Динамично програмиране

Зад. 1

Дадени са $n \in \mathbb{N}^+$ различни вида монети - coins[1 .. n] $\in (\mathbb{N}^+)^n$. Да се намери минималния брой монети, необходими да се получи дадена сума $S \in \mathbb{N}_0$ при условие, че:

***a*)** имаме по 1 монета от вид

***b*)** имаме неограничен брой монети

Пример:

$\begin{cases} \text{Вход : coins}[1\ ..4] = [1,\ 5,\ 6,\ 8],\ S = 11 \\ \text{Изход : } 2 \end{cases}$

***a*)**

Ясно е, че задачата можем да я решим **тъпо**, използвайки експоненциален алгоритъм - просто пробваме всички възможности и взимаме най-добрата от които. Сега ще разгледаме как да построим решение на задачата по схемата **Динамично Програмиране** и ще покажем защо привидно полиномиалното решение, всъщност е експоненциално.

Нека да разгледаме директно попълнена табличката (всяка задача решена по схемата ДП си има такава) и ще коментираме по нея:

	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{1}	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{1, 5}	0	1	∞	∞	∞	1	2	∞	∞	∞	∞	∞
{1, 5, 6}	0	1	∞	∞	∞	1	1	2	∞	∞	∞	2
{1, 5, 6, 8}	0	1	∞	∞	∞	1	1	2	1	2	∞	**2**

На всяка клетка имаме:

$\begin{cases} \infty & \text{, ако не можем да го получим сумата от текущия стълб с монетите за текущия ред} \\ \text{минималния брой монети} & \text{, ако можем да го получим сумата от текущия стълб с монетите за текущия ред} \end{cases}$.

Примерно в клетката DP[{1, 5, 6}][7] = 2 имаме, че 7 се получава минимално чрез 2 монети от {1, 5, 6}.

Разбира се търсеното от ***a*)** се намира в клетката DP[{1, 5, 6, 8}][11] = 2.

Как се образува таблицата:

· Първия ред и първата колона са ясни - нули за колоната и ∞ за реда

· Образуваме редовете последователно от горе надолу и всеки ред от ляво надясно по следната ф-ла:
dp[r][c] ← min(dp[r − 1][c], 1 + dp[r − 1][c − coins[c]]), като внимаваме индексите да не излязат извън матрицата.

```
1. task1a(coins[1 .. n], S) : // coins ∈ (ℕ⁺)ⁿ, n ∈ ℕ⁺, S ∈ ℕ₀
2.       dp[0 .. n][0 .. S] ← [[∞, …, ∞], …, [∞, …, ∞]]
3.       for r ← 0 to n
4.                dp[r][0] ← 0
5.       for r ← 1 to n
6.                for c ← 1 to S
7.                         dp[r][c] ← min(dp[r − 1][c], 1 + dp[r − 1][c − coins[r]])
8.       return dp[n][S]
```

Забележка Ако излезем извън матрицата, то приемаме че стойността там е +∞

Важно Този алгоритъм очевидно е с времева сложност $\theta(n.S)$, което е полином на n и S.. как така сложността се смъкна до привидно полиномиална? Нека да разгледаме следните две прости функции за илюстрация:

```
1. f(n) : // n ∈ ℕ0
2.     for i ← 1 to n
3.         print 'a'
```

```
1. g(A[1 .. n]) : // A ∈ ℤ^n, n ∈ ℕ0
2.     for i ← 1 to n
3.         print 'a'
```

Очевидно и двете функции са със сложност $f(n) = g(n) = \theta(n)$. Уловката е в това какво дефинираме като големина на входа.

· Когато сме във втроя случай (с който сме свикнали), то големината на входа е броя елементи на масива - тоест големината на входа е n. Оттук сложността спрямо големината на входа е $\theta(n)$. **[1]**

· Когато сме във първия случай, то големината на входа е броя символи, от които е съставено числото n или по-конкретно $\log_{10}(n)$, като основата е без значение. Ще използваме основа 10 за по-добра илюстрация. Нека положим $m = \log_{10}(n)$ е големината на входа. Тогава сложността спрямо големината на входа m е $\theta(n) = \theta(10^m)$ или с други думи - експоненциална спрямо големината на входа.

Деф (псевдо-полиномиален алгоритъм)

Алгоритми, които са с полиномиална сложност спрямо числовата стойност на вх. параметър, наричаме псевдо-полиномиални

Забележка task1a(n, S) е псевдо-полиномиален алгоритъм със сложност $\theta(n.S)$, а е с експоненциална сложност спрямо входа

b)

Не е трудно да съобразим, че схемата за изчисление е абсолютно аналогична с една единствена разлика: $\text{dp}[r][c] \leftarrow \min\left(\text{dp}[r-1][c],\ 1 + \text{dp}[\underbrace{\quad r \quad}_{\text{беше } r-1}][c - \text{coins}[c]]\right)$. Идеята е, че може оптималното решение за $c - \text{coins}[c]$ може да използва текущата монета, а на предходната подточка искаме да си гарантираме, че не не сме я използвали до момента. Нека да разгледаме попълнена табличката ѝ:

	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{1}	0	1	2	3	4	5	6	7	8	9	10	11
{1, 5}	0	1	2	3	4	1	2	3	4	5	2	3
{1, 5, 6}	0	1	2	3	4	1	1	2	3	4	2	2
{1, 5, 6, 8}	0	1	2	3	4	1	1	2	1	2	2	**2**

Съответно и псевдокода, като отново внимаваме за индексите:

```
1. task1b(coins[1 .. n], S) : // coins ∈ (ℕ+)^n, n ∈ ℕ+, S ∈ ℕ0
2.     dp[0 .. n][0 .. S] ← [[∞, …, ∞], …, [∞, …, ∞]]
3.     for r ← 0 to n
4.         dp[r][0] ← 0
5.     for r ← 1 to n
6.         for c ← 1 to S
7.             dp[r][c] ← min(dp[r − 1][c], 1 + dp[r][c − coins[r]])
8.     return dp[n][S]
```

Забележка Ако излезем извън матрицата, то приемаме че стойността там е $+\infty$

Зад. 2

Дадени са $n \in \mathbb{N}^+$ различни вида монети - coins[1 .. n] $\in (\mathbb{N}^+)^n$. При условие, че имаме неограничен брой монети от всеки вид, да се намери броя различни начини да се получи дадена сума $S \in \mathbb{N}_0$.

Пример:

{ Вход : coins[1 ..3] = [1, 2, 5], $S = 5$
  Изход : 4 // 1 + 1 + 1 + 1 + 1 = 1 + 1 + 1 + 2 = 1 + 2 + 2 = 5

Ясно е, че схемата за изчисление ще е много подобна на тази от **Зад. 1**. Даже може да направим аналогични случаи за *a*) и *b*). В случая ни изискват само аналога на *b*). Схемата за изчисление е следната: dp[r][c] ← dp[$r - 1$][c] + dp[r][$c -$ coins[r]]. Съответно таблицата с историята на изчисления е:

	0	1	2	3	4	5
{}	1	0	0	0	0	0
{1}	1	1	1	1	1	1
{1, 2}	1	1	2	2	3	3
{1, 2, 5}	1	1	2	2	3	**4**

Остана да напишем и псевдокода:

```
1. task2(coins[1 .. n], S) : // coins ∈ (ℕ⁺)ⁿ, n ∈ ℕ⁺, S ∈ ℕ₀
2.       dp[0 .. n][0 .. S] ← [[0, …, 0], …, [0, …, 0]]
3.       for r ← 0 to n
4.                dp[r][0] ← 1
5.                for c ← 1 to S
6.                         dp[r][c] ← dp[r − 1][c] + dp[r][c − coins[r]]
7.       return dp[n][S]
```

Заблежка Ако излезем извън матрицата, то приемаме че стойността там е 0

Зад. 3

Даден е регулярен израз regex[1 .. n] $\in \{a, b, \ldots, z, .,*\}^n$ и дума w[1 .. m] $\in \{a, b, \ldots, z\}^m$. Да се провери дали думата се *чете* от регулярния израз. Нека за улеснение регулярните изрази се състоят само от малки латински букви и метасимволите '.' и '*'.

Примери (компактно описани):

regex : *a.b*
w : acb, aab, axb ⟼ TRUE
w : ab, *b*, cb, axyb ⟼ FALSE

regex : *a*b*
w : *b*, ab, aab, aaab ⟼ TRUE
w : *a*, acb ⟼ FALSE

regex : *a*b.*c*
w : bc, bxc, bxxc, bxyc, abxc, abxxc, abxyc ⟼ TRUE
w : ay, ab ⟼ FALSE

Схемата за изчисление тук не е толкова тривиална, даже напротив - изглежда като магия:

$$d[i][j] \leftarrow \begin{cases} d[i-1][j-1] & , w[i] = \text{regex}[j] \vee \text{regex}[j] = \text{'.'} \\ d[i][j-2] & , \text{regex}[j] = \text{'*'} \;\&\; d[i][j-2] = \text{TRUE} \\ d[i-1][j] & , \text{regex}[j] = \text{'*'} \;\&\; (w[i] = \text{regex}[j-1] \vee \text{regex}[j-1] = \text{'.'}) \\ \text{FALSE} & , \text{else} \end{cases}$$

Накратко какво прави всеки случай:

· Ако текущата буква на регекса съвпада с текущата буква на думата, то вземи стойността в табличката, отговаряща на същия

регекс и дума, но без последните им символи. Аналогично ако регекса е метасимвола '.'.

· Ако имаме, че текущия символ на регекса е '*', то може да не го гледаме нито него, нито предходния символ - затова взимаме стойността в табличката, отговаряща на същата дума и на същия регекс, но с дължина две по-малко. Да обърнем внимание, че го взимаме само ако стойността е истина, иначе разглеждаме третия случай.

· Ако видим, че текущия символ на регекса е '*' и предходната буква на регекса съвпада с последната буква на думата, то тогава регекса ще *чете* текущата дума ↔ регекса *чете* текущата дума без последния символ.

· Иначе връщаме лъжа - примерно се различават символите.

Пример:

$\begin{cases} \text{Вход : regex}[1..6] = [x, a, *, b, ., c],\ w = [x, a, a, b, y, c] \\ \text{Изход : TRUE} \end{cases}$

Таблица с история на изчисленията:

	eps	x	a	*	b	.	c
eps	T	F	F	F	F	F	F
x	F	T	F	T	F	F	F
a	F	F	T	T	F	F	F
a	F	F	F	T	F	F	F
b	F	F	F	F	T	F	F
y	F	F	F	F	F	T	F
c	F	F	F	F	F	F	**T**

Остана да напишем и псевдокода:

```
1. task3(regex[1 .. n], w[1 .. m]) : // regex ∈ {a, b, …, z, .,*}^n, w ∈ {a, b, …, z}^m, n, m ∈ ℕ0
2.     d[0 .. m][0 .. n] ← [[FALSE, …, FALSE], …, [FALSE, …, FALSE]]
3.     d[0][0] ← TRUE
4.     for i ← 2 to n with step 2
5.         d[0][i] ← (d[0][i − 2] and regex[i] = '*')
6.     for i ← 1 to m
7.         for j ← 1 to n
8.             if regex[j] = w[i] then
9.                 d[i][j] ← d[i − 1][j − 1]
10.            else if j > 2 and regex[j] = '*' and [i][j − 2] then
11.                d[i][j] ← TRUE
12.            else if regex[j] = '*' and (w[i] = regex[j] or regex[j] = '.') and d[i − 1][j] then
13.                d[i][j] ← TRUE
14.    return d[m][n]
```

[1] Ако имаме вход $A[1..5] = [1, 13, 12\,345, 1\,048\,576, 4\,294\,967\,295]$, то тогава големината на входа е де факто $\log_{10}(1) + \log_{10}(13) + \log_{10}(12\,345) + \log_{10}(1\,048\,576) + \log_{10}(4\,294\,967\,295)$. Практически обаче ни интересува само размера на масива, тъй като числата с които работим (т.е с които процесора може да изпълнява операциите в константно време) са ограничени отгоре - 2^{32} или 2^{64} или други в зависимост от процесора. Тоест всички те са с размер най-много $\log_{10}(2^{32}) = \text{const}$. Разбира се процесора работи с битове, затова константата е по-скоро $\log_2(2^{32}) = 32$. Тоест имаме следното неравенство: $\log_{10}(1) + \log_{10}(13) + \log_{10}(12\,345) + \log_{10}(1\,048\,576) + \log_{10}(4\,294\,967\,295) \le 32 + 32 + 32 + 32 + 32 = 5 * 32$. Тоест големината на входа е $O(32 * \textit{големината на масива})$. Разбира се, ако работим с така наречените "големи числа", то трябва да работим по-прецизно.