Примерни решения на контролното по ДАА на КН2 от 21.04.2018 г.

Зад. 1 Разгледайте следната изчислителна задача.

Изчислителна задача Пирамида-в-Сортировка
Общ пример: Двоична макс-пирамида (max binary heap) $A[1 \ldots n]$
Решение: сортирана редица от числата, съдържащи се в A

Предложете алгоритъм за Пирамида-в-Сортировка, работещ във време $O(n)$ и базиран на директни сравнения, или докажете, че такъв алгоритъм не съществува.

Решение: Такъв алгоритъм не съществува. Тъй като можем да построим пирамида от всеки масив в $O(n)$ с директни сравнения, ако можехме да сортираме пирамида в $O(n)$ с директни сравнения, то щяхме да можем да сортираме всеки масив в $O(n)$ с директни сравнения – а знаем, че това е невъзможно.

Зад. 2 Даден е масив от цели числа $A[1, 2, \ldots, 2n]$ за някое $n \geq 1$, такъв че:

$$\forall i \in \{1, 2, \cdots, n-1\}\big(\forall j \in \{i+1, \ldots, 2n-i\}(A[i] \leq A[j] \leq A[2n+1-i]) \;\; \textbf{или} \\ \forall j \in \{i+1, \ldots, 2n-i\}(A[i] \geq A[j] \geq A[2n+1-i]\big)$$

10 т. • Напишете итеративен алгоритъм със сложност по време $O(n)$ и сложност по памет $O(1)$, който сортира масива A. Обосновете кратко сложностите по време и памет.

10 т. • Докажете коректността на предложения от Вас алгоритъм, използвайки инварианта на цикъла.

Решение

Alg1($A[1, 2, \ldots, 2n]$)
1 $\ell \leftarrow 1, h \leftarrow 2n$
2 **while** $\ell < h$ **do**
3   **if** $A[\ell] > A[h]$
4   swap($A[\ell], A[h]$)
5     $\ell{+}{+}, h{-}{-}$

Алгоритъмът има линейна сложност по време, защото цикълът се изпълнява $O(n)$ пъти и всяко негово изпълнение отнема $O(1)$ време. По-точно казано, ред 2 се достига $n + 1$ пъти, защото управляващото условие е $h - l > 0$; в началото $h - l = 2n - 1$ и след това тази разлика намалява с 2, докато не стане -1. Алгоритъмът е in-place, защото ползва само допълнителните променливи ℓ и h.

Инвариантата е:

> При всяко достигане на ред 2, $h = 2n + 1 - \ell$, подмасивът $A[1, \ldots, \ell - 1]$ съдържа $\ell - 1$ най-малки елемента от входа в сортиран вид, а масивът $A[h + 1, \ldots, 2n]$ съдържа $2n - h$ най-големи елемента от входа в сортиран вид.

При първото достигане на ред 2, $\ell = 1$ и $h = 2n$. Твърдението е "$2n = 2n+1-1$, подмасивът $A[1, \ldots, 0]$ съдържа 0 най-малки елемента от входа в сортиран вид, а масивът $A[2n + 1, \ldots, 2n]$ съдържа 0 най-големи елемента от входа в сортиран вид", което е вярно. ✓
Да допуснем, че твърдението е вярно за някое достигане на ред 2, което не е последното. Съгласно първото съждение от инвариантата, $h = 2n + 1 - \ell$. Тогава по условие имаме:

$$\forall j \in \{\ell + 1, \ldots, h - 1\}(A[\ell] \leq A[j] \leq A[h]) \;\; \textbf{или}$$
$$\forall j \in \{\ell + 1, \ldots, h - 1\}(A[\ell] \geq A[j] \geq A[h])$$

Разглеждаме двете възможности поотделно:

1. $\forall j \in \{\ell+1, \ldots, h-1\}(A[\ell] \leq A[j] \leq A[h])$. В този случай условието на **if**-а е лъжа и ред 4 не се изпълнява. От транзитивността на релацията $\leq$ и от индуктивното предположение следва, че подмасивът $A[1, \ldots, \ell]$ съдържа ℓ най-малки елемента от входа в сортиран вид, а масивът $A[h, \ldots, 2n]$ съдържа $2n-h+1$ най-големи елемента от входа в сортиран вид. След изпълнението на ред 5 отново е вярно, че подмасивът $A[1, \ldots, \ell-1]$ съдържа $\ell-1$ най-малки елемента от входа в сортиран вид, а масивът $A[h+1, \ldots, 2n]$ съдържа $2n-h$ най-големи елемента от входа в сортиран вид. ✓

2. $\forall j \in \{\ell+1, \ldots, h-1\}(A[\ell] \geq A[j] \geq A[h])$. В този случай условието на **if**-а е истина и ред 4 се изпълнява. След това, от транзитивността на релацията $\leq$ и от индуктивното предположение следва, че подмасивът $A[1, \ldots, \ell]$ съдържа ℓ най-малки елемента от входа в сортиран вид, а масивът $A[h, \ldots, 2n]$ съдържа $2n-h+1$ най-големи елемента от входа в сортиран вид. След изпълнението на ред 5 отново е вярно, че подмасивът $A[1, \ldots, \ell-1]$ съдържа $\ell-1$ най-малки елемента от входа в сортиран вид, а масивът $A[h+1, \ldots, 2n]$ съдържа $2n-h$ най-големи елемента от входа в сортиран вид. ✓

При терминирането, $h = n$ и $\ell = n+1$. Замествайки в инвариантата, получаваме "подмасивът $A[1, \ldots, n]$ съдържа n най-малки елемента от входа в сортиран вид, а масивът $A[n+1, \ldots, 2n]$ съдържа n най-големи елемента от входа в сортиран вид", което веднага влече, че масивът е сортиран. ✓

Зад. 3 Намерете и обосновете накратко сложността по време на алгоритъма AlgX:

AlgX($A[0, \ldots, 2n-1]$: цели числа, точно половината от тях положителни, $n \geq 1$)

```
k ← 0
for i ← 0 to 2n − 1
    a ← 0, b ← 0
    for j ← 0 to 2n − 1
        if A[(i + j) mod 2n] > 0
            a++
        else
            a−−
        if a < 0
            b++
    if b = 0
        k++
if k ≠ 0
    k ← 1
return AlgY(2n, k + 1)
```

където AlgY е следният рекурсивен алгоритъм:

AlgY(m, k: цели числа)

```
if m = 0
    return 1
else
    s ← 1
    for i ← 1 to k
        s ← s + AlgY(m − 1, k)
    return s
```

Решение: За по-нагледно обяснение, нека положителните числа в масива са белите, а останалите са черните. Двойният цикъл има следното действие. Да разглеждаме масива като цикличен. Прави се точно една пълна обиколка, започвайки от всяка позиция, като чрез променливата a се поддържа разликата между видяните до момента бели и черни числа. При една пълна обиколка, променливата b остава нула тстк във всеки момент, броят на видяните бели числа е не по-малък от броя на видяните черни числа. В края на всяка обиколка, k бива инкрементирано тстк по време на обиколката, във всеки момент броят на видяните бели числа е не по-малък от броя на видяните черни числа. След приключване на обиколките, ако поне при една обиколка, във всеки момент броят на видяните бели числа е бил не по-малък от броя на видяните черни числа, то $k > 0$, в противен случай k е останал 0. И така, присвояването на ред 14 се случва тстк поне при една обиколка, във всеки момент броят на видяните бели числа е бил не по-малък от броя на видяните черни числа.

Математически факт е, че при произволно кръгово разполагане на еднакъв брой бели и черни обекта в кръгова наредба съществува такъв бял обект, че при пълна обиколка (няма значение по посока или обратно на часовниковата стрелка), във всеки момент броят на видяните до този момент бели неща е поне колкото броят на видяните до този момент черни неща. Това се доказва тривиално по индукция, ако забележим, че която и посока да си изберем за обикаляне, има една двойка непосредствени съседи, първият от които е бял, а вторият, черен. Махайки тази двойка, попадаме в индуктивната хипотеза, а връщайки двойката, очевидно твърдението се запазва.

И така, ред 14 непременно се изпълнява, поради което ALGY бива викан с втори аргумент 2. Тогава броят на рекурсивните викания винаги е 2 чак до стигане на дъното на рекурсията. Тогава рекурентното уравнение, описващо сложността на ALGY, е $T(m) = 2T(m-1) + 1$. Това се решава наум с метода с характеристичното уравнение, като решението е $T(m) \asymp 2^m$. Тогава само ред 15 се изпълнява в $\Theta(2^{2n})$, тоест $\Theta(4^n)$. Това доминира над сложността на останалата част, която сложност е $\Theta(n^2)$. Тогава сложността на целия алгоритъм е $\Theta(4^n)$.

Зад. 4 Дадени са два масива $A[1, \ldots, n]$ и $B[1, \ldots, n+1]$. Масивът A е сортиран и съдържа само цели положителни числа. Масивът B се получава от A чрез вмъкване на точно една нула на някаква позиция $t \in \{1, \ldots, n\}$ в A и отместване с една позиция нагоре на елементите от A, които са от позиция t включително нататък. Ясно е, че B съдържа точно същите положителни числа като A и те са в точно същата наредба (сортирана), в каквато са в A. Ето пример за такива масиви:

$$A = [2, 3, 6, 12, 13, 25, 47]$$
$$B = [2, 3, 6, 12, 0, 13, 25, 47]$$

В примера, $n = 7$ и $t = 5$. Предложете колкото е възможно по-ефикасен алгоритъм, който има вход A и B и който връща t, тоест позицията, на която е вмъкната нулата. Обосновете кратко, но съдържателно коректността и сложността на предложения от Вас алгоритъм.

Решение: Задачата се решава в $\Theta(\lg n)$ с двоично търсене. За всяко k, такова че $1 \leq k \leq n$, ако $A[k] = B[k]$, то търсеното k е по-голямо от t, а ако $A[k] \neq B[k]$ и $B[k] \neq 0$ (можем да кажем просто $A[k] < B[k]$), то търсеното t е по-малко от k. Имаме $t = k$ тогава и само тогава, когато $B[k] = 0$. Индексът t никога не може да е $n + 1$.

```
ALG2(A, B, n)
 1  ℓ ← 1, h ← n
 2  while True do
 3      if h = ℓ
 4          return ℓ
 5      else if h = ℓ + 1
 6          if B[ℓ] = 0
 7              return ℓ
 8          else
 9              return ℓ + 1
10      else
11          m ← ⌊(ℓ+h)/2⌋
12          if A[m] = B[m]
13              ℓ ← m + 1
14          else if A[m] < B[m]
15              h ← m − 1
16          else
17              return m
```

Доказателството за коректност се базира на горното наблюдение. Инвариантата казва, че при всяко достигане на ред 2, търсеният индекс е в интервала $[\ell, h]$. За пълно доказателство е важно наблюдението, че при произволни стартови стойности, този вид делене на интервала на два подинтервала, после те на по два подподинтервала и така нататък, ако продължи достатъчно дълго, неизбежно стига до интервали с дължини едно и две. Разбира се, това не се очаква на изпита.

Сложността по време се получава точно като при двоичното търсене: $T(n) = T\left(\frac{n}{2}\right) + 1$ с решение $T(n) \asymp \lg n$ чрез Мастър теоремата.

Зад. 5 Наредете по асимптотично нарастване следните функции. Обосновете отговорите си кратко. Напишете в явен вид наредбата. *Забележка: всички логаритми са с основа* 2.

$$f_1(n) = 2n, \quad f_2(n) = n^{n!}, \quad f_3(n) = 2^{(\lg n)^2}, \quad f_4(n) = 2^{\lg(n^2)}, \quad f_5(n) = \frac{\sqrt{n}}{\lg \lg n},$$

$$f_6(n) = \binom{2n}{2}^2, \quad f_7(n) = \lg \binom{2n}{n}, \quad f_8(n) = \sqrt[3]{n} \lg \lg n, \quad f_9(n) = n^{\lg \lg n}, \quad f_{10}(n) = n!^n$$

Решение:

(i) Ще покажем, че $f_8 \prec f_5$, използвайки наготово факта, че всяка полиномиална функция расте асимптотично по-бързо от всяка полилогаритмична функция. Наистина, $\sqrt[3]{n} \lg \lg n = n^{\frac{1}{3}} \lg \lg n \prec n^{\frac{1}{3}} n^{\frac{1}{24}} = n^{\frac{9}{24}} \prec n^{\frac{9}{24}} n^{\frac{1}{24}} \prec n^{\frac{12}{24}} n^{-\frac{1}{24}} \prec n^{\frac{1}{2}} (\lg \lg n)^{-1} = \frac{\sqrt{n}}{\lg \lg n}$.

(ii) Ще покажем, че $f_5 \prec f_1$. Но това е напълно очевидно, понеже $\sqrt{n} \prec n$, така че $\frac{\sqrt{n}}{\lg \lg n} \prec 2n$.

(iii) Ще покажем, че $f_1 \asymp f_7$. Първо да преобразуваме биномния коефициент, използвайки трикратно апроксимацията на Стирлинг:

$$\binom{2n}{n} = \frac{(2n)!}{n!n!} = \frac{\sqrt{2\pi 2n}\left(\frac{(2n)^{2n}}{e^{2n}}\right)}{\sqrt{2\pi n}\left(\frac{n^n}{e^n}\right)\sqrt{2\pi n}\left(\frac{n^n}{e^n}\right)} \asymp \frac{2^{2n}}{\sqrt{n}} = \frac{4^n}{\sqrt{n}}$$

Тогава $\lg \binom{2n}{n} \asymp \lg \left(\frac{4^n}{\sqrt{n}}\right) = n \lg 4 - \frac{1}{2} \lg n \asymp n$, така че $2n \asymp \lg \binom{2n}{n}$.

(iv) Ще покажем, че $f_1 \prec f_4$. Веднага се вижда, че

$$f_4(n) = 2^{\lg(n^2)} = n^2$$

Това, че $n \prec n^2$, е очевидно.

(v) Ще покажем, че $f_4 \prec f_6$. Помним, че $f_4(n) = n^2$. От друга страна,

$$f_6(n) = \binom{2n}{2}^2 = \left(\frac{2n(2n-1)}{2}\right)^2 \asymp n^4$$

Твърдението става очевидно.

(vi) Ще покажем, че $f_6 \prec f_9$. Помним, че $f_6(n) \asymp n^4$. От друга страна, двойният логаритъм е растяща функция на n, така че $n^4 \prec n^{\lg \lg n}$.

(vii) Ще покажем, че $f_9 \prec f_3$. Първо преобразуваме f_3 така:

$$f_3(n) = 2^{(\lg n)^2} = 2^{(\lg n) \cdot (\lg n)} = 2^{\lg\left(n^{\lg n}\right)} = n^{\lg n}$$

Това, че $n^{\lg \lg n} \prec n^{\lg n}$, се вижда веднага, понеже логаритъмът е растяща функция.

(viii) Ще покажем, че $f_3 \prec f_{10}$. Логаритмуваме двете функции:

$$\lg\left(n^{\lg n}\right) = (\lg n)(\lg n) = (\lg n)^2$$
$$\lg\left(n!^n\right) = n \lg n! \asymp n^2 \lg n$$

Използвахме наготово, че $\lg n! \asymp n \lg n$. Но $(\lg n)^2 \prec n^2 \lg n$, понеже всяка полиномиална функция расте по-бързо от всяка полилогаритмична функция. Заключаваме, че $n^{\lg n} \prec n!^n$.

Благодарности на Андрей Дренски за откритата и коригирана грешка в асимптотиката на f_3!

(ix) Ще покажем, че $f_{10} \prec f_2$. Вече видяхме, че $\lg f_{10}(n) \asymp n^2 \lg n$. От друга страна, $\lg \left(n^{n!}\right) \asymp n! \lg n$. Тъй като факториелът расте асимптотично по-бързо от всяка полиномиална функция, заключаваме, че $\lg f_{10}(n) \prec \lg f_2(n)$. Оттук веднага следва $f_{10} \prec f_2$.

Окончателната наредба е:

$$f_8 \prec f_5 \prec f_1 \asymp f_7 \prec f_4 \prec f_6 \prec f_9 \prec f_3 \prec f_{10} \prec f_2$$

Зад. 6 На лекции видяхме, че сложността по време в най-добрия случай не е особено смислена мярка за качеството на даден алгоритъм и поради това рядко се разглежда, но сега разглеждаме именно нея. Докажете, че Heapsort има сложност по време **в най-добрия случай** $\Theta(n \lg n)$, ако елементите от входа са уникални (няма повторения).

Решение: Тъй като елементите са различни, имаме право да ги разбием мислено на две половини: големите и малките. Ние не знаем как точно са разположени във входа малките и големите числа, но във всеки случай е вярно, че втората половина от пирамидата (това, което Build Heap конструира) съдържа линеен брой (в n) от малки числа. А втората половина от пирамидата са листата на пирамидата. И така, линеен брой листа са малки върхове. Измежду тях, линеен брой са такива, че след всеки от тях бъде сложен на първо позиция (на върха), той изминава логаритмичен път "надолу" в пирамидата, бивайки "бутан" от Heapify в посока към листата.

Защо това е така? Защото първата половина на пирамидата след построяването ѝ съдържа линеен брой големи елементи. Те образуват поддърво – очевидно коренът е голям елемент (по-точно, най-големият, но това няма значение в случая) и освен това големите индуцират свързан подграф; няма как свързани области от големи числа да бъдат разделени от малки числа, просто дефиницията на макс пирамида не позволява това. И така, извън листата има линеен броя големи върхове. Нещо повече, на логаритмична височина (височина е разстояние от корена) има линеен брой големи върхове. За всеки малък връх, който застава на върха на пирамидата е вярно, че ще изминава логаритмично дълъг път, бивайки бутан надясно от Heapify, докато има големи върхове на логаритмична височина, тоест на константна дълбочина. Това е така, защото Heapify разменя връх с **по-голямото** дете, ако изобщо разменя нещо.