Домашна работа № 2 по "Дизайн и анализ на алгоритми"
за специалност "Компютърни науки", 2. курс, 1. поток — СУ, ФМИ,
летен семестър на 2017 / 2018 уч. г.

СЪСТАВЯНЕ НА АЛГОРИТМИ

Задача	1	2	3, а	3, б	3, в	*Общо*
получен брой точки						
максимален брой точки	*20*	*20*	*20*	*20*	*20*	*100*

Задача 1. Функцията $f(n)$ е дефинирана за цели неотрицателни числа:

$$f(0) = 0, \quad f(1) = 1, \quad f(2n) = f(n), \quad f(2n+1) = f(n) + f(n+1).$$

Съставете алгоритъм за изчисляване на $f(n)$ със сложност по време $O(\log n)$ и сложност по памет $O(1)$. Опишете алгоритъма като програма на езика C. Анализирайте сложността на алгоритъма по време и памет.

Упътване: Опитайте се да изчислите f от конкретна стойност, напр. $f(99)$, чрез развиване на уравнението. Видът на получаваните изрази ще ви подскаже идеята за решението.

Задача 2. В един град всички улици са еднопосочни. В града има n кръстовища, номерирани с целите числа от 1 до n. Някои двойки кръстовища p и q са свързани с еднопосочна отсечка от улица, така че по тази отсечка от улица няма други кръстовища. За всеки две кръстовища p и q има най-много една еднопосочна отсечка от улица — или в посока от p към q, или от q към p. При спазване на правилата за движение в града невинаги е възможно да отидем с кола от кръстовище a до кръстовище b. С колко най-малко нарушения можем да се придвижим от a до b? Всяко навлизане в посока, обратна на разрешената, в еднопосочна отсечка от улица се брои за едно нарушение.

Ако n е броят на кръстовищата, а m е броят на еднопосочните отсечки от улици, които ги свързват, то алгоритъм с времева сложност $O(m+n)$ носи 20 точки, алгоритъм с времева сложност $O((m+n)\log(m+n))$ се оценява с 10 точки, а по-бавен алгоритъм не носи точки.

Задача 3. Когато искаме да съединим голяма купчина листове с телбод, може да се наложи да използваме няколко телчета. Например, ако всяко телче захваща максимум 10 листа, а ние искаме да съединим 25 листа, ще трябва да използваме три телчета: с едното съединяваме листовете от № 1 до № 10, с второто — листовете от № 10 до № 19, с третото — от № 19 до № 25. Има и други начини, например с първото телче можем да защипем листовете от № 1 до № 10, с второто — от № 8 до № 17, с третото — от № 16 до № 25. Тоест не е задължително всички телчета да защипват еднакъв брой листове. Ако обаче с първото телче защипем листовете с номера от № 1 до № 10, с второто телче — от № 12 до № 21, а с третото — от № 19 до № 25, то не всички листове ще бъдат съединени; по-точно, ще има три "папки": едната папка ще съдържа листовете от № 1 до № 10, втората ще се състои от единствен лист (№ 11), а третата папка ще бъде съставена от листовете с номера от № 12 до № 25.

Тези наблюдения дават повод да разгледаме следната алгоритмична задача: Имаме два масива $a_1, a_2, \ldots, a_n$ и $b_1, b_2, \ldots, b_n$ от цели положителни числа, като $a_i < b_i$ за всяко $i = 1, 2, 3, \ldots, n$. Нека $L = \max\left\{b_1, b_2, \ldots, b_n\right\}$. Тълкуваме входните данни като n телчета, които защипват общо L листа, номерирани с целите числа от 1 до L. При това, i-тото телче защипва листовете с номера от a_i до b_i включително. Условието $a_i < b_i$ гарантира, че всяко телче защипва поне два листа. Няма горна граница за броя на листовете, които едно телче може да защипва. Съставете алгоритми, които да пресмятат броя на получените папки (множества от пряко или косвено захванати листове) и да отговарят на следните изисквания към сложността:

а) Предложете алгоритъм с максимална времева сложност $\Theta(n)$, когато числото L е от порядъка на n.

б) Предложете алгоритъм, който независимо от стойността на L има времева сложност $\mathrm{O}(n \log n)$ в най-лошия случай.

Упътване: Използвайте алгоритъма от подусловие "а".

в) Съществува ли алгоритъм, основан на сравнения, който независимо от стойността на L работи във време $\mathrm{O}(n)$ в най-лошия случай?

Опишете алгоритмите с думи или на псевдокод. Илюстрирайте ги с примери. Анализирайте техните максимални времеви сложности.

РЕШЕНИЯ

Задача 1 се решава чрез ***динамично програмиране.*** Само че обикновеният итеративен вариант на схемата е неприложим, тъй като изчисляването на f за всички стойности на аргумента от 1 до даденото n ще бъде по-бавно от максимално допустимото време, а запаметяването на всички тези стойности надхвърля поставеното ограничение на паметта. Затова ще използваме идеята на рекурсивния вариант на динамичното програмиране (т. нар. мемоизация): при него се пресмятат само необходимите стойности на f, а това пести време.

Забелязваме, че рекурентната формула за f съдържа линейна комбинация от две стойности на f. Това ни подсеща да разгледаме следната функция: $g(n,\ i,\ j) = i \,.\, f(n) + j \,.\, f(n+1)$. За g има по-прости рекурентни формули: $g(0,\ i,\ j) = j$, $g(2n,\ i,\ j) = g(n,\ i+j,\ j)$, $g(2n+1,\ i,\ j) = g(n,\ i,\ i+j)$. Функцията f се изчислява така: $f(n) = g(n,\ 1,\ 0) = \ldots = g(0,\ i,\ j) = j$. Тъй като функцията g участва отдясно в рекурентните формули само веднъж, то рекурсията може да се замени с цикъл, което пести памет. Програма на C:

```
unsigned int f(unsigned int n) {
  int i = 1;
  int j = 0;
  while (n > 0) {
    if (n % 2)
      j += i;
    else
      i += j;
    n >>= 1;
  }
  return j;
}
```

Сложността по време на този алгоритъм е $\Theta(\log n)$, защото толкова стъпки са му нужни, за да стигне от n до 1 чрез последователно деление на 2. Сложността по памет е $\Theta(1)$: използват се две локални променливи (i и j) и те са от примитивен тип (цели числа).

Задачата е дадена на Московската ученическа олимпиада през 1981 г.

Задача 2 (авторска — Д. Кралчев и Е. Келеведжиев).
Данните могат да се моделират чрез граф: върхове на графа са кръстовищата, а ребра — еднопосочните отсечки от улици, които ги свързват. Даваме им тегла 0. Добавяме обратни ребра (забранените посоки на движение), но на тях даваме тегла 1. Търсим най-къс път от даден връх a до друг даден връх b.

Първи начин: Тъй като теглата на ребрата са неотрицателни (0 и 1), можем да приложим ***алгоритъма на Дейкстра.*** Това решение има скорост $\Theta(m+n\log n)=\mathrm{O}((m+n)\log(m+n))$ и носи 10 точки.

Втори начин: чрез ***търсене в ширина*** по обратните (забранените) ребра. Това решение има скорост $\Theta(m+n)$ и носи 20 точки. По-точно, алгоритъмът започва от върха a и се движи само по разрешени ребра; след изчерпването им върши едно нарушение, т.е. стъпка по обратните ребра. После пак използва прави ребра до изчерпването им, тогава върши второ нарушение; и т.н.

```
SearchShortestPath(V,a,b,PermittedEdges,ForbiddenEdges)
PermittedQueue ← empty queue
ForbiddenQueue ← empty queue
for each u ∈ V do
  visited[u] ← false
  pred[u] ← nil
ForbiddenQueue.Append(<a,0,nil>)
while ForbiddenQueue is not empty do
  <u,offences,predcessor> ← ForbiddenQueue.Extract
  PermittedQueue.Append(<u,offences,predcessor>)
  while PermittedQueue is not empty do
    <u,offences,predcessor> ← PermittedQueue.Extract
    if not visited[u]
      visited[u] ← true
      pred[u] ← predcessor
      if u = b
        return offences, pred // пътят в обратен ред
      for each v ∈ PermittedEdges[u] do
        PermittedQueue.Append(<v,offences,u>)
      for each v ∈ ForbiddenEdges[u] do
        ForbiddenQueue.Append(<v,offences+1,u>)
return -1, nil // no path found
```

Задача 3 (авторска — Д. Кралчев).

а) Когато L е от порядъка на n, използваме идеята на сортирането чрез броене.

```
Alg_1(A[1...n], B[1...n]: array of integer): integer
// търсим L
L ← B[1]
for k ← 2 to n
   if L < B[k]
      L ← B[k]
depth: integer // дълбочина = брой обхващащи интервали
C[1...L]: array of integer // измененията на depth
for k ← 1 to L
   C[k] ← 0
for k ← 1 to n
   C[A[k]] ← C[A[k]] + 1 // дълбочината расте
for k ← 1 to n
   C[B[k]] ← C[B[k]] - 1 // дълбочината намалява
count: integer // броят на папките
count ← 0
depth ← 0
for k ← 1 to L
   depth ← depth + C[k]
   if depth = 0 // несъединени листове
      count ← count + 1 // нова папка
return count
```

Коректност на алгоритъма: При всяка проверка, дали *depth* = 0, е в сила равенството: *depth* = броя на интервалите $[a_i\,;\,b_i)$, съдържащи числото k. Следователно *count* = броя на празнините, т.е. броя на числата, непокрити от никой интервал $[a_i\,;\,b_i)$. Всяка празнина е край на папка и обратно. Ето защо последната стойност на *count* (върнатата стойност на алгоритъма) е тъкмо броят на папките.

Времевата сложност на алгоритъма е $\Theta(\max(n\,,L))$, което е равно на $\Theta(n)$, когато L е от порядъка на n.

б) Задачата се решава за време $\Theta(n \log n)$ независимо от стойността на L с помощта на някоя бърза сортировка, например пирамидално сортиране. По-точно, сортираме масива A и разместваме елементите на B съответно.

```
Alg_2(A[1...n], B[1...n]: array of integer): integer
HeapSort(A[1...n]) and rearrange B according to A
count ← A[1]
maxB  ← B[1]
for k ← 2 to n
   if A[k] > maxB
      count ← count + A[k] – maxB
   if B[k] > maxB
      maxB ← B[k]
return count
```

Времевата сложност е $\Theta(n \log n)$ заради сортирането; всички други стъпки се изпълняват за линейно време.

Коректността на алгоритъма се доказва с помощта на следната *инвариантa:*
При всяка проверка за край на цикъла `k` ≤ `n` са в сила следните равенства:

maxB = най-голямото число в подмасива `B[1...k-1]`;

count = броя на папките, образувани от листовете № 1 – № maxB

(последната от тези папки може да не е завършена още).

Инвариантата се доказва с индукция по итерациите на цикъла, тоест с индукция по `k`. Базата е при `k` = 2; тогава maxB = `B[1]` = max `B[1...k-1]`
count = `A[1]` = броя на папките, намерени дотук (една от тях е образувана с първото телче, а останалите — от единичните листове преди първото телче).
Индуктивна стъпка: на всяка итерация maxB нараства, ако се открие по-голям елемент на `B`, а count се увеличава с 1 за всяко телче, което започва папка, и с една единица за всеки незащипан лист между новата и старата папка.

Цикълът завършва, защото е цикъл по брояч. При последната проверка:
`k` = `n` + 1, maxB = max `B[1...k-1]` = max `B[1...n]` = L,
count = броя на папките, образувани от листовете № 1 – № maxB,
тоест от листовете № 1 – № L. Понеже това са всички листове, то следва, че
count = броя на всички папки, което е и върнатата стойност на алгоритъма.

в) В общия случай (когато L е произволно) задачата не може да се реши за време $o(n \log n)$ чрез алгоритъм, основан на сравнения. Това изключва съществуването на такъв алгоритъм с линейна времева сложност.

Доказателство: чрез редукция от `ElementUniqueness` — задачата за разпознаване дали сред n числа има поне две равни. Както е известно, тази задача изисква време $\Omega(n \log n)$, когато се решава чрез сравнения.

Самата редукция, описана на псевдокод, изглежда така:

```
ElementUniqueness(M[1...n]: array of integer): boolean
min ← M[1]
for k ← 2 to n
   if min > M[k]
      min ← M[k]
A[1...n]: array of integer
B[1...n]: array of integer
for k ← 1 to n
   A[k] ← 2 * (M[k] - min) + 1   // A[k] става нечетно
   B[k] ← A[k] + 1
L ← B[1]
for k ← 2 to n
   if L < B[k]
      L ← B[k]
return L - n = Folders(A,B)
```

Коректност на редукцията: Нека `Folders(A,B)` е произволен (коректен) алгоритъм, който пресмята броя на папките. От това, че $B[k] = A[k] + 1$, следва, че всяко телче защипва два последователни листа. При това, две телчета или съвпадат (когато съответните им числа в масива M са равни), или нямат общ край (в обратния случай). Всяка папка съдържа един или два листа. Броят на папките с два листа е точно броят m на различните числа в масива M. Броят на листовете е L, затова броят на папките с един лист е $L - 2m$, а броят на всички папки е $L - m$. Алгоритъмът връща стойност истина точно когато $L - m = L - n \Leftrightarrow m = n \Leftrightarrow$ в масива M няма равни числа.

Бързина на редукцията: Описаната редукция изисква линейно време $\Theta(n)$, следователно е достатъчно бърза, понеже $n = o(n \log n)$.

СХЕМА ЗА ОЦЕНЯВАНЕ

Задача 1 носи общо 20 точки, от които 15 точки за алгоритъм, който отговаря на изискванията, и 5 точки за анализ на алгоритъма.

Задача 2 се оценява според бързината на алгоритъма:
— алгоритъм със сложност $O(m + n)$ носи 20 точки;
— алгоритъм със сложност $O((m + n) \log (m + n))$ се оценява с 10 точки;
— по-бавен алгоритъм не носи точки.

Задача 3 съдържа 60 точки, разпределени по следния начин:
а) 20 точки, от които 15 точки за алгоритъма и 5 точки за неговия анализ;
б) 20 точки, от които 15 точки за алгоритъма и 5 точки за неговия анализ;
в) 20 точки, от които 10 точки за описание на редукцията, 5 точки за доказване на нейната коректност и 5 точки за анализ на бързината на редукцията.