Д И Н А М И Ч Н О П Р О Г Р А М И Р А Н Е
КОНТРОЛНО № 6 ПО ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ — ЗА СТУДЕНТИТЕ ОТ СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 1. ПОТОК, СУ, ФМИ, 03. 06. 2021 ГОД.

Указания:
1) Названията на алгоритмите и на структурите от данни да бъдат на български език.
2) Да се анализира времевата сложност при най-лоши данни и да се обоснове коректността на всички предложени алгоритми.
3) Алгоритмите, изучени на лекции, могат да се използват наготово.

Задача 1. $\left\langle {n \atop m} \right\rangle$ е броят на пермутациите π без повторение на целите числа 1, 2, . . . , n, в които пермутации точно m елемента са по-големи от непосредствено предхождащите ги, тоест $\left| \left\{ k \in \{2\,;\,3\,;\ \ldots\,;\,n\} :\ \pi(k-1) < \pi(k) \right\} \right| = m$. Числата n и m са цели и $0 \le m < n$. Например пермутацията 5, 6, 3, 1, 4, 2 се състои от $n = 6$ числа и $m = 2$ от тях (6 и 4) са по-големи от своите непосредствени предшественици.

а) Опишете на псевдокод итеративен алгоритъм за пресмятането на функцията $\left\langle {n \atop m} \right\rangle$ със сложност по време и памет $O(n^2)$ при всякакви входни данни.

Упътване: Съставете рекурентно уравнение, като изтриете числото n от пермутацията и после го добавите на всички възможни места. Предложете подходящи гранични условия.

б) Попълнете таблица със стойностите на $\left\langle {n \atop m} \right\rangle$ за всички n и m, ненадхвърлящи 6.

в) Оптимизирайте алгоритъма по такъв начин, че сложността по памет да стане $O(n)$, а сложността по време да остане $O(n^2)$; двете сложности важат при всякакви входни данни. Опишете оптимизацията на псевдокод. (Ако това е направено в точка "а", не го повтаряйте.)

Задача 2. а) Съставете алгоритъм с времева сложност $O(n^2)$ при най-лоши данни, който по дадена редица $A[1...n]$ от цели числа търси най-дълга строго растяща подредица, редуваща четни и нечетни числа. Първото число в подредицата може да е четно или нечетно.

Задачата може да се реши поне по два начина: чрез ориентиран ацикличен граф или направо (тоест без построяване на граф). Ако решавате задачата по първия начин, опишете алгоритъма словесно: как се построява графът, какво представляват върховете и ребрата му, как се определя посоката на всяко ребро, защо графът е ацикличен, до коя известна задача за динамично програмиране при ориентирани ациклични графи се свежда задача 2.

Ако решавате задачата без граф, опишете алгоритъма на псевдокод. В този случай е достатъчно да намерите само дължината на най-дълга подредица без самата подредица. Получавате допълнителни 4 точки, ако псевдокодът отпечатва и самата подредица.

Точките се удвояват за алгоритъм с времева сложност $O(n \log n)$ при най-лоши данни; алгоритъмът може да се опише словесно (без програмен код).

б) Демонстрирайте алгоритъма върху масива $A = (7;\ 5;\ 6;\ 30;\ 201;\ 10)$.

СХЕМА НА ТОЧКУВАНЕ

Цялото контролно носи най-много 20 точки — по 4 точки за всяко от петте подусловия. Още 4 т. носи алгоритъм на псевдокод (без графи), който в зад. 2 намира самата подредица. Точките по задача 2 се удвояват за алгоритъм с време $O(n \log n)$ при най-лоши входни данни.

РЕШЕНИЯ

Задача 1. Разглеждаме пермутациите без повторение на числата 1, 2, . . . , n. Пермутациите, в които точно m числа са по-големи от своите непосредствени предшественици, са два вида.

Първи вид: след премахване на числото n остава пермутация на 1, 2, . . . , $n-1$, в която точно m числа са по-големи от непосредствените си предшественици. Има $\left\langle {n-1 \atop m} \right\rangle$ такива пермутации и числото n може да бъде вмъкнато във всяка от тях на общо $m+1$ места — между два поредни елемента, вторият от които е по-голям от първия, както и в началото на редицата.

Втори вид: след премахване на числото n остава пермутация на 1, 2, . . . , $n-1$, в която $m-1$ числа са по-големи от непосредствените си предшественици. Има $\left\langle {n-1 \atop m-1} \right\rangle$ такива пермутации и числото n може да бъде вмъкнато във всяка от тях на общо $n-m$ места — между два поредни елемента, вторият от които е по-малък от първия, както и в края на редицата.

Ето защо функцията $\left\langle {n \atop m} \right\rangle$ удовлетворява рекурентното уравнение

$$\left\langle {n \atop m} \right\rangle = (m+1)\,.\left\langle {n-1 \atop m} \right\rangle + (n-m)\,.\left\langle {n-1 \atop m-1} \right\rangle \text{ при } n-1 > m > 0,$$

както и следните гранични условия за всяко цяло положително число n:

$\left\langle {n \atop 0} \right\rangle = 1$**:** има само една пермутация, в която числата са наредени в низходящ ред;

$\left\langle {n \atop n-1} \right\rangle = 1$**:** има само една пермутация, в която числата са наредени във възходящ ред.

Таблицата по-долу съдържа първите няколко стойности на функцията $\left\langle {n \atop m} \right\rangle$.

n \ m	0	1	2	3	4	5	6	7	8
1	1								
2	1	1							
3	1	4	1						
4	1	11	11	1					
5	1	26	66	26	1				
6	1	57	302	302	57	1			
7	1	120	1191	2416	1191	120	1		
8	1	247	4293	15619	15619	4293	247	1	
9	1	502	14608	88234	156190	88234	14608	502	1

В комбинаториката тези числа са известни като ***числа на Ойлер от първи род.***

Псевдокод на алгоритъма:

```
Число на Ойлер от първи род (n, m: цели числа) // 0 ≤ m < n
dyn[1...n][0...n-1]: масив от цели положителни числа
for ñ ← 1 to n do
   dyn[ñ][0] ← 1
   dyn[ñ][ñ-1] ← 1
for ñ ← 3 to n do
   for m̃ ← 1 to ñ-2 do
      kind1 ← (m̃+1) × dyn[ñ-1][m̃]
      kind2 ← (ñ-m̃) × dyn[ñ-1][m̃-1]
      dyn[ñ][m̃] ← kind1 + kind2
return dyn[n][m]
```

Сложността по време и по памет е $\Theta(n^2)$ — колкото е обемът на динамичната таблица.

Може да се постигне сложност по памет $\Theta(n)$ с помощта на следното наблюдение: числата във всеки ред на таблицата зависят само от числата в предишния ред. Ето защо е достатъчно да пазим само един ред от таблицата (и да го пресмятаме отдясно наляво).

Псевдокод на оптимизирания алгоритъм:

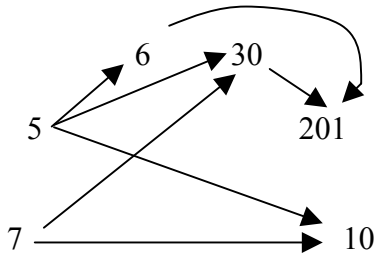
```
Число на Ойлер от първи род (n, m: цели числа) // 0 ≤ m < n
dyn[0...n-1]: масив от цели положителни числа
dyn[0] ← 1
for ñ ← 2 to n do
   dyn[ñ-1] ← 1
   for m̃ ← ñ-2 downto 1 do
      kind1 ← (m̃+1) × dyn[m̃]
      kind2 ← (ñ-m̃) × dyn[m̃-1]
      dyn[m̃] ← kind1 + kind2
return dyn[m]
```

Сложността по време на този алгоритъм е $\Theta(n^2)$ заради вложените цикли, а по памет е $\Theta(n)$ заради едномерната динамична таблица.

Нещо повече, сложността по време може да се свали до $\Theta(nm)$, а по памет — до $\Theta(m)$, като пазим динамична таблица `dyn[0...m]` и накараме брояча на вътрешния цикъл да започва от по-малкото от числата `m` и $\tilde{n}-2$. Добавяме проверка към първия ред от външния цикъл, която да гарантира, че въпросният ред ще се изпълнява само с допустими стойности на индекса. Тази оптимизация носи допълнителни 4 точки.

Задача 2. По дадения масив $A[1...n]$ построяваме ориентиран граф с върхове 1, 2, . . . , n (индексите на елементите на масива). От върха i към върха j има ребро тогава и само тогава, когато $i < j$, $A[i] < A[j]$ и от двете числа $A[i]$ и $A[j]$ едното е четно, а другото е нечетно. Този ориентиран граф е ацикличен, защото ребрата сочат само от връх с по-малък индекс към връх с по-голям индекс. Търсената най-дълга подредица съответства на най-дълъг път в построения ориентиран ацикличен граф; за тази задача разполагаме с готов алгоритъм, изучен на лекции — динамично програмиране върху ориентирани ациклични графи. Можем да си спестим топологичното сортиране на графа: върховете са сортирани поначало, защото ребрата сочат от връх с по-малък към връх с по-голям индекс.

При $A = (7;\ 5;\ 6;\ 30;\ 201;\ 10)$ графът изглежда така:



Най-дългите пътища в графа имат дължина 2 (тоест те се състоят от две ребра и три върха). Има няколко такива пътя и всеки от тях съответства на най-дълга подредица от (три) числа, от които всяко след първото е по-голямо и с различна четност от предишното число. Една такава подредица е (7; 30; 201). Има и други, например (5; 30; 201) и (5; 6; 201).

Разновидност на горното решение е да добавим два фиктивни върха s и t: от върха s излизат ребра към всички други върхове (вкл. t), а в t влизат ребра от всички други върхове. Сега търсим най-дълъг път от s до t вместо най-дълъг път между всеки два върха.

Построяването на графа изисква време $\Theta(n^2)$: трябва да проверим за всяка двойка числа дали има ребро между тях. Търсенето на най-дълъг път в получения граф изразходва време, линейно спрямо размера на графа, което е пак $\Theta(n^2)$: толкова са ребрата в най-лошия случай, например когато в първата половина на входния масив числата са нечетни и малки, а във втората половина на масива са четни и големи.

Времето на целия алгоритъм е сборът от тези две времена, тоест $\Theta(n^2)$.

Можем да използваме същата идея, без да построяваме граф в явен вид. За целта съставяме алгоритъм по схемата ***динамично програмиране.*** Състоянието е текущият индекс на елемент от масива с входните данни. В динамичната таблица запазваме дължината на най-дългата подредица от желания вид, която завършва с текущия входен елемент. Запазваме и индекса на предпоследния член на редицата, за да можем да я възстановим.

Псевдокод на алгоритъма:

```
LongestSubsequence(A[1...n]: масив от цели числа)
dyn[1...n], prev[1...n]: масиви от цели неотрицателни числа;
// dyn[k] = дължината на най-дългата подредица на A[1...k],
// завършваща с A[k], всеки член на която е по-малък
// и с различна четност от следващия;
// prev[k] = индекса от A на нейния предпоследен член.
dyn[1] ← 1
prev[1] ← 0 // Елементът A[1] няма предишен.
for j ← 2 to n do
   dyn[j] ← 0
   prev[j] ← 0 // Елементът A[j] е начало на подредица.
   for i ← 1 to j - 1 do
      if A[i] < A[j] and A[i] % 2 + A[j] % 2 = 1
         // b % d = остатъка, който b дава при деление на d.
         if dyn[i] > dyn[j]
            dyn[j] ← dyn[i]
            prev[j] ← i // A[j] продължава някоя подредица.
            // Понеже i < j, то prev[j] < j.
   dyn[j] ← dyn[j] + 1
bestEnd ← 1
for j ← 2 to n do
   if dyn[j] > dyn[bestEnd]
      bestEnd ← j

// Възстановяване на решението:
// индексите на най-дългата подредица се отпечатват
// в обратен ред (от най-големия към най-малкия).
j ← bestEnd
while j > 0 do
   print j
   j ← prev[j] // Понеже prev[j] < j, то j намалява строго.

// Алгоритъмът връща дължината на най-дълга подредица,
// всеки член на която е по-малък и с различна четност
// от следващия.
return dyn[bestEnd]
```

Анализ на времевата сложност: Двата вложени цикъла, попълващи динамичната таблица, изразходват време $\Theta(n^2)$. Цикълът след тях, който обхожда попълнената таблица и търси най-голяма дължина, изисква време $\Theta(n)$. Възстановяването изисква време $O(n)$, тъй като индексът j намалява с поне една единица на всяка стъпка. Окончателно, времето за работа на целия алгоритъм е $\Theta(n^2)$.

Демонстрация на алгоритъма при $A = (7;\ 5;\ 6;\ 30;\ 201;\ 10)$:

k	1	2	3	4	5	6
$A[k]$	7	5	6	30	201	10
dyn[k]	1	1	2	2	3	2
prev[k]	0	0	2	1	3	1

Най-голямото число в реда dyn е 3 с индекс 5, затова алгоритъмът започва възстановяването от петия елемент на масива A:

prev[5] = 3; prev[3] = 2; prev[2] = 0 (няма повече членове).

Намерената най-дълга подредица се състои от втория, третия и петия член на масива A, тоест това е редицата $(5;\ 6;\ 201)$.

Задача 2 може да се реши за време от порядък $n \log n$ чрез малка промяна на алгоритъма, известен под името ***сортиране чрез пасианс.*** Променениият алгоритъм изглежда така:

1) Числата от входния масив са написани върху карти за игра — по едно число на карта. Теглим картите една по една от началото към края на масива. Изтеглените карти подреждаме в редица от купчинки (отначало празна) по следните правила.
2) Намираме най-лявата купчинка, чиято най-горна карта е не по-малка от текущата. Ако няма такава купчинка (тоест ако новата карта е по-голяма от най-горните карти на всички купчинки), тогава слагаме новата карта най-отдясно в нова купчинка и насочваме указател от нея към най-горната карта на последната стара купчинка.
3) В противен случай сравняваме четностите на текущата карта и най-горната карта от намерената купчинка. Ако двете карти са с еднаква четност, слагаме текущата карта върху най-горната карта от намерената купчинка, след което насочваме указателя от текущата карта към най-горната карта на предходната купчинка (тоест купчинката, която се намира непосредствено отляво на намерената); ако намерената купчинка е най-лявата, то указателят не сочи наникъде.
4) Ако текущата карта и най-горната карта от намерената купчинка са с различна четност, то разглеждаме два случая:
   а) Първи случай: Най-горната карта в купчинката непосредствено отляво на намерената има същата четност като картата, за която търсим място. Тогава просто отстраняваме текущата карта, тоест изхвърляме я встрани от всички купчинки (тя не участва повече в работата на алгоритъма).
   б) Втори случай: Най-горната карта в купчинката непосредствено отляво на намерената има различна четност от картата, за която търсим място. Тогава слагаме текущата карта между двете купчинки, т.е. правим нова купчинка непосредствено вляво от намерената, и насочваме указател от текущата карта към най-горната карта на купчинката вляво от текущата карта.
5) Изпълняваме стъпките № 2, № 3 и № 4 за всяка карта, изтеглена от входния масив. Накрая масивът се изчерпва.

6) Стъпките № 3 и № 4 гарантират, че картите във всяка купчинка са с еднаква четност. Обхождаме редицата от купчинки, групираме поредните купчинки с еднакви четности и връщаме броя на получените групи.

   П р и м е р : Ако редицата от купчинки изглежда така:

$$\underbrace{\text{четна, четна, четна,}}_{\text{група № 1}}\ \underbrace{\text{нечетна, нечетна,}}_{\text{група № 2}}\ \underbrace{\text{четна, четна,}}_{\text{група № 3}}$$

   връщаме стойност 3, тъй като има три групи от купчинки.

7) Най-дългата строго растяща подредица, редуваща четни и нечетни числа, се получава в обратен ред по следния начин: тръгваме от най-горната карта на последната купчинка и вървим наляво по указателите от поредната карта към карта в предходната купчинка (прескачаме картите със същата четност като текущата карта, тоест измежду картите от една група от купчинки взимаме само една, без значение коя, например най-дясната).

Както знаем, първата фаза от сортирането чрез пасианс се изпълнява за време $\Theta(n \log n)$ при най-лоши входни данни. Следователно това е времевата сложност на нашия алгоритъм. Той се различава от оригиналното сортиране чрез пасианс по добавянето на стъпка № 4, но тя не увеличава порядъка на времевата сложност, защото се изпълнява най-много n пъти (не повече от веднъж за всяка карта) и всяко нейно изпълнение изразходва време $O(\log n)$, ако пазим купчинките в ***приоритетна опашка*** (реализирана например с двоична пирамида); ключът на всяка купчинка е числото на нейната най-горна карта.

Тази промяна гарантира, че картите от всяка купчинка имат еднаква четност.

По-различно в сравнение с оригиналната версия е намирането на редицата (стъпка № 7). Разликата се състои в това, че пропускаме картите със същата четност като текущата карта (взимаме по една карта от всяка група от купчинки, а не по една карта от всяка купчинка).

Както в оригиналната версия на алгоритъма, най-горните карти на купчинките образуват строго растяща редица (която не е непременно подредица на дадения масив). Това свойство се доказва чрез математическа индукция по поредния номер на текущата карта:

Отначало няма купчинки, а с изтеглянето на първата карта се получава една купчинка. Празната редица и редиците с дължина единица са строго растящи.

Нека k е текущата карта на някоя следваща стъпка. Ако сложим k в нова купчинка отдясно на съществуващите (вж. стъпка № 2), то числото k е по-голямо от най-горните карти на всички купчинки, затуй текущата строго растяща редица от числата на най-горните карти се удължава с още един член, запазвайки свойството си да е строго растяща.

В противен случай нека b е най-горната карта от намерената купчинка. Да означим с a и c съответно най-горните карти на купчинките непосредствено отляво и отдясно на намерената. Съгласно с индуктивното предположение важат неравенствата $a < b < c$. От стъпка № 2 следва, че $a < k \le b$. От двете двойни неравенства заключаваме, че $a < k \le b < c$.

Ако k и b са с еднаква четност, слагаме k върху b (според стъпка № 3), при което редицата от най-горните карти се превръща от $\ldots < a < b < c < \ldots$ в $\ldots < a < k < c < \ldots$ и остава строго растяща.

Ако k и b са с различна четност, а пък k и a са с еднаква четност, то според стъпка № 4а изхвърляме картата k и редицата от най-горните карти не се променя.

Ако картите k и b са с различна четност и картите k и a също са с различна четност, то a и b са с еднаква четност. В такъв случай прилагаме стъпка № 4б: вмъкваме картата k между купчинките a и b; редицата от най-горните карти добива вида $\ldots < a < k < b < c < \ldots$ и остава строго растяща (неравенството $k \le b$ е строго, щом k и b имат различна четност).

В това доказателство може да липсва всяка от картите a и c, но изводите остават в сила.

Коректност на предложения алгоритъм: Да означим с K броя на групите от купчинки, а с L — дължината на най-дълга строго растяща подредица, редуваща четни и нечетни числа. Тоест K е отговорът на алгоритъма, а L е верният отговор. Трябва да докажем, че $K = L$ и че върнатата редица е най-дълга строго растяща подредица на дадения масив, редуваща четни и нечетни числа.

Върнатата редица е подредица на входния масив, защото указателят на всяка карта сочи непременно към по-рано изтеглена карта.

Върнатата подредица редува четни и нечетни числа, тъй като при нейното образуване пропускаме картите със същата четност като картата, току-що добавена към редицата (според стъпка № 7), тоест взимаме само по една карта от всяка група от купчинки.

Върнатата подредица расте строго, тъй като е образувана чрез проследяване на веригата от указателите на картите: указателите винаги сочат от по-голямо към по-малко число. Това е ясно от стъпки № 2, № 3 и № 4б (само те добавят новата карта към някоя купчинка):

— На стъпки № 2 и № 3 указателят се насочва от текущата карта към карта, която тъкмо е била подмината като по-малка от текущата.

— На стъпка № 4б указателят се насочва от k към a, а пък $a < k$, казано с обозначенията от предишната страница.

И така, върнатата подредица отговаря на всички изисквания с евентуално изключение на изискването за най-голяма дължина. K е дължината на една строго растяща подредица, редуваща четни и нечетни числа, а пък L е дължината на най-дълга такава подредица. Ето защо $K \le L$.

Остава да докажем, че $K \ge L$, с което доказателството за коректност ще бъде завършено, защото от двете неравенства следва, че $K = L$, а пък алгоритъмът връща редица с дължина K.

Действително, нека $k_1 < k_2 < k_3 < \ldots < k_L$ е най-дълга строго растяща подредица, редуваща четни и нечетни числа. Тъй като е подредица, то тези L карти се подреждат върху купчинките по растящ ред на номерата си. В момента, когато търсим място за k_i, за всяко $j < i$ важи двойното неравенство $k_i > k_j \ge$ най-горната карта на онази купчинка, в която е била поставена по-рано картата k_j. Последното неравенство (нестрогото) следва от стъпки № 2 и № 3 на алгоритъма: във всяка купчинка числата на картите отдолу нагоре намаляват или остават същите. Щом k_i е по-голяма от най-горните карти на купчинките на всички свои предшественици в избраната най-дълга растяща подредица, то картата k_i ще се разположи в нова купчинка отдясно на всички съществуващи (според стъпка № 2). Понеже това важи за всяко i от 1 до n включително, то всяка карта от най-дългата подредица става начало на нова купчинка. Следователно в края на алгоритъма има поне L купчинки, тоест $K \ge L$, което трябваше да се докаже.

П р и м е р и :

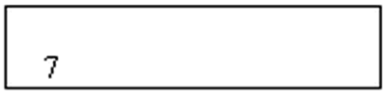
1) Ако най-горните карти на купчинките са 1, 4, 5, 8, 9 и 11, а следващата карта е 12, слагаме я най-отдясно, в нова купчинка: 1, 4, 5, 8, 9, 11 и 12.

2) Ако най-горните карти на купчинките са 1, 4, 5, 8, 9 и 11, а следващата карта е 6, слагаме я върху осмицата.

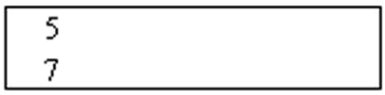
3) Ако най-горните карти на купчинките са 1, 4, 5, 8, 9 и 11, а следващата карта е 10, слагаме я между числата 9 и 11; получава се 1, 4, 5, 8, 9, 10 и 11.

4) Ако най-горните карти на купчинките са 1, 4, 5, 8, 9 и 11, а следващата карта е 7, не я слагаме никъде, а просто я премахваме: мястото на седмицата е между числата 5 и 8, които са с различна четност. Премахването на седмицата е основателно, защото тя може да бъде заместена с петицата във всяка строго растяща подредица, редуваща четностите, обаче обратното не е вярно: петицата невинаги може да бъде заместена със седмицата (ако например след петицата идва шестица).

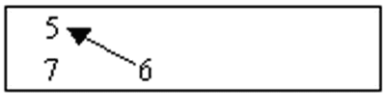
Демонстрация на алгоритъма върху масива $A = (7;\ 5;\ 6;\ 30;\ 201;\ 10)$:



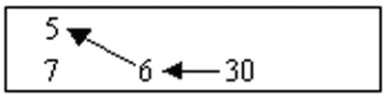
Поставяне на първата карта.



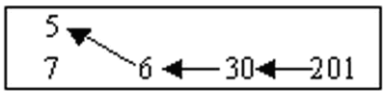
Поставяне на втората карта.



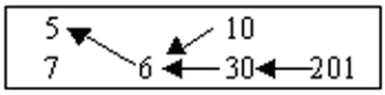
Поставяне на третата карта.



Поставяне на четвъртата карта.



Поставяне на петата карта.



Поставяне на шестата карта.

Втората и третата купчинка съдържат само четни числа, а първата и четвъртата — нечетни. Следователно има три групи от купчинки: първа — № 1; втора — № 2 и № 3; трета — № 4. Затова най-дълга строго растяща подредица, редуваща четни и нечетни числа, е с дължина 3. Една такава подредица намираме, като тръгнем от числото 201, което е най-горната карта в най-дясната купчинка. Движейки се по указателите, построяваме редицата $(201;\ 30;\ 6;\ 5)$. След четното число 30 не може да стои четното число 6, затова задраскваме шестицата. Остава редицата $(201;\ 30;\ 5)$, която след обръщане дава търсената подредица: $(5;\ 30;\ 201)$. Това е една най-дълга строго растяща подредица, редуваща четни и нечетни числа.