

Зад. 1 Нека f е асимптотично положителна функция. Докажете или опровергайте, че

$$o(f) = O(f) \setminus \Theta(f)$$

Решение: Твърдението не е вярно. Нека

$$f(x) = 2^{2^{\lfloor x \rfloor}} \text{ за всяко положително реално } x$$

$$g(x) = 2^{2^{\lfloor x \rfloor}} \text{ за всяко положително реално } x$$

В сила са следните твърдения.

- $g(x) = O(f(x))$, понеже съществува положителна константа c , такава че $2^{2^{\lfloor x \rfloor}} \leq c \cdot 2^{2^{\lfloor x \rfloor}}$ за всички достатъчно големи стойности на x . Примерно, $c = 1$.
- $g(x) \neq \Theta(f(x))$, понеже $(g(x))^2 = f(x)$ за не-цели стойности на x .
- $g(x) \neq o(f(x))$, защото за всяко положително x_1 съществува $x_2 > x_1$, такава че $g(x_2) = f(x_2)$; а именно, x_2 е цяло число, по-голямо от x_1 .

Тогава $g \in O(f) \setminus \Theta(f)$, но $g \notin o(f)$. Тогава множествата $o(f)$ и $O(f) \setminus \Theta(f)$ не са равни.

Зад. 2 Даден е масив $A[1..n]$ от числа, които са две по две различни. Известно е, че за някакво k , всеки елемент на A е на разстояние не по-голямо от k от позицията, на която би бил, ако A бъде сортиран.

Предложете колкото е възможно по-ефикасен като сложност по време, в асимптотичния смисъл, алгоритъм, който сортира A . Точки се дават само на алгоритми с оптимална, в асимптотичния смисъл, сложност по време.

Докажете формално и прецизно коректността на предложението от Вас алгоритъм.

Изследвайте сложността му по време.

Решение: Следният алгоритъм решава задачата.

Почти СОРТИРАН($A[1..n]$: числа; k : цяло положително число)

- 1 (* Всеки елемент на A е на разстояние $\leq k$ от позицията си в сортирания масив *)
- 2 Увеличаваме A с $k + 1$ нови клетки $A[n + 1], \dots, A[n + k + 1]$
- 3 На всяка нова клетка на A присвояваме ∞
- 4 Построяваме пирамида min тип $S[1..k + 1]$ от елементите на $A[1..k + 1]$
- 5 Построяваме масив $C[1..n]$
- 6 **for** $i \leftarrow 1$ **to** n
- 7 $C[i] \leftarrow \text{EXTRACT-MIN}(S)$
- 8 $\text{INSERT}(S, A[k + i + 1])$
- 9 **return** C

Добавените към A елементи ∞ играят ролята на сентинели – алгоритъмът е по-прост с тях, а асимптотичната сложност не се влошава.

Очевидно $0 \leq k \leq n - 1$.

Инвариант за цикъла е следното твърдение: при всяко достигане на ред 6,

1. $C[1..i - 1]$ съдържа $i - 1$ на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите на входа, които са били в S и са били извадени от S ,
2. S е пирамида от тип min ,
3. i -ият по големина елемент на входа е минималният елемент в S ,
4. елементите $A[1], A[2], \dots, A[k + i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S .

Базата е $i = 1$. (1) е вярно в празния смисъл, понеже $i - 1$ е нула. (2) е вярно заради действието, което се върши на ред 4. (3) е вярно, понеже най-малкият елемент на входа се намира във входния $A[1..k + 1]$, а ние построяваме пирамидата на ред 4 именно от елементите на входния $A[1..k + 1]$. (4) е вярно, понеже става дума за $A[1], A[2], \dots, A[k + i]$ от входа: на ред 4 точно тези елементи влизат в S , а в този момент извадени елементи от пирамидата няма.

Да допуснем, че е инвариантът е в сила за някое достигане на ред 6, което не е последното. Нека това е момента t от изпълнението, а t' е моментът на следващото достигане на ред 6. Ще докажем, че инвариантът е в сила при следващото достигане на ред 6. Инвариантът е конюнкция от четири съждения и ще ги докажем поотделно.

1. Ще докажем (1). По допускане, в момента t е вярно, че $C[1..i - 1]$ съдържа $i - 1$ на брой минимални елементи на входния A , в сортиран вид и това са точно извадените от S елементи до момента, а i -ият най-малък елемент на входа е в S . На ред 7, функцията EXTRACT-MIN премахва същия този елемент от пирамидата и го слага в C на позиция i . Вече е вярно, че $C[1..i]$ съдържа i на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите, извадени от S .

Функцията INSERT не променя C .

При следващото достигане на ред 6, i се инкрементира. Спрямо новото i отново е вярно, че $S[1..i-1]$ съдържа $i-1$ на брой минимални елементи на входния A , в сортиран вид, и това са точно елементите, извадени от S .

2. Ще докажем (2). Ако S е пирамида в момента t , тя остава пирамида след изпълненията на $\text{EXTRACT-MIN}(S)$ на ред 7 и $\text{INSERT}(S, A[k+i+1])$ на ред 8, защото EXTRACT-MAX и INSERT запазват “пирамидалността”, както знаем от лекции.
3. Ще докажем (3). Очевидно е, че $(i+1)$ -ият по големина елемент на входния масив не може да се намира на позиция, по-голяма от $k+i+1$, във входа, заради ограничението всеки елемент във входа да е на разстояние $\leq k$ от мястото си в сортирания масив. Но тогава $(i+1)$ -вият по големина елемент е или един от елементите $A[1], A[2], \dots, A[k+i]$, за които говори (4) от инварианта, или е $A[k+i+1]$. Съгласно (4), $A[1], A[2], \dots, A[k+i]$ от входа са точно елементите, които или са били в S и извадени от S преди момента t , или са в S в момента t . Съгласно (1), най-малките $i-1$ на брой елементи вече не се намират в S в момента t . Заключаваме, че $(i+1)$ -ият по големина елемент или се намира в S , или е $A[k+i+1]$. След изпълнението на $\text{INSERT}(S, A[k+i+1])$ на ред 8, $(i+1)$ -ият по големина елемент се намира в S . След инкрементирането на i на следващото достигане на ред 6, спрямо новата стойност на i е вярно, че i -ият по големина елемент се намира в S ; това е в момента t' .

Вече видяхме, че в момента t' , $i-1$ на брой минимални елементи на входния масив са били в S , но извадени от S . Заключаваме, че i -ият по големина елемент не просто е в S , а е най-малкият елемент в S .

4. Ще докажем (4). Това е елементарно: по допускане, елементите $A[1], A[2], \dots, A[k+i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S , в момента t . След изпълнението на $\text{INSERT}(S, A[k+i+1])$ на ред 8, $(i+1)$ -ият по големина елемент се намира в S . Имаме право да кажем, че вече елементите $A[1], A[2], \dots, A[k+i], A[k+i+1]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S .

След инкрементирането на i на следващото достигане на ред 6, спрямо новата стойност на i е вярно, че елементите $A[1], A[2], \dots, A[k+i]$ на входния масив са точно елементите, които в момента са в S или са били в S , но са извадени от S ; това е в момента t' .

Доказахме инварианта. При последното достигане на ред 6, $i = n+1$. От част (1) на инварианта имаме “ $S[1..n]$ съдържа n на брой минимални елементи на входния A , в сортиран вид”. На ред 9 алгоритъмът връща S , така че той всъщност връща входа в сортиран вид.

Сега да изследваме сложността по време. Действието на ред 2, което се състои в увеличаване на A с $k+1$ елемента и присвояване на стойности на тези нови елементи, се извършва във време $O(n+k+1)$. Построяването на пирамидата става във време $\Theta(k+1)$, ако използваме бързия алгоритъм BUILDHEAP . Построяването на S става във време $O(n)$. Цикълът се “извърта” n пъти, а всяко негово изпълнение става във време $\Theta(\lg(k+1)) + \Theta(\lg(k+1))$ в най-лошия случай – на лекции сме доказали, че както EXTRACT-MIN , така и INSERT , работят във време, пропорционално на логаритъм от големината на пирамидата, в най-лошия случай. Тогава сложността по време, в асимптотичния смисъл, е

$$O(n+k+1) + \Theta(k+1) + O(n) + n(\Theta(\lg(k+1)) + \Theta(\lg(k+1)))$$

Предвид факта, че $k \leq n$, можем да опростим този израз до $\Theta(n \lg(k+1))$. Причината да не ползваме просто “ $\lg k$ ” е, че k може да бъде нула.

Зад. 3 Решете следното рекурентно уравнение колкото е възможно по-формално и прецизно.

$$T(n) = 8T(n-1) - 20T(n-2) + 16T(n-3) + n^{\log_2 n} \quad (1)$$

Решение: Уравнението не може да се реши с метода с характеристичното уравнение, защото нехомогенната част не е от правилния вид. Ако обаче игнорираме нехомогенната част и разгледаме хомогенното уравнение

$$T(n) = 8T(n-1) - 20T(n-2) + 16T(n-3) \quad (2)$$

веднага виждаме, че то е податливо на решаване чрез метода с характеристичното уравнение, като характеристичното уравнение е $x^3 - 8x^2 + 20x - 16$. Мултимножеството от корените е $\{2, 2, 4\}_M$, следователно $T(n) \asymp 4^n$ е решението на (2).

Допускаме, че $T(n) \asymp 4^n$ е също така и решение на (1). Причина да допуснем това е, че дори нехомогенното уравнение да беше

$$T(n) = 8T(n-1) - 20T(n-2) + 16T(n-3) + 2^n \quad (3)$$

мултимножеството от корените в крайна сметка щеше да е $\{2, 2, 2, 4\}_M$, така че решението пак щеше да е $T(n) \asymp 4^n$, а нехомогенната част на (3) расте асимптотично по-бързо от нехомогенната част на (1) – в сила е $n^{\log_2 n} < 2^n$. Последното се доказва елементарно чрез логаритмуване на двете страни на $n^{\log_2 n}$ vs 2^n , водещо $(\log_2 n)^2$ vs n , и ползване на резултата от лекции, казващ, че всяка полилогаритмично растяща функция расте асимптотично по-бавно от всяка полиномиално растяща функция.

И така, допускаме, че $T(n) \asymp 4^n$ е решението на (1). Ще докажем това по индукция. Ще докажем, че $T(n) = O(4^n)$ и $T(n) = \Omega(4^n)$. Твърдението, което ще докажем по индукция, е, че съществуват положителни константи c , d и k , такива че

$$T(n) \leq c4^n - d3^n \quad (4)$$

$$T(n) \geq k4^n \quad (5)$$

за всички достатъчно големи стойности на n . Тук използваме засилване на твърдението в (4) по технически причини. Доказателството е със силна индукция. Допускаме, че

$$T(n-1) \leq c4^{n-1} - d3^{n-1} \quad (6)$$

$$T(n-2) \geq k4^{n-2} \quad (7)$$

$$T(n-3) \leq c4^{n-3} - d3^{n-3} \quad (8)$$

$$T(n-1) \geq k4^{n-1} \quad (9)$$

$$T(n-2) \leq c4^{n-2} - d3^{n-2} \quad (10)$$

$$T(n-3) \geq k4^{n-3} \quad (11)$$

Тогава

$$8T(n-1) \leq 8c4^{n-1} - 8d3^{n-1} \quad (12)$$

$$-20T(n-2) \leq -20k4^{n-2} \quad (13)$$

$$16T(n-3) \leq 16c4^{n-3} - 16d3^{n-3} \quad (14)$$

$$8T(n-1) \geq 8k4^{n-1} \quad (15)$$

$$-20T(n-2) \geq -20c4^{n-2} + 20d3^{n-2} \quad (16)$$

$$16T(n-3) \geq 16k4^{n-3} \quad (17)$$

От (12), (13), (14), (15), (16), (17) и (1) заключаваме, че

$$T(n) \leq 8c4^{n-1} - 8d3^{n-1} - 20k4^{n-2} + 16c4^{n-3} - 16d3^{n-3} + n^{\log_2 n}$$

$$T(n) \geq 8k4^{n-1} - 20c4^{n-2} + 20d3^{n-2} + 16k4^{n-3} + n^{\log_2 n}$$

Остава да покажем, че съществуват положителни c , d и k , такива че за всички достатъчно големи n :

$$\begin{aligned} 8c4^{n-1} - 8d3^{n-1} - 20k4^{n-2} + 16c4^{n-3} - 16d3^{n-3} + n^{\log_2 n} &\leq c4^n - d3^n \\ 8k4^{n-1} - 20c4^{n-2} + 20d3^{n-2} + 16k4^{n-3} + n^{\log_2 n} &\geq k4^n \end{aligned}$$

Но тази система е еквивалентна на

$$\begin{aligned} 2c4^n - 8d3^{n-1} - 5k4^{n-1} + c4^{n-1} - 16d3^{n-3} + n^{\log_2 n} &\leq c4^n - d3^n \\ 2k4^n - 5c4^{n-1} + 20d3^{n-2} + k4^{n-1} + n^{\log_2 n} &\geq k4^n \end{aligned}$$

която е еквивалентна на

$$\begin{aligned} c4^n - 8d3^{n-1} - 5k4^{n-1} + c4^{n-1} - 15d3^{n-3} + n^{\log_2 n} &\leq 0 \\ k4^n - 5c4^{n-1} + 20d3^{n-2} + k4^{n-1} + n^{\log_2 n} &\geq 0 \end{aligned}$$

която е еквивалентна на

$$\begin{aligned} 4^{n-1}(4c - 5k + c) - 8d3^{n-1} - 15d3^{n-3} + n^{\log_2 n} &\leq 0 \\ 4^{n-1}(4k - 5c + k) + 20d3^{n-2} + n^{\log_2 n} &\geq 0 \end{aligned}$$

която е еквивалентна на

$$\begin{aligned} 5 \cdot 4^{n-1}(c - k) - 8d3^{n-1} - 15d3^{n-3} + n^{\log_2 n} &\leq 0 \\ 5 \cdot 4^{n-1}(k - c) + 20d3^{n-2} + n^{\log_2 n} &\geq 0 \end{aligned}$$

Ако $k = c$, това става

$$\begin{aligned} -8d \cdot 3^{n-1} - 15d \cdot 3^{n-3} + n^{\log_2 n} &\leq 0 \\ 20d3^{n-2} + n^{\log_2 n} &\geq 0 \end{aligned}$$

което е изпълнено за $d = 1$ и за всички достатъчно големи n .

Зад. 4 Напишете алгоритъма PARTITION-LOMUTO, изучаван на лекции, обяснете какво прави и докажете това строго формално чрез инвариант на цикъла.

Решение: Това е правено на лекции.

Зад. 5 Даден е полином

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Полиномът е представен чрез своите коефициенти като числен масив $A[0..n]$, като a_i е $A[i]$ за $0 \leq i \leq n$. Предложете колкото може по-ефикасен алгоритъм, който при даден полином p , представен чрез $A[0..n]$, и число x , изчислява стойността на $p(x)$. Докажете формално коректността на алгоритъма. Намерете сложността му по време. Точки се дават само за решения-алгоритми с оптимална, в асимптотичния смисъл, сложност по време.

Решение: Наивният алгоритъм за оценяване на полином за дадена стойност на променливата x е

EVAL-POLY-SLOW($A[0..n]$: коефициентите на полинома; x : число)

```
1 res ← 0
2 for k ← 0 to n
3     tmp ← 1
4     for i ← 1 to k
5         tmp ← tmp * x
6     res ← res + A[k] * tmp
7 return res
```

Той е прекалено бавен – сложността му очевидно е $\Theta(n^2)$ заради вложените цикли.

В тази задача става дума за известния алгоритъм на Хорнер за оценяване на полином върху числена стойност на променливата.

EVAL-POLY-HORNER($A[0..n]$: коефициентите на полинома; x : число)

```
1 res ← 0
2 for k ← n downto 0
3     res ← A[k] + x * res
4 return res
```

Интуитивно, коректността му се обосновава с това, че полиномът може да се представи така:

$$p(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-2} + x(a_{n-1} + xa_n))\dots))$$

Ще докажем формално коректността. Инвариант на цикъла е следното твърдение: при всяко достигане на ред 2 на EVAL-POLY-HORNER,

$$\text{res} = \sum_{j=0}^{n-(k+1)} A[j + (k + 1)] \cdot x^j \tag{18}$$

Базовият случай е първото достигане на ред 2. Тогава, от една страна $\text{res} = 0$ заради присвояването на ред 1, а от друга страна, $k = n$, така че сумата в (18) е

$$\sum_{j=0}^{n-(n+1)} A[j + (n + 1)] \cdot x^j = \sum_{j=0}^{-1} A[j + (n + 1)] \cdot x^j = 0$$

Да допуснем, че (18) е в сила при някое достигане на ред 2, което не е последното. След присвояването

на ред 3 вече е вярно, че

$$\begin{aligned}
 \text{res} &= A[k] + x \cdot \left(\sum_{j=0}^{n-(k+1)} A[j + (k + 1)] \cdot x^j \right) \\
 &= A[k] + \sum_{j=0}^{n-k-1} A[(j + 1) + k] \cdot x^{j+1} \\
 &= A[k] + \sum_{0 \leq j \leq n-k-1} A[(j + 1) + k] \cdot x^{j+1} \\
 &= A[k] + \sum_{1 \leq j+1 \leq n-k} A[(j + 1) + k] \cdot x^{j+1} \\
 &= A[k] + \sum_{1 \leq j \leq n-k} A[j + k] \cdot x^j \\
 &= A[k] + \sum_{j=1}^{n-k} A[j + k] \cdot x^j \\
 &= A[0 + k] \cdot x^0 + \sum_{j=1}^{n-k} A[j + k] \cdot x^j \\
 &= \sum_{j=0}^{n-k} A[j + k] \cdot x^j
 \end{aligned}$$

След това k се декрементира на ред 2. Изразено чрез новото k , имаме

$$\text{res} = \sum_{j=0}^{n-(k+1)} A[j + (k + 1)] \cdot x^j$$

Доказахме инварианта. Да видим терминацията. Когато изпълнението е на ред 2 за последен път, $k = -1$. Замествайки в инварианта, получаваме

$$\text{res} = \sum_{j=0}^n A[j] \cdot x^j$$

което е точно стойността на полинома, оценен в x . И на ред 4 алгоритъмът връща тази стойност.

Сложността по време е $\Theta(n)$, понеже тялото на цикъла се изпълнява $n + 1$ пъти и всяко изпълнение се извършва във време $\Theta(1)$.