С О Р Т И Р А Н Е И Т Ъ Р С Е Н Е

ПРИМЕРНО КОНТРОЛНО № 2 ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ" — СУ, ФМИ
(ЗА СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 2. ПОТОК, ФЕВРУАРИ – ЮНИ 2019 Г.)

Забележка: Уточнявайте името на всеки използван алгоритъм за сортиране или търсене!

Задача 1. Числовият масив $A[1...n]$ съдържа дължините на n филма (измерени в секунди). Имаме M секунди и искаме да изгледаме възможно най-голям брой филми за това време. Предложете алгоритъм за подбор на филмите с времева сложност $O(n)$ в най-лошия случай.
а) Опишете алгоритъма с думи. **(15 точки)**
б) Анализирайте времевата сложност на алгоритъма. **(10 точки)**

Задача 2. Масивът от дробни положителни числа $A[1...n]$ съдържа радиусите на n звезди. Астрономът, извършил наблюденията и пресмятанията, предполага, че някои от тези обекти може да са двойници — образи на една и съща звезда, създадени от гравитационна леща. Помогнете на астронома да намери предполагаемите двойници, като съставите алгоритъм, който проверява за време $O(n \log n)$ в най-лошия случай дали измежду дадените радиуси има два, чиято разлика не надхвърля 3% от по-малкия радиус. (Ако има няколко подходящи двойки звезди, достатъчно е алгоритъмът да изведе една от тези двойки.)
а) Опишете алгоритъма на псевдокод. **(15 точки)**
б) Анализирайте времевата сложност на алгоритъма. **(10 точки)**

Задача 3. Нека $A[1...n]$ е логически масив, елементът $A[1]$ е истина, а пък $A[n]$ е лъжа. Предложете най-бърз алгоритъм за намиране на два последователни елемента, от които първият е истина, а вторият — лъжа. Алгоритъмът да връща по-малкия от двата индекса. Само най-бърз алгоритъм носи точки!
а) Опишете алгоритъма на псевдокод. **(15 точки)**
б) Анализирайте времевата сложност на алгоритъма. **(10 точки)**

Задача 4. Масивът $A[1...n]$ съдържа цели числа — възрастите на няколкостотин деца в години. Съставете алгоритъм, който за време $O(n)$ избира пет деца така, че разликата във възрастите на най-голямото и най-малкото от избраните деца да бъде минимална.
а) Опишете алгоритъма на псевдокод. **(15 точки)**
б) Анализирайте времевата сложност на алгоритъма. **(10 точки)**

Р Е Ш Е Н И Я

Задача 1. За да изгледаме най-много филми за даденото време M, избираме най-късите. Чрез сортиране на филмите не можем да постигнем линейна сложност, затова се отказваме от сортирането. Вместо това използваме ***алгоритъма PICK в съчетание с двоично търсене.*** По-конкретно:

1) Намираме медианата на дължините на филмите посредством алгоритъма PICK.
2) Разделяме филмите на къси и дълги спрямо медианата: слагаме късите филми вляво от медианата, а дългите — вдясно. (Смятаме и медианата за къс филм.) Ако има няколко филма с дължини, равни на медианата, то някои от тях слагаме сред късите, а други — сред дългите филми, така че двата подмасива (на късите и на дългите филми) да имат равен или почти равен брой елементи (дължините на двата подмасива да се различават най-много с единица).
3) Пресмятаме сбора S от дължините на късите филми. Разглеждаме три случая:
   а) Ако $S = M$, ще гледаме всички къси филми и само тях; край на алгоритъма.
   б) Ако $S > M$, то времето, с което разполагаме, не стига дори за късите филми. Затова се отказваме от дългите филми и изпълняваме алгоритъма рекурсивно върху масива от късите филми, като запазваме стойността на M.
   в) Ако $S < M$, то времето, което имаме, е достатъчно за всички къси филми, а вероятно ще остане време и за някои от дългите филми. Затова решаваме, че ще гледаме всички къси филми, и изпълняваме алгоритъма рекурсивно върху масива от дългите филми, като подаваме $M - S$ вместо M. Тоест времето, което ни остава, след като сме изгледали всички къси филми, служи като ограничение при избора на дълги филми.

Дъното на рекурсията е при $n \le 2$. Ако $n = 0$, алгоритъмът трябва да върне празен списък При $n = 1$ гледаме единствения филм само ако дължината на филма не надвишава M. При $n = 2$ гледаме по-късия филм, ако дължината му не надвишава M; а после — по-дългия, ако дължината му не надхвърля оставащото време (M минус дължината на по-късия филм). Търсим времевата сложност $T(n)$ на алгоритъма в най-лошия случай: когато алгоритъмът не попада в подточка "а" на точка 3. Функцията $T(n)$ удовлетворява рекурентното уравнение

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Делението на две идва от това, че при рекурсията дължината на масива намалява два пъти, защото разделянето на филмите на къси и дълги се извършва относно медианата. Коефициентът пред неизвестната функция в дясната страна на уравнението е единица, понеже се изпълнява само едно от рекурсивни извиквания в подточки "б" и "в" на т.3: не е възможно сборът S да бъде едновременно по-голям и по-малък от стойността на M. Свободният член на уравнението е общото време, изразходвано от алгоритъма PICK, разделянето на филмите на къси и дълги спрямо медианата, събирането на дължините, маркирането на филми дали ще бъдат гледани. Всяка от тези стъпки изисква линейно време, затова и общото им време е линейно: $\Theta(n)$.

Полученото рекурентно уравнение решаваме с мастър-теоремата, откъдето намираме, че времевата сложност на алгоритъма в най-лошия случай е линейна: $T(n) = \Theta(n)$.

Задача 2. Сортираме масива с някой бърз алгоритъм, например с ***пирамидално сортиране.*** Ако масивът съдържа близки стойности, то след сортирането те ще се окажат една до друга. Затова след сортирането е достатъчно да проверим двойките от съседни елементи. За да можем да отпечатаме оригиналните индкеси на елементите, разместваме индексите заедно със стойностите на елементите. За целта създаваме допълнителен масив от обекти, всеки от които притежава две полета — стойност и индекс на съответния елемент от входния масив.

Псевдокод:

```
STARS(A[1...n])
B[1...n]: array of pairs <key, index>
for k ← 1 to n do
   B[k].key ← A[k]
   B[k].index ← k
HeapSort(B[1...n]) // сортирането е по ключовете (key)
for k ← 2 to n do
   if B[k].key < 1,03 × B[k-1].key
      print B[k].index
      print B[k-1].index
      return true // Намерена е подходяща двойка звезди.
print "Няма"
return false // Няма подходяща двойка звезди.
```

Анализ на времевата сложност на алгоритъма: Най-лошият случай е, когато масивът A не съдържа подходяща двойка елементи или такава двойка образуват най-големите числа. Времето за инициализиране на помощния масив B е $\Theta(n)$, пирамидалното сортиране изразходва време $\Theta(n \log n)$, а времевата сложност на цикъла е $\Theta(n)$. Затова общото време на целия алгоритъм е $\Theta(n \log n)$. Очевидно помощният масив B не увеличава порядъка на времевата сложност.

Задача 3 се решава за време $\Theta(\log n)$ с помощта на ***двоично търсене:***

```
Problem3(A[1...n]) // n ≥ 2, A[1] = true, A[n] = false
first ← 1
last ← n
while last > first + 1 do
   mid ← ⌊(first + last) / 2⌋
   if A[mid] = true
      first ← mid
   else
      last ← mid
return first
```

Така се проверяват дълги сметки по някоя формула. Ако последният ред (отговорът) е верен, приемаме, че сметките са верни (въпреки че няколко грешки могат да се унищожат). Ако отговорът е грешен, търсим къде е грешката. Проверяваме първия ред: ако е грешен, значи грешката е в самото начало. Ако е верен, намираме грешката с двоично търсене. Така си спестяваме проверката на цялото решение: проверяваме само отделни редове от решението.

Задача 4. Възрастите на децата са малки цели числа — от 0 до 17 включително. Има само 18 различни стойности на възрастта, което е много по-малко от броя на децата (няколкостотин). Затова можем да използваме ***сортиране чрез броене.*** След сортирането децата с близки възрасти идват едно до друго. Затова е достатъчно да проверим само петорките от последователни деца. В условието на задачата не е указан идентификатор на децата (име, ЕГН или нещо друго), ето защо ще посочваме децата с техните индекси в масива $A[1...n]$. Сортирането размества децата, затова в друг масив $B[1...n]$ ще пазим първоначалните индекси.

```
Problem4(A[1...n])
C[0...17]: array of lists // масив от списъци
for p ← 0 to 17 do
   C[p] ← empty list // празен списък
for k ← 1 to n do
   C[A[k]].Append(k) // добавяме индекса k в края на списъка C[A[k]]
B[1...n]: array of integers // масив от първоначалните индекси
j ← 1
for p ← 0 to 17 do
   while C[p] is not empty do
      k ← C[p].ExtractFirst // четем и изтриваме първия елемент от списъка C[p]
      A[j] ← p
      B[j] ← k
      j ← j + 1
minDiff ← +∞
for j ← 1 to n - 4 do    // търсим пет деца с най-малка възрастова разлика
   currentDiff ← A[j + 4] - A[j]
   if currentDiff < minDiff            // намерена е по-добра петорка деца
      minDiff ← currentDiff
      minIndex ← j
for j ← minIndex to minIndex + 4 do   // извеждаме най-добрата петорка
   print B[j]                         // отпечатваме първоначалните индекси
```

Анализ на времевата сложност на алгоритъма:
— Първият цикъл for — този, който инициализира масива $C[0...17]$ с празни списъци — изисква константно време $\Theta(1)$. Теоретичното основание за този извод е, че броят на повторенията (18) на тялото на цикъла не зависи от n. От практическа гледна точка е важно още времето на цикъла да е много по-малко от n. Това изискване също е изпълнено, защото броят на повторенията (18) е много по-малък от дължината n на входния масив (по условие n е няколкостотин).
— Вторият цикъл for — този, който попълва списъците — изразходва време $\Theta(n)$: толкова отива за едно обхождане на масива $A[1...n]$.
— Третият цикъл for — този, който заедно с вложения цикъл while презаписва масива $A[1...n]$, подреждайки децата по възраст — изразходва време $\Theta(n)$, защото най-вътрешните оператори се изпълняват точно n пъти: по веднъж за всяка стойност на променливата j, която се увеличава с единица на всяка стъпка, започвайки от 1 и достигайки до n.
— Тялото на четвъртия цикъл for — който търси най-малката възрастова разлика — се изпълнява точно $n - 4 = \Theta(n)$ пъти, затова има линейна времева сложност.
— Петият цикъл for — този, който отпечатва намерената петорка — има времева сложност $\Theta(1)$: броят (5) на повторенията на тялото му не зависи от n и е малко число (много по-малко от n, което е от порядъка на няколкостотин).

Окончателно, времевата сложност на алгоритъма е $\Theta(1) + \Theta(n) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.

Забележки по решението на задача 4. По тази задача са възможни различни оптимизации. Тук ще ги обсъдим накратко.

Щом възрастите на децата са цели числа от 0 до 17 включително, то по принципа на Дирихле следва, че ако има поне 73 деца (а по условие те са няколкостотин), то между тях ще се намерят поне пет на еднаква възраст. Това опростява втората фаза на алгоритъма (същинското сортиране): не е нужно да променяме масива $A[1...n]$. Вместо това е достатъчно да обходим списъците $C[p]$ един по един, докато открием списък с дължина поне 5 (непременно има такъв). Така спестяваме предпоследния цикъл — обхождането на сортирания масив $A[1...n]$ в търсене на най-малка възрастова разлика. Времевата сложност на алгоритъма остава линейна: $\Theta(n)$, тоест не се променя по порядък, обаче константният множител пред n намалява, тъй като отпадат някои операции от алгоритъма. Затова тази малка оптимизация носи допълнителни 3 точки.

Много по-съществена оптимизация на времевата сложност — по порядък! — ще постигнем, ако се сетим, че е достатъчно да изследваме възрастите само на първите 73 деца: както казахме, между тях със сигурност има поне пет на една и съща възраст. Как точно ще ги изследваме, е подробност. Важното е, че броят им 73 не зависи от n (и е много по-малък от няколкостотин), така че този алгоритъм има константна времева сложност: $\Theta(1)$. Откриването на това решение удвоява точките (за всяко от двете подусловия).

Обратно, следното решение, въпреки че има линейна времева сложност, е непрофесионално, затова носи само 4 точки (по 2 точки за всяко подусловие): 18 пъти обхождаме масива $A[1...n]$ — по веднъж за всяка възможна възраст. Спираме търсенето при първото обхождане, при което успеем да открием пет деца на съответната възраст. В най-лошия случай това обхождане ще бъде последното (осемнадесетото). Сложността е линейна ($18n$), обаче константният множител е голям (правим твърде много обхождания на масива).