Примерни решение на задачите от изпита по Дизайн и Анализ на Алгоритми, КН2, 27.06.2015 г.

Зад. 1 Даден е ориентиран граф $G = (V, E)$. Докажете или опровергайте всяко от следните две твърдения.

10 т.
- Ако произволно пускане на DFS върху G открива само едно ребро e от вида "ребро назад" (back edge), то $G' = (V, E \setminus \{e\})$ е ацикличен.

10 т.
- Ако има ребро $e \in E$, такова че $G' = (V, E \setminus \{e\})$ е ацикличен, то произволно пускане на DFS върху G ще открие точно едно ребро от вида "ребро назад" (back edge).

Решение. Първото твърдение е вярно: пускането на DFS върху G' няма да открие нито едно ребро назад, което, съгласно изучаваното на лекции е същото като да няма цикли. Второто твърдение не е вярно. Разгледайте $G = (\{u, v, x, y\}, \{(u, v), (v, x), (x, y), (y, u), (u, x), (v, y)\})$. Очевидно G е цикличен и премахването на реброто (y, u) води до ацикличен граф. Обаче пускането на DFS от връх y води до две ребра назад, а именно (v, y) и (x, y).

20 т. **Зад. 2** Професор Петров е изправен пред избор. Дадени са n различни лекции (на различни теми), които той може да води. Времето е дискретно: дадени са моменти 0, 1, $\ldots$, n, в които може да започват лекциите. Всяка лекция ℓ_i има:

- Продължителност единица,
- Краен срок на приключване $d_i \in \{1, 2, \ldots n\}$. Тоест, ако не започне до момент $d_i - 1$, няма да се проведе изобщо, но може да започне във всеки момент от 0 до $d_i - 1$ включително, стига в този момент да не започва друга лекция,
- Цена p_i лева, която се заплаща за провеждането ѝ.

Можем да мислим за времето като за слотове: първият слот е $[0, 1]$, вторият слот е $[1, 2]$, и така нататък, n-ият слот е $[n - 1, n]$. Всеки слот може да поеме не повече от една лекция и всяка лекция може да влезе в най-много един слот. Ако нямаше ограничението за крайни срокове (иначе казано, ако $\forall n : d_i = n$), професорът можеше да проведе всички n лекции, слагайки, да речем, лекция ℓ_i в слот $[i - 1, i]$ и да вземе общо $\sum_{i=1}^{n} p_i$ лева. Заради крайните срокове обаче професорът може да не може да проведе всички лекции – примерно, ако $\forall i : d_i = 1$, професорът може да проведе най-много една лекция, и ще избере тази с най-голяма печалба. Предложете алгоритъм със сложност $O(n^2)$, който изчислява кои лекции да вземе професорът, така че да максимизира печалбата си (като спазва казаните ограничения).

Входът е масив $\mathsf{Lec}[1 \ldots n]$, като $\mathsf{Lec}[i]$ е наредена двойка от заплащането за лекция i, наречено $\mathsf{Lec}[i]$.profit, и крайния срок за лекция i, наречен $\mathsf{Lec}[i]$.deadline. Изходът на алгоритъма да е масив $\mathsf{Slots}[1 \ldots n]$, всеки елемент от който е или някой уникален $\mathsf{Lec}[i]$, или `NULL`. Този масив-изход представлява слотовете от време; ако $\mathsf{Slots}[j]$ е $\mathsf{Lec}[i]$, това означава, че лекция i отива в слот j, а ако $\mathsf{Slots}[j]$ е `NULL`, това означава, че слот j не се ползва.

Бонус 1, 20 т: *обяснете в общи линии, без подробности, как може да подобрите сложността до* $O(n \lg n)$. *Ако предложите направо* $O(n \lg n)$ *алгоритъм, без първо да посочвате* $O(n^2)$ *алгоритъм, имате* ***40 точки*** *за задачата и първия бонус.*

Бонус 2, 20 т: *докажете коректността на алгоритъма си накратко.*

Решение. Следният алгоритъм решава задачата.

```
Alg1(Lec[1 ... n] of (profit : posint, deadline : posint))
 1  Sort Lec[] by profits
 2  for i ← 1 to n
 3      Slots[i] ← NULL
 4  for i ← n downto 1
 5      k ← Lec[i].deadline
 6      while Slots[k] ≠ NULL do
 7          k ← k − 1
 8      if k ≥ 1
 9          Slots[k] ← Lec[i]
10  return Slots[]
```

Сложността на алгоритъма очевидно е не по-лоша от квадратична. Алгоритъмът може да се опише на по-високо ниво с думи така: разглеждаме последователно лекциите в обратен ред на печалбите (от големи към малки) и слагаме всяка лекция в най-късния възможен слот, който е съвместим с нейния краен срок; ако такъв слот няма, не слагаме тази лекция в никой слот.

Коректност Първо забелязваме, че масивът Slots[] в края на алгоритъмът задава решение, което не нарушава ограниченията на задачата:

- Това, че всеки елемент на Slots[] е или NULL, или някой Lec[i], е изпълнено заради кода на редове 3 и 9.
- Това, че всеки Lec[i] се появява най-много веднъж в Slots[] е изпълнено заради това, че няма как при две различни итерации на **for**-цикъла, променливата i да има една и съща стойност, така че при различни достигания на ред 9, Lec[i] представлява различен елемент на Lec[].
- Никоя лекция не се назначава след крайния си срок заради присвояването на ред 5 и факта, че след това (в рамките на текущата итерация на **for**-цикъла) променливата k само намалява (ред 7).

Освен това, не може да има опит за записване в Slots[] извън рамките на масива заради проверката на ред 8 и инициализацията на k на ред 5 (по условие, Lec[i].deadline не надхвърля n).

Аргументацията, че решението освен това винаги е оптимално, е по-трудна. Да допуснем, че Alg1 греши върху поне един вход. Разглеждаме това i, за което алгоритъмът греши за първи път – такова i трябва да има, иначе алгоритъмът е коректен върху този вход! Съобразяваме, че алгоритъмът записва стойност на ред 9 само ако Slots[k] е бил NULL преди записването – това е вярно заради **while**-цикъла, след неговото приключване или Slots[k] е NULL, или k е нула. Тогава Alg1, за някаква стойност на i, такава че $1 \leq i \leq n$, записва на ред 9 Lec[i] в някакво Slots[k] и то остава там до края на алгоритъма. Щом i е първата стойност на управляващия параметър на **for**-цикъла, за която алгоритъмът греши, то сложените преди това елементи в Slots[] са коректни в смисъл, че задават подмножество на някое оптимално решение. Чак след слагането на Lec[i] в Slots[k], масивът Slots[] задава лекции, които не са подмножество на никое оптимално решение. Ерго, би трябвало или Lec[i] да не се слага изобщо, или да се сложи на по-лява позиция.

Да разгледаме възможността Lec[i] да не е част от никое оптимално решение. Забелязваме, че за всяко оптимално решение, Slots[k] не е NULL, защото, ако беше, можеше да сложим там Lec[i] и да получим дори по-добро решение. Разглеждаме някое оптимално решение OPT. В него, както казахме, Slots[k] е някое Lec[i′], където i' е различно от i. Нещо повече, i' е <u>по-малко</u> от i. Но тогава $\text{Lec}[i'].\text{profit} \leq \text{Lec}[i].\text{profit}$ заради начина, по който работи Alg1, който "разглежда" лекциите в ненарастващ порядък на печалбите от тях. Тогава, ако заменим в OPT Slots[k] с Lec[i], получаваме решение, което е не по-лошо или дори по-добро. Това противоречи на допускането, че алгоритъмът е сбъркал, слагайки Lec[i].

А дали може грешката да не е била, че Lec[i] е сложен в решението, а че е сложен именно на позиция k? Да допуснем, че е така. Тогава има оптимално решение OPT, което съдържа Lec[i], но Lec[i] е в Slots[k′] за някакво $k' < k$. Пак забелязваме, че Slots[k] не е празен – ако беше празен, можеше да преместим Lec[i] там и щяхме да имаме решение със същата печалба, ненарушаващо условията, което противоречи на допускането, че сме сбъркали с Lec[i]. Значи, Slots[k] съдържа някакво $\mathsf{Lec}[i']$, като $\mathsf{Lec}[i']$.deadline не е по-голям от k. Тогава можем да разменим местата на $\mathsf{Lec}[i]$ и $\mathsf{Lec}[i']$ в $\mathsf{Slots}[]$, получавайки решение със същата цена, което не нарушава условията. Отново получихме противоречие с допускането, че сме сбъркали с $\mathsf{Lec}[i]$.

Сложност $O(n \lg n)$ Сортирането в началото може да се направи във време $O(n \lg n)$ с, примерно, HEAPSORT. За да се постигне обща сложност $O(n \lg n)$, трябва кодът на редове 2–9 да работи във време $O(n \lg n)$. Това може да се постигне, ако можем да намираме *бързо* първото свободно място, на което да сложим $\mathsf{Lec}[i]$. Последователното търсене на това място с **while**-цикъла става недопустимо. Възможно е да ползваме структура данни, подобна на Union-Find. Да си припомним, че Union-Find поддържа разбиване на множество, като множествата са сливащи се. В нашия случай, по отношение на произволно изпълнение на **for**-цикъла, въпросните множества са следните:

- Всеки максимален по включване подмасив на $\mathsf{Slots}[\,]$, нито един елемент на който не е `NULL`, е едно множество.

- Всеки елемент на $\mathsf{Slots}[]$, който е `NULL`, е едно множество сам по себе си.

Очевидно в началото, тоест преди започването на **for**-цикъла, всеки елемент на $\mathsf{Slots}[]$ е едно самостоятелно множество. После започваме да сливаме по начин, подобен на Union-Find. Има и особености. Всяко множество си има "лидер", както е при Union-Find, но този лидер на множество съхранява някаква стойност, да я наречем `next_available`, която е индексът на първия свободен слот вляво от множеството; ако няма такъв слот, `next_available` е 0 (която не може да е индекс). На елементите, които са `NULL`, `next_available` стойността е собствената им позиция (индекс). Очевидно в началото, `next_available` на всяко (едноелементно) множество е собственият индекс. Когато започнат сливанията, за всяко множество от значение е единствено `next_available` на лидера; `next_available` на останалите елементи не се ползва. Когато присвоим стойността на k (ред 5), проверяваме дали $\mathsf{Slots}[k]$ е `NULL`.

- Ако $\mathsf{Slots}[k]$ е `NULL`, има два подслучая: или поне една от съседните позиции не е `NULL`, или и двете са `NULL`. Ако и двете са `NULL`, няма сливане: броят на множествата си остава същият, като $\mathsf{Slots}[k]$ остава едноелементно, но вече `next_available` е $k-1$ (преди беше k). В противен случай се правят сливания на две или три множества в едно.
  - Ако сливаме $\mathsf{Slots}[k]$ с множество вдясно, `next_available` на лидера става $k-1$
  - Ако сливаме $\mathsf{Slots}[k]$ с множество вляво, `next_available` на лидера остава/става `next_available` на лидера на множеството вляво.
  - Ако сливаме $\mathsf{Slots}[k]$ с множества вляво и вдясно, това може да стане на две стъпки, като `next_available` на лидера ще стане/остане `next_available` на лидера на множеството вляво.

- Ако $\mathsf{Slots}[k]$ не е `NULL`, то можем да открием лидера на множеството (към което принадлежи $\mathsf{Slots}[k]$) с не повече от $\log n$ "скока", което свойство на Union-Find структурите сме доказвали на лекции. Нека m е `next_available` стойността на лидера.
  - Ако $m = 0$, не правим нищо (прескачаме $\mathsf{Lec}[i]$).
  - В противен случай слагаме $\mathsf{Lec}[i]$ в $\mathsf{Slots}[m]$ и сливаме множеството на $\mathsf{Slots}[k]$ с това на $\mathsf{Slots}[m]$.
    - Ако $m = 1$, `next_available` на лидера става 0.
    - Ако $m > 1$ и $\mathsf{Slots}[m-1] = $ `NULL`, `next_available` на лидера става $m-1$.
    - Ако $m > 1$ и $\mathsf{Slots}[m-1] \neq$ `NULL`, сливаме множеството с множеството, на което принадлежи $\mathsf{Slots}[m-1]$. Тогава `next_available` на лидера на новото множество става `next_available` на лидера на множеството, в което беше $\mathsf{Slots}[m-1]$.

20 т. **Зад. 3** На улица X от едната страна има n къщи, номерирани от 1 до n. За всяка къща i е дадена нейната височина h_i. Освен това е дадено едно положително число h^*, което има смисъл на някаква идеална средна височина за къща. За всяка къща i, разликата $s_i = h_i - h^*$ е *излишъкът*

(излишъкът може да е и отрицателен). Предложете колкото е възможно по-ефикасен алгоритъм, който намира непрекъснат интервал от къщи на тази улица с максимален сумарен излишък. Иначе казано, който намира i и j, такива че $1 \leq i \leq j \leq n$ и $\sum_{k=i}^{j} s_i$ е максимално.

Решение. Задачата е същата като задачата, даден е масив от цели числа $A[1 \dots n]$, да се намери непразен (поне един елемент) подмасив, сумата от елементите на който е максимална. Очевидно ако всички числа са неотрицателни, отговорът е самият масив, а ако всички са отрицателни, отговорът е кой да е елемент с минимална абсолютна стойност. Интересният случай е тогава, когато има и положителни, и отрицателни числа.

Нека $i \in \{2, \dots, n\}$. Ключово наблюдение е, че ако знаем оптимално решение $A[\ell \dots r]$ за подмасива $A[1 \dots i-1]$ (където $1 \leq \ell \leq r \leq i-1$), то оптимално решение за $A[1 \dots i]$ е:

- споменатото $A[\ell \dots r]$, тоест не ползваме $A[i]$,
- или подмасивът $A[i' \dots i]$ за някое i', такова че $1 \leq i' \leq i$, тоест ползваме $A[i]$.

Използваме включващо "или", защото може двете решения да имат еднаква стойност. Това наблюдение може да се имплементира в алгоритъм, състоящ се от едно единствено сканиране на масива отляво надясно. За всяко $i > 1$, първото решение $A[\ell \dots r]$ се получава директно, ако съхраняваме стойностите ℓ и r и сумата на $A[\ell \dots r]$ от предната итерация, а второто решение $A[i' \dots i]$ се получава от $A[i' \dots i-1]$ с добавяне на $A[i]$; стойността i' и сумата на $A[i' \dots i-1]$ се съхраняват от предната итерация. В следния алгоритъм променливата sum съхранява сумата на $A[\ell \dots r]$, а променливата tmpsum, на $A[i' \dots i]$.

```
Alg2(A[1...n]: int)
if ∀i_{1≤i≤n} : (A[i] ≤ 0)
    return (i, i), такова че A[i] е максимално
sum ← 0
tmpsum ← 0
i' ← 1
for i ← 1 to n
    tmpsum ← tmpsum + A[i]
    if tmpsum > sum
        sum ← tmpsum
        ℓ ← i'
        r ← i
    if tmpsum < 0
        tmpsum ← 0
        i' ← i + 1
return (ℓ, r)
```

Очевидно сложността по време е $\Theta(n)$.

15 т. **Зад. 4** Докажете или опровергайте, че ва всеки свързан тегловен неориентиран граф $G = (V, E)$ с тегловна функция w е вярно, че за всеки два върха $u, v \in V$ е изпълнено $\mathsf{dist}_T(u, v) = \mathsf{dist}_G(u, v)$, където

- T е графът, който връща Алгоритъмът на Крускал с вход (G, w).
- $\mathsf{dist}_T(u, v)$ е най-малката претеглена дължина на път между u и v в T
- $\mathsf{dist}_G(u, v)$ е най-малката претеглена дължина на път между u и v в G
- *претеглена дължина на път* p в някакъв тегловен граф е сумата от теглата на ребрата на p.

Решение. С други думи, пита се дали МПД-то на G е също така дърво на най-късите пътища. Отговорът е, не. Като контрапример разгледайте $G = (\{u, v, x, y\}, \{(u, v), (u, x), (x, y), (y, v)\})$ и тегла $w(u, v) = 2$, $w(u, x) = w(x, y) = w(y, v) = 1$. Тоест, цикъл от четири ребра, едно от които с тегло 2, останалите с тегло 1. Минималното (то е единствено) покриващо дърво T е пътят $u-x-y-v$ със сумарно тегло 3. Алгоритъмът на Крускал ще върне T. В дървото T, най-късият претеглен път между u и v има претеглена дължина 3. В оригиналния G, най-късият претеглен път между u и v има претеглена дължина 2.

15 т. **Зад. 5** Дадена е следната изчислителна задача.

Изчислителна Задача XYZ
Общ пример: Множество U от булеви променливи, дизюнктивна нормална форма C над U
Въпрос: Съществува ли булева оценка (валюация) на променливите от U, такава че C да се оцени на TRUE? □

Дискутирайте сложността на задачата XYZ. Според Вас, тя дали принадлежи на **P**? Дали принадлежи на **NP**? Дали принадлежи на **NP**-complete?

Решение. Задачата XYZ е тривиална. Отговорът винаги е ДА, защото всяка функция, която има дизюнктивна нормална форма, е различна от функцията константа-нула (константа-нула няма дизюнктивна нормална форма – ако има поне една елементарна конюнкция, очевидно съществува поне една удовлетворяваща валюация). Следователно задачата е в **P**. Тогава тя е и в **NP**, защото $\mathbf{P} \subseteq \mathbf{NP}$.

Ако задачата XYZ е в **NP**-complete, от това веднага следва, че $\mathbf{P} = \mathbf{NP}$; знаем, че ако дори една **NP**-пълна задача е в **P**, то $\mathbf{P} = \mathbf{NP}$. Въпросът дали $\mathbf{P} = \mathbf{NP}$ или $\mathbf{P} \neq \mathbf{NP}$ е основният нерешен въпрос на теоретичната информатика, така че категоричен отговор на третия въпрос на тази задача няма.

20 т. **Зад. 6** Задачата SET COVER се дефинира така:

Изчислителна Задача SET COVER
Общ пример: Множество A, множество $B \subseteq 2^A$, число $k \in \mathbb{N}$
Въпрос: Съществува ли $C \subseteq B$, такова че $|C| \leq k$ и $\cup C = A$? □

Докажете, че SET COVER е **NP**-пълна. Не е необходимо да правите строго доказателство в двете посоки за коректността на редукцията.

Решение. Очевидно SET COVER $\in$ **NP**, тъй като за всеки ДА-пример, недетерминираната машина на Тюринг може да отгатне някакво подмножество на C на B, да провери, че наистина $|C| \leq k$, и също така, че обединението на елементите на C дава A.

Нека е даден произволен пример на VERTEX COVER. Този пример е наредена двойка $(G = (V, E), k')$, където G е граф, а k', естествено число. Ще построим пример (A, B, k) на SET COVER. Първо, $k \leftarrow k'$. Второ, $A \leftarrow E$. Трето, B се състои от $|V|$ на брой елемента $B = \{B_1, B_2, \ldots, B_{|V|}\}$, като всяко B_i съответства на точно един връх v_i от графа. Нещо повече, B_i е множеството от точно тези ребра от E, които са инцидентни с v_i.

Фактът, че $(G = (V, E), k')$ е ДА-пример на VERTEX COVER тогава и само тогава, когато (A, B, k) е ДА-пример на SET COVER, вземаме за очевиден. В условието е казано, че няма нужда това да се доказва.

Фактът, че конструкцията може да се направи в полиномиално време, е очевиден. Така че действително тук описахме полиномиална редукция.