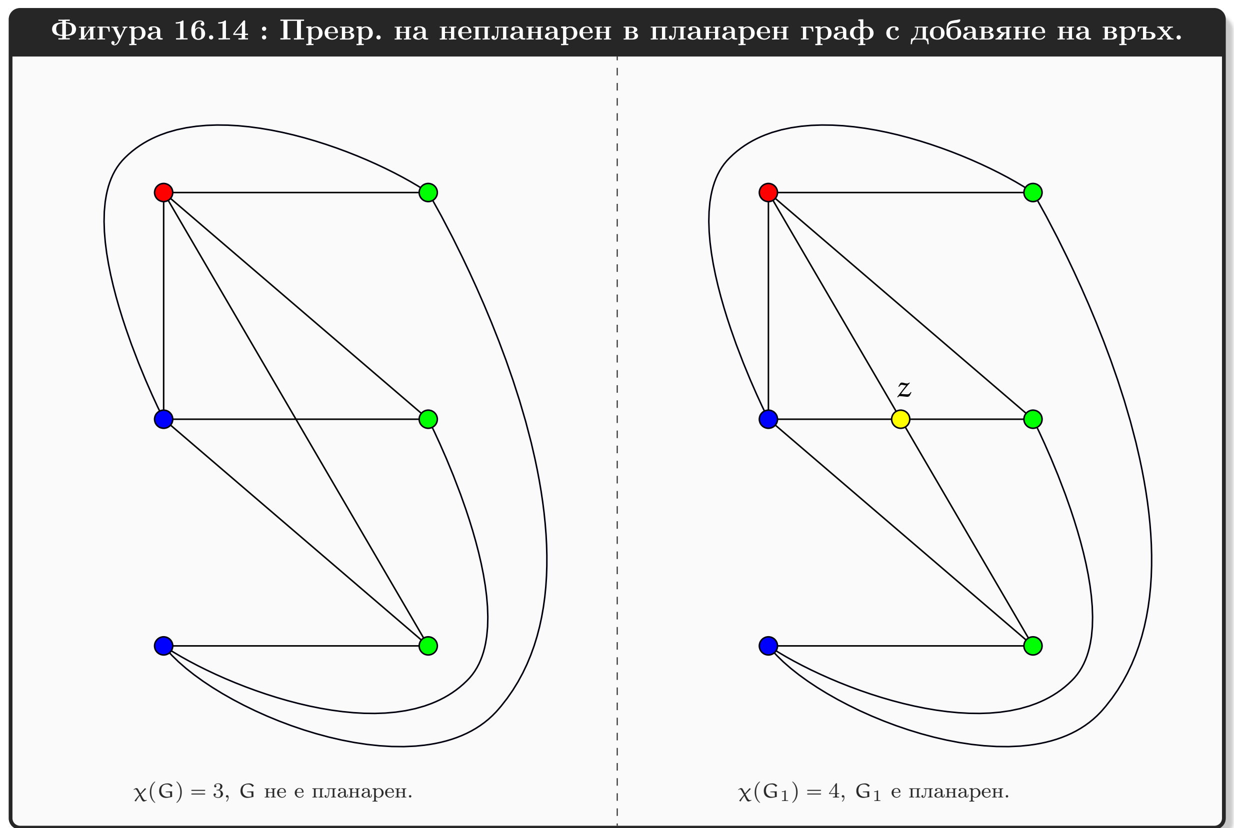
- $\mathcal{V}_1 = \mathcal{V} \cup \{p_1, \ldots, p_t\}$.

- Всяко ребро на $\mathcal{E}$ бива подразделено на $k + 1$ ребра от k-те точки измежду $p_1, \ldots, p_t$, които му принадлежат, където $k \geqslant 0$. $\mathcal{E}_1$ 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.



$\chi(G) = 3$, G не е планарен.

$\chi(G_1) = 4$, G_1 е планарен.

Допълнение 71: Намиране на хроматичното число чрез Maple (TM)

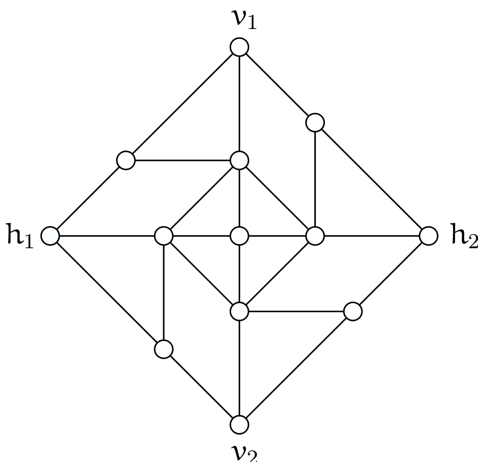
Хроматичното число на граф може да бъде изчислено с Maple (TM), стига графът да е достатъчно малък. Да вземем за пример G_1 от Фигура 16.14, който е достатъчно малък, така че да може да бъде оцветен оптимално от алогритъм с брутална сила. Командата на Maple (TM) за целта е `ChromaticNumber`. Преди това 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\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.
> 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. Затова ще трансформираме G по друг начин в планарен граф G_2, който начин запазва 3-оцветимостта. За целта въвеждаме това приспособление:



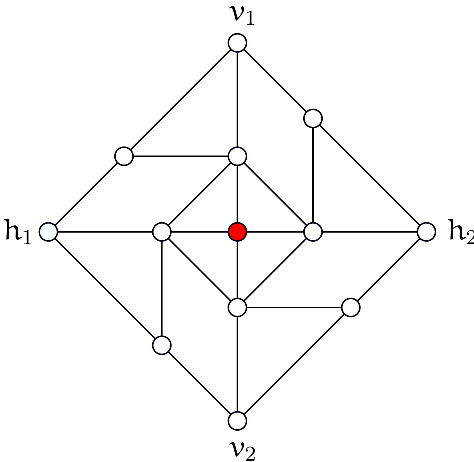
Приспособлението е планарен граф с 13 върха и 24 ребра. Четирите върха с имена на илюстрацията са *краищата* на приспособлението. Те се групират в две двойки на *вертикалните краища* v_1 и v_2, и *хоризонталните краища* h_1 и h_2. Останалите върхове остават анонимни. Ето важните за нашата конструкция свойства на приспособлението.

Факт 1 Хроматичното число на приспособлението е 3.

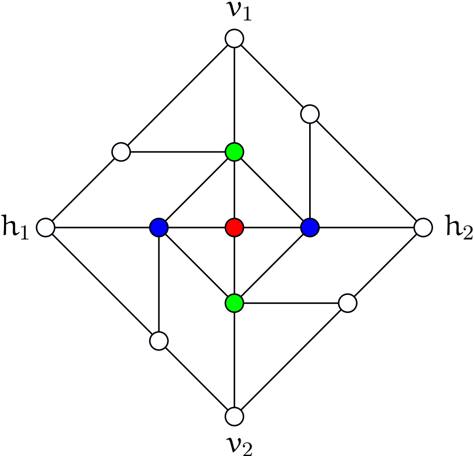
Факт 2 За всяко 3-оцветяване на приспособлението, вертикалните краища са в един и същи цвят c_v. Хоризонталните краища също са в един и същи цвят c_h, който може да съвпада или да не съвпада с c_v.

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 $c_v \neq c_h$.

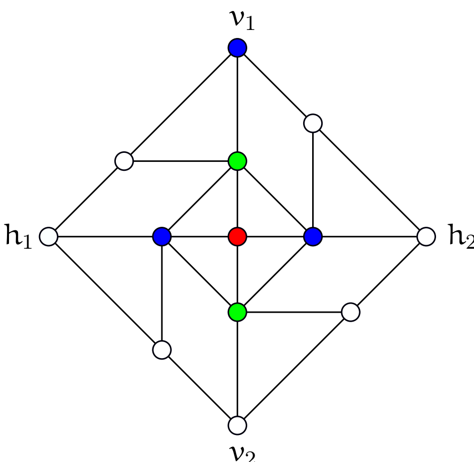
Ето конструктивно доказателство и на трите факта. Да използваме червен, зелен и син цвят. БОО, нека върхът в центъра е червен:



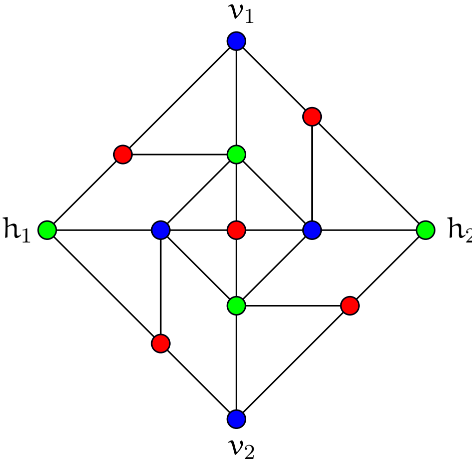
Това форсира син и зелен цвят върху съседите му, които алтернират по цикъла, който образуват. Да кажем:



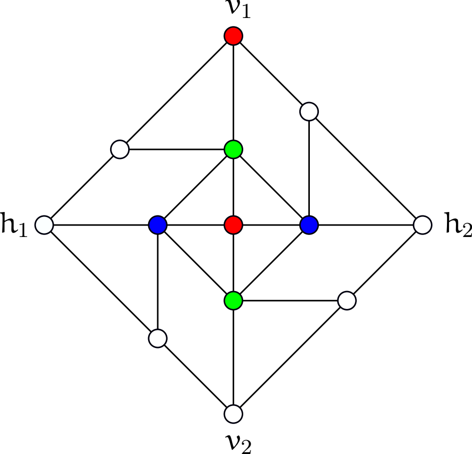
За v_1 има избор между синьо и червено. Първо да допуснем, че v_1 е син.



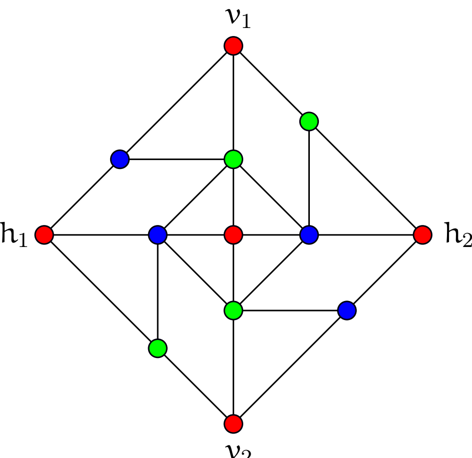
Това форсира оцветяването на останалите върхове.



Сега да допуснем, че v_1 е червен.

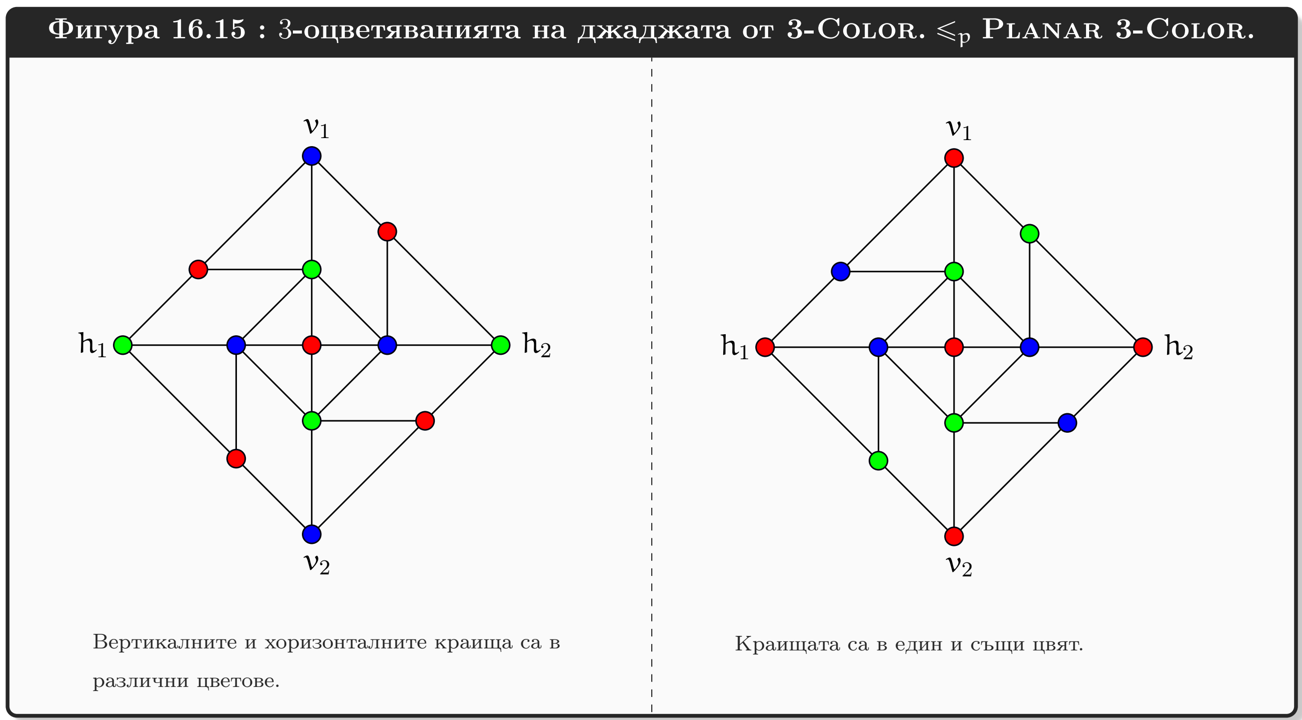


Това също форсира оцветяването на останалите върхове.

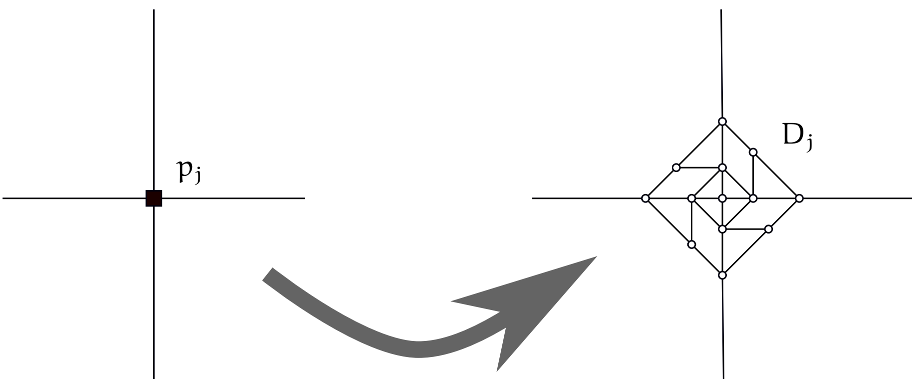


Фигура 16.15 показва двете възможни, с точност до симетрия, 3-оцветявания на приспособлението, ако централният връх е червен.

Фигура 16.15 : 3-оцветяванията на джаджата от 3-COLOR. $\leqslant_p$ PLANAR 3-COLOR.



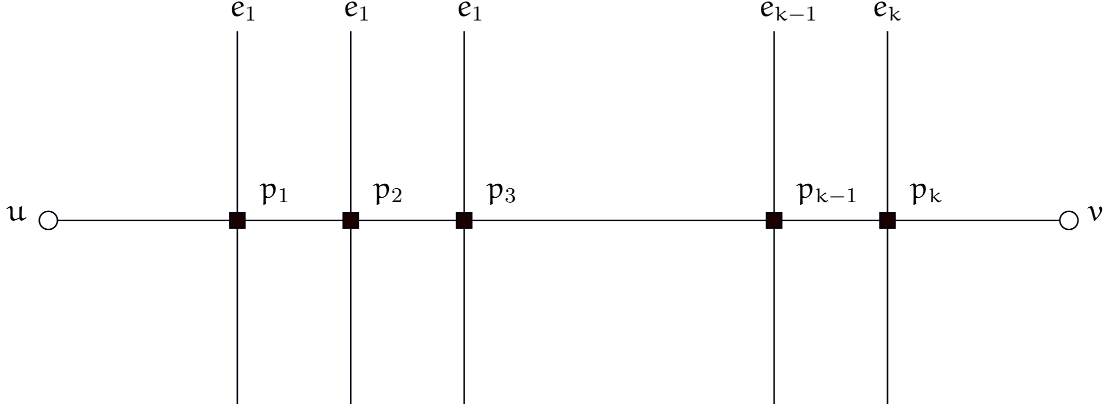
Ето обяснение на идеята на конструкцията, използваща приспособленията. За $j \in \{1, \ldots, t\}$, заменяме всяка p_j плюс достатъчно малка околност около нея с (планарното вписване на) едно приспособление D_j ето така:



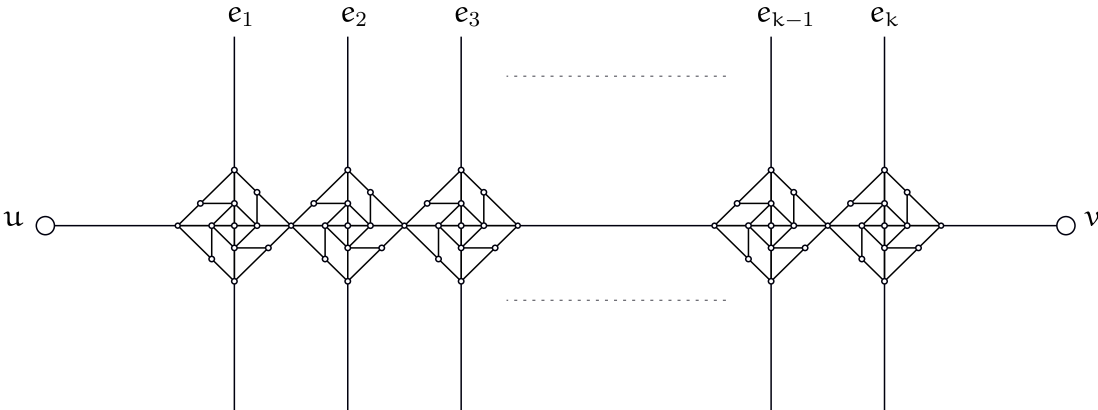
Това е добре дефинирано, понеже всяка p_j е обща крайна точка за точно четири планарни ребра на $\mathcal{G}_1$, а приспособленията имат по четири краища.

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.

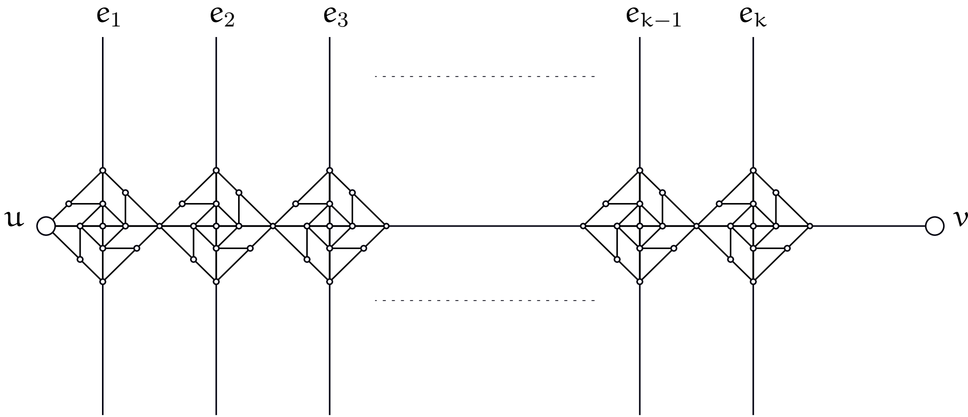
Нека $e = (u, v)$ е произволно планарно ребро на $\mathcal{G}$, което съдържа k точки от P за някое $k \in \{1, \ldots, t\}$. БОО, нека точките са $p_1, \ldots, p_k$, в този ред от u към v. Тези точки са общи за e и някакви други ребра $e_1, \ldots, e_k$:



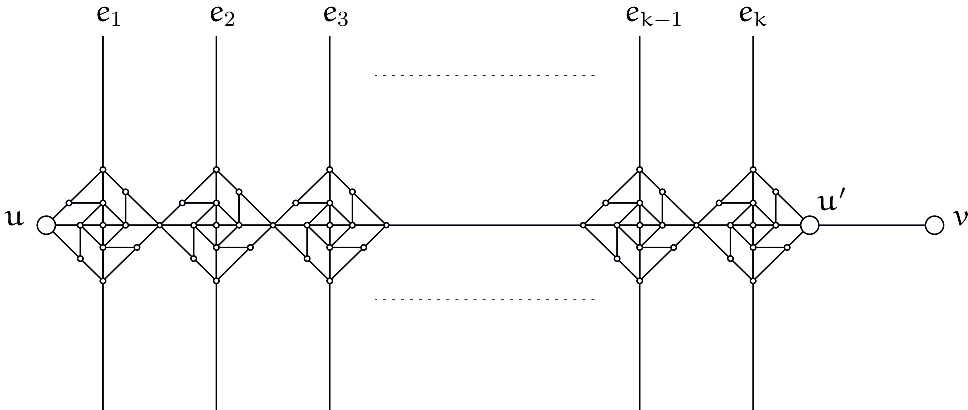
Слагаме по едно приспособление на мястото на всяко p_j, $1 \leqslant j \leqslant k$, като ги "слепваме" по следния начин:



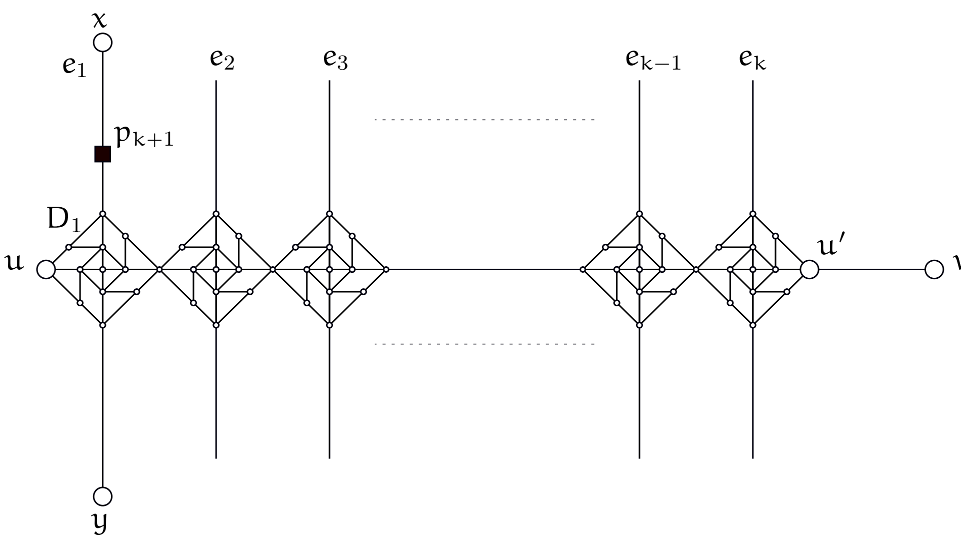
При това въпросните $p_1, \ldots, p_k$ изчезват. След това избираме единия край на e, няма значение кой, за *особен*. Да кажем, че u е особеният край в примера. "Слепваме" u с най-близкия край на най-близкото до u приспособление, при което те стават един връх, а реброто между тях изчезва:



Разглеждаме и най-отдалеченият от u край на веригата от слепени приспособления. Това е върхът, който е съсед на v. Да го наречем u':

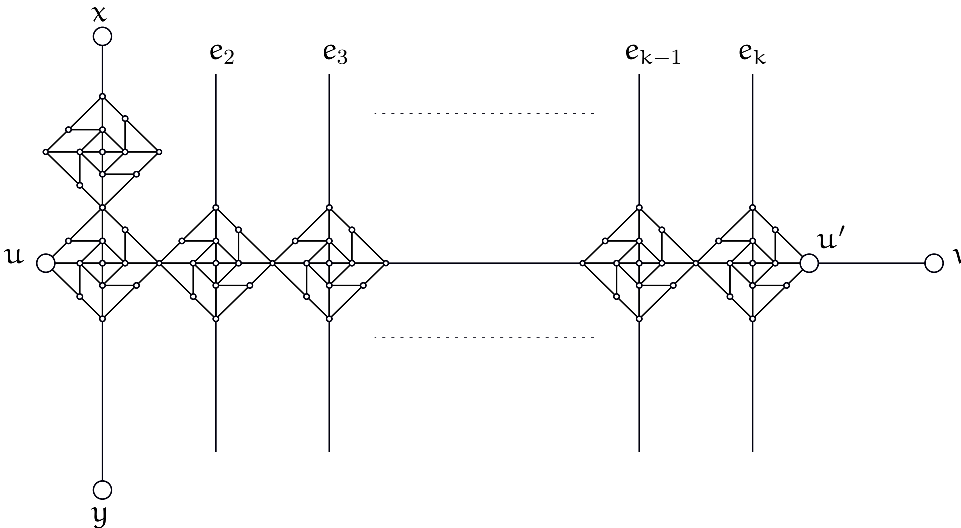


Според Garey и Johnson [51, стр. 88–89], с това описанието е конструкцията приключва. Според автора на записките, уместно е да се добави, че описаното слагане на верига от пристособления и идентифициране на два върха се извършва по точно този начин само за първото ребро $e = (u, v)$, върху което действаме. 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 $e_1, \ldots, e_k$, няма да действаме по точно същия начин. БООО[†], нека е това друго ребро е $e_1 = (x, y)$ 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:

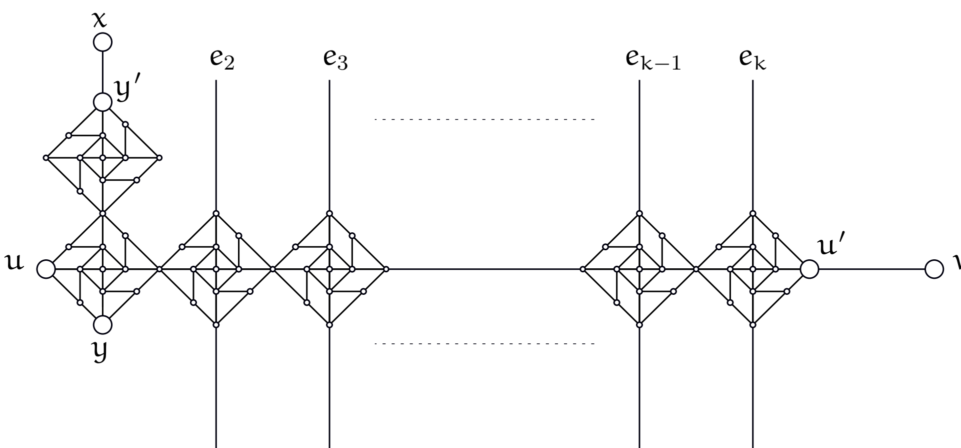


p_{k+1} бива заменена с приспособление, като се получава верига от две слепени в краищата си приспособления:

[†]Без Особено Ограничение на Общността.



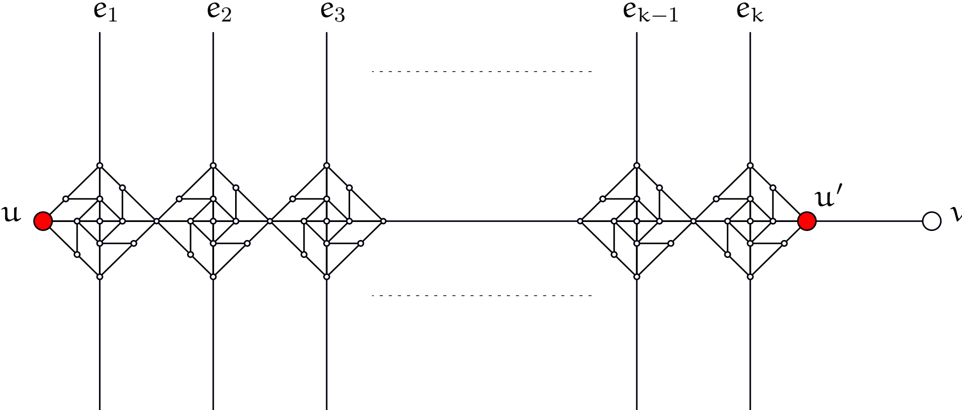
След което единият край на (бившето) e_1, да кажем y, бива обявен за особен, което означава да го слепим с най-близкия край на веригата от приспособления, а другият край на веригата да стане y':



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.

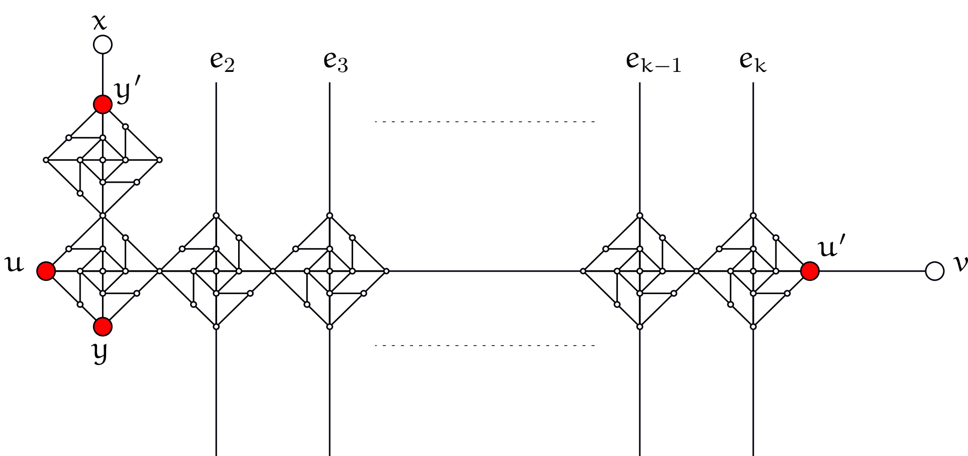
Ще докажем коректността. В едната посока, нека G е 3-оцветим. Ще докажем, че G_2 е също 3-оцветим. Нека f е оцветяване на върховете на G в не повече от 3 цвята. f се явява и 3-оцветяване на $\mathcal{G}$, ако дефинираме "оцветяване на планарно вписване" по най-естествения начин. Ще покажем, че f може 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.

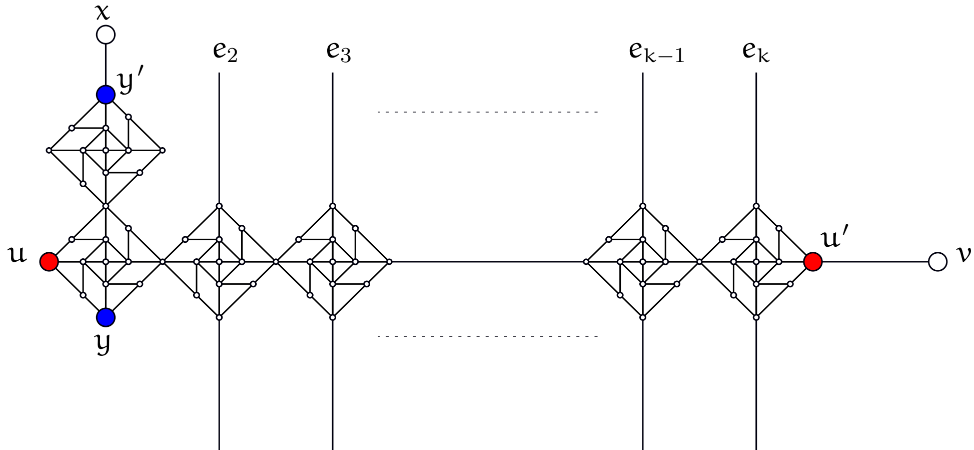


Иначе казано, за да получим цветове за новодобавените върхове, правим цвета на u' същия като цвета на u и знаем, че можем да раздадем цветове и на останалите върхове от приспособленията по протежение на e, така че 3-оцветимостта да е налице.

Нещо повече. Приспособленията, разположени по протежение на e, могат да пренасят цветове и в "ортогоналното направление"; тоест, по веригите, сложени върху планарните ребра $e_1, \ldots, e_k$, 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 особеният 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 $\mathcal{G}_\in$ 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 $f = f_2\big|_{\mathcal{V}}$ 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, особеният 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 $f(v) = f_2(v)$. Заключаваме, че $f(u) \neq f(v)$, в противоречие с прежде направеното допускане. $\lightning$

Показахме, че конструкцията е коректна. Това, че може да се извърши в полиномиално време, стига броят на точките на пресичане $p_1, \ldots, p_t$ в $\mathcal{G}$ да е полиномиално ограничен от размера на G, е очевидно. □

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 78) 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 [71]). 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

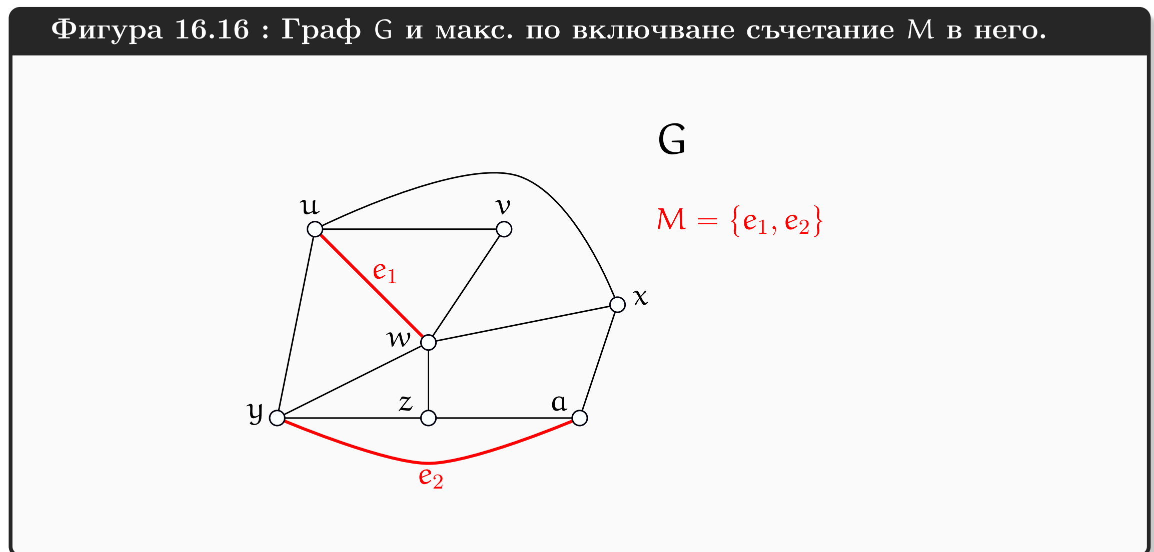
планарен граф G е вярно, че $\chi(G) \leqslant 4$. Нещо повече: има ефикасни алгоритми, изчисляващи 4-оцветяване на произволен планарен граф (вижте например [122]). Ерго, по отношение на PLANAR 3-COLORABILITY, неподатливостта се е само в това, че не можем да определим ефикасно дали $\chi(G) = 3$ или $\chi(G) = 4$. 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.

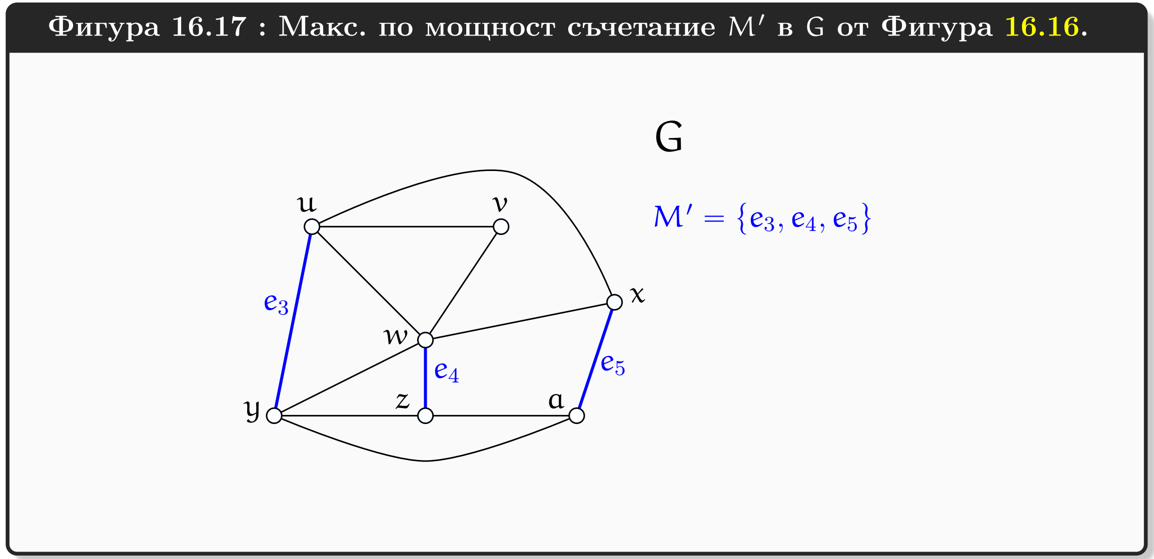


Фигура 16.16 : Граф G и макс. по включване съчетание M в него.

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 включване 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 квартичен алгоритъм за нея през 1965 г. [38]. Най-бързият известен алгоритъм за СЪЧЕТАНИЕ е на Micali и Vazirani, който има сложност $O(\sqrt{n}m)$ [105].

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 [119].

Допълнение 72: 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 [119, стр. vii-viii]. Да кажем, че са дадени n ра́боти (computer jobs) $j_1, \ldots, j_n$ 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 наличитието на ограничения усложнява нещата.
Ще моделираме задачата с **неориентиран** граф G, чиито върхове са $j_1, \ldots, j_n$, 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 което двата компютъра работят едновременно. Лесно се вижда, 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 $\frac{n}{2}$.
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 съчетение. 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, $\ldots$, j_{n-1} преди j_n, графът няма ребра, понеже всеки две рáботи 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 оженят. Иска се да ожиеним 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 [31, 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 [70] 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 съществувана на перфектно съчетание в двуделен граф следва веднага от Теорема 91, често наричана “the marriage theorem”.

Допълнение 73: Теореми на Kőning 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 89, е на унгарски. Статията е преведена на английски и е свободно достъпна [137].

Теорема 89: Теорема на Kőning, 1931 г.

Във всеки двуделен граф, размерът на максималното съчетание е равен на размера на минималното върхово покриване.

С други думи, $\nu(G) = \tau(G)$.

Доказателство: Известни са много доказателства. Примерно, в книгата на Diestel "Graph Theory" [34, 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. [121].

Това, че $\nu(G) \leqslant \tau(G)$, 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, $|M| \leqslant |U|$.
Нека G е минимален контрапример за теоремата. Тогава $\nu(G) \neq \tau(G)$. Щом $\nu(G) \leqslant \tau(G)$, следва, че

$$\nu(G) < \tau(G) \tag{16.5}$$

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 $\geqslant 3$. 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

$$|U' \cup \{w\}| \leqslant \nu(G) \tag{16.6}$$

От друга страна, $U' \cup \{w\}$ покрива върхово G. Тогава $\tau(G) \leqslant |U' \cup \{w\}|$. Предвид (16.5), заключаваме, че

$$\nu(G) < |U' \cup \{w\}| \tag{16.7}$$

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 $d_G(u) \geqslant 3$, 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 $\nu(G - e) \leqslant nu(G)$, така че $\tau(G - e) \leqslant \nu(G)$. 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). □

Щом в двуделните графи минималното върхово покриване е равно на максималното съчетание и можем да намираме ефикасно максимално съчетание в двуделен граф, можем да намиране ефикасно и размера на минималното върхово покриване на двуделен граф.

Теорема 90: 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 91 неслучайно е известна като "the marriage theorem". И така, има n жени, n мъже и са дадени двойките на взаимно харесване. Иска 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 $N(S)$[б] 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 $|N(S)| \geqslant |S|$, задачата има поне едно решение. Условието "$\forall S : |N(S)| \geqslant |S|$" е известно като "the marriage condition". Това, че the marriage condition влече наличие на решение, изобщо не е очевидно и е същността на Теорема 91, доказана от Philip Hall през 1935 г. [60].

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 равномощни. Теорема 91 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 равномощни, това е перфектно съчетание.

Теорема 91: Теорема на Hall, 1931 г.

В G има съчетание, което съчетава всички върхове на X тстк $\forall S \subseteq X : |N(S)| \geqslant |S|$.

Доказателство: Diestel [34, стр. 38–39] предлага три различни доказателства на Теоремата на Hall. Тук ще видим друго доказателство, ползващо Теоремата на Kőnig.

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 89, $|M| = |U|$. Тогава $|U| < |X|$. Нека

$$
\begin{aligned}
U \cap X &= X^+ \\
U \cap U &= Y^+ \\
X \backslash X^+ &= X^- \\
Y \backslash Y^+ &= Y^-
\end{aligned}
$$

Очевидно

$$
\begin{aligned}
|X| &= |X^+| + |X^-| \\
|U| &= |X^+| + |Y^+|
\end{aligned}
$$

Тогава

$$
|X| - |U| = |X^-| - |Y^+|
$$

Щом $|U| < |X|$, то $|X| - |U| > 0$, така че $|X^-| - |Y^+| > 0$, тоест

$$|X^-| > |Y^+| \tag{16.8}$$

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

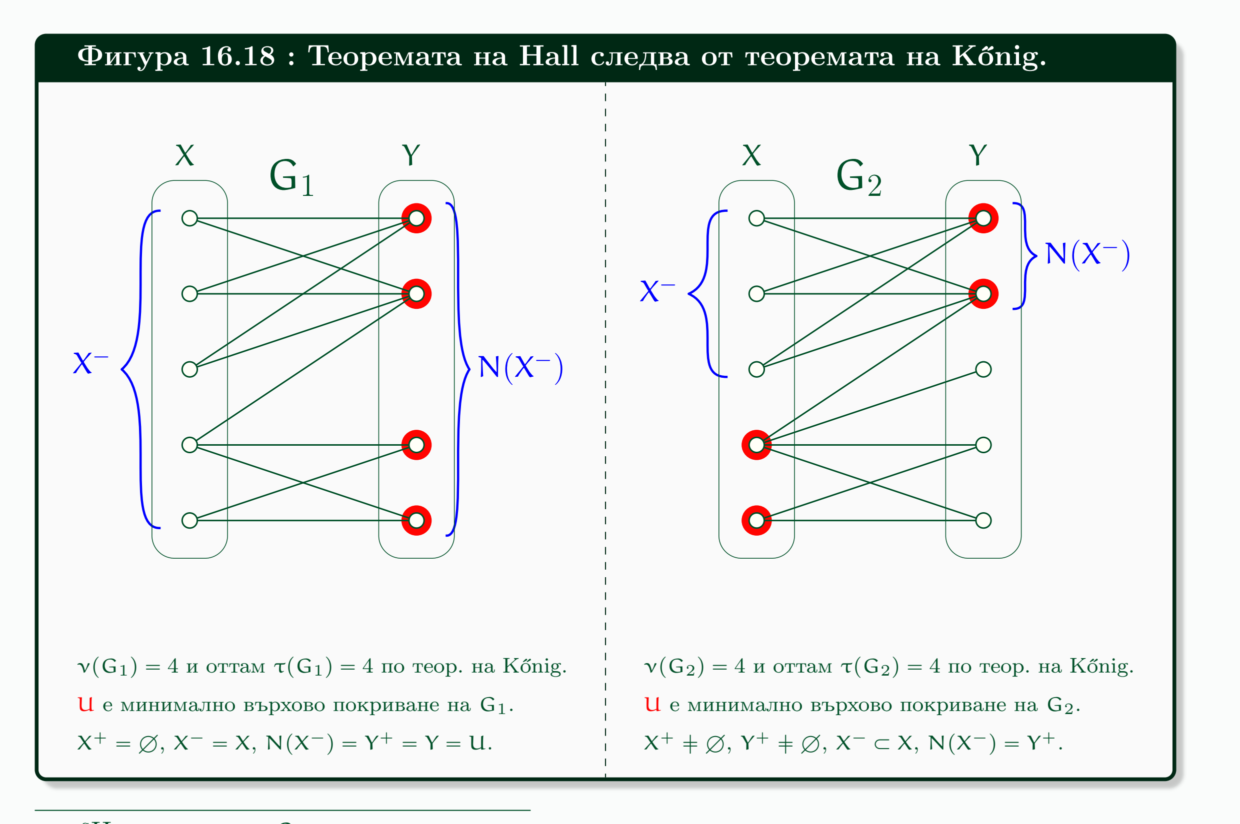
$$|Y^+| \geqslant |N(X^-)| \tag{16.9}$$

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 дяловете 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 Kőnig 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 Kőnig, 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.

[а] Четен, защото G е двуделен.
[б] Вижте Нотация 5.

16.2.7.2 Хиперграфите като обобщение на обикновените графи

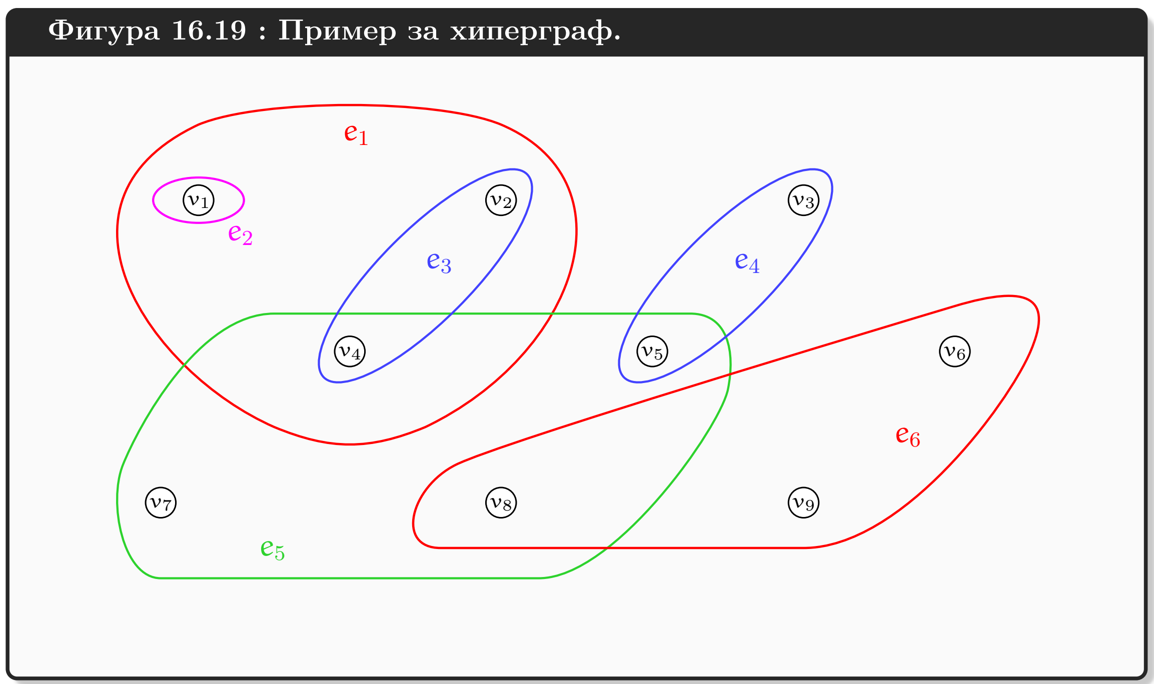
Хиперграф, на английски *hypergraph*, е естествено обобщение на "граф". Нека $G = (V, E)$ е граф. Можем да мислим за V като за опорно множество, а за E като за фамилия над V. И то доста ограничен вид фамилия: всяко ребро е двуелементно множество от върхове. Мислейки за E като за фамилия над V, може ли 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.

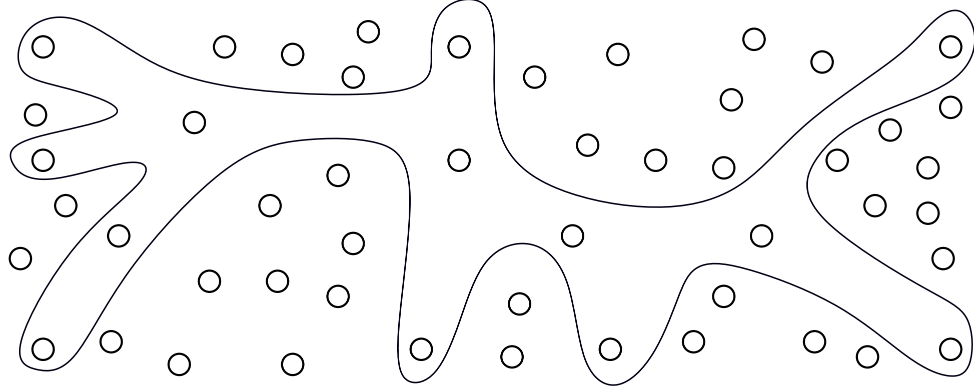
Фигура 16.19 показва пример за хиперграф. Типично, хиперребрата се рисуват с прости затворени криви—окръжности или овали, ако е възможно—като образа на хиперребро на рисунката огражда образите на неговите върхове в хиперграфа.

[†] За задълбочено изложение върху хиперграфите вижте класическите книги на Berge "Graphs and Hypergraphs" [18] и "Hypergraphs: Combinatorics of Finite Sets" [17], съвременната "Hypergraph Theory: An Introduction" на Bretto [25] и обзорната статия "Hypergraphs: an introduction and review" на Ouvrard [113]. За приложения на хиперграфите вижте статията "Learning with Hypergraphs: Clustering, Classification, and Embedding" [148].

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, какъвто 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 цветове. Читателят лесно може да си представи ситуация, в която хиперребра се 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 $\{v_1, \ldots, v_9\}$, а множеството от статиите е $\{e_1, \ldots, e_6\}$. 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 74: 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 степеннния 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

$$2^{\binom{n}{3}} = 2^{\frac{n(n-1)(n-2)}{6}}$$

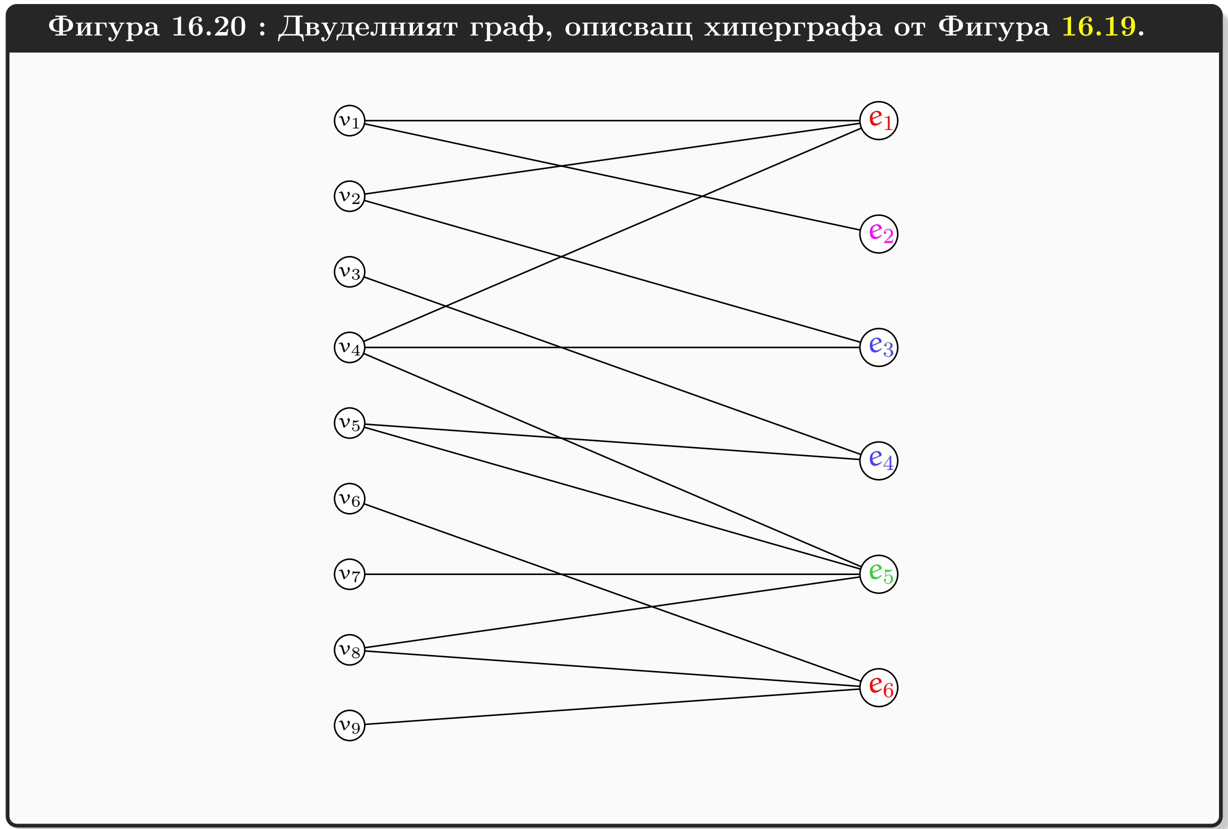
тъй като различните хиперребра са $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. Очевидно е, че

$$\lim_{n\to\infty} \frac{2^{\frac{n(n-1)(n-2)}{6}}}{2^{\frac{n(n-1)}{2}}} = \infty$$

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 $|V|$.

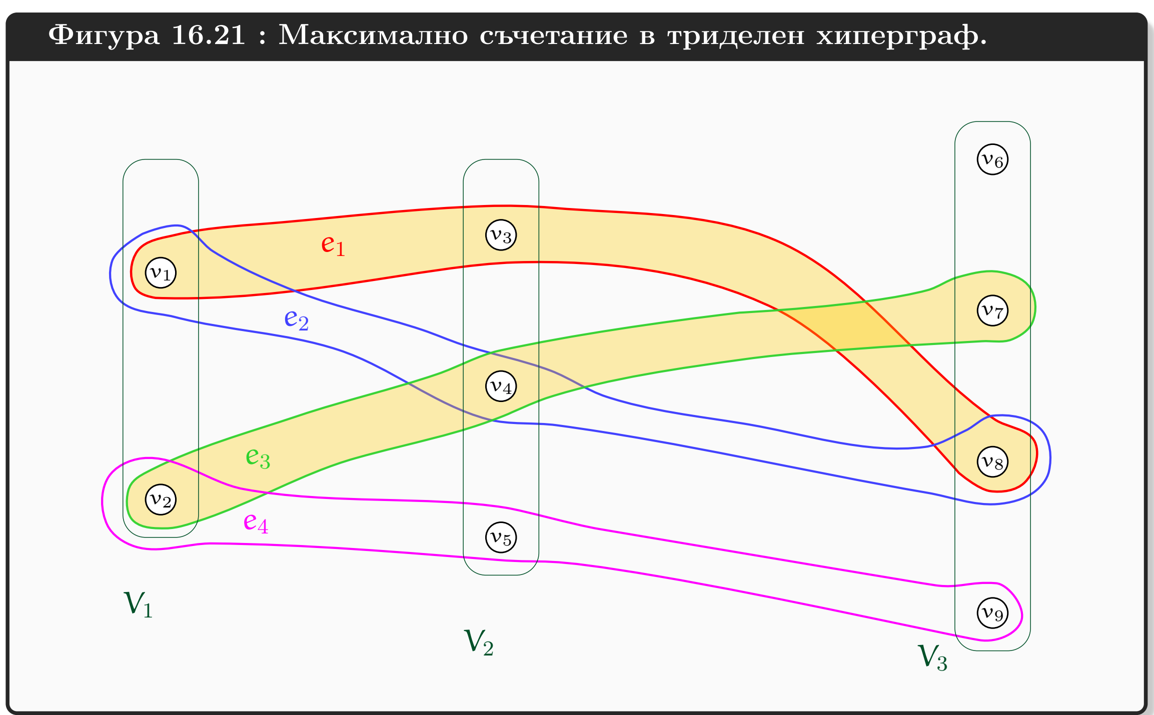
Ако видим само съответния двуделен граф, без да сме видели оригиналния хиперграф H, можем ли да разберем кой от дяловете (на двуделния) отговаря на върховете на H и кой, на хиперребрата на H? Отговорът е категорично **не**. Примерно, G на Фигура 16.20 би могъл да представлява хиперграф с върхове $e_1, \ldots, e_6$ и ребра $v_1, \ldots, v_9$. 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 91) и задачата за съществуване на перфектно съчетание е решима ефикасно чрез алгоритъм за потоци в графи ([31, стр. 732–726]) или алгоритъма на Hopcroft и Karp ([70]). 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 79: 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

$$\forall X, Y \in A' : X \neq Y \rightarrow X \cap Y = \varnothing?$$

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 $\leqslant_p$ 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 равномощни множества с празни сечения по двойки и $A \subseteq \{\{b, c, d\} \,|\, b \in B, c \in C, d \in D\}$, по такъв начин, че A има подмножество-перфектно съчетание тстк ϕ е удовлетворима. Нека $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$.

Първо конструираме множествата B, C и D:

$$\begin{aligned}
B &= \{u_{i,j} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\} \cup \{v_{i,j} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\} \\
C &= \{y_{i,j} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\} \cup \{s_{1,j} \,|\, 1 \leqslant j \leqslant m\} \cup \{g_{1,j} \,|\, 1 \leqslant j \leqslant m(n-1)\} \\
D &= \{z_{i,j} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\} \cup \{s_{2,j} \,|\, 1 \leqslant j \leqslant m\} \cup \{g_{2,j} \,|\, 1 \leqslant j \leqslant m(n-1)\}
\end{aligned}$$

Две по две не имат празни сечения. Очевидно, $|B| = |C| = |D| = 2nm$, така че $|B \cup C \cup D| = 6nm$. Множеството A е

$$A = \left(\bigcup_{i=1}^{n} A_{1,i}\right) \bigcup \left(\bigcup_{j=1}^{m} A_{2,j}\right) \bigcup \left(\bigcup_{k=1}^{m(n-1)} A_{3,k}\right)$$

където

$$\begin{aligned}
A_{1,i} = &\{\{u_{i,j}, y_{i,j}, z_{i,j}\} \,|\, 1 \leqslant j \leqslant m\} \cup \\
&\{\{v_{i,j}, y_{i,j+1}, z_{i,j}\} \,|\, 1 \leqslant j < m\} \cup \{\{v_{i,m}, y_{i,1}, z_{i,m}\}\}
\end{aligned}$$

$$A_{2,j} = \{\{u_{i,j}, s_{1,j}, s_{2,j}\} \,|\, \overline{x_i} \text{ е литерал във } \phi_j\} \cup \{\{v_{i,j}, s_{1,j}, s_{2,j}\} \,|\, x_i \text{ е литерал във } \phi_j\}$$

$$\begin{aligned}
A_{3,k} = &\{\{u_{i,j}, g_{1,k}, g_{2,k}\} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\} \cup \\
&\{\{v_{i,j}, g_{1,k}, g_{2,k}\} \,|\, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m\}
\end{aligned}$$

Това е конструкцията, описана формално и прецизно. Лесно се вижда, че

$$\begin{aligned}
|A_{1,i}| &= 2m && \text{за } 1 \leqslant i \leqslant n \\
|A_{2,j}| &= 3 && \text{за } 1 \leqslant j \leqslant m \\
|A_{3,k}| &= 2mn && \text{за } 1 \leqslant k \leqslant m(n-1)
\end{aligned}$$

откъдето

$$\begin{aligned}
\left|\bigcup_{i=1}^{n} A_{1,i}\right| &= 2mn \\
\left|\bigcup_{j=1}^{m} A_{2,j}\right| &= 3m \\
\left|\bigcup_{k=1}^{m(n-1)} A_{3,k}\right| &= 2m^2(n-1)
\end{aligned}$$

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

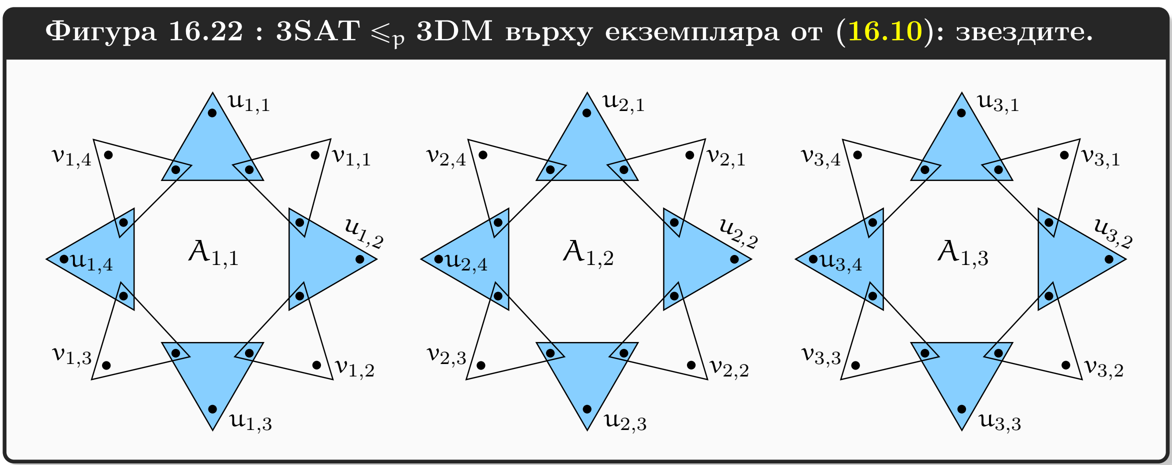
$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\phi_1}\underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{\phi_2}\underbrace{(\overline{x_1} \vee x_2 \vee \overline{x_3})}_{\phi_3}\underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{\phi_4} \quad (16.10)$$

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 $1 \leqslant i \leqslant n$, 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. Именувани са върховете $u_{i,j}$ и $v_{i,j}$.

Фигура 16.22 : 3SAT $\leqslant_p$ 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 90

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.

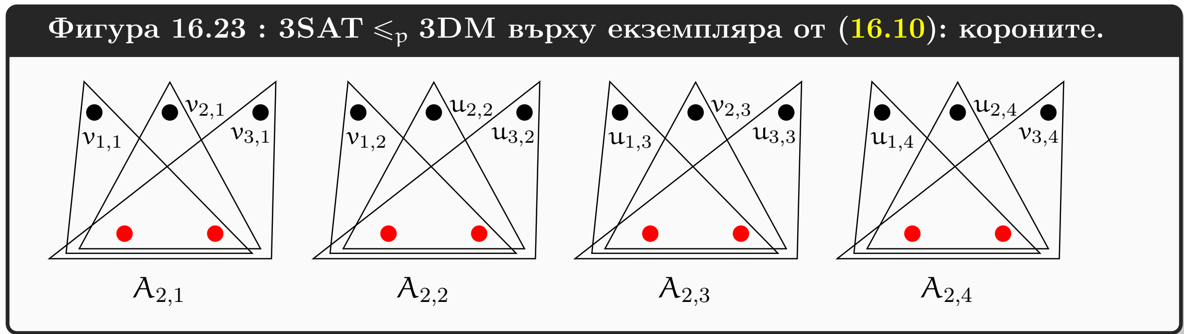
Читателят сигурно се досеща, че щом звездите отговарят на променливите, то дали за дадена звезда вземаме точно u-тройките или точно v-тройките в перфектно съчетание моделира дали тази променлива е истина или лъжа в съответната удовлетворяваща валюация на екземпляра на 3SAT. Дали u-тройките или v-тройките 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 $1 \leqslant j \leqslant m$, 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 $v_{i,j}$.

Фигура 16.23 : 3SAT $\leqslant_p$ 3DM върху екземпляра от (16.10): короните.



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 91

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.

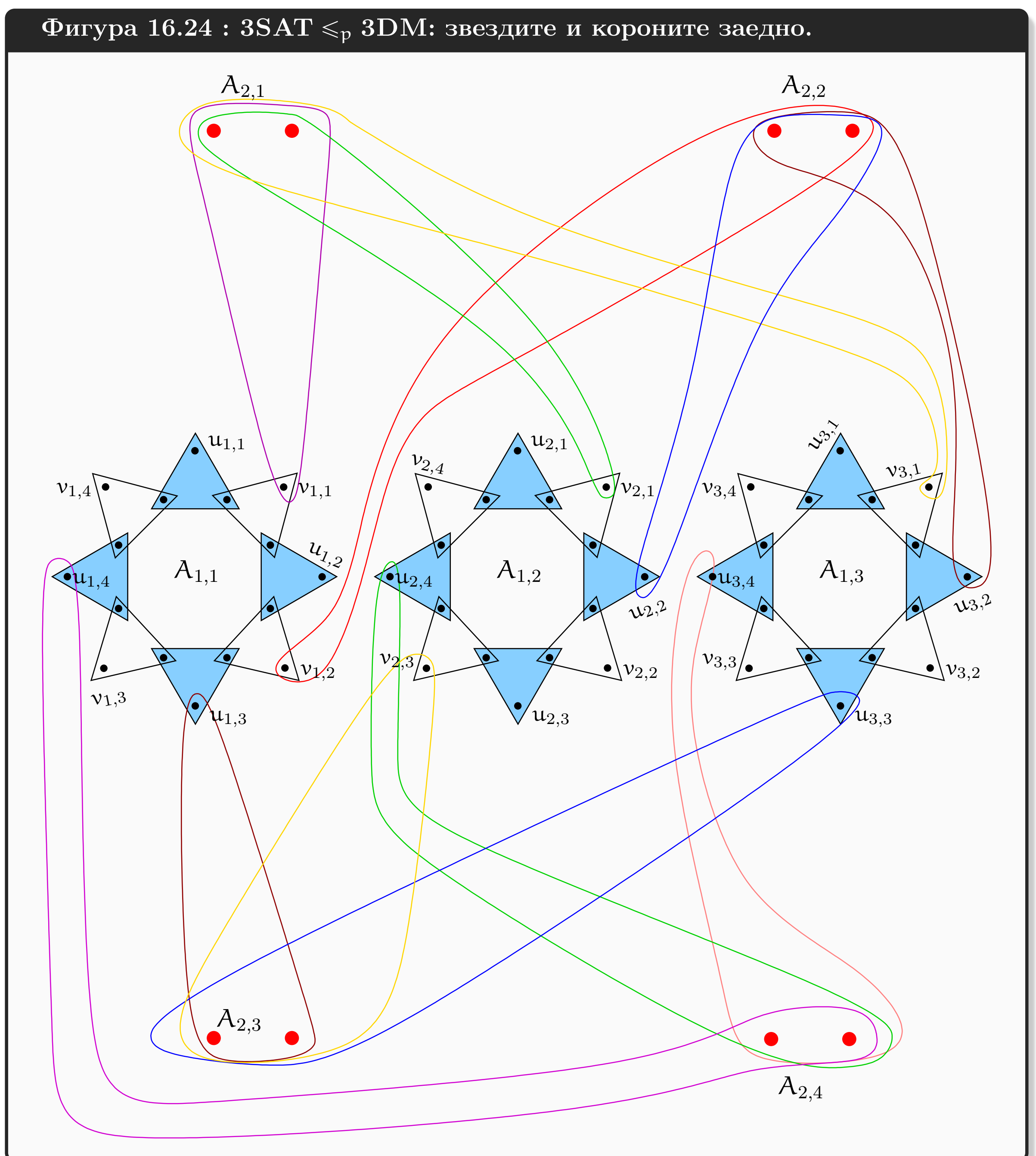
Читателят сигурно се досеща, че щом короните отговарят на клаузите и всяка от трите тройки моделира участието на една от трите променливи в клаузата—било с положителен, било с отрицателен литерал—то коя от трите тройки на короната участва в перфектно съчетание моделира през коя променлива клаузата получава 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 валюация 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.

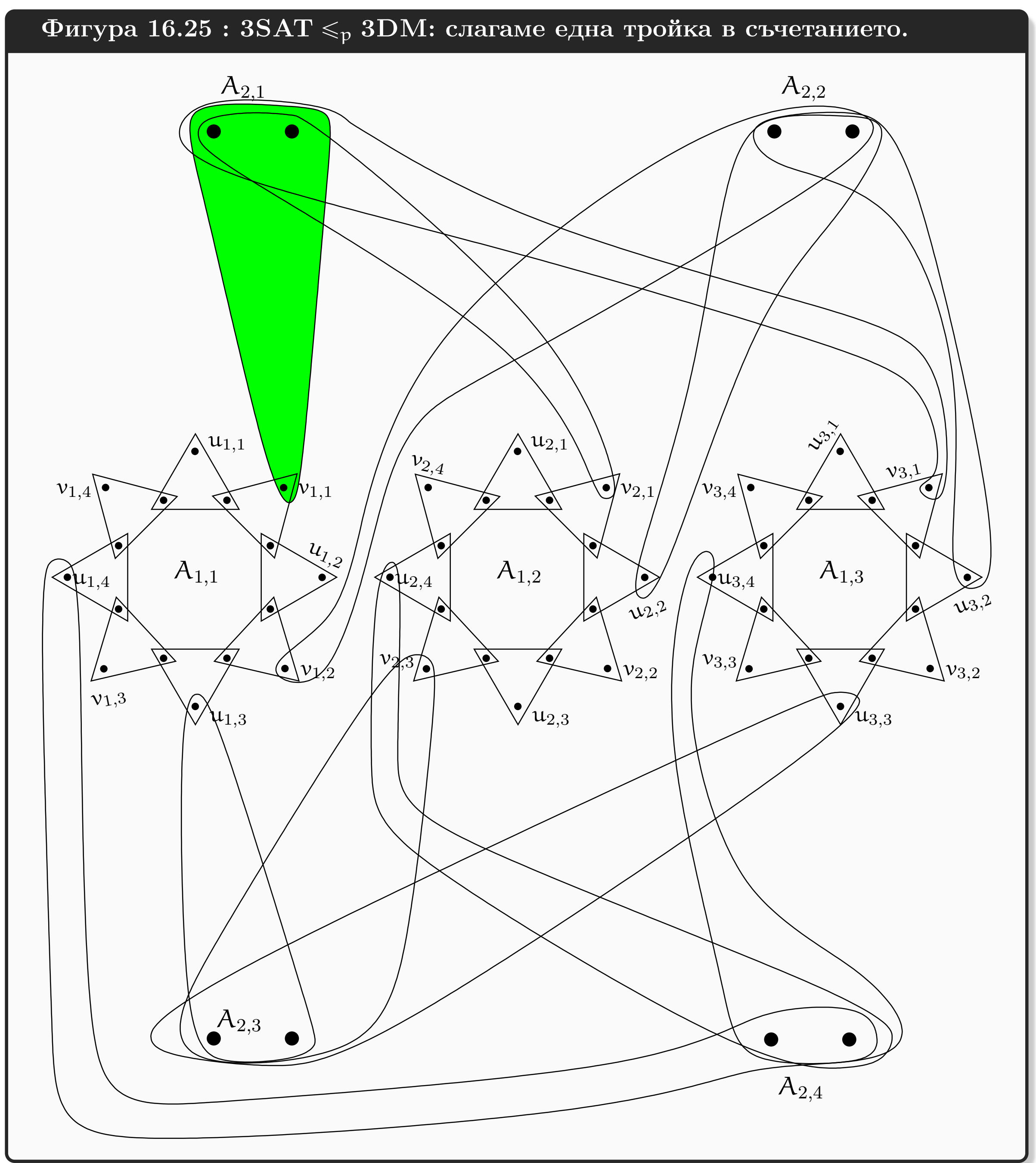
Фигура 16.24 : 3SAT $\leqslant_p$ 3DM: звездите и короните заедно.



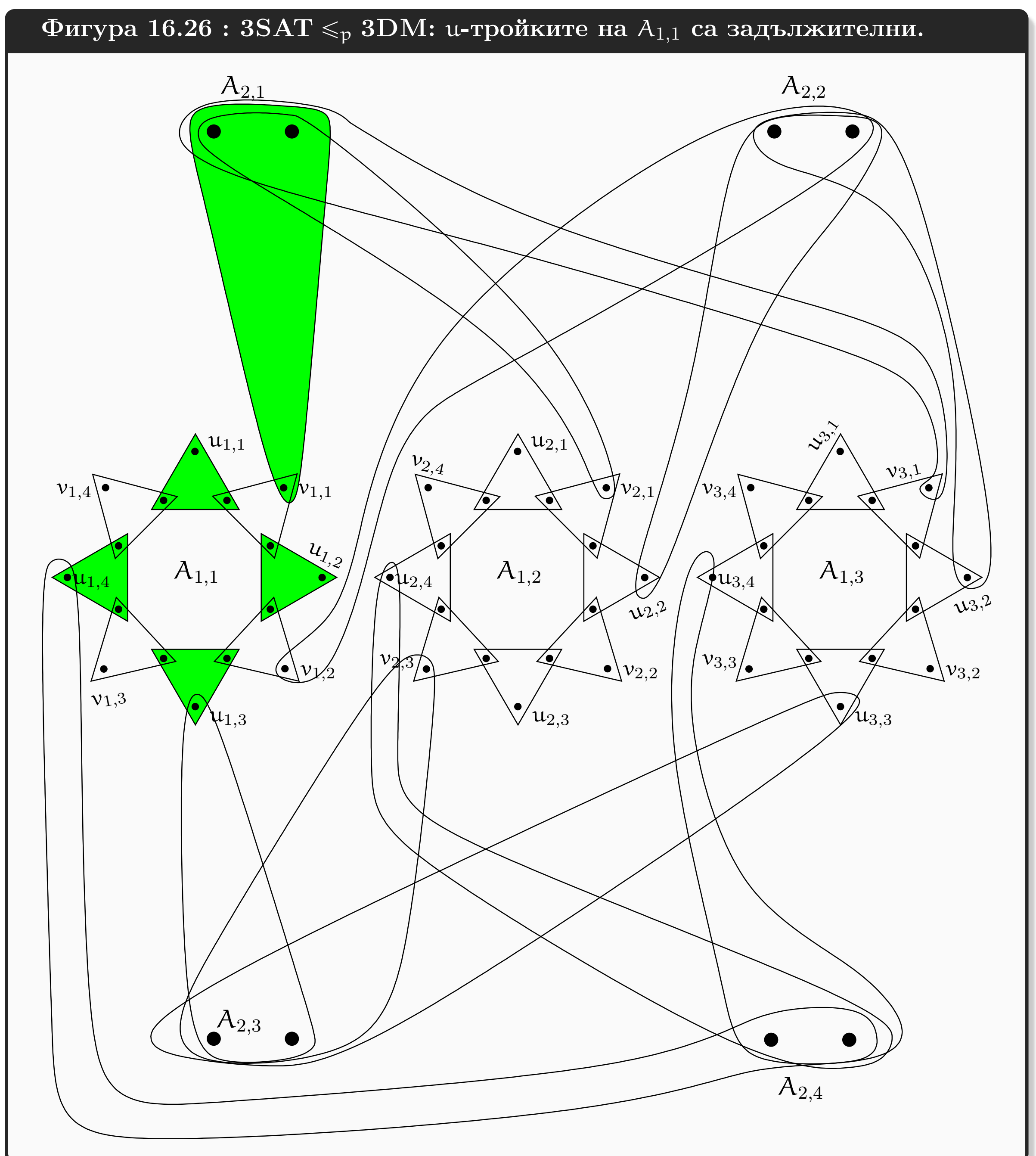
Сега да видим същината на конструкцията – защо това “работи”. За да има перфектно съчетание, точно една тройка на короната $A_{2,1}$ трябва да е в него. Да кажем, тройката с връх $v_{1,1}$. Това е показано на Фигура 16.25.

[†]Ситуацията напомня на привидното противоречие от 3SAT $\leqslant_p$ Vertex Cover (Наблюдение 88).

Фигура 16.25 : 3SAT $\leqslant_p$ 3DM: слагаме една тройка в съчетанието.



Изборът да сложим тази тройка в съчетанието има последици. Припомняме си Наблюдение 90: 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 $A_{1,1}$, длъжни сме да сложим в съчетанието u-тройките на $A_{1,1}$. Това е показано на Фигура 16.26.



Фигура 16.26 : 3SAT $\leqslant_p$ 3DM: u-тройките на $A_{1,1}$ са задължителни.

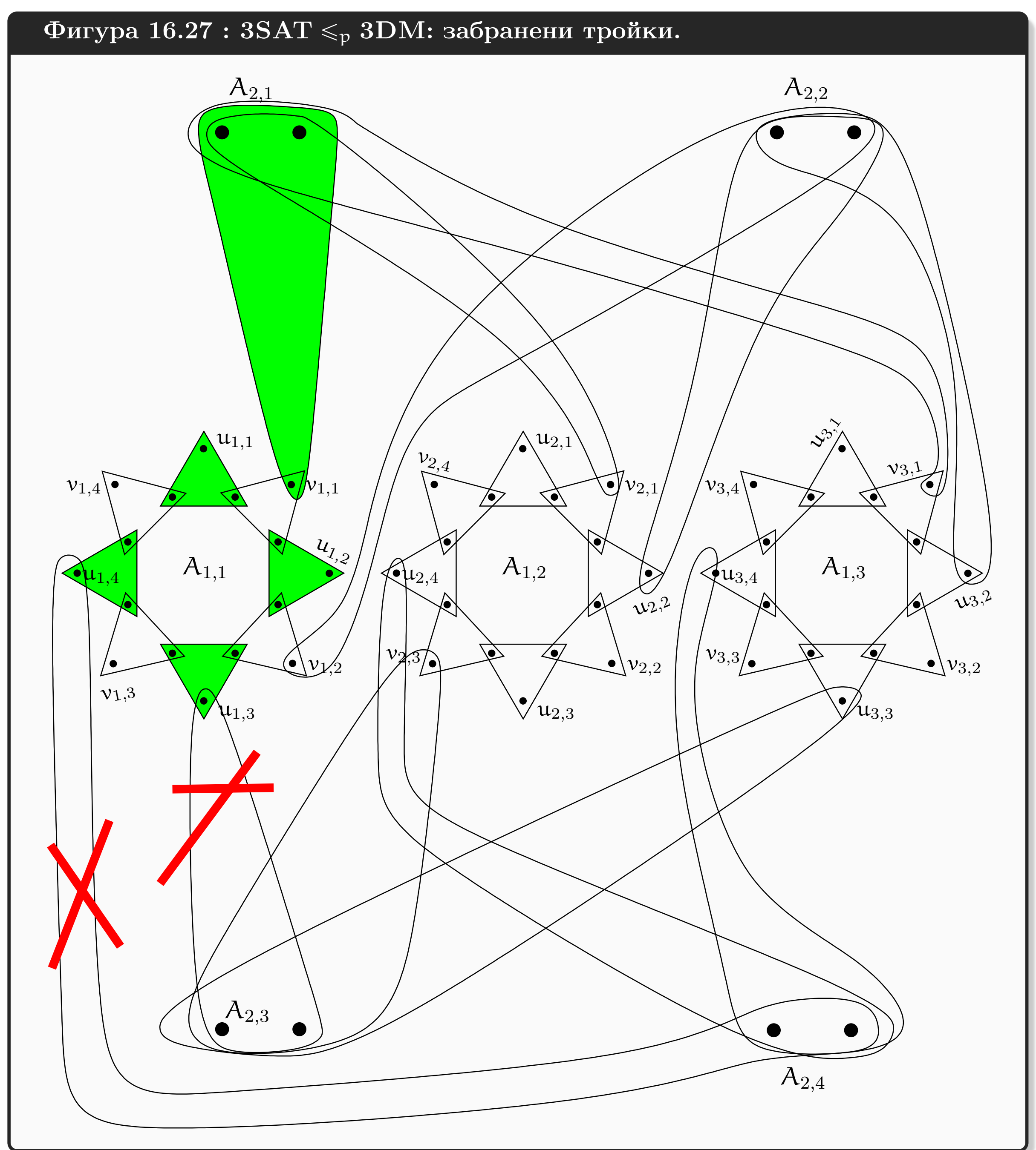
На свой ред, слагането на u-тройките на $A_{1,1}$ в перфектното съчетание води до това, че от всяка от трите засега необработени корони $A_{2,2}$, $A_{2,3}$ и $A_{2,4}$ трябва да изберем за съчетанието тройка, която няма връх $u_{1,j}$. Защо? – защото четирите върха $u_{1,j}$ вече се намират в тройка 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 валюация, в която x_1 е 1, и тази стойност на x_1 бива "наложена" и по отношение на останалите клаузи.

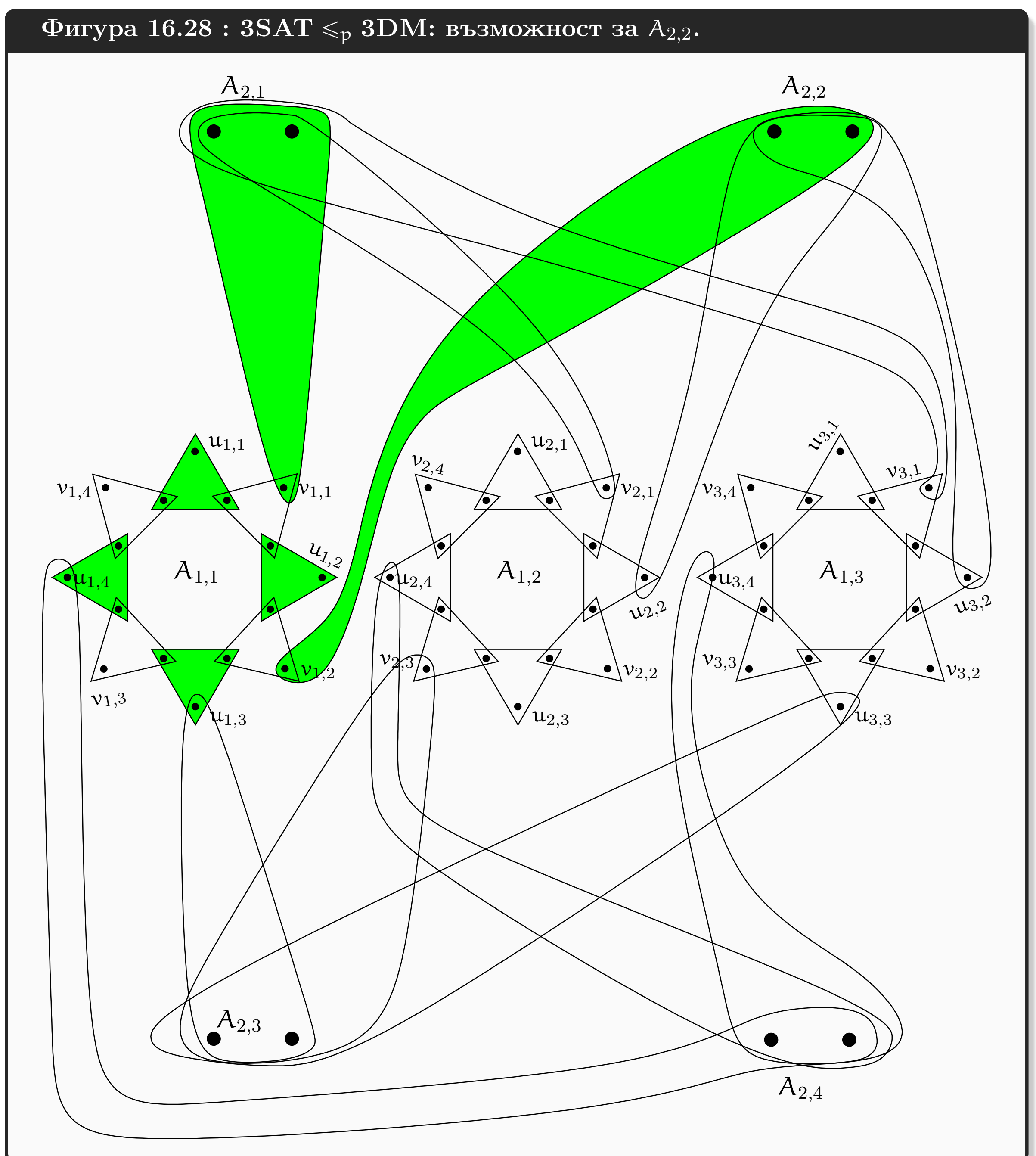
Фигура 16.27 показва как изборът на u-тройките от звездата $A_{1,1}$ прави невъзможно да се

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. Което е смислено, понеже променливата x_1 участва с отрицателния си литерал във ϕ_3 и ϕ_4.

Фигура 16.27 : 3SAT $\leqslant_p$ 3DM: забранени тройки.



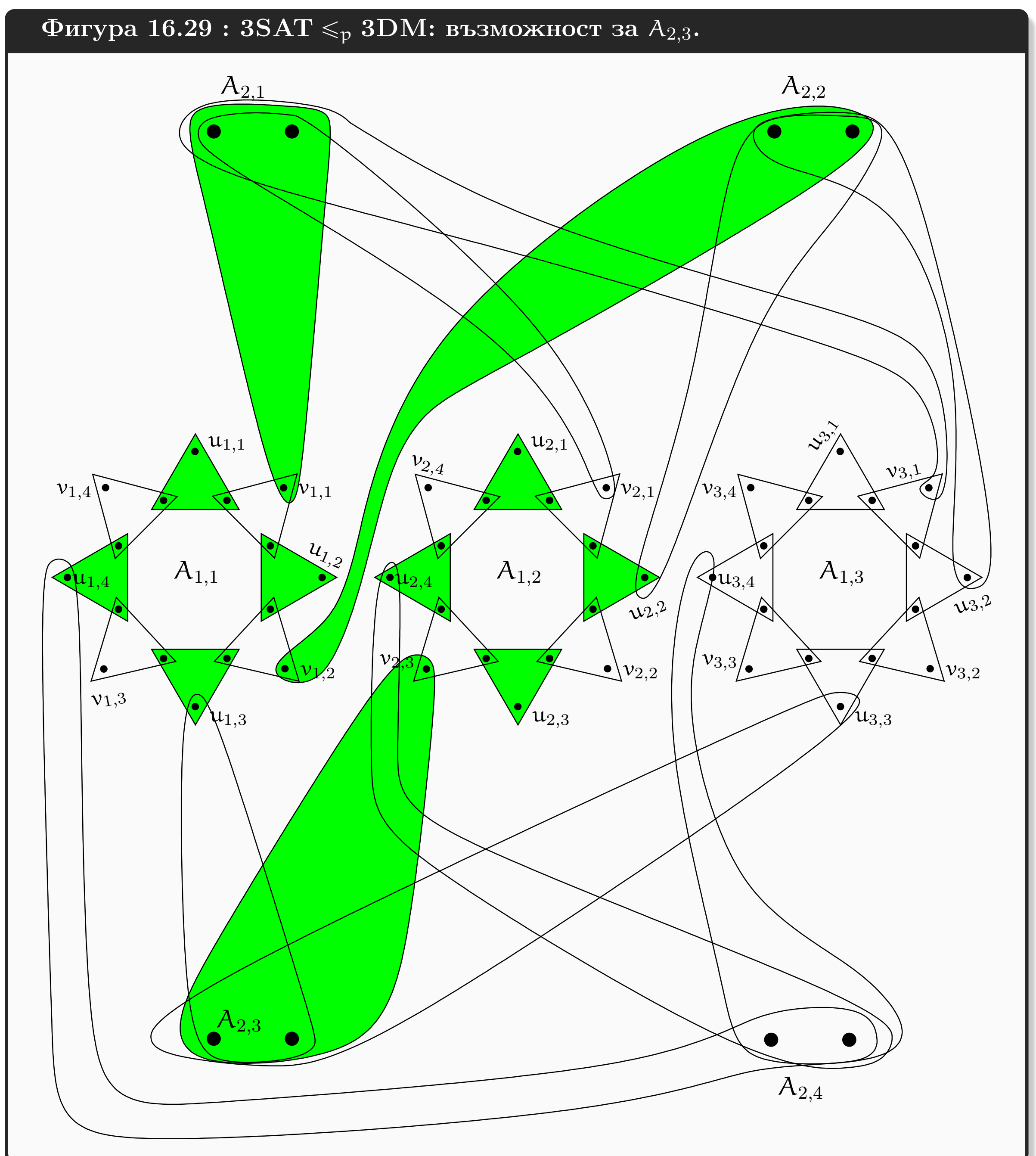
Във ϕ_2 обаче променливата x_1 обаче участва с положителния си литерал. Ерго, ϕ_2 бива удовлетворена от валюация, даваща 1 на x_1. По отношение на екземпляра на 3DM, това се превежда така: можем да изберем за короната $A_{2,2}$ тази тройка, която съдържа $v_{1,2}$. Фигура 16.28 показва точно това.



Фигура 16.28 : 3SAT $\leqslant_p$ 3DM: възможност за $A_{2,2}$.

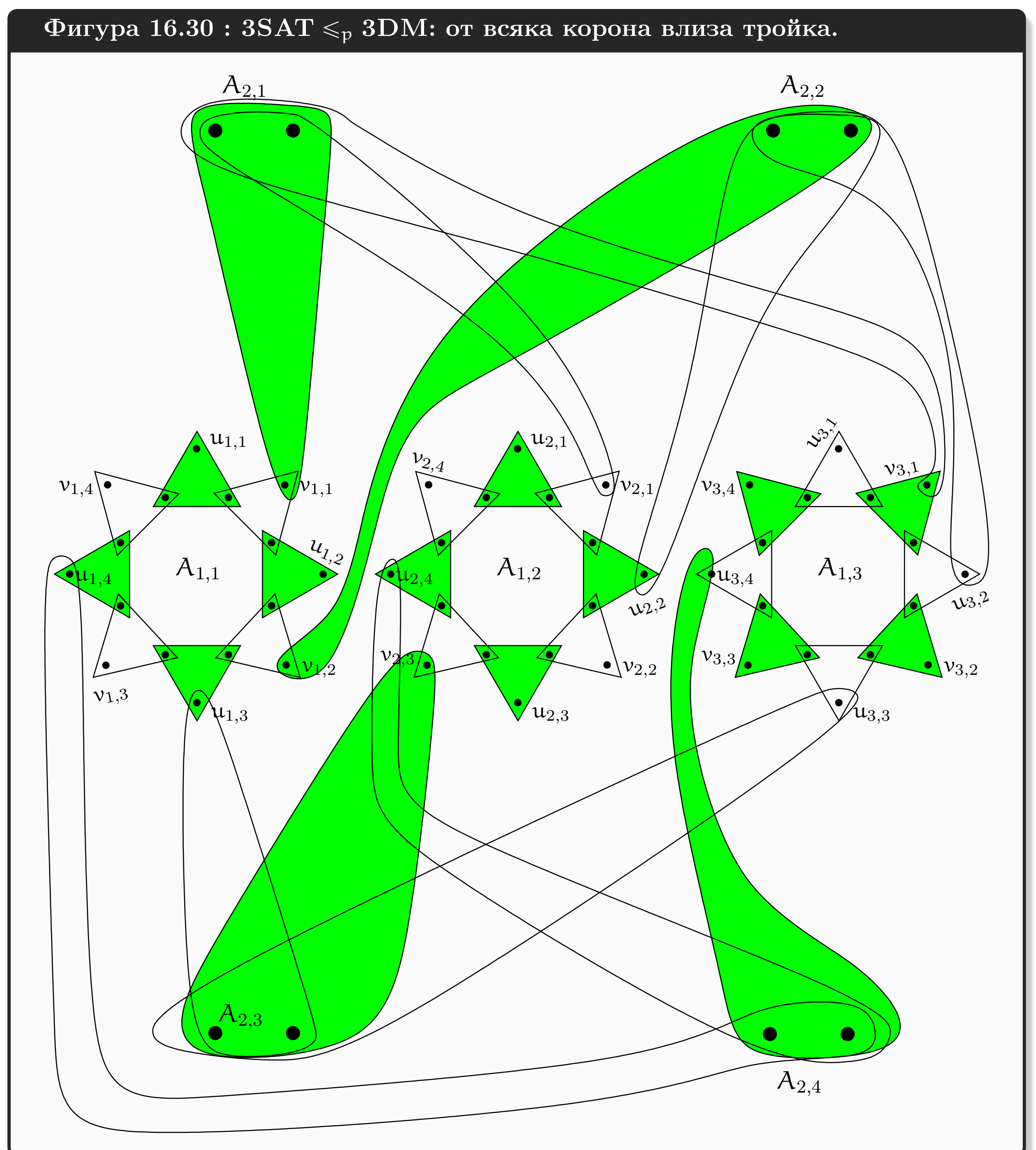
Сега да разгледаме короната $A_{2,3}$. Както вече знаем, не можем да изберем нейната тройката, съдържаща $u_{1,3}$. Преведено на езика на 3SAT, клаузата ϕ_3 трябва да си "намери" удовлетворение не през литерала на x_1, тъй като вече избрахме стойност 1 за x_1, а във ϕ_3, променливата x_1 участва с отрицателния 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 $\leqslant_p$ 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 $\overline{x_1}$. Фигура 16.30 показва състоянието на нещата дотук.



Фигура 16.30 : 3SAT $\leqslant_p$ 3DM: от всяка корона влиза тройка.

Това обаче не може да е краят 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 $1 \leqslant k \leqslant m(n-1)$. Това са третият вид приспособления и те са най-многобройните. Съвкупността от тях е известна като *garbage collector* [51, 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. □

16.2.8 Hamiltonian Cycle, Ham. Path, Longest Cycle, Longest Path

Разглеждаме Hamiltonian Cycle (Задача 2).

> Редукция 23: Vertex Cover $\leqslant_p$ 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 $\leqslant k$. 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:

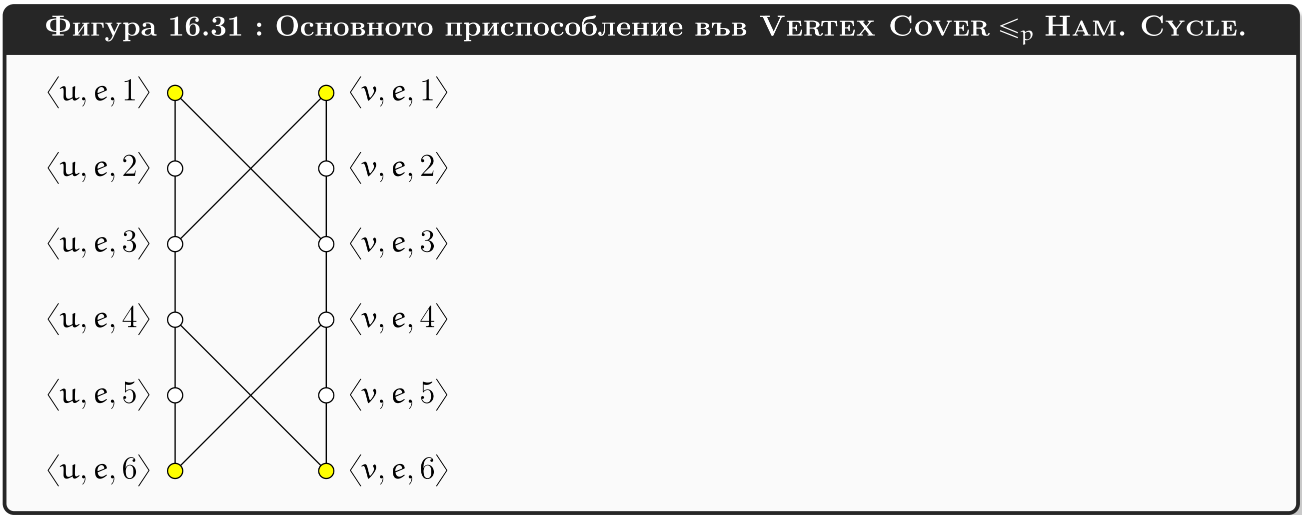
$$\begin{aligned} E'_e = &\big\{\big(\langle u, e, i\rangle, \langle u, e, i+1\rangle\big) \,|\, 1 \leqslant i \leqslant 5\big\} \cup \big\{\big(\langle v, e, i\rangle, \langle v, e, i+1\rangle\big) \,|\, 1 \leqslant i \leqslant 5\big\} \cup \\ &\big\{\big(\langle u, e, 1\rangle, \langle v, e, 3\rangle\big), \big(\langle u, e, 3\rangle, \langle v, e, 1\rangle\big)\big\} \cup \\ &\big\{\big(\langle u, e, 4\rangle, \langle v, e, 6\rangle\big), \big(\langle u, e, 6\rangle, \langle v, e, 4\rangle\big)\big\} \end{aligned}$$

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.

Фигура 16.31 : Основното приспособление във Vertex Cover $\leqslant_p$ Ham. Cycle.



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 $\langle u, e, 1\rangle, \ldots, \langle u, e, 6\rangle$ индуцират подграф-път с дължина 5, който наричаме u-*страната* на приспособлението. Аналогично, върховете $\langle v, e, 1\rangle, \ldots, \langle v, e, 6\rangle$ индуцират подграф-път с дължина 5, който наричаме v-*страната* на приспособлението.

Наблюдение 92: Точно три начина за Хамилтонов цикъл през джаджа

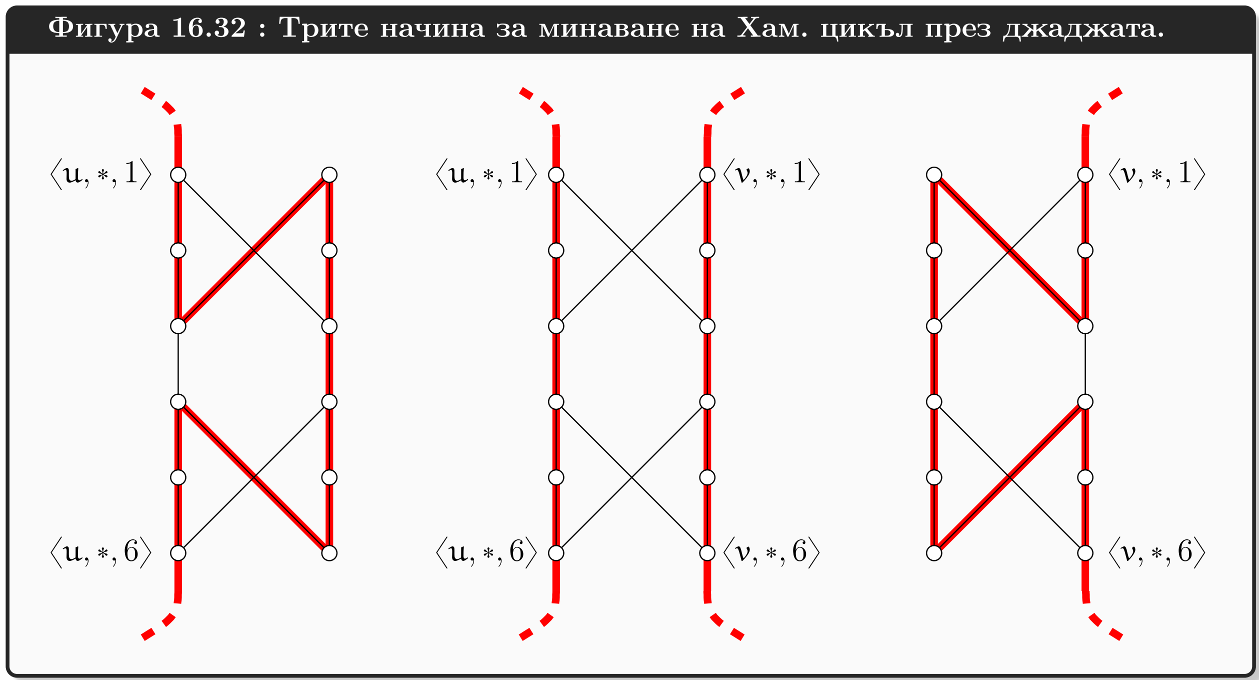
В контекста на цялостната конструкция има точно три начина, по които Хамилтонов цикъл C в J може да мине през дадено приспособление H. Ако краищата на H са $\langle u, *, 1\rangle$, $\langle v, *, 1\rangle$, $\langle u, *, 6\rangle$ и $\langle v, *, 6\rangle$, въпросните начини са следните.

- 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

  $$\langle u, *, 1\rangle, \langle u, *, 2\rangle, \ldots, \langle u, *, 6\rangle$$
  $$\langle v, *, 1\rangle, \langle v, *, 2\rangle, \ldots, \langle v, *, 6\rangle$$

  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 $e_1, \ldots, e_t$. Избираме една произволна наредба на тези ребра и я наричаме r_z, да кажем

$$r_z = \langle e_1, \ldots, e_t \rangle$$

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:

$$E''_z = \left\{ \left(\langle z, e_i, 6\rangle, \langle z, e_{i+1}, 1\rangle \right) \,|\, 1 \leqslant i \leqslant t-1 \right\}$$

И правим това за всеки $z \in V$. Тогава

$$E'' = \bigcup_{z \in V} E''_z$$

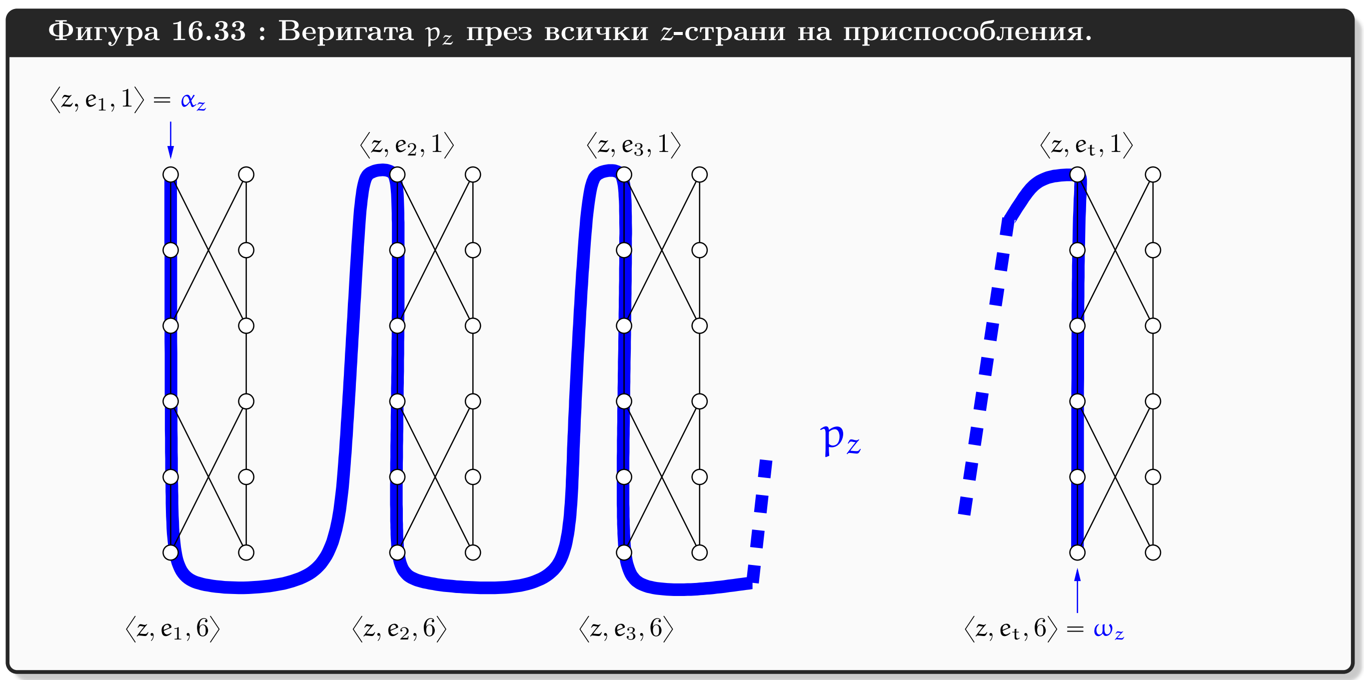
За всеки $z \in V$, *z-веригата в* J ще наричаме пътя p_z, в който алтернират z-страна и ребро от E''_z, и то в реда на r_z. Не особено формално казано, p_z започва със z-страна на първото приспособление, което има z-страна, минава през ребро от E''_z, после през z-страна на следващото приспособление, 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.

Краищата 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 краищата 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.

Фигура 16.33 : Веригата p_z през всички z-страни на приспособления.



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

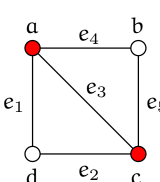
$$E'' = 2m - n$$

понеже за всяко $z \in V$, $|E''_z| = d(z) - 1$.

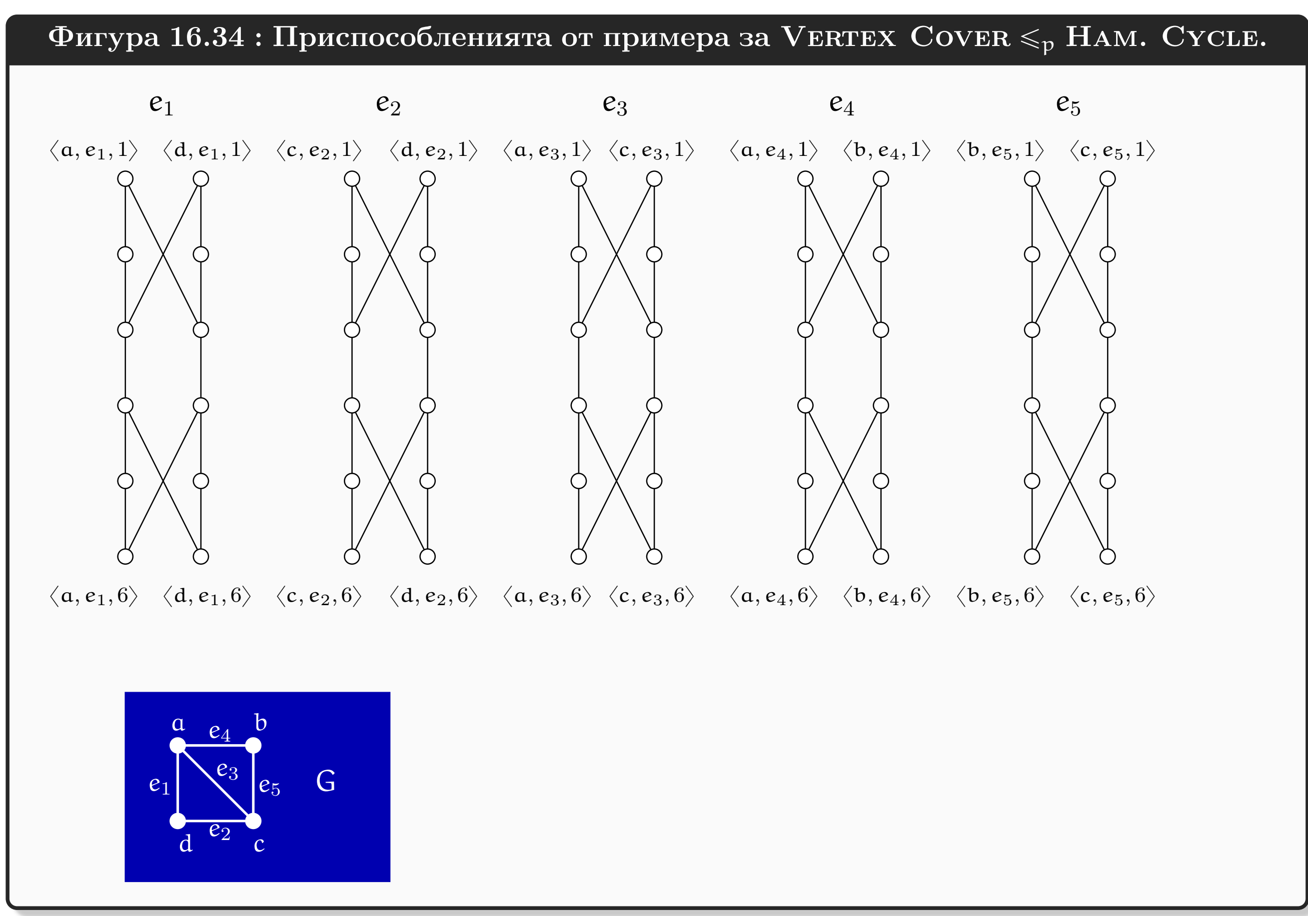
И накрая, дефинираме E''' така:

$$E''' = \{(\ell_i, \alpha_z) \mid i \in \{1, \dots, k\}, z \in V\} \cup \{(\ell_i, \omega_z) \mid i \in \{1, \dots, k\}, z \in V\}$$

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.

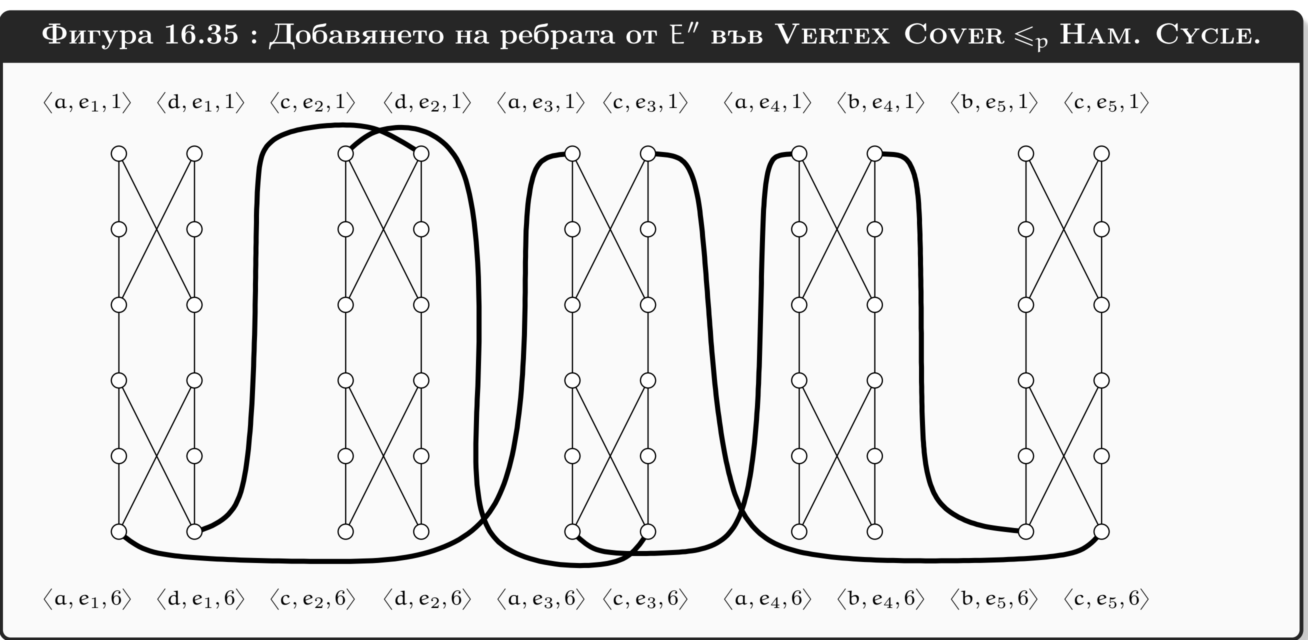
Конструкцията добавя множество от ребра E'', което се генерира спрямо някакви наредби на ребрата, инцидентни с върховете на G, като за всеки връх z наредбата r_z е произволна, но, след като я изберем веднъж, трябва да я спазваме. Нека наредбите са тези:

- $r_a = \langle e_1, e_3, e_4\rangle$,
- $r_b = \langle e_5, e_4\rangle$,
- $r_c = \langle e_5, e_3, e_2\rangle$,
- $r_d = \langle e_1, e_2\rangle$.

След като сме избрали наредбите, образуваме четирите множества от ребра

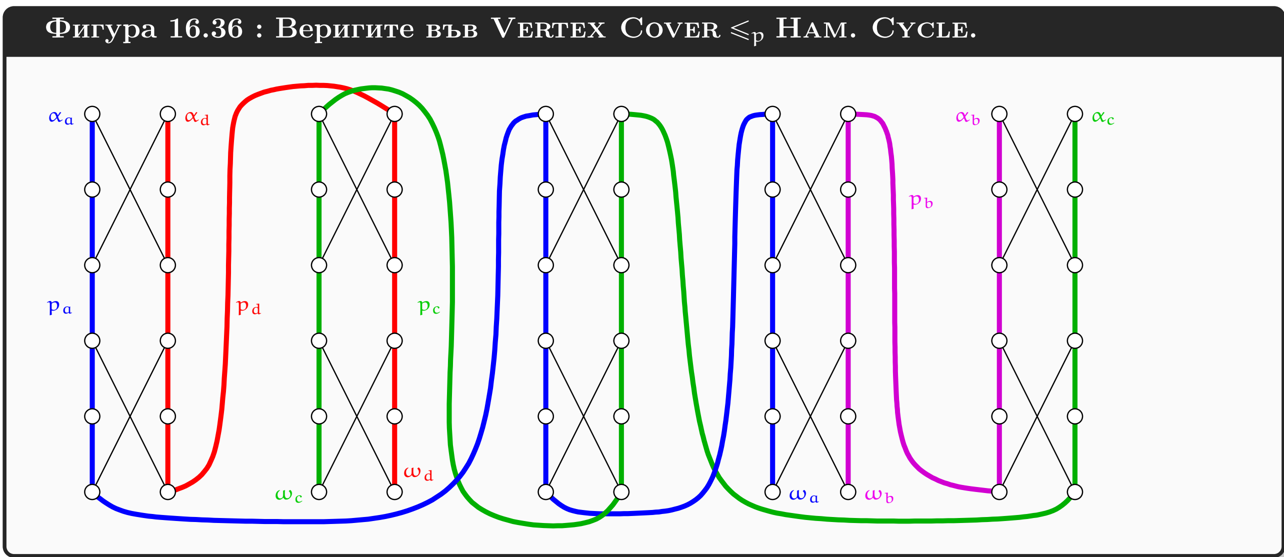
$$E''_a = \{(\langle a, e_1, 6\rangle, \langle a, e_3, 1\rangle), (\langle a, e_3, 6\rangle, \langle a, e_4, 1\rangle)\}$$
$$E''_b = \{(\langle b, e_4, 1\rangle, \langle b, e_5, 6\rangle)\}$$
$$E''_c = \{(\langle c, e_2, 1\rangle, \langle c, e_3, 6\rangle), (\langle c, e_3, 1\rangle, \langle c, e_5, 6\rangle)\}$$
$$E''_d = \{(\langle d, e_1, 6\rangle, \langle d, e_2, 1\rangle)\}$$

Тяхното обединение е E''. Фигура 16.35 показва E'' с дебели криви. Виждаме, че наистина E'' съдържа $2m - n = 10 - 4 = 6$ ребра.



Фигура 16.35 : Добавянето на ребрата от E'' във Vertex Cover $\leqslant_p$ Ham. Cycle.

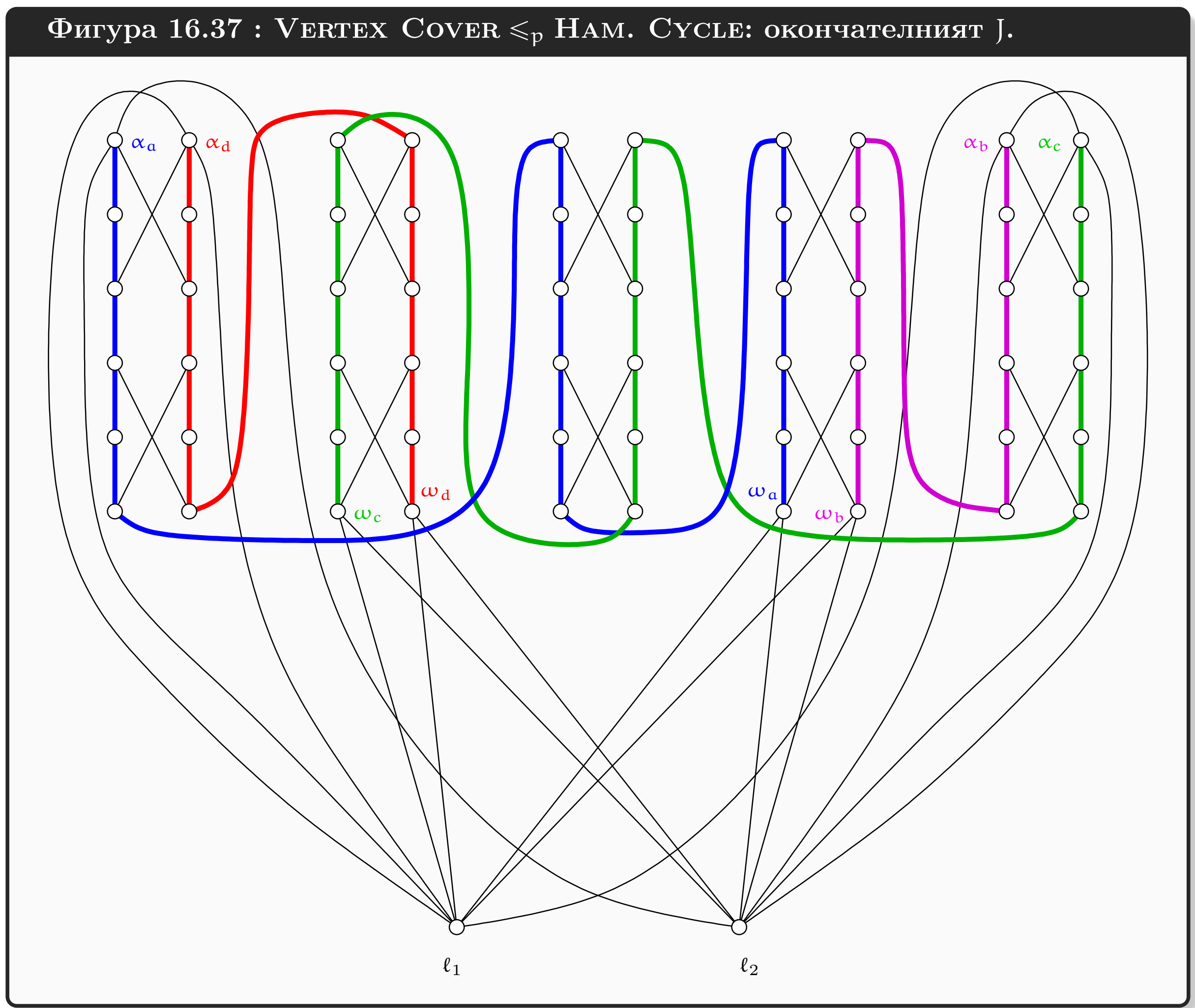
Да видим веригите в построеното досега. Те са четири, колкото са върховете на G. Именуваме ги p_a с краища α_a и ω_a, p_b с краища α_b и ω_b, и така нататък. Вижте Фигура 16.36.



Фигура 16.36 : Веригите във Vertex Cover $\leqslant_p$ Ham. Cycle.

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.



Фигура 16.37 : Vertex Cover $\leq_p$ Ham. Cycle: окончателният J.

Да разгледаме коректността на редукцията. Започваме с наблюдението, че преди слагането на върховете $\ell_1, \ldots, \ell_k$ 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 92: 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 $\ell_1, \ldots, \ell_k$ 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 $\ell_1, \ldots, \ell_n$. 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 92: Коректността на VERTEX COVER $\leqslant_p$ 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 $\leqslant k$, то има и върхово покриване с размер точно k. Нека покриващото множество за G е $U = \{u_1, u_2, \ldots, u_k\}$. Ще построим Хамилтонов цикъл C в J.

Разглеждаме множеството от вериги

$$P_U = \{p_{u_i} \,|\, 1 \leqslant i \leqslant k\}$$

Разглеждаме и следното подмножество на E''', което ще наречем E_U:

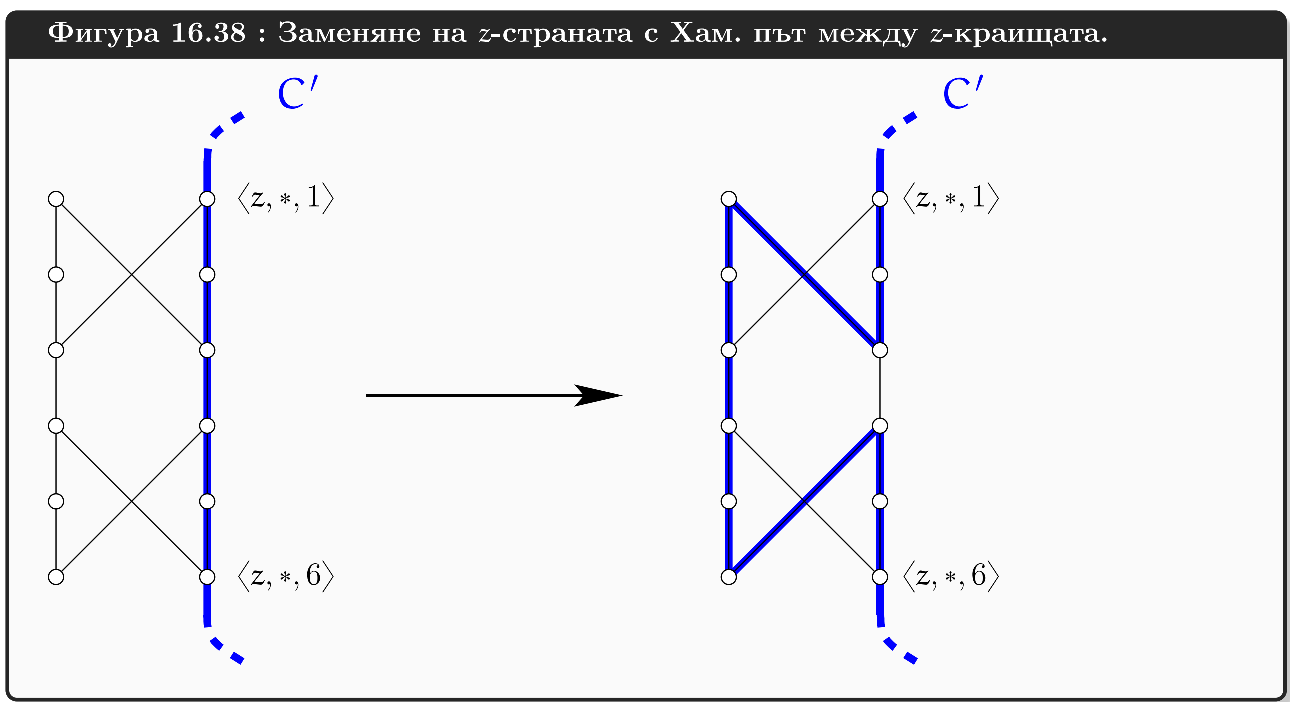
$$E_U = \{(\ell_1, \alpha_{u_1}), (\ell_1, \omega_{u_k})\} \cup \{(\ell_i, \omega_{u_{i-1}}) \,|\, 2 \leqslant i \leqslant k\} \cup \{(\ell_i, \alpha_{u_i}) \,|\, 2 \leqslant i \leqslant k\}$$

"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.

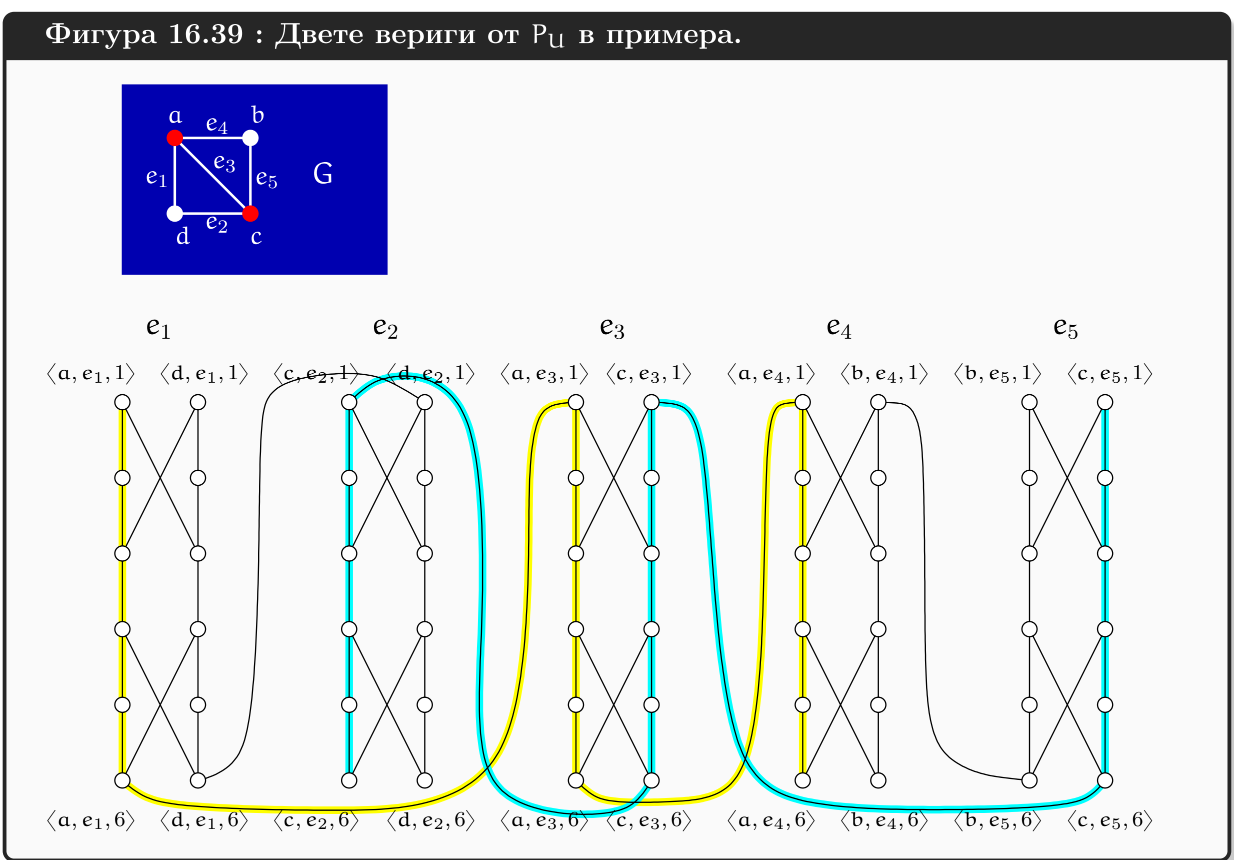
Фигура 16.38 : Заменяне на z-страната с Хам. път между z-краищата.



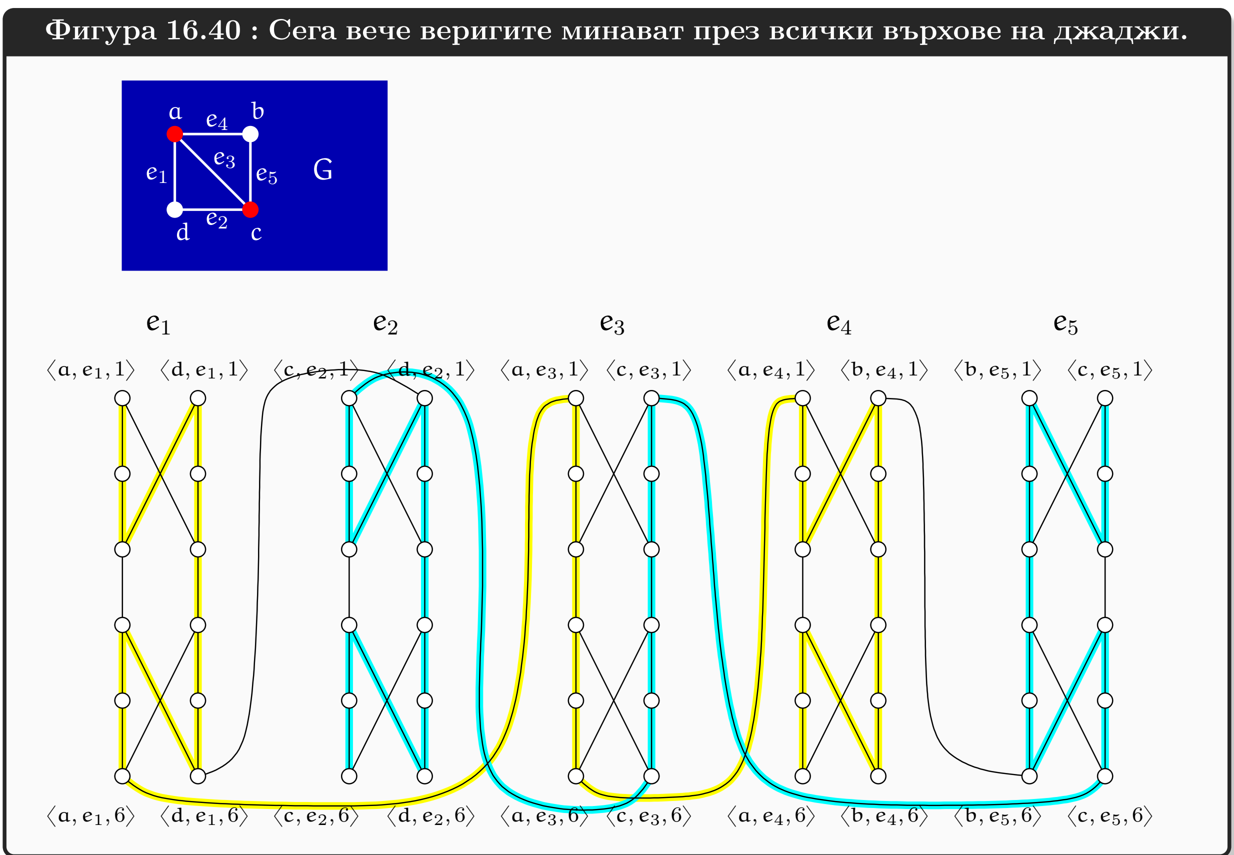
След като направим това за всички приспособления, през които минава точно една верига от P_U, наричаме получения C' с името "C" и забелязваме, че C е Хамилтонов цикъл в J.

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.



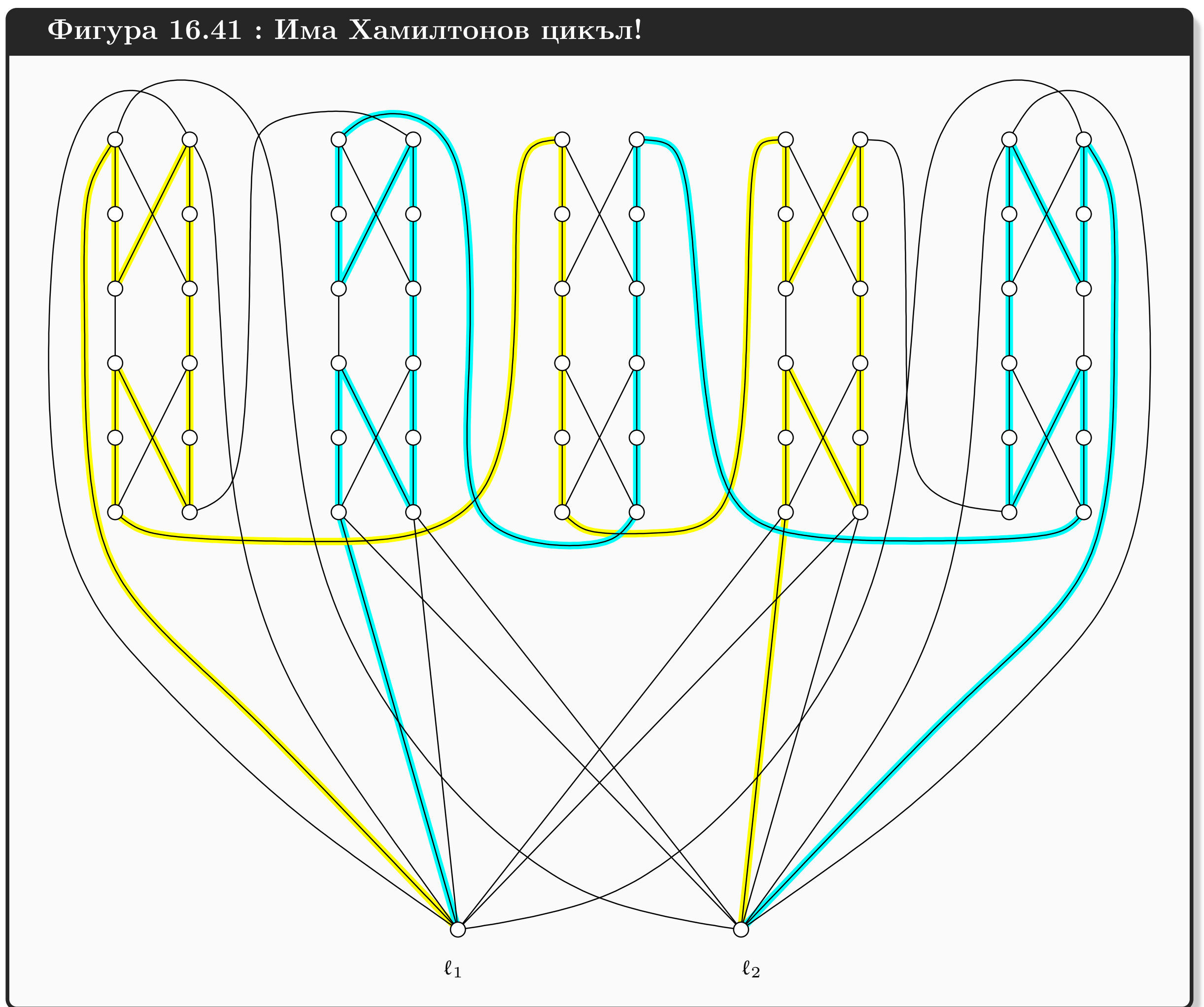
Ясно е, че от тези вериги и ребрата от E''' не можем да направим Хамилтонов цикъл, понеже четири от приспособленията има върхове, непринадлежащи нито на p_a, нито на p_c. Модифицираме минаването на веригите през приспособленията по такъв начин, че всеки връх на приспособление да е в точно една верига 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 $\ell_1, \ldots, \ell_k$ се среща точно веднъж в C. Да изтрием $\ell_1, \ldots, \ell_k$ от C. Получаваме колекция от k на брой пътя $q_1, \ldots, q_k$. 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 $q_1, \ldots, q_k$, минаващи по този начин през приспособленията, е същото като всяко ребро на G да е инцидентно с поне един връх от $u_1, \ldots, u_k$. Показахме, че G има върхово покриване с размер k. □

Разглеждаме Hamiltonian Path (Задача 57). Вече показахме, че Hamiltonian Cycle се редуцира до нея с Karp редукция (Редукция 3). Тогава Hamiltonian Path ∈ **NP-c**.

Дефинирахме задачата Longest Path между два върха като оптимизационна задача в Допълнение 43 в Лекция 11 (Задача 28). 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 80: Longest Cycle, версия за разпознаване

екземпляр: Неориентиран граф $G = (V, E)$, естествено число k.
въпрос: Дали в G има цикъл с дължина $\geqslant k$?

Изч. Задача 81: Longest Path, версия за разпознаване

екземпляр: Неориентиран граф $G = (V, E)$, естествено число k.
въпрос: Дали в G има път с дължина $\geqslant k$?

Редукция 24: Hamiltonian Cycle $\leqslant_p$ 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 Hamiltonian Cycle. □

Редукция 25: Hamiltonian Path $\leqslant_p$ 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 Hamiltonian Path. □

16.2.9 2-Partition

Разглеждаме Задача 33. 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 $\leqslant_p$ 2-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 Hamiltonian Cycle $\leqslant_p$ 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 $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$ и $D = \{d_1, \ldots, d_n\}$ са множества с празни сечения по двойки и $A \subseteq \{\{b, c, d\} \mid b \in B, c \in C, d \in D\}$.

Нека $A = \{a_1, \ldots, a_m\}$, където $a_j = \{b_{p_j}, c_{q_j}, d_{r_j}\}$ за $j \in \{1, \ldots, m\}$, където $p_j, q_j, r_j \in \{1, \ldots, n\}$. Ще конструираме съответен екземпляр $y = \langle S, w \rangle$ на 2-PARTITION, който е

БОО, $m \geqslant n$, 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 редукцията, необходимо и достатъчно е да дефинираме $w(s)$ за всяко $s \in S$. Да кажем, че $S = \{s_1, \ldots, s_m, s_{m+1}, s_{m+2}\}$. Елементите на S са два вида:

- $s_1, \ldots, s_m$ 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 "$\nu(x)$" числото, което е записано с x в двоична позиционна бройна система.

Теглата на $s_1, \ldots, s_m$. Конструираме двоични стрингове $\sigma_1, \sigma_2, \ldots, \sigma_m$, като за всяко $j \in \{1, \ldots, m\}$, $w(s_j) = \nu(\sigma_j)$. 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 $j \in \{1, \ldots, m\}$, 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 $j \in \{1, \ldots, m\}$, за всяко $\ell \in \{B, C, D\}$, $\sigma_{j,\ell}$ е конкатенация от n стринга $\sigma_{j,\ell,1}$, $\sigma_{j,\ell,2}$, $\ldots$, $\sigma_{j,\ell,n}$, 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 $j \in \{1, \ldots, m\}$:

- $\sigma_{j,B,p_j} = 00\cdots01$ и $\forall i \in \{1, \ldots, n\} \setminus \{p_j\} : \sigma_{j,B,i} = 00\cdots0$;
- $\sigma_{j,C,q_j} = 00\cdots01$ и $\forall i \in \{1, \ldots, n\} \setminus \{q_j\} : \sigma_{j,C,i} = 00\cdots0$;
- $\sigma_{j,D,r_j} = 00\cdots01$ и $\forall i \in \{1, \ldots, n\} \setminus \{r_j\} : \sigma_{j,D,i} = 00\cdots0$.

И така, в σ_j има точно три единици. Лесно се вижда, че

$$\nu(\sigma_j) = 2^{g(3n-p_j)} + 2^{g(2n-q_j)} + 2^{g(n-r_j)} \tag{16.11}$$

за всяко $j \in \{1, \ldots, m\}$.

[†]Забележете, че g е броят на битовете в записа 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

$$A = \{\{b_1, c_1, d_1\}, \{b_3, c_2, d_2\}, \{b_2, c_3, d_4\}, \{b_4, c_4, d_3\}, \{b_1, c_2, d_2\}, \{b_2, c_3, d_3\}, \{b_3, c_1, d_1\}, \{b_4, c_4, d_1\}\}$$

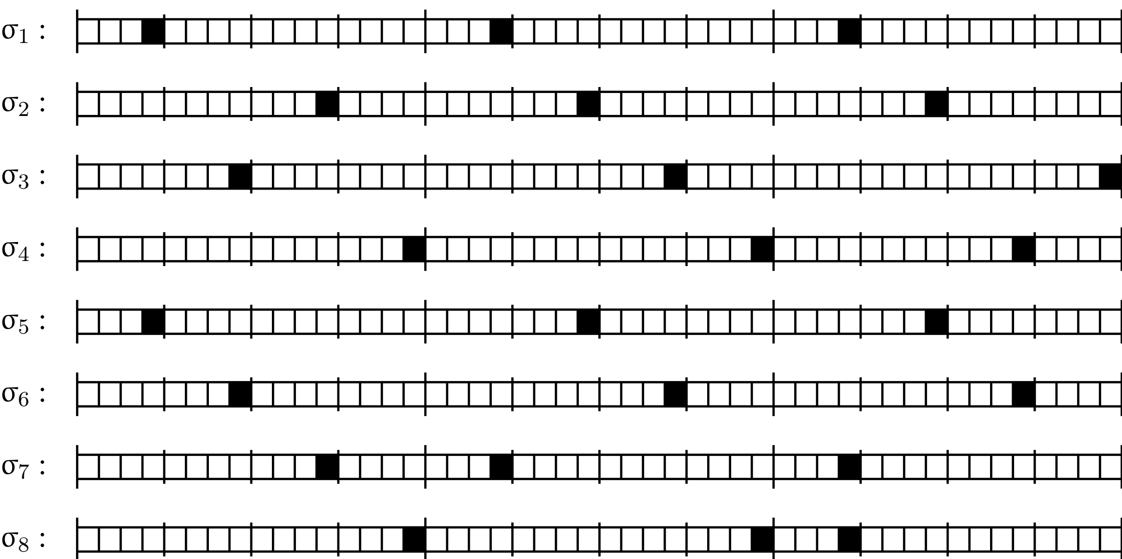
В този пример $n = 4$, $m = 8$ и $g = 4$. Конструкцията генерира осем двоични стринга $\sigma_1, \ldots, \sigma_8$, всеки от които има дължина $3 \times 4 \times 4 = 48$. Всеки σ_j е конкатенация от три подстринга, всеки с дължина 16, като на свой ред всеки от тези подстрингове е контатенация от 4 подподстринга с дължина 4. Ето общата схема на кой да е σ_j.

$\sigma_{j,B}$ $\sigma_{j,C}$ $\sigma_{j,D}$

$\sigma_{j,B,1}$ $\sigma_{j,B,2}$ $\sigma_{j,B,3}$ $\sigma_{j,B,4}$ $\sigma_{j,C,1}$ $\sigma_{j,C,2}$ $\sigma_{j,C,3}$ $\sigma_{j,C,4}$ $\sigma_{j,D,1}$ $\sigma_{j,D,2}$ $\sigma_{j,C,3}$ $\sigma_{j,D,4}$

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.

$$\begin{aligned}
\nu(\sigma_1) &= 2^{44} + 2^{28} + 2^{12} = 17\,592\,454\,483\,968\\
\nu(\sigma_2) &= 2^{36} + 2^{24} + 2^{8} \;\; = 68\,736\,254\,208\\
\nu(\sigma_3) &= 2^{40} + 2^{20} + 2^{0} \;\; = 1\,099\,512\,676\,353\\
\nu(\sigma_4) &= 2^{32} + 2^{16} + 2^{4} \;\; = 4\,295\,032\,848\\
\nu(\sigma_5) &= 2^{44} + 2^{24} + 2^{8} \;\; = 17\,592\,202\,821\,888\\
\nu(\sigma_6) &= 2^{40} + 2^{20} + 2^{4} \;\; = 1\,099\,512\,676\,368\\
\nu(\sigma_7) &= 2^{36} + 2^{28} + 2^{12} = 68\,987\,916\,288\\
\nu(\sigma_8) &= 2^{32} + 2^{16} + 2^{4} \;\; = 4\,295\,032\,848
\end{aligned}$$

Да се убедим, че (16.11) е в сила за $\nu(\sigma_1)$. Наистина, $p_1 = q_1 = r_1 = 1$, така че (16.11) дава

$$\nu(\sigma_1) = 2^{g(3n-p_1)} + 2^{g(2n-q_1)} + 2^{g(n-r_1)} = 2^{4(12-1)} + 2^{4(8-1)} + 2^{4(4-1)} = 2^{4\times 11} + 2^{4\times 7} + 2^{4\times 3}$$

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, …, $(3n-1)g$, 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.

$$w(s_{m+1}) = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H \tag{16.12}$$

$$w(s_{m+2}) = \left(\sum_{j=1}^{m} \nu(\sigma_j)\right) + H \tag{16.13}$$

И това е краят на описанието на конструкцията.

В примера, който разглеждаме, $\sum_{i=1}^{8} \nu(\sigma_i) = 37\,529\,996\,894\,769$. Тогава

$$w(s_{m+1}) = 2 \times 37\,529\,996\,894\,769 - 18\,764\,998\,447\,377 = 56\,294\,995\,342\,161$$
$$w(s_{m+2}) = 37\,529\,996\,894\,769 + 18\,764\,998\,447\,377 = 56\,294\,995\,342\,146$$

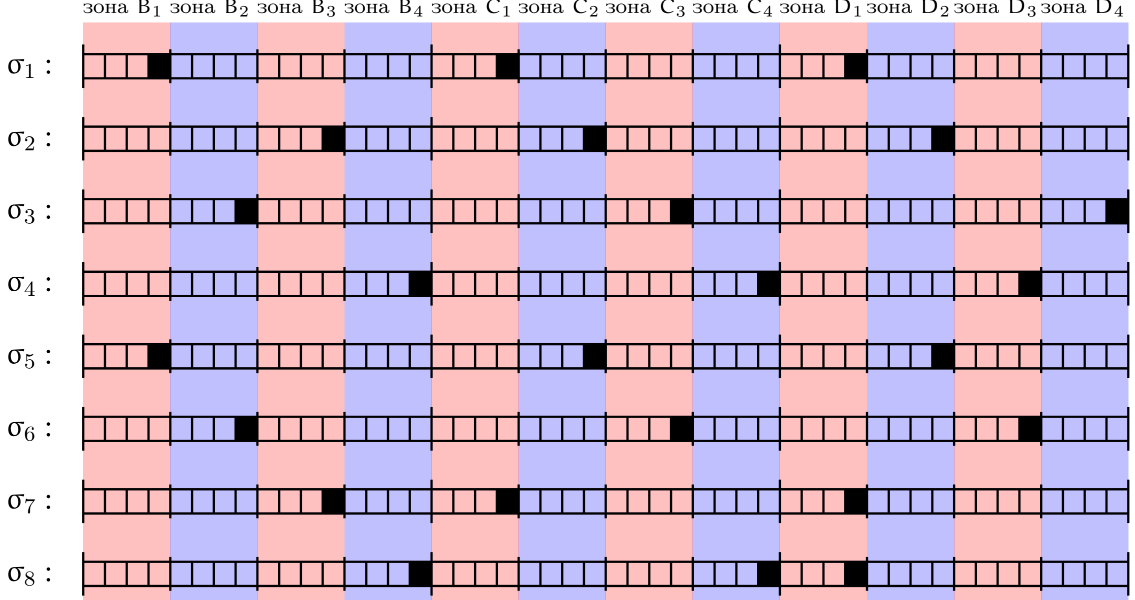
Коректност на конструкцията Когато става дума за сумиране на числата $\nu(\sigma_j)$, както е в (16.12) и (16.13), удобно е да мислим за сумиране на директно на стринговете σ_j. Тъй като тези стрингове са записите на числата в двоична бройна система, това е нещо напълно естествено.

Удобно е да мислим за всички стрингове σ_1, …, σ_m, наредени един над друг и подравнени вляво (и в дясно, щом са с еднакви дължини). Да дефинираме, че

- подподстринговете $\sigma_{1,B,1}$, …, $\sigma_{m,B,1}$ образуват *зона* B_1,
- подподстринговете $\sigma_{1,B,2}$, …, $\sigma_{m,B,2}$ образуват *зона* B_2,
- и така нататък,
- подподстринговете $\sigma_{1,B,n}$, …, $\sigma_{m,B,n}$ образуват *зона* B_n,
- подподстринговете $\sigma_{1,C,1}$, …, $\sigma_{m,C,1}$ образуват *зона* C_1,
- подподстринговете $\sigma_{1,C,2}$, …, $\sigma_{m,C,2}$ образуват *зона* C_2,
- и така нататък,

- подподстринговете $\sigma_{1,C,n}, \ldots, \sigma_{m,C,n}$ образуват *зона* C_n,
- подподстринговете $\sigma_{1,D,1}, \ldots, \sigma_{m,D,1}$ образуват *зона* D_1,
- подподстринговете $\sigma_{1,D,2}, \ldots, \sigma_{m,D,2}$ образуват *зона* D_2,
- и така нататък,
- подподстринговете $\sigma_{1,D,n}, \ldots, \sigma_{m,D,n}$ 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 x е ДА-екземпляр на 3DM. Има смисъл да разсъждаваме върху това, понеже дефиницията (16.12) на $w(s_{m+1})$ използва разлика. Щом 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 B_1. След сумирането $\sigma_1 + \cdots + \sigma_m$ 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. Заключаваме, че със сигурност в записа на удвоената сума има единица на позиция, която е вляво от позицията на най-лявата единица на H. Тогава $2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H > 0$.

Да видим каква е сумата от теглата на всички елементи. Очевидно $w(s_{m+1}) + w(s_{m+2}) = 3\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. Тогава

$$\sum_{j=1}^{m+2} w(s_j) = \left(\sum_{j=1}^{m} \nu(\sigma_j)\right) + w(s_{m+1}) + w(s_{m+2}) = 4\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) \tag{16.14}$$

Наблюдение 93: При сумиране на единиците от зона не се получава препълване

Дори да има зона, която съдържа m единици, при сумирането $\sigma_1 + \cdots + \sigma_m$ не може да се получи препълване и прехвърляне на единици от тази зона в зоната вляво, ако има такава. Ширината $\lfloor \log_2 m \rfloor + 1$ на всяка зона е достатъчна, за да не се получи такова препълване. Следователно, след сумирането $\sigma_1 + \cdots + \sigma_m$, 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 69

$A' \subseteq A$ е перфектно съчетание тстк

$$\sum_{\sigma \in Z'} \nu(\sigma) = H$$

където Z' е подмножеството на $\{\sigma_1, \ldots, \sigma_m\}$, 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 93, 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. □

Теорема 93: Коректността на 3DM $\leqslant_p$ 2-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 69 знаем, че сумата от числата, съответни на стринговете от Z', е точно H. Но тогава

$$\sum_{\sigma \in Z'} \nu(\sigma) + w(s_{m+1}) = H + 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$$

Но от (16.14) знаем, че $\sum_{j=1}^{m+2} w(s_j) = 4\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. Тогава $\sum_{\sigma \in Z'} \nu(\sigma) + w(s_{m+1})$ 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 $\exists S' \subset \{s_1, s_2, \ldots, s_m, s_{m+1}, s_{m+2}\}$, такова че $\sum_{s \in S'} w(s) = \sum_{s \in S \setminus S'} w(s)$. Както вече знаем, $\sum_{j=1}^{m+2} w(s_j) = 4\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. Тогава $\sum_{s \in S'} w(s) = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$.

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: $w(s_{m+1}) + w(s_{m+2}) = 3\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. БОО, нека $s_{m+1} \in S'$. Тогава

$$\sum_{s \in S' \setminus \{s_{m+1}\}} w(s) = \underbrace{2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)}_{\text{сумалното тегло на } S'} - \left(\underbrace{2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H}_{\text{теглото на } s_{m+1}}\right) = H$$

И така, има подмножество от двоични стрингове, такова че числата, съответни на неговите елементи се сумират до H. Съгласно Лема 69, A съдържа перфектно съчетание. □

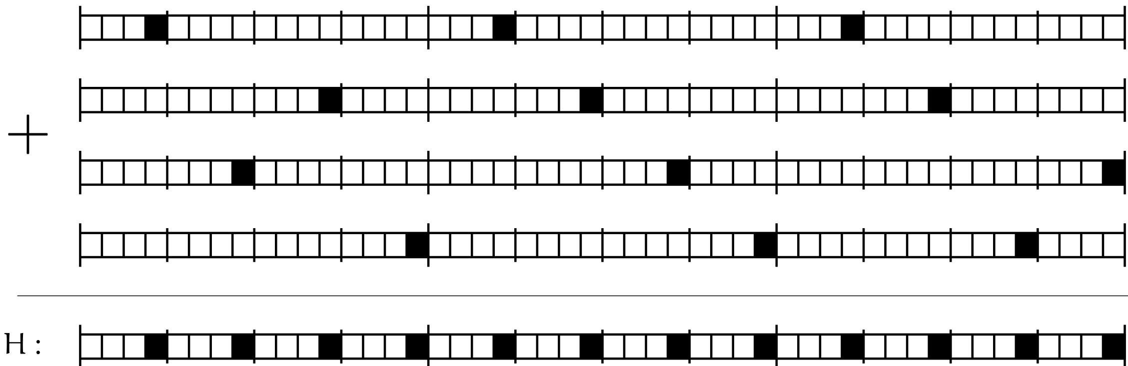
Да разгледаме отново примера, който въведохме горе:

$$A = \{\{b_1, c_1, d_1\}, \{b_3, c_2, d_2\}, \{b_2, c_3, d_4\}, \{b_4, c_4, d_3\},$$
$$\{b_1, c_2, d_2\}, \{b_2, c_3, d_3\}, \{b_3, c_1, d_1\}, \{b_4, c_4, d_1\}\}$$

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 H.



16.2.10 Knapsack

Тази задача въведохме в Лекция 12, Секция 12.6.3, и то в три варианта. Тук ни интересува задачата във варианта 0-1 Knapsack, който видяхме в Подсекция 12.6.3.2. Но в Подсекция 12.6.3.2 задачата беше оптимизационна, а за целите на редукцията ни трябва задачата във версия за разпознаване.

Изч. Задача 82: 0-1 Knapsack, версия за разпознаване

екземпляр: Множество от предмети $A = \{a_1, \ldots, a_n\}$. За всяко a_i, $1 \leqslant i \leqslant n$, 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:

$$\sum_{a \in X} w(a) \leqslant C$$
$$\sum_{a \in X} v(a) \geqslant G?$$

Редукция 27: 2-Partition $\leqslant_p$ 0-1 Knapsack, разпознаване

Конструкция: По даден екземпляр $x = \langle S, f \rangle$ на 2-Partition, където $S = \{s_1, s_2, \ldots, s_n\}$ е множество и $f : S \to \mathbb{N}^+$ е тегловна функция върху него конструираме пример $y = \langle A, v, w, C, G \rangle$ на Knapsack, където A е множество, $v : A \to \mathbb{N}^+$ е функцията на стойностите,

$w : A \to \mathbb{N}^+$ е функцията на теглата, $C \in \mathbb{N}^+$ е капацитетът и $G \in \mathbb{N}^+$. А именно,

$$
\begin{aligned}
A &\leftarrow S \\
w &\leftarrow f \\
v &\leftarrow f \\
C &\leftarrow \left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor \\
G &\leftarrow \left\lceil \frac{1}{2} \sum_{s \in S} f(s) \right\rceil
\end{aligned}
$$

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 $\sum_{s \in X} w(s) \leqslant C$ и $\sum_{s \in X} v(s) \geqslant G$ са спазени за така дефинираните C и G. В обратната посока, ако съществува подмножество от предмети X, за което $\sum_{a \in X} w(a) \leqslant C$ и $\sum_{a \in X} v(a) \geqslant G$ са спазени, то X е единият дял на разбиване, каквото се иска в 2-PARTITION. □

16.2.11 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 83: BIN PACKING

екземпляр: Крайно непразно множество S. Функция $w : S \to \mathbb{N}^+$. Числа $C, k \in \mathbb{N}^+$.
въпрос: Дали съществува разбиване $\{S_1, S_2, \ldots, S_k\}$ на S, такова че

$$\forall j \in \{1, \ldots, k\} : \sum_{s \in S_j} w(s) \leqslant C \text{ ?}$$

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 тегава и технически сложна ([114, 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 $\leqslant_p$ 2-PARTITION ползва големи числа. Поради което както 2-PARTITION, така и KNAPSACK[†], 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

[†]Доказателството за неподатливостта на KNAPSACK, което направихме, е редукция от 2-PARTITION.

тази лекция това е достатъчно. Ако обаче искаме да изследваме BIN PACKING задълбочено, трябво 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 28: 2-PARTITION $\leqslant_p$ BIN PACKING

Конструкция: Даден е екземпляр $x = \langle S, w \rangle$ на 2-PARTITION. Ще конструираме екземпляр $y = \langle A, f, C, k \rangle$ 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 x е **Да**-екземпляр на 2-PARTITION тстк 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 x е **Да**-екземпляр на 2-PARTITION. Тогава $\sum_{s \in S} w(s)$ е четно число, така че $\left\lfloor \frac{1}{2} \sum_{s \in S} w(s) \right\rfloor = \frac{1}{2} \sum_{s \in S} w(s)$, 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 y е ДА-екземпляр на BIN PACKING. Тогава съществува разбиване $\{S_1, S_2\}$ на S, такова че $\sum_{s \in S_1} f(s) \leqslant \left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor$ и $\sum_{s \in S_2} f(s) \leqslant \left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor$. Тогава $\sum_{s \in S} f(s)$ е четно, така че $\left\lfloor \frac{1}{2} \sum_{s \in S} f(s) \right\rfloor = \frac{1}{2} \sum_{s \in S} f(s) = \sum_{s \in S_1} w(s) = \sum_{s \in S \setminus S_1} w(s)$. Тогава x е ДА-екземпляр на 2-PARTITION. □

16.2.12 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 $T = \{t_1, \ldots, t_n\}$. Задача t_i се изпълнява за $\ell(t_i)$ единици време, за $i \in \{1, \ldots, n\}$, 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 с

$$f : T \to \{1, \ldots, p\}$$

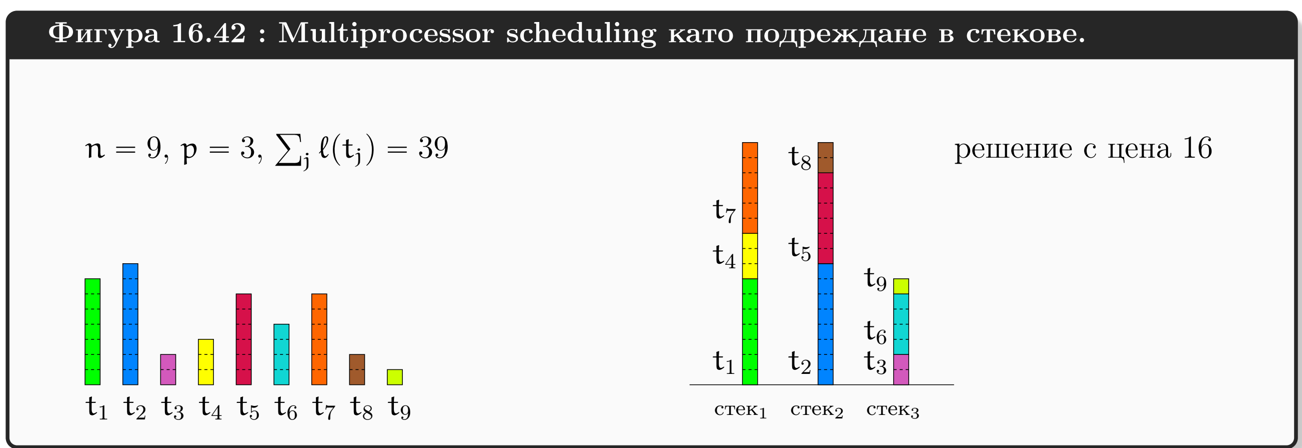
то цената на назначаването f е

$$\max_{1 \leqslant i \leqslant p} \sum_{t_j \in T,\, f(t_j)=i} \ell(t_j)$$

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 онагледи 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. Иска 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.



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.

Изч. Задача 84: MULTIPROCESSOR SCHEDULING, ВЕРСИЯ ЗА РАЗПОЗНАВАНЕ

екземпляр: Крайно непразно множество S. Функция $w : S \to \mathbb{N}^+$. Числа $p, D \in \mathbb{N}^+$.
въпрос: Дали съществува разбиване $\{S_1, S_2, \ldots, S_p\}$ такова че

$$\max_{1 \leqslant i \leqslant p} \sum_{s \in S_i} w(s) \leqslant D \text{ ?}$$

"D" идва от deadline – краен срок за изпълнението на задачите.

Редукция 29: 2-PARTITION $\leqslant_p$ 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 $D = \left\lfloor \frac{1}{2} \sum_{s \in S} w(s) \right\rfloor$. Коректността е очевидна. □

16.3 Когато принадлежността на задача към NP не е очевидна.

16.3.1 ⋆-Free Regexp Ineq

Въпросът дали два регулярни израза генерират един и същи език, тоест, дали са еквивалентни, е от голям интерес предвид широкото използване на регулярните езици. В Лекция 14 на стр. 699 видяхме, че задачата дали два регулярни израза, 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** [134]. Това са задачите, имащи полиномиална сложност по памет. Не е известно дали **PSPACE** е различен от **NP**, въпреки че преобладаващото мнение е, че $\mathbf{NP} \subset \mathbf{PSPACE}$.

Да си припомним формалната дефиниция на основното понятие.

Определение 136: 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 $p^\star$ 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 плюсът. Използваме скоби като мета-символи само когато има нужда.

Примери за регулярни изрази над $\{a, b\}$ са

$$
\begin{array}{lll}
\epsilon & a & b \\
a + b & (a + b)(a + b) & \epsilon aab(a + a + (b + aa)bb) \\
(a^\star)^\star & (a + b)^* & \emptyset(a + b)^\star \\
\emptyset^\star & (a^\star a^\star ba^\star + b^\star a^\star)^\star & (a^\star b^\star)^\star
\end{array}
$$

Определение 137: 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 $r = \varnothing$, то $L(r) = \varnothing$. 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 $r = p^\star$ за някой регулярен израз над Σ r, то $L(r) = (L(p))^\star$.

Примерно,

$$
\begin{aligned}
&L(\epsilon) = \{\epsilon\}, \quad L(a) = \{a\}, \quad L(b) = \{b\}, \quad L(a+b) = \{a, b\}\\
&L((a+b)(a+b)) = \{aa, ab, ba, bb\}\\
&L(\epsilon aab(a + a + (b + aa)bb)) = \{aaba, aabbbb, aabaabb\}\\
&L((a^\star)^\star) = \{\epsilon, a, aa, aaa, \dots\}\\
&L((a+b)^*) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}\\
&L(\varnothing(a+b)^\star) = \varnothing\\
&L(\varnothing^\star) = \{\epsilon\}\\
&L((a^\star a^\star ba^\star + b^\star a^\star)^\star) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}\\
&L((a^\star b^\star)^\star) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}
\end{aligned}
$$

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 136 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

$$
r = \underbrace{(a+b)(a+b)\cdots(a+b)}_{n \text{ “множители”}}
$$

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’ [108]. 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 75.

Изч. Задача 85: ⋆-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 94: ⋆-Free Regexp Ineq $\in$ NP

Съществува НМТ M, такава че за всеки Да-екземпляр x на ⋆-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 $\boldsymbol{\sigma}$ и след това, ползвайки сертификата, установява <u>в полиномиално време</u> , 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 136 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,

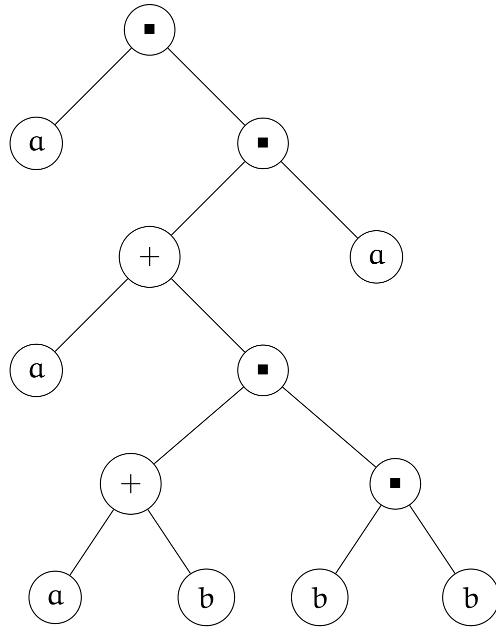
- всяка поява на $\emptyset$ като $\emptyset + p$ се редуцира до p,
- всяка поява на $\emptyset$ като $\emptyset p$ се редуцира до $\emptyset$, но тъй като $\emptyset$ не е в езика, в крайна сметка това $\emptyset$ ще изчезне,
- всяка поява на ϵ като $\epsilon + p$ може да се редуцира до p, понеже ϵ 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 $\boldsymbol{\sigma} \in \Sigma^*$, такъв че $\boldsymbol{\sigma} \in L(r_1) \triangle L(r_2)$. Магически се отгатва такъв стринг $\boldsymbol{\sigma}$ и след това се проверява, че единият от r_1, r_2 може да генерира $\boldsymbol{\sigma}$, 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 ща наречем u, е вярно, че $T[\![u]\!]$ може да генерира τ.

Тази идея може да се реализира чрез полиномиален в $|\boldsymbol{\sigma}|$, $|r_1|$ 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 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 $\boldsymbol{\sigma} \in S_z$, което е вярно тстк $\boldsymbol{\sigma}$ е в езика на съответния регулярен израз. В крайна сметка, машината M проверява, че $\boldsymbol{\sigma} \in L(r_1) \wedge \boldsymbol{\sigma} \notin L(r_2)$ или $\boldsymbol{\sigma} \in L(r_2) \wedge \boldsymbol{\sigma} \notin L(r_1)$.

Предвид това, че непразните подстрингове на $\boldsymbol{\sigma}$ са $\Theta(|\boldsymbol{\sigma}|^2)$ на брой и че комбинациите от връх на дърво и непразен подстринг на $\boldsymbol{\sigma}$ са само полиномиално много, алгоритъмът е с полиномиална сложност по време, ако се реализира грамотно. □

Наблюдение 94

Идеята от доказателството на Теорема 94 **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 30: 3SAT $\leqslant_p$ $\star$-FREE REGEXP INEQ

Конструкция: Даден е екземпляр ϕ на 3SAT, като ϕ е КНФ

$$\phi = \phi_1 \cdots \phi_m$$

Да кажем, че $\mathsf{Var}(\phi) = \{x_1, \ldots, x_n\}$. 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 $j \in \{1, \ldots, m\}$,

  $$p_j = q_{j,1} q_{j,2} \cdots q_{j,n}$$

  За $i \in \{1, \ldots, n\}$,

  $$q_{j,i} = \begin{cases} 1, & \text{ако литералът } \overline{x_i} \text{ участва във } \phi_j \\ 0, & \text{ако литералът } x_i \text{ участва във } \phi_j \\ (0+1), & \text{ако променливата } x_i \text{ не участва във } \phi_j \text{ с литерал} \end{cases}$$

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

$$\phi = (x_1 \vee x_3 \vee x_6)(\overline{x_2} \vee x_4 \vee x_6)(\overline{x_3} \vee \overline{x_5} \vee \overline{x_6})(\overline{x_2} \vee \overline{x_4} \vee x_5)$$

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.

$$101010 \notin L(\ 0(0+1)0(0+1)(0+1)0\ ) \tag{16.15}$$
$$101010 \notin L(\ (0+1)1(0+1)0(0+1)0\ ) \tag{16.16}$$
$$101010 \notin L(\ (0+1)(0+1)1(0+1)11\ ) \tag{16.17}$$
$$101010 \notin L(\ (0+1)1(0+1)10(0+1)\ ) \tag{16.18}$$

Това, че примерът използва удовлетворяваща валюация като стринг, който не е в $L(r_2)$, не е случайно. Също така не е случайно, че

- (16.15) 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,
- (16.16) 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,
- (16.17) 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,
- (16.18) 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 $t \in \mathsf{Val}(\mathsf{Var}(\phi))$ 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 $\mathbf{t}$ означава t като стринг с дължина n (в примера горе имахме $\mathbf{t} = 101010$). Ще покажем, че $\mathbf{t} \notin L(r_2)$. Ще покажем, че $\forall j \in \{1, \ldots, m\} : \mathbf{t} \notin L(p_j)$. Щом t е удовлетворяваща валюация, клаузата ϕ_j съдържа поне един литерал, да го наречем α_j, такъв че $\mathsf{TA}(\alpha_j, t) = 1$. 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 $\mathbf{y} \in L(p_j)$ е вярно, че $\mathbf{y}$ и $\mathbf{t}$ се различават в i-тата позиция. И това е за всяко j. Следователно, $\mathbf{t}$ не може да бъде генериран от r_2.

Сега да допуснем, че $\exists \boldsymbol{z} \in L(r_1) \backslash L(r_2)$. Това означава, че $\forall j \in \{1, \ldots, m\} : \boldsymbol{z} \notin L(p_j)$. Подробно написано

$$\forall j \in \{1, \ldots, m\}\ \exists i \in \{1, \ldots, n\} : (z_i = 0 \wedge q_{j,i} = 1) \vee (z_i = 1 \wedge q_{j,i} = 0)$$

Но тогава $\boldsymbol{z} \models \phi$. □

Допълнение 75: Задачи-допълнения и клас на сложност coNP

Определение 138: Задачата-допълнение на задача за разпознаване

Нека Π е задача за разпознаване и $\mathcal{I}$ е множеството от нейните екземпляри. *Задачата-допълнение на* Π, която бележим с $\overline{\Pi}$, също има за множество от екземпляри $\mathcal{I}$, но ДА-екземплярите и НЕ-екземплярите са разменени. Тоест, $\overline{\Pi}_Y = \Pi_N$ и $\overline{\Pi}_N = \Pi_Y$.

На прост български, $\overline{\Pi}$ се получава от Π 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 86: $\overline{\text{Свързаност на графи}}$

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 87: $\overline{\text{Hamiltonian Cycle}}$

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 Π, то може да модифицираме този алоритъм тривиално, разменяйки **return** True и **return** False, получавайки по този начин алгоритъм за $\overline{\Pi}$. 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 Π в смисъла на Определение 109. Тогава можем много лесно да конструираме МТ $\overline{M}$, която решава $\overline{\Pi}$: просто разменяме q_Y и q_N в таблицата на преходите на M и получаваме (таблицата на преходите на) $\overline{M}$.

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.

- M е машина с размножаване. M приема или отхвърля (кодирания на) графи по начин, близък до описанието на стр. 720: 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 M **не можем да получим** машина за $\overline{\text{Hamiltonian Cycle}}$ чрез проста размяна на q_Y и q_N по следните две причини.

1. Приемащото състояние и отхвърлящото състояние участват по различен начин в M: 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 (кодиране 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 "c е Хамилтонов".

   Ерго, задачата $\overline{\text{HAMILTONIAN CYCLE}}$ е дефинирана чрез универсален квантор, понеже нейният въпрос е негация на въпроса на HAMILTONIAN CYCLE. И така, в дефиницията на $\overline{\text{HAMILTONIAN CYCLE}}$ 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.

- M може да е със сертификат и работи, както е казано на стр. 722: сертификатът се генерира магически за единица време, а в детерминираната фаза M 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 M **не изглежда да има начин да получим** машина за $\overline{\text{HAMILTONIAN CYCLE}}$ 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 къс сертификат за $\overline{\text{HAMILTONIAN CYCLE}}$ няма, но опитите на компютърните учени в продължение на десетилетия да открият къс сертификат за задача-допълнение на **NP**-пълни задача не са довели до успех. Прочее, както предстои да видим, откриването на такъв къс сертификат би означавало, че **P** = **NP**.

Дано от този пример става ясно, че в общия случай недетерминираните машини на Turing за задачите-допълнения нямат нищо общо с недетерминираните 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 139: Клас на сложност coNP

Класът на сложност **NP**, по отношение на дадена азбука Σ, е

$$\mathbf{coNP} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \overline{L} \in \mathbf{NP}\}$$

В това обаче има малък формален проблем: както знаем (Наблюдение 76), езикът-допълнение $\overline{L}$ включва не само кодиранията на Не-екземплярите, а и стринговете, които са невалидни кодирания. Papadimitriou адресира този проблем [114, 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 [114, стр. 219], че задачите от **coNP** притежават къси *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 139 говори за допълнение (на език), класовете **NP** и **coNP** не са взаимно комплементарни! Най-малкото, те имат непразно 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 28.

Класът **coNP** съдържа пълни спрямо полиномиални редукции задачи, също като **NP**. Сравнете Определение 140 с Определение 125.

Определение 140: Клас на сложност coNP-c

Нека Π е задача за разпознаване. Казваме, че Π е ***coNP**-пълна*, ако

1. $\Pi \in \mathbf{coNP}$ и
2. за всяка задача $\Pi' \in \mathbf{coNP}$ е вярно, че $\Pi' \leqslant_p \Pi$.

Класът на сложност **coNP-c** се състои точно от **coNP**-пълните задачи.

На английски терминът е ***coNP**-complete*.

Да си припомним, че булева формула ϕ е тавтология тстк $\forall t \in \mathsf{Val}(\mathsf{Var}(\phi)) : t \models \phi$.

Изч. Задача 88: 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 85, понеже ще ползваме Теорема 85.

Теорема 95: Tautology $\in$ coNP-c

Tautology е **coNP**-пълна.

Доказателство: Първо ще покажем, че Tautology $\in$ **coNP**. Но това е почти очевидно, понеже $\overline{\text{Tautology}} \in$ **NP**: задачата $\overline{\text{Tautology}}$ пита дали булева формула ϕ има поне една фалшифицираща валюация 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 $\forall \Pi \in$ **coNP** : $\Pi \leqslant_p$ Tautology. Щом $\Pi \in$ **coNP**, вярно е, че $\overline{\Pi} \in$ **NP**. Щом $\overline{\Pi} \in$ **NP**, от Теорема 85 следва, че $\overline{\Pi} \leqslant_p$ SAT. Karp-редукцията на $\overline{\Pi}$ към SAT е функция, която изобразява всеки екземпляр x на $\overline{\Pi}$ в булева формула $\varphi(x)$, която е удовлетворима тстк $x \in \overline{\Pi}_Y$.
Но $x \in \overline{\Pi}_Y$ тстк $x \in \Pi_N$ съгласно Определение 138. Следва, че $\varphi(x)$ е удовлетворима тстк $x \in \Pi_N$. Но тогава $\varphi(x)$ е противоречие тстк $x \in \Pi_Y$. Прилагаме Наблюдение 82 и заключаваме, че $\neg\varphi(x)$ е тавтология тстк $x \in \Pi_Y$. Забележете, че $\neg\varphi(x)$ е екземпляр на Tautology, понеже е булева формула.
И това е краят на доказателството. Показахме, функция, която изобразява произволен екземпляр x на Π в екземпляр на Tautology, и то така, че x е Да-екземпляр тстк изображението е тавтология. □

Tautology не е единствената **coNP**-пълна задача, чиято задача-допълнение е **NP**-пълна. За всяка **NP**-пълна задача Π е вярно, че $\overline{\Pi}$ е **coNP**-пълна [114, стр. 220, Proposition 10.1].

Както знаем от Теорема 85, 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-c**,
2. то задачата-допълнение дали КНФ ϕ е противоречие е в **NP-c**.

Но тогава заради дуалността е вярно, че задачата дали ДНФ ψ е тавтология също е в **NP-c**. Таблица 16.1 резюмира тези резултати.

Вид на ϕ \ Въпрос за ϕ	ϕ е удовлетворима?	ϕ е фалшифицируема?	ϕ е тавтология?	ϕ е противоречие?
ϕ е КНФ	**NP-c**	винаги	никога	**coNP-c**
ϕ е ДНФ	винаги	**NP-c**	**coNP-c**	никога

Таблица 16.1: Сложност на базови задачи върху КНФ/ДНФ.

Интересен въпрос е $\mathbf{NP} \stackrel{?}{=} \mathbf{coNP}$. За съжаление, или за щастие, да се отговори на него е поне толкова трудно, колкото да се отговори на $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Теорема 96: P = NP влече NP = coNP

Ако се окаже, че $\mathbf{P} = \mathbf{NP}$, ще следва, че $\mathbf{NP} = \mathbf{coNP}$.

Доказателство: $\mathbf{P}$ е затворен спрямо допълнение; тоест, $\mathbf{P} = \mathbf{coP}$, където

$$\mathbf{coP} \stackrel{\text{def}}{=} \{L \subseteq \Sigma^* \mid \overline{L} \in \mathbf{P}\}$$

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 $\mathbf{P} = \mathbf{NP}$ и при положение, че $\mathbf{P} = \mathbf{coP}$, веднага следва $\mathbf{NP} = \mathbf{coNP}$. □

Конверсното на Теорема 96 обаче не е вярно. Възможно е $\mathbf{P} \neq \mathbf{NP}$ и въпреки това $\mathbf{NP} = \mathbf{coNP}$.

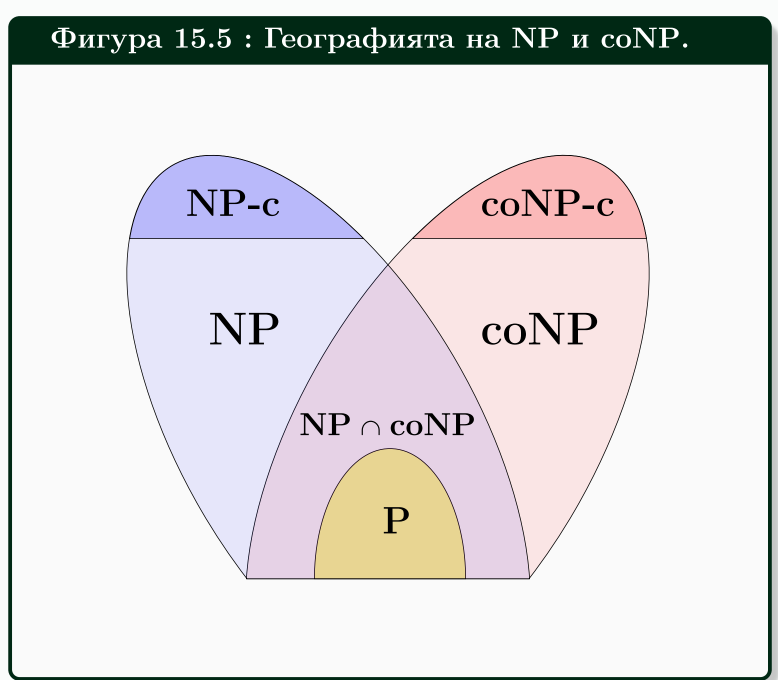
Това е Proposition 10.2 в [114, стр. 220].

Теорема 97: $\Pi \in$ coNP-c $\wedge$ $\Pi \in$ NP влече NP = coNP

Ако се окаже, че **coNP**-пълна задача е в **NP**, ще следва, че $\mathbf{NP} = \mathbf{coNP}$.

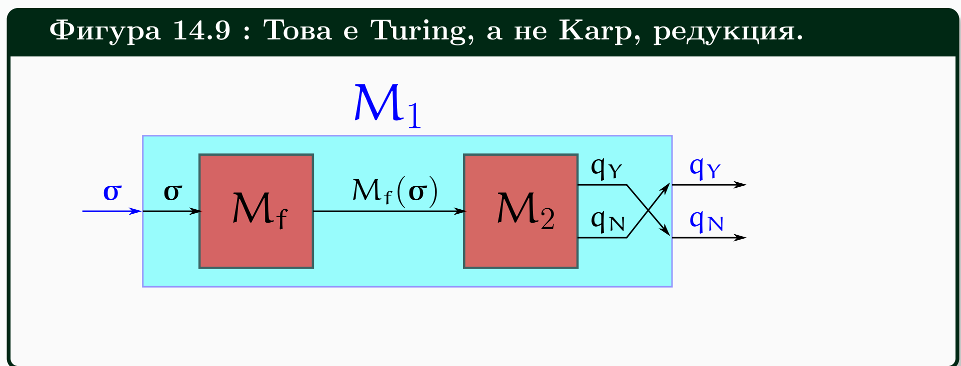
Доказателство: Ще покажем, че ако задача Π е в **NP** е **coNP**-пълна, то $\mathbf{coNP} \subseteq \mathbf{NP}$. В обратната посока доказателството става симетрично.
Щом $\Pi \in \mathbf{NP}$, съществува НМТ M, която приема (недетерминирано) екземплярите на Π.
Нека Π' е произволна задача от **coNP**. Щом Π е **coNP**-пълна, $\Pi' \leqslant_p \Pi$ с полиномиална редукция f. Нека $\mathbf{x}$ е екземпляр на Π'. Тогава $f(\mathbf{x})$ е екземпляр на Π, построим от $\mathbf{x}$ в полиномиално време. Тогава M приема $f(\mathbf{x})$ в полиномиално в $|\mathbf{x}|$ време. Тогава Π' е в **NP**. □

Широко разпространеното вярване сред специалистите е, че $\mathbf{NP} \neq \mathbf{coNP}$, понеже десетилетни усилия да се намери къс 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 Karp редукциите (Определение 120). Turing редукциите (Определение 124) са прекалено мощни и не различават тези два класа на сложност. За да се убедим в това, да си представим редукция, която прилича на Karp 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 Karp, 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 машина/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** Karp редукция е, че разменя q_Y с q_N в окончателния отговор. Макар и просто действие, то не е разрешено при Karp редуциите.
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.

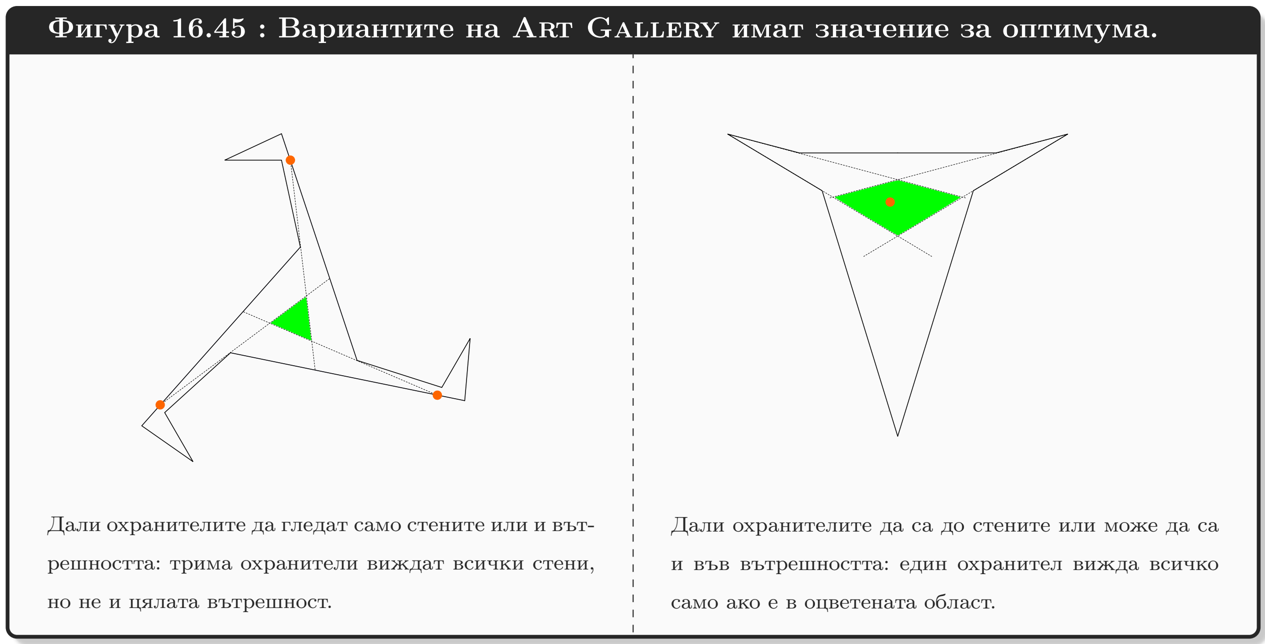
16.3.2 Art Gallery

Това е задача от изчислителната геометрия с приложение в области като роботика, sensor networks и surveillance [136]. Има книга, посветена на нейните варианти [111], 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 [120]. "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" насред 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 [27]:

> 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 [27], 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 [43], в която доказва на няколко реда, че $\left\lfloor \frac{n}{3} \right\rfloor$ 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 [111, стр. 12]. В [111, 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 $\left\lfloor \frac{n}{3} \right\rfloor$.

Желаният извод следва веднага.

Допълнение 76: Point Visibility Graphs

Терминът "dominates" в цитата на Chvátal на предишната страница не е случаен. Сега ще видим, че нашата дефиниция на "доминиране" (Задача 49) 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 тстк 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 $O(n)$ [52]. 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 [52]. 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 [99].

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 89: 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 98: Art Gallery ∈ 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 [42, 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 $O(n^2k^2)$. 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 [64]. □

Оригиналната редукция от **NP**-пълна задача към Art Gallery е от 3SAT [95]. Тази редукция е разказана и в [111, стр. 239–242]. Известна е обаче и много по-лесно разбираема редукция, която е от Vertex Cover Сега ще видим нея.

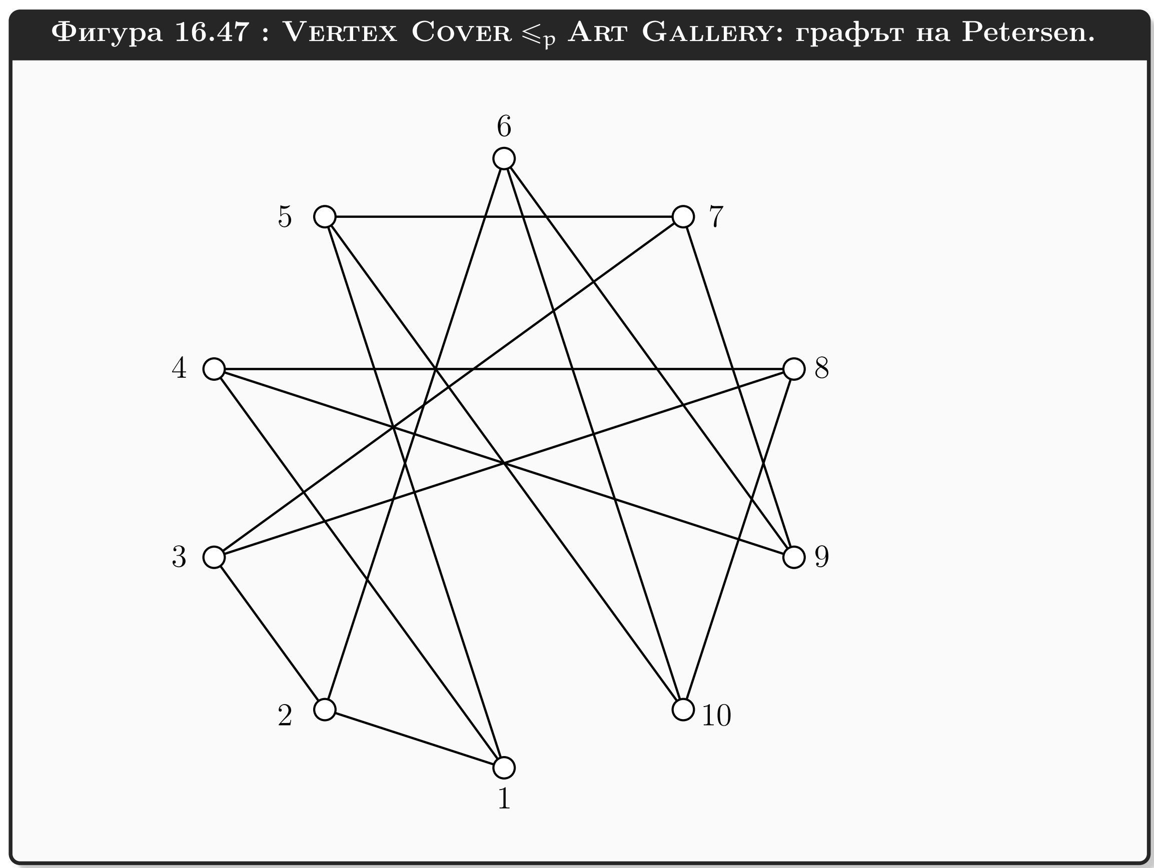
Редукция 31: Vertex Cover $\leqslant_p$ Art Gallery

Конструкция: Тази конструкция е описана, например, в [108, 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-екезмпляр 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. 588, 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.

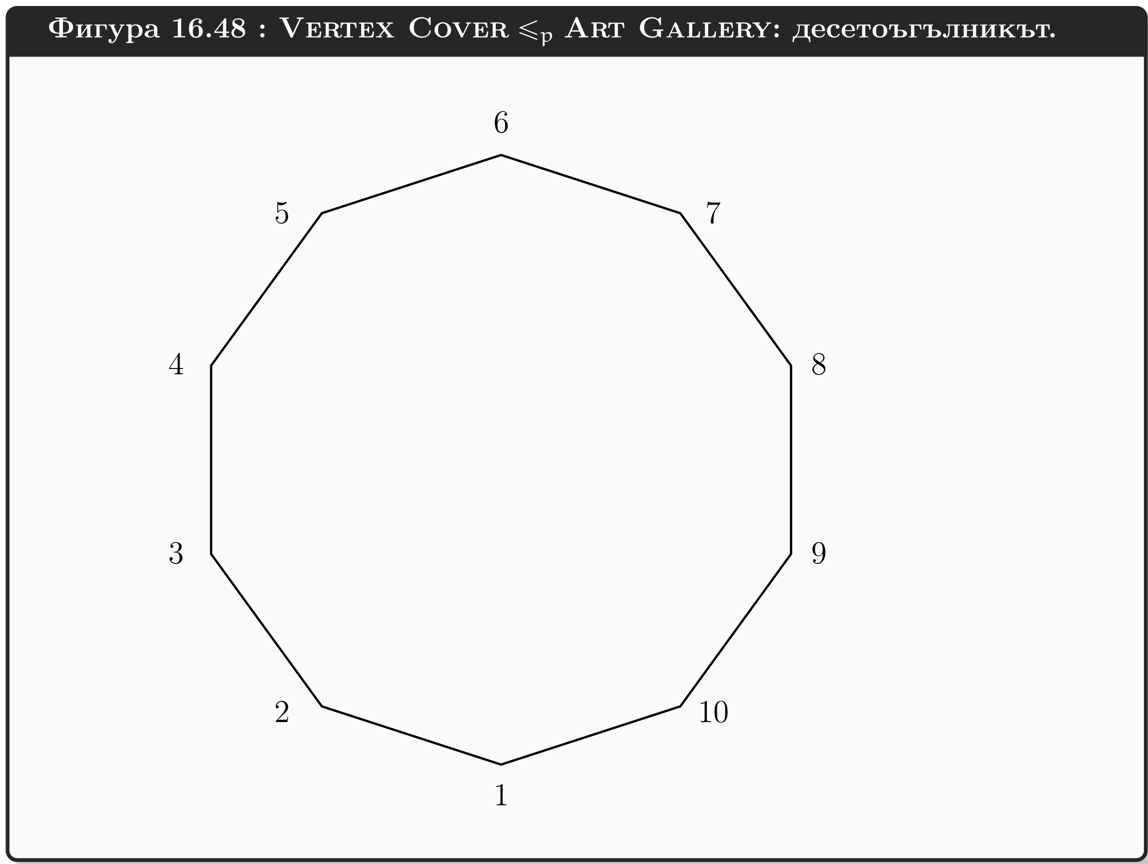
Каноничната рисунка на графа на Petersen в учебниците е тази, която е показана на Фигура 12.1. Тук ще нарисуваме графа на Petersen по друг начин: върховете са върхове на правилен десетоъгълник, именувани с 1, ..., 10, а ребрата са петнадесет отсечки между тях. Фигура 16.47 показва това.

Фигура 16.47 : Vertex Cover $\leqslant_p$ Art Gallery: графът на Petersen.



Конструкцията започва с правилен десетоъгълник, чиито върхове са именувани с 1, ..., 10. Вижте Фигура 16.48.

Фигура 16.48 : Vertex Cover $\leqslant_p$ Art Gallery: десетоъгълникът.

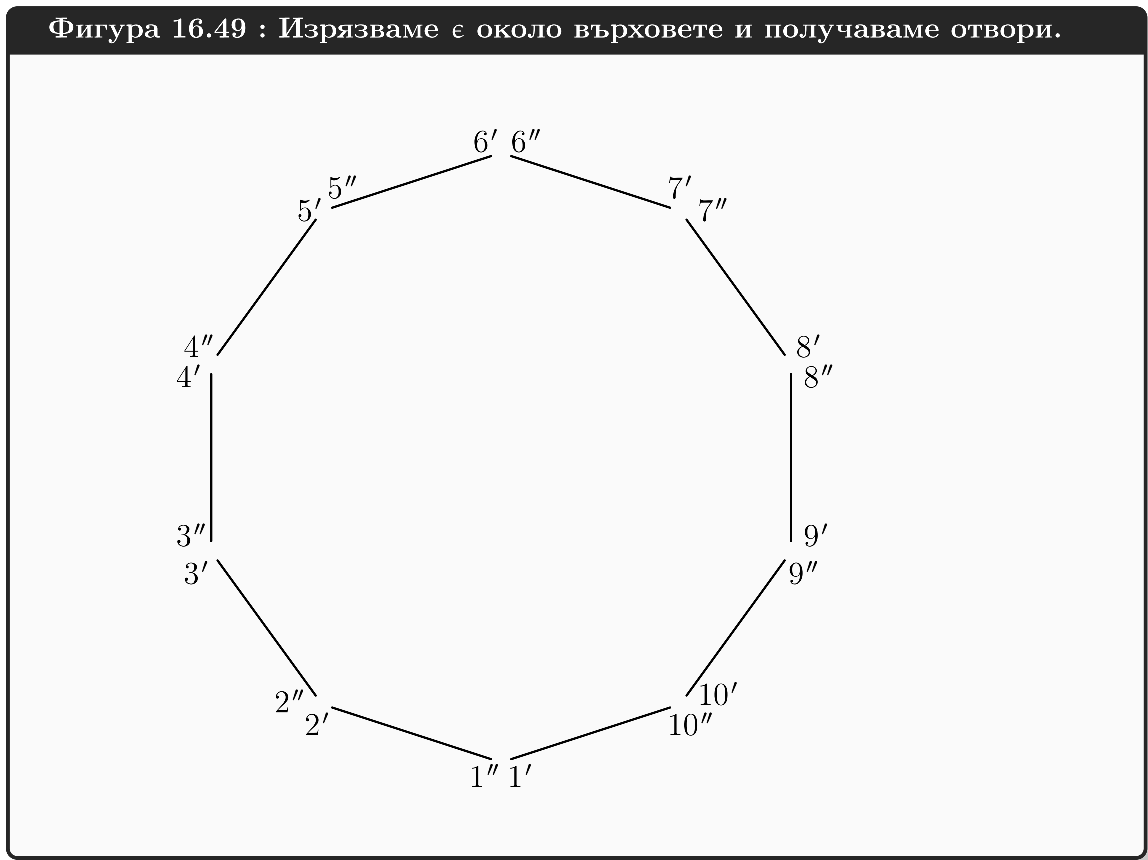


Махаме десет малки парчета от десетоъгълника: около всеки връх построяваме окръжност с радиус ϵ, за някакво достатъчно малко ϵ, и изтриваме сечението на десетоъгълника с вътрешността на окръжността. С други думи, изтриваме тези части от околните страни, които са на разстояние $< \epsilon$ от върховете.

По този начин получаваме десет отсечки с празни сечения по двойки, всяка от които е "остатъкът" от една от страните на бившия десетогълник. На всеки връх $\mathtt{i}$ от бившия десетоъгълник съответсват два от краищата на тези отсечки, които именуваме $\mathtt{i}'$ и $\mathtt{i}''$. Нека $\mathtt{i}'$ се появява преди $\mathtt{i}''$ при обикаляне по часовниковата стрелка. Вижте Фигура 16.49.

Точките $1'$, $1''$, $2'$, $2''$, ..., $10'$, $10''$ наричаме *особените точки*. Между всяка $1'$ и $1''$ се намира *отворът* $\mathtt{i}$.

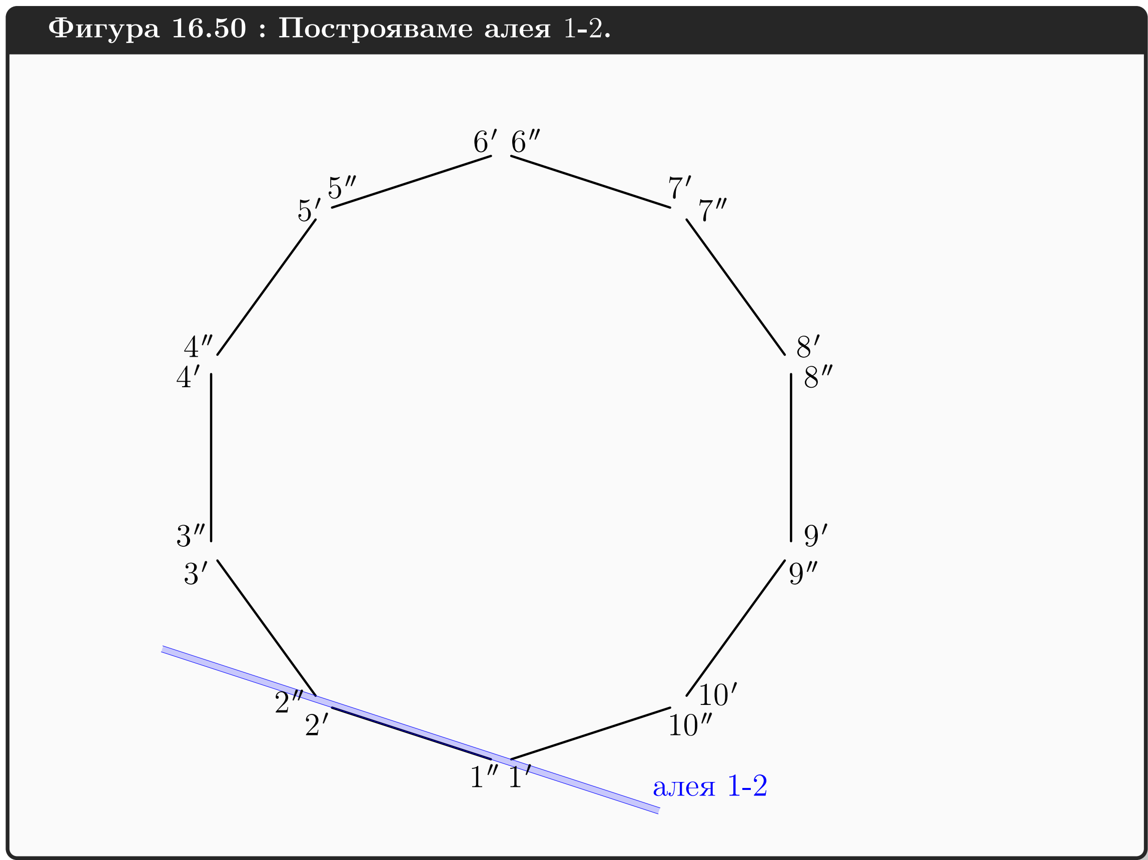
Фигура 16.49 : Изрязваме ϵ около върховете и получаваме отвори.



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.

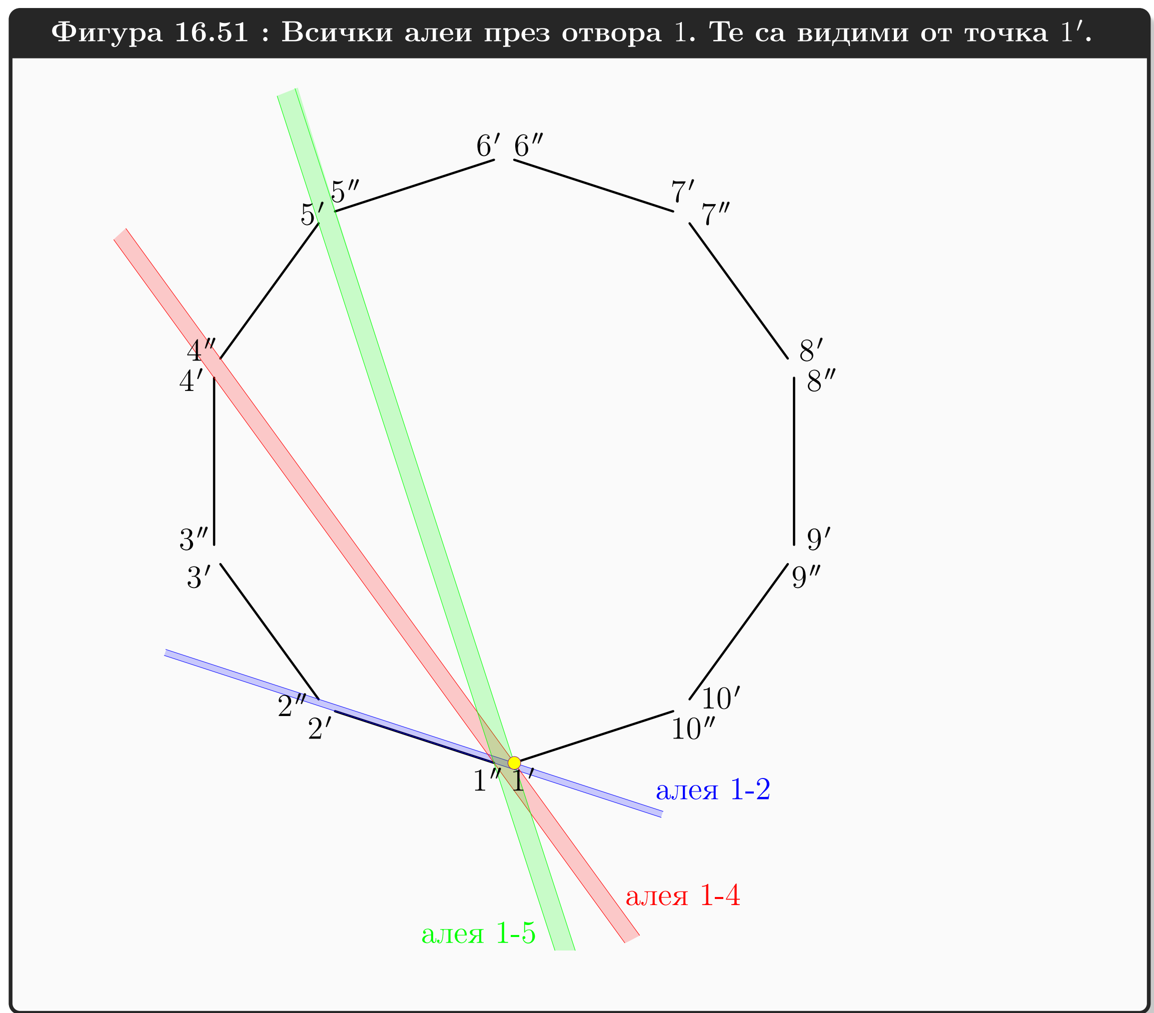
Фигура 16.50 : Построяваме алея 1-2.



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. Забежете, 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 $\mathtt{i}'$, за $1 \leqslant \mathtt{i} \leqslant 10$, като точките, от които се вижда всичко.

Фигура 16.51 : Всички алеи през отвора 1. Те са видими от точка 1′.

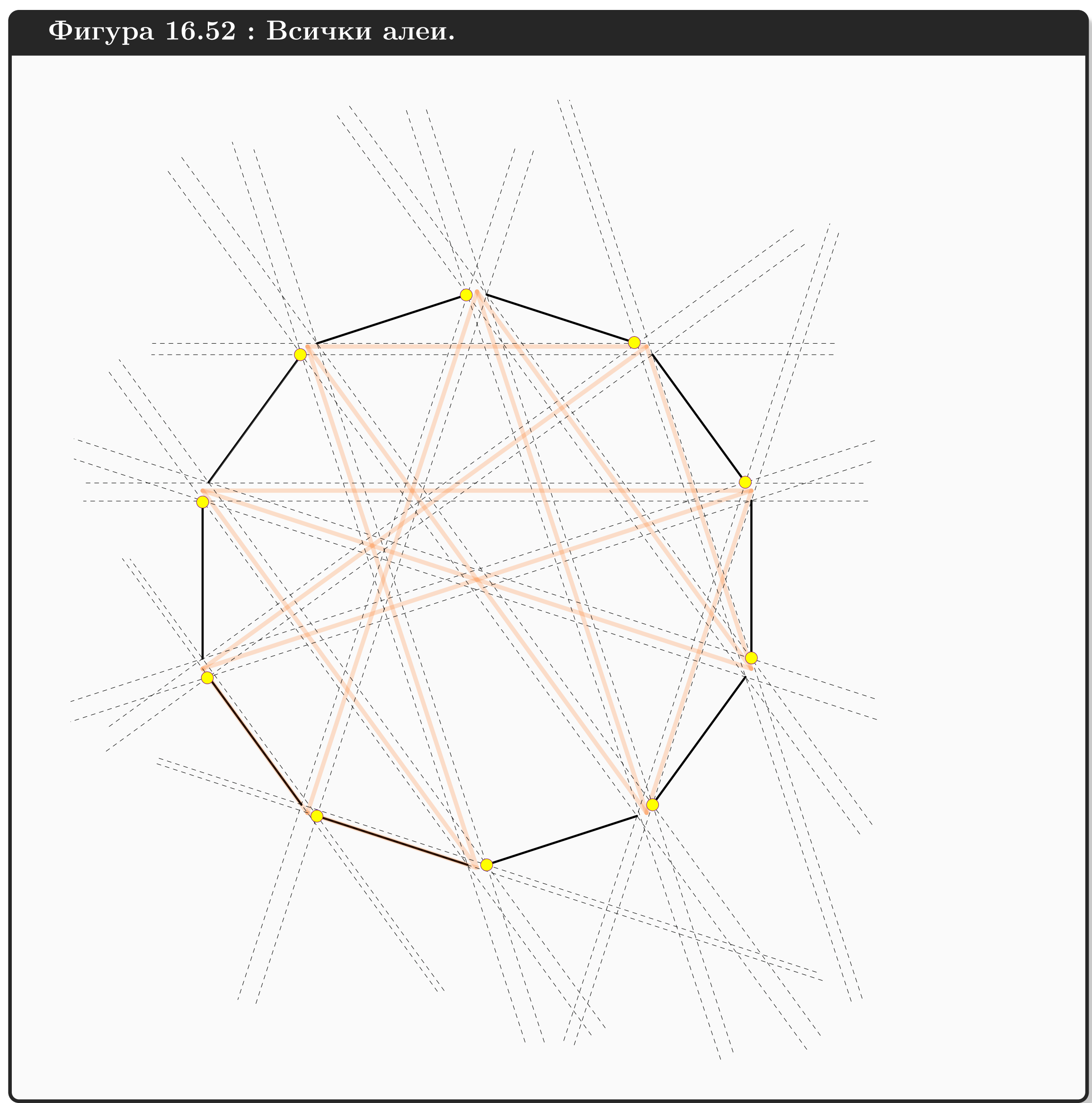


Конструираме по една такава алея и за останалите ребра на графа. Тъй като графът няма изолирани върхове, през всеки отвор минава поне една алея. За нашият пример, получаваме това, което е показано на Фигура 16.52. Страните на петнадесетте алеи са очертани с пунктирани линии.

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 $\mathtt{i}'$, за $1 \leqslant \mathtt{i} \leqslant 10$. 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.

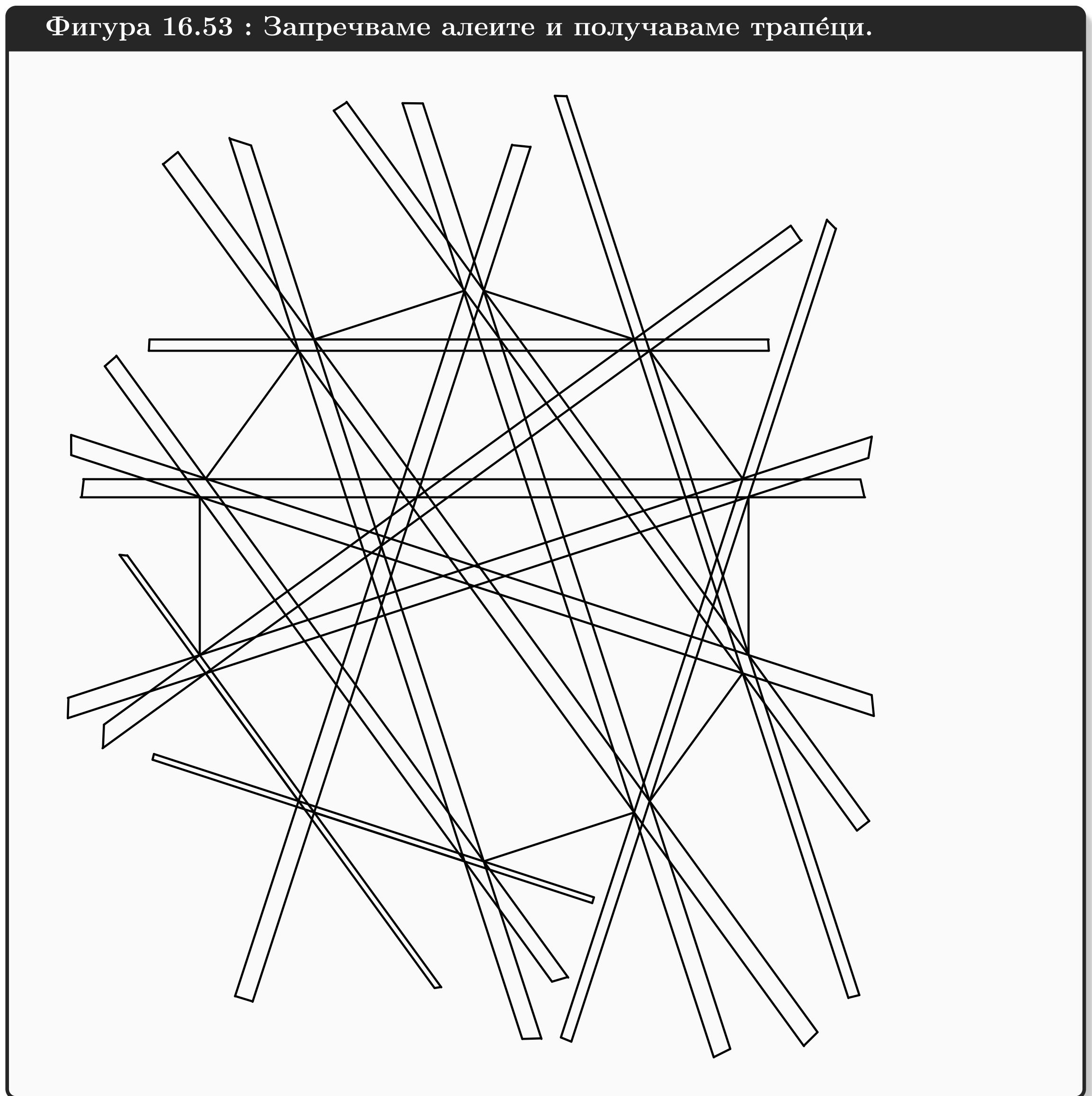


Този обект не е многоъгълник, понеже алеите са безкрайни. За да го превърнем в многоъгълник, във всяка алея слагаме две отсечки напреки и вземаме само частта от алеята между тях. Тази част е трапец, чиито бедра са въпросните две отсечки, а основите са подмножества на правите на алеята. За всяка алея, 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.

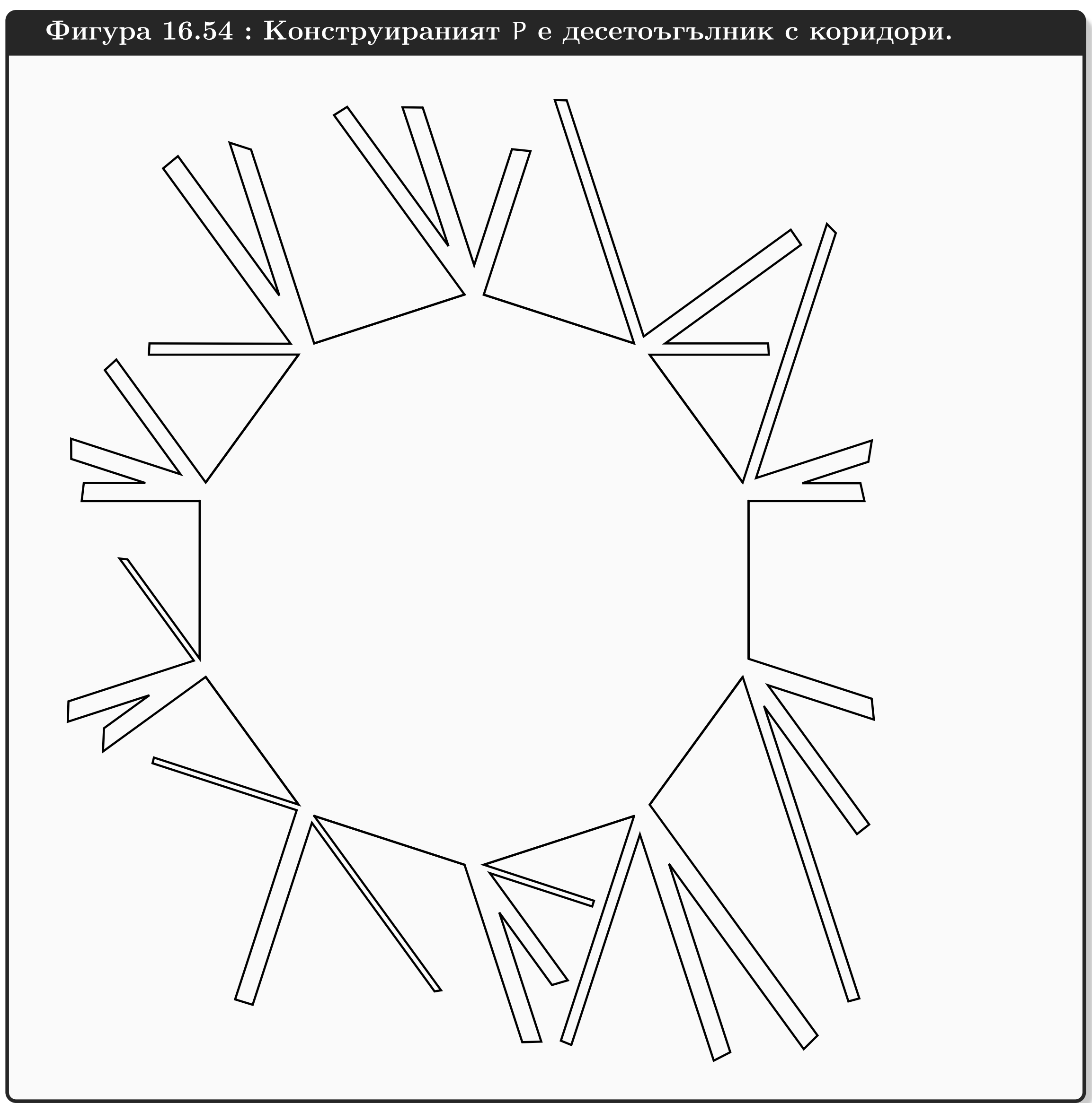
Тази част от конструкцията е илюстрирана на Фигура 16.53.

Фигура 16.53 : Запречваме алеите и получаваме трапéци.



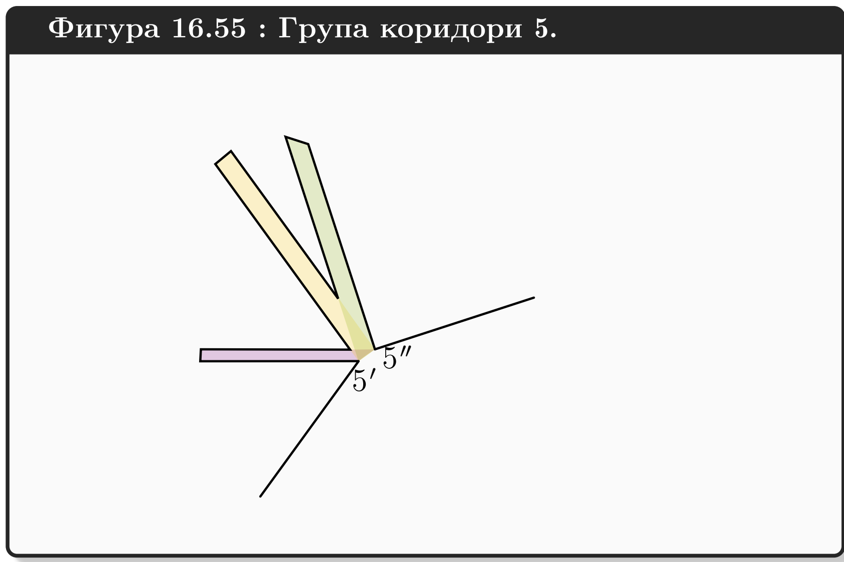
И накрая, вземаме обединението на десетоъгълника (от времето преди да махаме части от него) и трапéците и получаваме обекта, показан на Фигура 16.54. Той се състои от десетоъгълник с *коридори*, като коридорите са задънени, с единствен вход-изход към десетоъгълника, и са частите от трапéците, "стърчащи" 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.



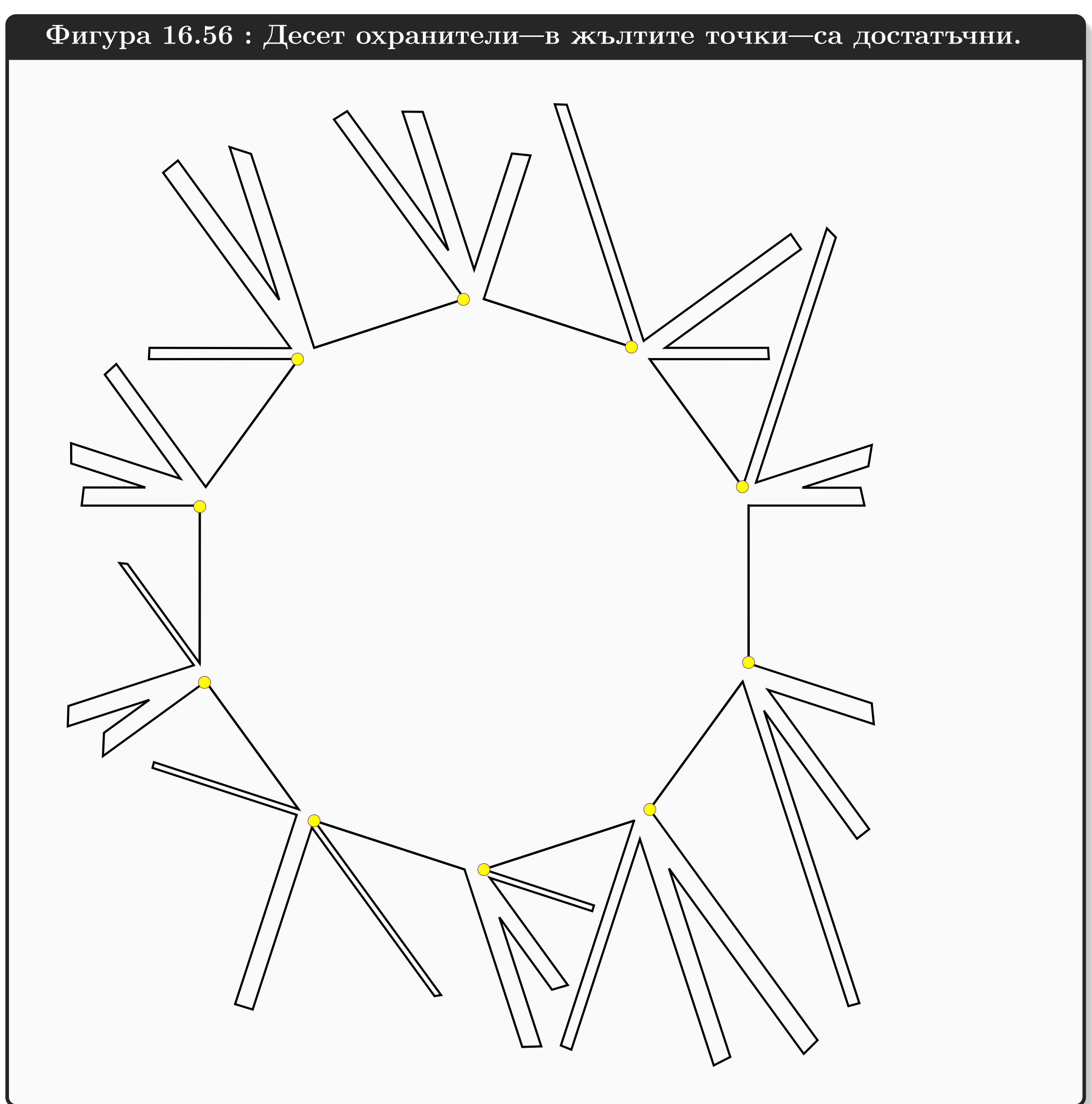
Нека дефинираме за $1 \leqslant j \leqslant 10$, *група коридори* j като обединението от коридорите, "излизащи" от отвор j. В нашия пример всяка група коридори има точно три коридора, но това се дължи на факта, че графът на Petersen е 3-регулярен. Фигура 16.55 показва група коридори 5, като трите нейни коридори са в различни цветове.

Фигура 16.55 : Група коридори 5.



Ако сложим по един охранител във всяка точка $\mathtt{i}'$, за $1 \leqslant \mathtt{i} \leqslant 10$, както е показано на Фигура 16.56 с десетте жълти точки, ще осигурим охрана на целия многоъгълник, защото точка $\mathtt{i}'$ вижда всяка точка на група коридори $\mathtt{i}$, а освен това вижда и десетоъгълника.

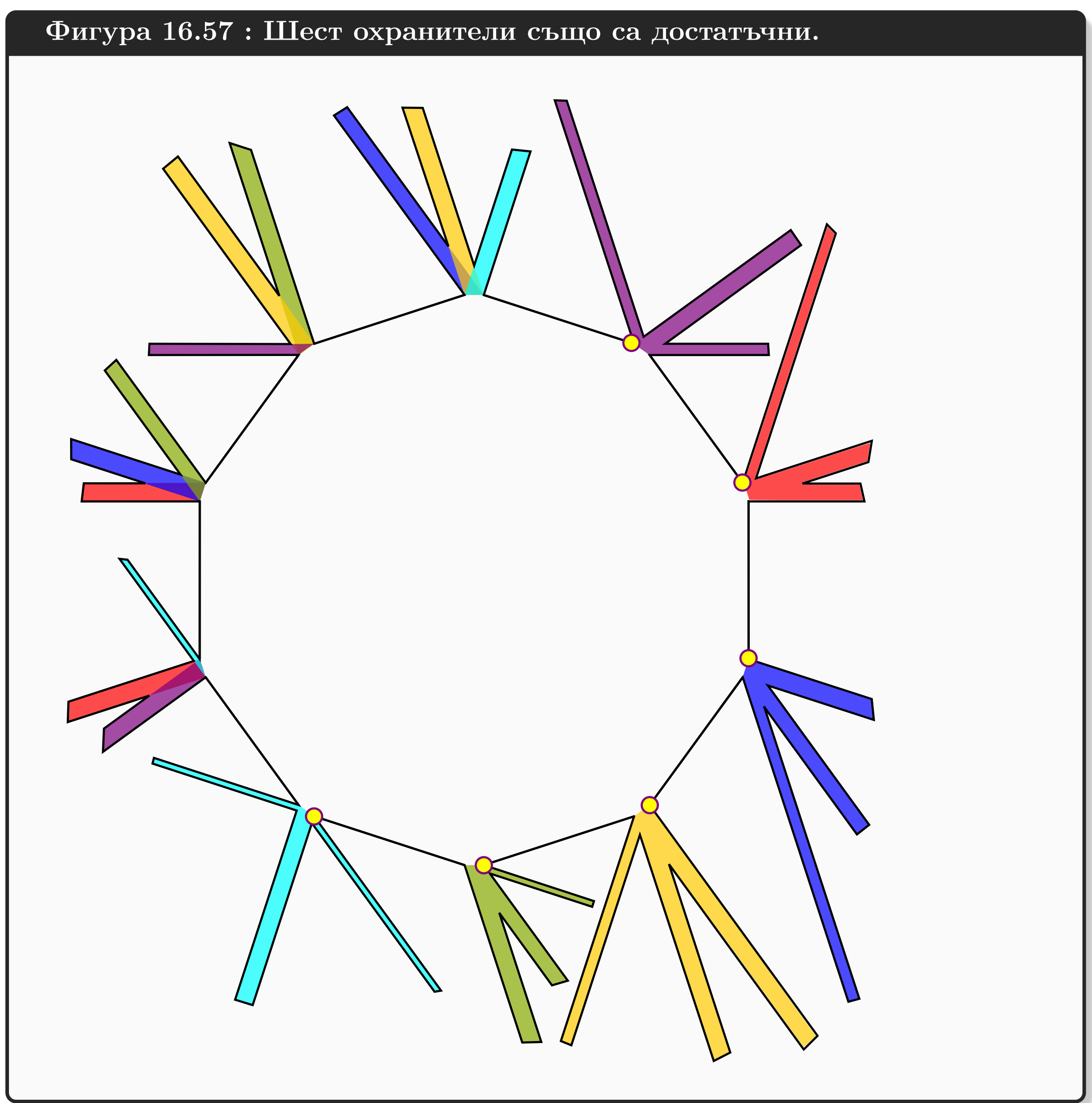
Фигура 16.56 : Десет охранители—в жълтите точки—са достатъчни.



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 $I \subseteq \{1', \ldots, 10'\}$, 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 ключовово 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 89), 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 разместванния 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 $\leqslant k$.

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 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 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 с размер, не по-голям от полиномиален.

  По-горе, а именно на стр. 901, бе споменато, че описанието на редукцията от Vertex Cover в [108, 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 [108, стр. 247–249]. За целите на тези лекционни записки, изложената конструкция е напълно удовлетворителна, предавайки акуратно основната идея. □

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. 439. 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. 37. 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. 39, както и на грешки във формулировките на Лема 8 и Лема 11. Благодарности на **Костадин Гаров** за откритата грешка в доказателството на Лема 27. Благодарности на **Ясен Алексиев** за откритата грешка в доказателството на Теорема 42. 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 61). Благодарности на **Иван-Асен Чакъров** за откритите грешки в рекапитулацията на графите. Благодарности на **Ива Петрова Караджова** за откритите грешки във формулировката на Наблюдение 43. 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 20. Благодарности на **Николай Пашов** за откритата и коригирана грешка в заключението на доказателството на Теорема 22. Благодарности на **Бойко Красимиров Борисов** за открит неверен резултат с невалидно “доказателство” в Допълнение 19. Благодарности на **Христо Трендафилов** за откритата груба грешка във формулировката и доказателството на Лема 33. Благодарности на **Цветан Алексиев** за откритата грешка в в кода на Partition от Подсекция 6.2.2. Благодаря на **Ивайло Панайтов** за откритата и коригирана грешка в записа на Лема 29. 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 и Фигура 5.3. 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 Захариева** 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 Карагьозов** за откритата и коригирана грешка в индексирането на ред 7 в Alg Matrix-Chain Multiplication. Благодарности на **Тодор Дуков** за откритата неточност в дефиницията на нормална форма на Чомски на стр. 525. 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 20.

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. 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. 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. 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.

[8] Giorgio Ausiello и др. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. 1st. Berlin, Heidelberg: Springer-Verlag, 1999. ISBN: 3540654313.

[9] 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.

[10] 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://dmtcs.episciences.org/3512.

[11] Richard Bellman. *Eye of the Hurricane: An Autobiography*. Series in modern applied mathematics. World Scientific, 1984. ISBN: 9789971966010.

[12] 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.

[13] 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.

[14] Jon Bentley. "Programming Pearls: Little Languages". В: *Commun. ACM* 29.8 (авг. 1986), с. 711—721. ISSN: 0001-0782. DOI: 10.1145/6424.315691. URL: http://doi.acm.org/10.1145/6424.315691.

[15] 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.

[16] 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. URL: https://doi.org/10.1145/800113.803652.

[17] 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.

[18] 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=X32GlVfqXjsC.

[19] Garrett Birkhoff. *Lattice Theory*. 3rd. American Mathematical Society, 1967.

[20] 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.

[21] 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. URL: https://doi.org/10.1145/321386.321395.

[22] 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.

[23] 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.

[24] 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.

[25] A. Bretto. *Hypergraph Theory: An Introduction*. Mathematical Engineering. Springer International Publishing, 2013. ISBN: 9783319000800. URL: https://books.google.bg/books?id=lb5DAAAAQBAJ.

[26] Neville Campbell. Непубликувана, достъпна онлайн на https://nevillecampbell.com/Recurrences.pdf.

[27] 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.

[28] 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.

[29] Louis Comtet. *Analyse Combinatoire Tome II*. Presses Universitaires de France, 1970. ISBN: 978-2130309819.

[30] 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.

[31] Thomas H. Cormen и др. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 9780262033848.

[32] 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.

[33] Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. 1st. USA: A. K. Peters, Ltd., 2011. ISBN: 1466505192.

[34] Reinhard Diestel. *Graph Theory*. Fourth. Т. 173. Graduate Texts in Mathematics. Springer, 2010. ISBN: 9783642142789 3642142788 9783642142796 3642142796.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Saber N. Elaydi. *An Introduction to Difference Equations*. 3rd. Springer-Verlag New York, 2005. ISBN: 978-0-387-23059-7. DOI: 10.1007/0-387-27602-5.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] J.B. Fraleigh, R.A. Beauregard и V.J. Katz. *Linear Algebra*. 3rd. Addison-Wesley, 1995. ISBN: 0-201-52675-1. URL: https://books.google.bg/books?id=ViqBvgAACAAJ.

[49] 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.

[50] 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/pdfft?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.

[51] 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.

[52] 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.

[53] Kurt Godel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Превод B. Meltzer; има свободно достъпно копие на превода на https://homepages.uc.edu/~martinj/History_of_Logic/Godel/Godel%20%E2%80%93%20On%20Formally%20Undecidable%20Propositions%20of%20Principia%20Mathematica%201931.pdf. Dover Publications, 1992. ISBN: 0486669807.

[54] 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.

[55] 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.

[56] Ronald L. Graham, Donald E. Knuth и Oren Patashnik. *Concrete Mathematics*. second. Addison-Wesley, 1994. ISBN: 0-201-55802-5.

[57] B. Guenin, J. Könemann и L. Tunçel. *A Gentle Introduction to Optimization*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781107282094.

[58] 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=Ofw5w1yuD8kC.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] C. A. R. Hoare. "Algorithm 63: Partition". В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: 10.1145/366622.366642. URL: http://doi.acm.org/10.1145/366622.366642.

[67] C. A. R. Hoare. "Algorithm 64: Quicksort". В: *Commun. ACM* 4.7 (юли 1961), с. 321—. ISSN: 0001-0782. DOI: 10.1145/366622.366644. URL: http://doi.acm.org/10.1145/366622.366644.

[68] 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.

[69] L. Hogben. *Handbook of Linear Algebra, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2013. ISBN: 9781466507289.

[70] John E. Hopcroft и Richard M. Karp. "An n5/2 Algorithm for Maximum Matchings in Bipartite Graphs". В: *SIAM J. Comput.* 2 (1973). Достъпна свободно онлайн на http://cse.unl.edu/~choueiry/Documents/MaxMatching-HopdcroftKarp.pdf, с. 225—231.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Ravi Jain, David Molnar и Zulfikar Ramzan. "Towards a Model of Energy Complexity for Algorithms". В: *IEEE Wireless Communications & Networking Conference* (2005).

[76] 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.

[77] 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. URL: http://www.sciencedirect.com/science/article/pii/0196677487900435.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. 1st. Addison-Wesley Professional, 2011. ISBN: 0201038048.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] J C Lagarias. "The 3x+1 problem: An annotated bibliography (1963–1999)". Достъпна онлайн на https://arxiv.org/abs/math/0608208. 2012.

[93] J C Lagarias. "The 3x+1 Problem: An Annotated Bibliography, II (2000-2009)". Достъпна онлайн на https://arxiv.org/abs/math/0309224. 2011.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] Zohar Manna. *Mathematical Theory of Computation*. Dover Publications, Incorporated, 2003.

[103] Silvano Martello и Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0471924202.

[104] 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.

[105] 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.

[106] 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.

[107] 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=IVZBHAAACAAJ.

[108] Bernard M. Moret. *The Theory of Computation*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201258285.

[109] 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.

[110] 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.

[111] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. USA: Oxford University Press, Inc., 1987. ISBN: 0195039653.

[112] 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.

[113] Xavier Ouvrard. *Hypergraphs: an introduction and review*. Свободно достъпна на https://arxiv.org/pdf/2002.05014. 2020.

[114] Christos H. Papadimitriou. *Computational Complexity*. Chichester, UK: John Wiley и Sons Ltd., с. 260—265. ISBN: 0-470-86412-5.

[115] Christos H. Papadimitriou и Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. USA: Prentice-Hall, Inc., 1982. ISBN: 0131524623.

[116] 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.

[117] 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.

[118] M. Pirlot и P. Vincke. *Semiorders: Properties, Representations, Applications*. Theory and Decision Library B. Springer Netherlands, 1997. ISBN: 9780792346173. URL: https://books.google.com/books?id=rXroQjbQHn0C.

[119] 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.

[120] 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.

[121] Romeo Rizzi. "A Short Proof of KöNig'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.

[122] 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.

[123] 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.

[124] Frank Ruskey. *Combinatorial Generation*. Достъпна онлайн на http://page.math.tu-berlin.de/~felsner/SemWS17-18/Ruskey-Comb-Gen.pdf. 2003.

[125] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201895390.

[126] 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.

[127] 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.

[128] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[129] 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.

[130] 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.

[131] Robert Sedgewick и Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011, с. I—XII, 1—955. ISBN: 978-0-321-57351-3.

[132] Michael Sipser. *Introduction to the theory of computation: second edition*. 2-е изд. Boston: PWS Pub., 2006. ISBN: 978-0534950972.

[133] Steven S. Skiena. *The Algorithm Design Manual*. 2nd. Springer Publishing Company, Incorporated, 2008. ISBN: 9781848000698.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] 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.

[142] Y. Wang. *The Goldbach Conjecture*. Series in pure mathematics. World Scientific, 2002. ISBN: 9789812776600. URL: https://books.google.com/books?id=VAY9nTreXkcC.

[143] 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.

[144] Lutz M. Wegner. *Sorting – The Turku Lectures*. Достъпна онлайн на http://tucs.fi/publications/attachment.php?fname=bWegner_LutzMx14a.full.pdf. 2014.

[145] 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.

[146] J. W. J. Williams. "Algorithm 232: Heapsort". В: *Communications of the ACM* 7.6 (1964), с. 347—348.

[147] 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.

[148] 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.