С О Р Т И Р А Н Е   И   Т Ъ Р С Е Н Е

КОНТРОЛНО № 2 ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ" — СУ, ФМИ
(ЗА СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 1. ПОТОК; 28 МАРТ 2018 Г.)

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

Задача 1. Две цели положителни числа ще наричаме подобни, ако съвпадат след зачертаване на последните две цифри (например 391452 и 391418).

а) Даден е масив $A[1...n]$ от цели положителни числа.
Съставете алгоритъм, който за време $O(n \log n)$ намира две подобни числа, стоящи на различни места в масива.

Опишете алгоритъма на псевдокод като функция от вида
`FindSimilarNumbers(A[1...n]: array of integers)`

Входни данни: масив A от n цели положителни числа.

Изход (върната стойност): false — ако A не съдържа подобни числа; true — ако съдържа. Във втория случай алгоритъмът да отпечатва стойностите на намерените две подобни числа.

Изпълнете алгоритъма стъпка по стъпка с входни данни: $A = (9104;\ 7208;\ 3015;\ 7241)$. Покажете междинните резултати (след всяка стъпка) и крайния резултат (върнатата стойност и двете отпечатани числа, ако има такива). **(1 точка)**

б) Докажете, че поставената алгоритмична задача изисква време $\Omega(n \log n)$. **(1 точка)**

Задача 2. Масивът `B[1...n]` съдържа теглата на n предмета (в килограми), а ние можем да носим най-много K килограма. Съставете бърз алгоритъм за избор на максимален брой предмети, които можем да пренесем наведнъж.

Алгоритъмът трябва да намира не само максималния брой, но и самите предмети.

Опишете алгоритъма с думи и го демонстрирайте с подходящи примерни данни. Демонстрацията трябва да показва както междинните сметки, така и крайния резултат. Анализирайте времето на изпълнение в най-лошия случай. Не се приемат алгоритми без коректен анализ на времевата сложност.

Ако максималната времева сложност на алгоритъма е $O(n)$, ще получите **2 точки.**

В противен случай, ако максималната сложност е $O(n \log n)$, ще получите **1 точка.**

По-бавни алгоритми не носят точки.

Оценката = 2 + броя на получените точки.
Не се зачитат решения, които не отговарят на изискванията!

Р Е Ш Е Н И Я

Задача 1.

а) Използваме някоя бърза сортировка, например пирамидалното сортиране.

```
FindSimilarNumbers(A[1...n]: array of integers)
Sort(A) // HeapSort или MergeSort, или QuickSort
for k ← 2 to n
   if ⌊A[k]/100⌋ = ⌊A[k-1]/100⌋
      print A[k], A[k-1]
      return true // има подобни числа
return false // няма подобни числа
```

Анализ на алгоритъма: сортирането изразходва време $\Theta(n \log n)$ в най-лошия случай, обхождането на сортирания масив — $\Theta(n)$; общо: $\Theta(n \log n)$.

Демонстрация на алгоритъма при $A = (9104;\ 7208;\ 3015;\ 7241)$.

1) Сортираме масива: $A = (3015;\ 7208;\ 7241;\ 9104)$.
2) Последователно сравняваме частните при деление на 100, закръглени надолу, тоест числата 30, 72, 72 и 91:
   — на първия и втория елемент: $30 \neq 72$;
   — на втория и третия елемент: $72 = 72$.
3) Тук алгоритъмът прекъсва търсенето, тъй като е намерил подобни числа: 7208 и 7241. Алгоритъмът отпечатва тези две числа (7208 и 7241) и връща true.

б) Използваме редукция от алгоритмичната задача `ElementUniqueness` (проверка за липса на повторения).

```
ElementUniqueness(A[1...n]: array of integers)
for k ← 1 to n
   A[k] ← 100 × A[k]
return not FindSimilarNumbers(A[1...n])
```

Цикълът, умножавайки числата по 100, добавя по две нули в края на всяко число. После тези нули се пренебрегват от `FindSimilarNumbers`, какъвто и алгоритъм да сме избрали за решаването на задачата (защото такава е дефиницията на самата задача). Следователно алгоритъмът `FindSimilarNumbers` (който и да е той) сравнява оригиналните числа, т.е. редукцията е *коректна*.

По-формално, ако `A[1...n]` е масивът, подаден на `ElementUniqueness`, тогава е в сила следната еквивалентност:

`A[i] = A[j]` $\Leftrightarrow$ числата `100 × A[i]` и `100 × A[j]` са подобни.

Оттук следва друга еквивалентност:

входните данни на `ElementUniqueness` съдържат поне две равни числа $\Leftrightarrow$
входните данни на `FindSimilarNumbers` съдържат поне две подобни числа.

Тази еквивалентност е формален израз на коректността на редукцията.

Редукцията се състои от:
— цикъла, който обхожда масива и умножава числата по 100 — това изисква време $\Theta(n)$;
— пресмятането на отрицанието на върнатата стойност — това изисква време $\Theta(1)$.

Следователно общото време на редукцията е $\Theta(n)$. Понеже $n \prec n \log n$, то редукцията е достатъчно бърза за целите на доказателството.

За да бъде решението напълно точно, трябва да бъде поправен следният недостатък. Входните данни на задачата `ElementUniqueness` могат да бъдат всякакви цели числа — положителни, отрицателни, нули. А входните данни на задачата `FindSimilarNumbers` са само положителни цели числа. Затова при тази редукция има опасност да бъдат подадени недопустими входни данни на функцията `FindSimilarNumbers`. Това може да бъде предотвратено по следния начин: най-малкото число във входния масив, намалено с 1, го изваждаме от всички числа (така те стават положителни). Ако най-малкото число е например -7, то от всички числа изваждаме -8, тоест добавяме 8.

```
ElementUniqueness(A[1...n])
min ← A[1]
for k ← 2 to n do
   if A[k] < min
      min ← A[k]
min ← min - 1
for k ← 1 to n do
   A[k] ← 100 × (A[k] - min)
return not FindSimilarNumbers(A[1...n])
```

Бързина на редукцията: $\Theta(n)$ заради двата последователни цикъла. Понеже $n \prec n \log n$, то редукцията е достатъчно бърза.

Коректност на редукцията: Прибавянето на достатъчно голямо число гарантира, че всички числа в масива са станали положителни, тоест функцията `FindSimilarNumbers` получава допустими входни данни. От друга страна, събирането на едно и също число с всички числа запазва техните разлики, а значи запазва и върнатата стойност от `FindSimilarNumbers`, поради което можем да се позовем на разсъжденията от предишната страница.

Проблемът с недопустимите стойности на `FindSimilarNumbers` е логическа тънкост, поради което се приемат и решения, в които този проблем не е забелязан. Решения, в които проблемът е забелязан и отстранен, носят допълнителна единица към оценката.

Задача 2 може да бъде решена по различни начини.

Първи начин: чрез някоя от бързите сортировки, например пирамидалното сортиране.
1) Сортираме предметите, т.е. подреждаме ги в растящ ред на теглата им.
2) Товарим предметите, като започваме от най-леките и продължаваме последователно, докато ги изчерпим или до надхвърляне на максималното допустимо тегло K.
Тази стъпка се реализира с цикъл, в който обхождаме сортирания масив `B` и натрупваме сбора на елементите му, докато ги изчерпим или докато сборът им надхвърли K.

Анализ на алгоритъма: сортирането изисква време $\Theta(n \log n)$ в най-лошия случай, обхождането и сумирането на сортирания масив — $\Theta(n)$; общо: $\Theta(n \log n)$.

Втори начин: без сортиране. Вместо това използваме алгоритъма PICK.

1) Намираме медианата на масива `B` с помощта на алгоритъма PICK.
2) Разделяме предметите на леки и тежки относно медианата.
3) Пресмятаме сбора L от теглата на леките предмети.
4) Ако L = K, товарим всички леки предмети, отказваме се от всички тежки. Край.
5) Ако L > K, отказваме се от всички тежки предмети и рекурсивно извикваме алгоритъма върху леките предмети, като използваме същия капацитет K. Край на алгоритъма.
6) Ако L < K, товарим всички леки предмети и рекурсивно извикваме алгоритъма върху тежките предмети, като използваме останалия капацитет K – L в ролята на K. Край.

Дъното на рекурсията е при $n = 1$. Тогава сравняваме капацитета K с теглото `B[1]` на единствения предмет: товарим предмета, ако и само ако `B[1]` $\leq$ K.

Анализ на алгоритъма: Най-лошият случай е, когато стъпка № 4 не се изпълнява, тоест на всеки етап алгоритъмът изпълнява или стъпка № 5, или стъпка № 6.

Стъпки № 1, № 2, № 3, отказването и товаренето на предмети в стъпки № 5 и № 6 — всички тези операции изискват линейно време $\Theta(n)$. От двете рекурсивни извиквания в стъпки № 5 и № 6 се изпълнява само едното; всяко от тях работи върху половината масив, понеже разбиването на масива на големи и малки стойности е извършено спрямо медианата. Ето защо времевата сложност на алгоритъма удовлетворява рекурентното уравнение

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

От третия случай на мастър-теоремата следва, че решението на уравнението е $T(n) = \Theta(n)$, тоест алгоритъмът има линейна времева сложност.

Демонстрация на алгоритмите. Нека `B` = (50; 20; 80; 10) и K = 45.

Първият алгоритъм сортира масива: `B` = (10; 20; 50; 80). После започва да събира елементите му, като сравнява получените суми с капацитета K = 45:
10 ≤ 45, следователно предметът с тегло 10 кг ще бъде натоварен;
10 + 20 = 30 ≤ 45; следователно и предметът с тегло 20 кг ще бъде натоварен;
обаче 30 + 50 = 80 > 45, следователно за другите предмети нямаме достатъчно сили.

Извод: Можем да натоварим най-много два предмета — тези с тегла 10 и 20 килограма.

Вторият алгоритъм намира медианата 35 и разбива масива: `B` = (20; 10 | 50; 80). Сборът от теглата на леките предмети е L = 20 + 10 = 30 < K = 45. Затова алгоритъмът товари двата леки предмета и преминава рекурсивно към изследване на тежките предмети, като сега разполага с капацитет K – L = 15. Това е новото K, а новият масив `B` = (50; 80). Неговата медиана е 65, разбиването е `B` = (50 | 80), сборът от теглата на леките предмети е L = 50 > K = 15, затова алгоритъмът се отказва от тежкия предмет (с тегло 80). Следва рекурсия върху първата половина от масива, тоест новият масив е `B` = (50), капацитетът остава същият: K = 15. Достигнато е дъното на рекурсията — масив с дължина единица. Неравенството 50 > 15 показва, че единственият останал предмет не може да бъде натоварен.

Извод: Можем да натоварим най-много два предмета — тези с тегла 10 и 20 килограма.