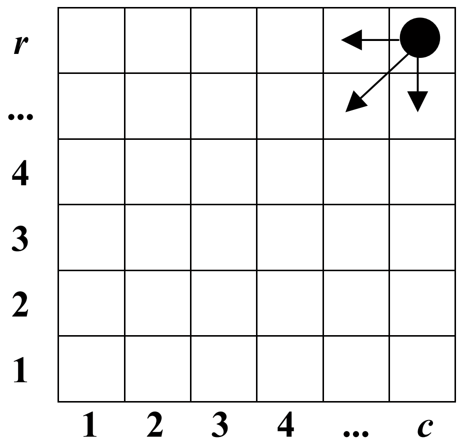
ЗАД. 1. — ДИНАМИЧНО ПРОГРАМИРАНЕ



Зад. 1. Правоъгълно поле се състои от клетки, подредени в r реда и c стълба, номерирани отляво надясно с числата от 1 до c и отдолу нагоре с числата от 1 до r. На това поле играят двама души. Отначало в горния десен ъгъл $(r ; c)$ се намира пул. Играчите се редуват да местят пула с една клетка наляво по хоризонтал, надолу по вертикал или наляво и надолу по диагонал. Печели играчът, който успее да закара пула в долния ляв ъгъл $(1 ; 1)$. Търси се булева функция $f : \mathbb{N} \times \mathbb{N} \to \{\text{false}, \text{true}\}$, за която е в сила следното: $f(r ; c) = \text{true} \iff$ първият играч има печеливша стратегия.

а) Съставете рекурентна зависимост за функцията f **(4 точки)**

и я опишете като алгоритъм от вида `bool f(int r, int c)`,

съставен по схемата динамично програмиране. **(4 точки)**

б) Оценете времевата сложност на алгоритъма от подточка “а” **(2 точки)**

и го изпълнете при $r = c = 6$, попълвайки подходяща таблица. **(2 точки)**

в) От таблицата изведете явна формула за функцията $f(r ; c)$ **(4 точки)**

и я опишете във вид на алгоритъм `bool f(int r, int c)`. **(2 точки)**

г) Оценете времевата сложност на алгоритъма от подточка “в” **(1 точка)**

и го сравнете по бързодействие с алгоритъма от подточка “а”. **(1 точка)**

Решение:

а) Търсената рекурентна зависимост е следната:

$$f(r ; c) = \overline{f(r-1 ; c) \wedge f(r ; c-1) \wedge f(r-1 ; c-1)},$$

където в конюнкцията под линията на отрицанието участват само тези членове, чиито аргументи приемат допустими стойности. С други думи,
ако $r > 1$ и $c > 1$, то конюнкцията има три члена (така, както е описана);
ако $r = 1$ и $c > 1$, то остава само вторият член: $f(1 ; c) = \overline{f(1 ; c-1)}$;
ако $r > 1$ и $c = 1$, то остава само първият член: $f(r ; 1) = \overline{f(r-1 ; 1)}$;
ако $r = 1$ и $c = 1$, то не остават членове, тоест $f(1 ; 1) = \text{false}$, защото празната конюнкция има стойност истина, а нейното отрицание е неистинно.

Доказателство: Играчът, започващ от дадено поле $(r;c)$, има печеливша стратегия тогава и само тогава, когато поне един от ходовете, с които разполага, води до ситуация, губеща за неговия противник. Иначе казано, полето $(r;c)$ е благоприятно за играча, започващ от него, само когато поне едно от полетата $(r-1;c)$, $(r;c-1)$ и $(r-1;c-1)$ е губещо, т.е. когато не всичките три полета са печеливши.

Чрез динамично програмиране от формулата се получава алгоритъм:

```
bool f(int r, int c)
dyn[1...r , 1...c]: array of bool
dyn[1,1] ← false
for r̃ ← 2 to r
  dyn[r̃,1] ← not dyn[r̃-1,1]
for c̃ ← 2 to c
  dyn[1,c̃] ← not dyn[1,c̃-1]
for r̃ ← 2 to r
  for c̃ ← 2 to c
    dyn[r̃,c̃] ← not (
      dyn[r̃-1,c̃] and dyn[r̃,c̃-1] and dyn[r̃-1,c̃-1]
    )
return dyn[r,c]
```

б) Всяка отделна клетка от таблицата `dyn` се пресмята за константно време, затова времевата сложност е равна по порядък на броя на клетките, тоест $\Theta(r.c)$. При $r = c = 6$ таблицата `dyn` изглежда, както е показано вдясно.

6	T	T	T	T	T	T
5	F	T	F	T	F	T
4	T	T	T	T	T	T
3	F	T	F	T	F	T
2	T	T	T	T	T	T
1	F	T	F	T	F	T
	1	**2**	**3**	**4**	**5**	**6**

в) От таблицата е ясно, че $f(r;c) = \text{false}$ само когато r и c са нечетни; в останалите случаи $f(r;c) = \text{true}$. Това се доказва по индукция.

```
bool f(int r, int c)
return (r mod 2 = 0) or (c mod 2 = 0)
```

г) Времевата сложност на последния алгоритъм е константа $\Theta(1)$, тоест той е по-бърз от предишния алгоритъм.

ЗАД. 2. — ГРАФИ И SUBSETSUM

Разглеждаме следната алгоритмична задача за разпознаване:

Дадени са естествено число K и двуделен граф $G(V_1 \cup V_2\ ,\ E)$; върховете от двата дяла V_1 и V_2 са дадени поотделно (в два отделни масива). Търси се възможно ли е да бъдат изтрити няколко върха от V_1 (заедно с ребрата си) така, че да останат точно K ребра.

Намерете класа на времева сложност на задачата. С други думи, докажете, че тя е NP-пълна, или предложете полиномиален алгоритъм.

Решение: Тази алгоритмична задача принадлежи на класа **P**. Тя се решава чрез SubsetSum: числовият масив се състои от степените на върховете от V_1, а търсеният сбор е K. Както знаем, времевата сложност на SubsetSum е $\Theta(nK)$, където n е броят на върховете от първия дял. Това е псевдополиномиална сложност, защото K е стойност, а не дължина на входа. Входът се състои от две части: масива от степените на върховете от първия дял (с дължина $\Theta(n)$) и числото K (с дължина $\Theta(\log K)$). Дължината на целия вход е $\Theta(n + \log K)$.

В общия случай (за SubsetSum) числото K може да бъде толкова голямо, че $\log K \succ n$. Тогава дължината на входа е $\Theta(\log K)$, а времето $\Theta(nK)$ е експоненциална функция на $\log K$.

Трябва да съобразим обаче, че в разглеждания частен случай числото K не може да бъде твърде голямо. По-точно, ако $K > m$ (броя на ребрата на G), то очевидно няма как да получим граф с K ребра (при изтриване на ребра броят им може само да намалява). Затова в началото на алгоритъма правим проверка за стойността на K:

```
Alg(G(V₁ ∪ V₂, E), K)
if K > m
   return false
D[1...n] ← Degrees(V₁)
return SubsetSum(D,K)
```

Изчисляването на степените на върховете изисква едно обхождане на графа, следователно изразходва време $\Theta(n+m)$. Алгоритъмът за задачата SubsetSum се вика само при $K \leq m$. Но за всеки граф $m \leq n(n-1)/2$. Тогава $K \preceq n^2$, $\log K \preceq 2\log n \prec n$, дължината на входа е $\Theta(n + \log K) = \Theta(n)$, а времето на SubsetSum е $\Theta(nK) = \mathrm{O}\left(n^3\right)$. Окончателно, времето на целия алгоритъм е полиномиално: $\mathrm{O}\left(n^3\right)$, следователно задачата принадлежи на класа **P**.

ЗАД. 3. — АНАЛИЗ НА АЛГОРИТЪМ

Даден е следният алгоритъм.

```
f(a,b: non-negative integers)
s ← 0
p ← a
r ← b
while r > 0
   if r is odd
      r ← r – 1
      s ← s + p
   r ← r / 2
   p ← 2 × p
return s
```

а) Какво връща алгоритъмът?
Дайте строго доказателство (например чрез инварианта на цикъла).

б) Намерете времевата сложност на алгоритъма.

Решение:

а) Инварианта на цикъла: Всеки път, когато се изпълнява проверката $r \overset{?}{>} 0$, е в сила равенството $\boxed{s + pr = ab.}$

Доказателство: с индукция.

База: При първата проверка (т.е. при влизането в цикъла) са в сила равенствата $s = 0$, $p = a$, $r = b$, откъдето $s + pr = 0 + ab = ab$.

Индуктивна стъпка: Нека $s + pr = ab$ при някоя проверка и $r > 0$. Тогава алгоритъмът влиза в тялото на цикъла. Ако r е нечетно, то след изпълнение на тялото на оператора `if` е в сила равенството

новото $s + pr$ = старото $(s + p) + p(r-1) = s + \cancel{p} + pr - \cancel{p} = ab$,

където последното равенство следва от индуктивното предположение.

След това r е четно и се дели на 2, а пък p се умножава по 2. Затова pr запазва предишната си стойност. Тогава и целият израз $s + pr$ запазва стойността си ab, което доказва индуктивното заключение: при следващата проверка на условието за край на цикъла пак важи равенството $s + pr = ab$.

Завършек на цикъла: На всяка итерация числото r намалява (строго) и остава цяло (понеже се дели на 2 само ако е четно). Тъй като не съществува безкрайна строго намаляваща редица от цели положителни числа, то броят на итерациите е краен, т.е. алгоритъмът все някога завършва.

Последната проверка на условието за край на цикъла е била неуспешна, т.е. последното $r \le 0$. Но при предишната проверка r е било положително. Делението на 2 запазва знака, а изваждането на 1 (ако r е нечетно, т.е. поне 1) също не може да направи r отрицателно. Значи случаят $r < 0$ е невъзможен. Остава една възможност: $r = 0$. Следователно $s + pr = s + p \times 0 = s$. От това равенство и от доказаната инварианта $s + pr = ab$ следва, че при завършване на цикъла $s = ab$. Алгоритъмът връща s, тоест ab.

Отговор: Алгоритъмът връща произведението на входните параметри.

б) При всяка итерация стойността на r намалява два пъти (закръглено надолу заради изваждането на единицата); след k итерации стойността е $r = \left\lfloor \frac{b}{2^k} \right\rfloor$.
Алгоритъмът завършва, щом r стане 0. След предпоследната итерация $r = 1$. Тоест алгоритъмът завършва след $k + 1$ итерации, където $\left\lfloor \frac{b}{2^k} \right\rfloor = 1$. Оттук $T(a,b) = \Theta(k+1) = \Theta(k) = \Theta\left(\log_2 b\right) = \Theta(\log b)$. Следва, че времето за изпълнение на алгоритъма зависи само от параметъра b.

Отговор: $T(a,b) = \Theta(\log b)$.

Забележка към подусловие "а": Върнатата стойност може да бъде намерена и по друг начин (без инварианта на цикъл). Заменяме итеративния алгоритъм с рекурсивен, който връща същата стойност:

```
f(a,b: non-negative integers)
if b = 0
   return 0 // дъно на рекурсията
else if b is odd
   return f(a, b - 1) + a
else
   return f(2 × a, b / 2)
```

За функцията $f(a,b)$ получаваме рекурентната формула:

$$f(a,b) = \begin{cases} 0, \text{ ако } b = 0; \\ f(a,b-1)+a, \text{ ако } b \text{ е нечетно и } b > 0; \\ f(2a,b/2), \quad \text{ ако } b \text{ е четно и } b > 0. \end{cases}$$

Оттук чрез силна математическа индукция по b се доказва, че $f(a,b) = ab$.

ЗАД. 4. — ГРАФИ И СОРТИРОВКИ

Алгоритъмът на Дейкстра е бил пуснат върху някакъв неориентиран тегловен граф $G\,(V, E)$ от върха A и е върнал дървото T на най-късите пътища от A до всички други върхове на графа. След това графът G е бил унищожен. Останало е само дървото T с посочен корен A и теглата на ребрата на T (неотрицателни цели числа). Търси се редът, в който ребрата на T са били присъединени към дървото. Алгоритъмът на Дейкстра не може да бъде пуснат върху G повторно, тъй като графът G вече е унищожен.

Докажете, че тази алгоритмична задача има времева сложност $\Theta(n \log n)$, където n е броят на върховете на дървото T. За целта:

а) Съставете алгоритъм, решаващ задачата за време $\mathrm{O}(n \log n)$.

Демонстрирайте работата на алгоритъма с пример.

б) Докажете, че всеки алгоритъм, решаващ задачата, изисква време $\Omega(n \log n)$.

Решение:

а) *Първи начин:* Въпреки че не можем да пуснем алгоритъма на Дейкстра върху оригиналния граф G (тъй като той е унищожен), можем да го пуснем върху дървото T (все едно че G съвпада с T). Ще се получи дърво T_2, което ще съвпадне с T, т.е. ще получим дървото T за втори път, но сега можем да проследим реда на присъединяване на ребрата.

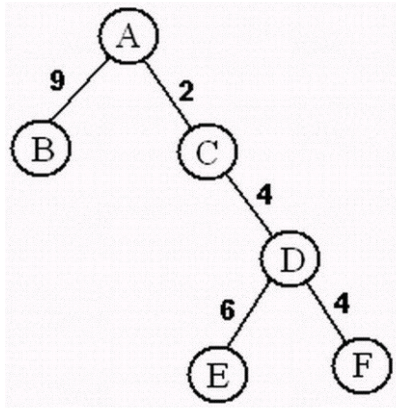
Алгоритъмът на Дейкстра (с приоритетна опашка, реализирана например чрез пирамида на Фибоначи) изисква време $\Theta(m + n \log n)$, където n е броят на върховете, m е броят на ребрата на графа, върху който пускаме алгоритъма. В случая алгоритъмът се пуска върху дървото T, така че имаме $m = n - 1$. Затова $\Theta(m + n \log n) = \Theta(n - 1 + n \log n) = \Theta(n \log n)$, тоест този алгоритъм е достатъчно бърз. Да отбележим, че използването на двоична пирамида води до същата сложност по време.

Втори начин: чрез сортиране. Обхождаме дървото (няма значение как — в ширина или в дълбочина), като започваме от корена A. На всеки връх пишем по едно цяло число dist — разстоянието от корена до този връх:
$\mathrm{dist}\,[A] = 0$, $\mathrm{dist}\,[Y] = \mathrm{dist}\,[X] + \mathrm{w}\,(X, Y)$, където X е родителят на Y в дървото T, тоест X е върхът, от който сме стигнали до Y. Тази стъпка изисква време $\Theta(m + n)$, където m е броят на ребрата на T. От теорията на графите е известно, че $m = n - 1$. Следователно изразходваното дотук време е $\Theta(m + n) = \Theta(n - 1 + n) = \Theta(2n - 1) = \Theta(n)$.

Без ограничение, нека коренът A е връх № 1, т.е. $\mathrm{dist}[1] = \mathrm{dist}[A] = 0$. В такъв случай първият елемент на масива dist не ни интересува. Сортираме останалите върхове, т.е. масива $\mathrm{dist}[2...n]$, с някой от бързите алгоритми, например с пирамидалното сортиране, за време $\Theta(n \log n)$.

Едновременно с размените в масива $\mathrm{dist}[2...n]$ разместваме и ребрата на дървото: на всеки връх Y (без корена) съответства реброто към родителя X.

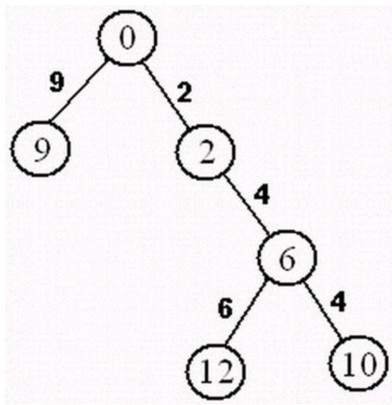
Получената подредба на ребрата е точно тази, която търсим. Това следва от факта, че на всяка стъпка алгоритъмът на Дейкстра избира онзи отворен връх, чието разстояние до корена е най-малко.



П р и м е р : Дадено е дърво с корен A.

индекси	1	2	3	4	5	6
върхове	A	B	C	D	E	F
ребра		AB	AC	CD	DE	DF
dist	0	9	2	6	12	10

Масивът dist е попълнен по следния начин:



$\mathrm{dist}[A] = 0$,
$\mathrm{dist}[B] = \mathrm{dist}[A] + \mathrm{w}(A, B) = 0 + 9 = 9$,
$\mathrm{dist}[C] = \mathrm{dist}[A] + \mathrm{w}(A, C) = 0 + 2 = 2$,
$\mathrm{dist}[D] = \mathrm{dist}[C] + \mathrm{w}(C, D) = 2 + 4 = 6$,
$\mathrm{dist}[E] = \mathrm{dist}[D] + \mathrm{w}(D, E) = 6 + 6 = 12$,
$\mathrm{dist}[F] = \mathrm{dist}[D] + \mathrm{w}(D, F) = 6 + 4 = 10$.

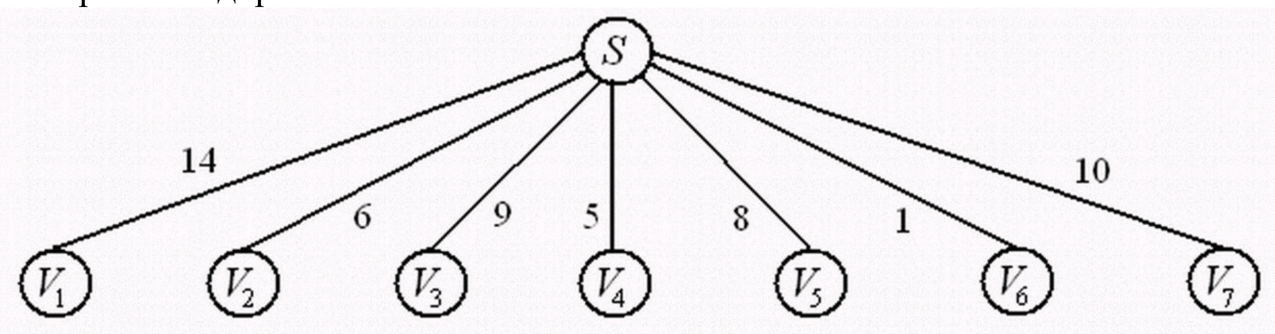
Сортираме масива $\mathrm{dist}[2...6]$ и едновременно с това разместваме ребрата:

ребра	AC	CD	AB	DF	DE
dist	2	6	9	10	12

Сега ребрата са подредени тъкмо в реда, в който са били присъединени към дървото от алгоритъма на Дейкстра: AC, CD, AB, DF, DE.

б) Че разглежданата алгоритмична задача изисква време $\Omega(n \log n)$, се доказва с помощта на следната редукция от задачата за сортиране чрез сравняване: по даден числов масив $A[1 \ldots n]$ строим дърво с върхове $S, V_1, V_2, \ldots, V_n$ и ребра $SV_1, SV_2, \ldots, SV_n$ с тегла съответно $A[1], A[2], \ldots, A[n]$. Върхът S е коренът на дървото. Редът на присъединяване на ребрата е точно редът на нарастване на техните тегла (тъй като излизат от един връх — S).

П р и м е р **:** Нека трябва да сортираме масива $A = (14, 6, 9, 5, 8, 1, 10)$. Построяваме дърво T.



Извикваме произволен алгоритъм, който да намери реда на присъединяване на ребрата на T (ако го разглеждаме като получено в резултат на работата на алгоритъма на Дейкстра върху някакъв граф G, например $G = T$). Редът на присъединяване на ребрата е следният: SV_6, SV_4, SV_2, SV_5, SV_3, SV_7, SV_1. Това е точно сортираният масив A: 1, 5, 6, 8, 9, 10, 14.

Коректността на редукцията следва от това, че щом всички ребра излизат от корена S на дървото, то алгоритъмът на Дейкстра ще ги присъедини по реда на нарастване на теглата им. Обаче алгоритъмът на Дейкстра работи правилно само върху графи с неотрицателни тегла на ребрата. Затова описаната редукция е коректна само за числови масиви с неотрицателни елементи.

Тъй като масивът A може да съдържа и отрицателни числа, налага се да поправим редукцията. Допълваме я със следните стъпки:

Първо намираме x = най-малкия елемент на масива A.

След това изваждаме $x - 1$ от всички елементи на A. Сега най-малкият елемент на A е числото 1, следователно всички числа от A са положителни.

За получения масив от положителни числа прилагаме редукцията, описана по-горе (построяването на дърво).

След като новият масив A бъде сортиран, обхождаме го и прибавяме $x - 1$ към всичките му елементи.

Поправената редукция е коректна, защото прибавянето (и изваждането) на едно и също число от всички елементи на масив не променя тяхната подредба.

П р и м е р : Ако е даден масивът $A = (7, -1, 2, -2, 1, -6, 3)$, първо намираме $x = \min A = -6$, изваждаме $x - 1 = -7$, т.е. прибавяме 7 към всички елементи на масива и получаваме $A = (14, 6, 9, 5, 8, 1, 10)$; после сортираме новия масив чрез дърво: $A = (1, 5, 6, 8, 9, 10, 14)$; накрая прибавяме $x - 1 = -7$, т.е. изваждаме 7 от елементите на масива: $A = (-6, -2, -1, 1, 2, 3, 7)$; това са първоначалните числа в растящ ред.

Бързина на редукцията: Намирането на най-малкото число, изваждането и прибавянето на число към всички елементи на масива, както и построяването на дървото изискват линейно време $\Theta(n)$ – по едно обхождане на масива за всяка от тези четири операции. Тъй като $n \prec n \log n$, то редукцията е достатъчно бърза за целите на доказателството.

ЗАД. 5. — СОРТИРАНЕ В ГЕОМЕТРИЧНА ЗАДАЧА

Масивът $A[1\ldots n]$ съдържа положителни числа — дължините на n отсечки. Предложете алгоритъм с времева сложност $O(n \log n)$, намиращ три отсечки, които могат да бъдат страни на триъгълник (ако има такива).

Решение: Следният алгоритъм решава задачата.

```
FindTrigon(A[1...n])
Sort(A)
for j ← 2 to n-1
   if A[j-1] + A[j] > A[j+1]
      print A[j-1], A[j], A[j+1]
      return
print "Няма триъгълник."
```

Цикълът изисква линейно време $\Theta(n)$, а сортирането — време $\Theta(n \log n)$, ако използваме някоя от бързите сортировки (напр. пирамидалното сортиране). Така че целият алгоритъм има времева сложност $\Theta(n \log n)$, т.е. отговаря на изискванията от условието на задачата.

Коректността на алгоритъма следва от неравенството на триъгълника. Тъй като търсим три отсечки, най-дългата от които е по-малка от сбора на другите две, то все едно търсим (след сортирането на масива) три индекса i, j и k, такива, че $i < j < k$ и $A[i] + A[j] > A[k]$. При произволно, но фиксирано j трябва $A[i]$ да е максимално, а $A[k]$ — минимално, за да има възможност неравенството да бъде изпълнено. Понеже масивът е сортиран, достатъчно е да проверим само $i = j - 1$ и $k = j + 1$.