КОРЕКТНОСТ И СЛОЖНОСТ НА АЛГОРИТМИ

ПРИМЕРНО КОНТРОЛНО № 1 ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ" — СУ, ФМИ (ЗА СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 1. ПОТОК, ФЕВРУАРИ – ЮНИ 2018 Г.)

Разглеждаме следната алгоритмична задача: по даден числов масив `A[1...n]` да се пресметне сборът от произведенията на всяко отрицателно с всяко четно число.

Пример: Ако `A = (4, –3, 7, –6, 8, 9)`, то алгоритъмът трябва да върне

`(–3).4 + (–3).(–6) + (–3).8 + (–6).4 + (–6).(–6) + (–6).8 = –54.`

Задачата може да се реши чрез следните два алгоритъма:

```
alg_1(A[1...n])
s ← 0
for j ← 1 to n do
  if A[j] е четно
    for k ← 1 to n do
      if A[k] < 0
        p ← A[j] × A[k]
        s ← s + p
return s
```

```
alg_2(A[1...n])
return alg_2_rec(A[1...n], 1, n, 1, n)

alg_2_rec(A[1...n], i, j, k, r)
// пресмята сбора от произведенията на
// всяко отрицателно число от A[k...r]
// с всяко четно число от A[i...j]
if (i > j) or (k > r) // празен масив
  return 0
if (i = j) and (k = r)
  if (A[j] е четно) and (A[k] < 0)
    return A[j] × A[k]
  return 0
y ← ⌊(i + j) / 2⌋
z ← y + 1
u ← ⌊(k + r) / 2⌋
w ← u + 1
s1 ← alg_2_rec(A[1...n], i, y, k, u)
s2 ← alg_2_rec(A[1...n], z, j, k, u)
s3 ← alg_2_rec(A[1...n], i, y, w, r)
s4 ← alg_2_rec(A[1...n], z, j, w, r)
return s1 + s2 + s3 + s4
```

Задача 1. Докажете коректността на всеки от двата алгоритъма **(2 × 10 т. = 20 точки)**
При вложените цикли е достатъчно да използвате инварианта само за външния цикъл.

Задача 2. Намерете сложността на алгоритмите по време и по памет в най-лошия случай. Посочете най-лошия случай за всяка от четирите подзадачи. **(4 × 10 т. = 40 точки)**

Задача 3. Предложете възможно най-бърз алгоритъм за тази задача. Опишете го с думи и с псевдокод. Демонстрирайте работата му с масива от примера по-горе. **(20 точки)**
Намерете времевата сложност на предложения алгоритъм в най-лошия случай. **(10 точки)**
Сравнете го по бързодействие с алгоритмите `alg_1` и `alg_2`. **(5 точки)**
Обяснете защо не съществува по-бърз алгоритъм (поне по порядък). **(5 точки)**

РЕШЕНИЯ

Задача 1. Коректността на `alg_1` се доказва чрез инварианта на външния цикъл: **?**

Всеки път, когато се извършва проверката за край на цикъла $j \le n$, е в сила следното:
`s` = сбора от произведенията на всяко четно число от подмасива `A[1...j-1]`
с всяко отрицателно число от масива `A[1...n]`. **(5 точки)**

Доказателство на инвариантата: с индукция по броя на итерациите на външния цикъл.

База: Първата проверка е при влизане във външния цикъл. Тогава инвариантата се свежда до вярното равенство 0 = 0. Лявата страна е нулата от инициализацията на променливата `s`, а дясната страна е стойността на празния сбор. **(1 точка)**

Индуктивна стъпка: Ако инвариантата е в сила при някоя проверка, която не е последна, то тя е в сила и при следващата проверка, защото новите събираеми (ако има такива) се натрупват в променливата `s` с помощта на вътрешния цикъл. **(1 точка)**

Алгоритъмът все някога ще завърши, защото циклите са по брояч, тоест горна граница за броя на итерациите се знае отнапред: вътрешните оператори (събирането и умножението) се изпълняват не повече от n^2 пъти. **(1 точка)**

При последната проверка за край на външния цикъл е в сила равенството $j = n + 1$. От инвариантата следва, че в този миг е изпълнено следното равенство:
`s` = сбора от произведенията на всяко четно число от масива `A[1...n]`
с всяко отрицателно число от масива `A[1...n]`. **(1 точка)**

Алгоритъмът връща тази стойност на `s`, тоест връща сбора от произведенията на всяко четно число с всяко отрицателно число от масива `A[1...n]`. **(1 точка)**
Точно това се иска в алгоритмичната задача, следователно алгоритъмът `alg_1` е коректен.

Коректността на алгоритъма `alg_2` се доказва с разглеждане на случаи.

Първи случай: $i > j$ или $k > r$. Следователно поне единият подмасив е празен, затова е празен и търсеният сбор, така че алгоритъмът правилно връща нула. **(1 точка)**

Втори случай: $i \le j$ и $k \le r$. Тогава използваме силна индукция по $j - i + r - k$ (това е сборът от дължините на подмасивите минус две). Логично е да се използва този параметър, тъй като алгоритъмът разделя големите подмасиви на по-малки. **(2 точки)**

База: при $j - i + r - k = 0$. Тогава $i = j$ и $k = r$, тоест двата подмасива имат по един елемент. Ако елементите отговарят на изискванията, алгоритъмът връща произведението им. В противен случай алгоритъмът връща нула, което е правилно, тъй като сборът е празен. **(1 точка)**

Индуктивна стъпка: Алгоритъмът разбива всеки от двата големи подмасива на два по-малки и намира поотделно четирите сбора, съответни на възможностите за всяко произведение. (За всеки множител има две възможности според това, в кой по-малък подмасив се пада. За цялото произведение има $2 \times 2 = 4$ възможности.) Че четирите рекурсивни извиквания работят правилно, следва от индуктивното предположение. **(2 точки)**

Алгоритъмът завършва, тъй като не съдържа цикли и дължината на масива намалява двойно при всяко рекурсивно извикване, затова достига 1 (дъното на рекурсията). **(2 точки)**

Резултатът е верен: сумата от четирите сбора съдържа всички произведения. **(2 точки)**

Задача 2. Анализ на сложността на двата дадени алгоритъма:

— Сложността по време на `alg_1` е $T_1(n) = \Theta(n^2)$, тъй като, от една страна, най-вътрешните оператори се изпълняват не повече от n^2 пъти, **(5 точки)**
а от друга страна, се изпълняват точно n^2 пъти в най-лошия случай — когато всички числа са и четни, и отрицателни. Случаят, когато всички числа от масива са четни, но не всички са отрицателни, е също толкова лош по порядък, въпреки че изпълнението може да е няколко пъти по-бързо: въпреки че умножението и събирането не се изпълняват, проверката за отрицателност се изпълнява n^2 пъти, а това е достатъчно за порядъка. **(5 точки)**

— Сложността по памет на `alg_1` е $M_1(n) = \Theta(1)$, понеже алгоритъмът използва фиксиран брой (две) променливи от примитивен (целочислен) тип: `s` и `p`. **(5 точки)**
Тук всички входни данни са еднакво лоши. **(5 точки)**

— Сложността по време на `alg_2` е $T_2(n) = \Theta(n^2)$. Това следва с помощта на първия случай на мастър-теоремата, приложена към рекурентното уравнение

$$T_2(n) = 4T_2\left(\frac{n}{2}\right) + \Theta(1).$$

Делението на 2 идва от разделянето на масивите на две части с (почти) равни дължини. Множителят 4 се дължи на четирите рекурсивни извиквания. Свободният член представлява времето за всички останали оператори на алгоритъма (без рекурсията). То е ограничено, защото алгоритъмът съдържа само разклонения, тоест няма цикли. **(5 точки)**
И тук най-лошият случай е, когато всички числа в масива `A` са и четни, и отрицателни: тогава се изпълняват най-много умножения. Но ако се интересуваме само от порядъка, то всички входни данни са еднакво лоши: при всякакви числа в масива `A` функцията извиква себе си четири пъти, докато останат масиви с не повече от един елемент. **(5 точки)**

— Сложността по памет на `alg_2` е $M_2(n) = \Theta(\log n)$, понеже при последователно разбиване на масива на две части рекурсията има $\log_2 n$ равнища, всяко от които изисква количество памет $\Theta(1)$ за локалните променливи (фиксиран брой от примитивен тип). Рекурсията е неопашкова, затова всяко равнище използва собствена памет. **(5 точки)**
Тук всички входни данни са еднакво лоши. **(5 точки)**

Задача 3. Сборът от произведенията на всяко отрицателно с всяко четно число е равен на произведението на сбора от отрицателните числа със сбора от четните числа. **(5 точки)**

Пример: Ако `A` = (4, –3, 7, –6, 8, 9), то търсеното число е
[(–3) + (–6)] . [4 + (–6) + 8] = (–9) . 6 = –54. **(5 точки)**

Описание на алгоритъма с помощта на псевдокод:

```
alg_3(A[1...n])
s1 ← 0
s2 ← 0
for k ← 1 to n do
  if A[k] е четно
    s1 ← s1 + A[k]
  if A[k] < 0
    s2 ← s2 + A[k]
return s1 × s2
```

(10 точки)

Понеже тялото на цикъла се изпълнява n пъти, то времевата сложност на `alg_3` е $T_3(n) = \Theta(n)$ в най-лошия случай — когато всички числа в масива са и четни, и отрицателни; тогава се извършват максимален брой събирания. Обаче по порядък всички входни данни са еднакво лоши: времето за работа на `alg_3` винаги е $\Theta(n)$, защото, каквито и да са числата в масива `A`, проверките за четност и отрицателност се изпълняват точно по n пъти всяка. **(10 точки)**

Алгоритъмът `alg_3` е по-бърз по порядък от алгоритмите `alg_1` и `alg_2`, защото

$$n = o(n^2).$$

Това се доказва чрез граничен преход:

$$\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0.$$

(5 точки)

Не съществува алгоритъм, който да е по-бърз по порядък от алгоритъма `alg_3`, т.е. не е възможно да решим алгоритмичната задача за време $o(n)$. Причината е, че всеки алгоритъм трябва да прочете всички числа от масива $A[1...n]$, а за тази цел е нужно време $\Theta(n)$. Това ограничение е известно като тривиална долна граница по размера на входа: времевата сложност на всеки алгоритъм с неструктуриран вход е поне линейна. (Двоичното търсене има сублинейна сложност по време, но то работи със структурирани входни данни — сортиран масив, — затова не е нужно да се четат всички числа от входа.)

Възможно е да съществува алгоритъм, по-бърз от `alg_3` няколко пъти, но не и по порядък. Няма да изследваме този въпрос, тъй като се интересуваме само от порядъка на сложността. **(5 точки)**