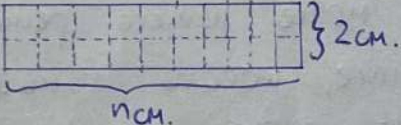


## DAA Упражнение 11 (Динамично програмиране)

Динамичното програмиране/оптимизиране се използва в задачи с оптимална подструктура и припокриващи се задачи. При решаване на задачи от такъв тип гледане как да сведем задачата до по-лесна подзадача. Запазваме резултата на вече изчислени подзадачи (мемоизация). Например при числата на Фибоначи чрез запазване на стойностите на  $F_{n-1}$  и  $F_n$  може директно да изчислим  $F_{n+1}$ , вместо да пресмятаме  $F_{n-1}$  и  $F_n$  от нулата, чрез което намаляваме сложността за изчисление на  $n$ -тото число на Фибоначи от експоненциална на линейна.

зад. 1/ Дадени са ни  $n$  на брой плочки с размери 2cm и 1cm и ни е дадена правоъгълна дъска с размери  $n$ cm и 2cm. По колко начина можем да запълним дъската с дадените плочки?

Решение: 

Нека да сведем задачата до неин по-прост вариант. За  $n=1$  имаме само 1 начин да запълним дъската и той е ако я поставим вертикално  $\boxed{0}$ . За  $n=2$  имаме 2 начина да запълним дъската и те са следните  $\boxed{00}$ ,  $\boxed{8}$ . За  $n=3$  можем да запълним дъската по 3 начина и те са  $\boxed{000}$ ,  $\boxed{80}$  и  $\boxed{08}$ . Можем да забележим, че решението за 3 е броя на начини за 2, като поставим една вертикална плочка + броя начини за 1, като добавим две вертикални плочки. Тоест ползваваме, че за дължина  $n$  на дъската решението е сумата от решенията на дъска с дължина  $n-1$  + дъска с дължина  $n-2$ , като базовите случаи са за дъски с дължина 1 и 2.

Важно: Тук рекурсивната декомпозиция е  $F_n = F_{n-1} + F_{n-2}$  и тя е реализирана чрез псевдокод на следващата страница. Решения без рекурсивна декомпозиция, а само псевдокод ползват 0 точки!



SOLUTION ( $N$ : дължина на дъската)

1.  $M[1...N]$  - задаваме празен масив
2.  $M[1] \leftarrow 1; M[2] \leftarrow 2$
3. for  $i \leftarrow 3$  to  $N$
4.  $M[i] \leftarrow M[i-1] + M[i-2]$
5. return  $M[N]$

// За конкретната задача няма нищо да се задава уел масив, а може решението да се реализира с 2 променливи. Конкретното решение е подходящо, ако имаме много на брой заявки от вида "по колко пъти може да се нареди дъска с дължина  $m \times 2$ ".

зад. 2/ Даден ни е масив  $C[1...n]$ , който ни казва какви видове монети имаме (напр. 1, 2, 10, 47, 1001) и  $S \in \mathbb{N}^+$ . Всички монети са с положителна стойност и имаме неограничен брой от всяка от тях. Предложете оптимален алгоритъм по време и памет, който връща минималният брой монети, чиято сума е равна на  $S$ .

Решение: Ако имаме за  $C = [1, 4, 9]$  и  $S = 12$ , то ако тръгнем да търсим решение чрез алген алгоритъм (тест взимаме възможно най-голямата монета във всеки момент), то може нашето решение да е грешно или да не намерим решение, когато такова съществува. В конкретният пример такъв алген подход би ни дал решение  $\{9, 1, 1, 1\}_n$ , което са 4 монети, докато решение по схемата ДП би ни дало отговор  $\{4, 4, 4\}_n$ , което е минималният брой монети.

Идеята отново е да развием задачата на по-проста подзадача. Например нека търсим сумата от монетите да е  $S'$ . За сума 0 имаме, че тя ще се получи със 0 монети. За сума 1 имаме, че тя ще се получи или с 1 монета (ако имаме такова), или няма да можем да я получим. За сума 2 имаме два варианта: или имаме монета 2 и следователно ще получим сумата с 1 монета, или ако имаме монета 1 и вземем оптималното решение за нея, тоест ползваваме отговор 2 (има и трети вариант в който нямаме нито монета 1, нито 2, следователно не можем да получим тези суми). Тоест за всякава сума  $S''$  тя може да се състави от 1 монета  $S''$  или от някакви монети  $S' +$  оптималното решение за сума  $S'' - S'$ , като трябва да вземем минимума от всички монети.

Пример:  $C = [1, 4, 9], S = 12$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	- сума
0	1	2	3	1	2	2	3	2	1	2	3	3	2	- брой монети



Тъй рекурсивната декомпозиция е  $f(x) = \min(+\infty, 1 + f(x - C[i])) \forall i, 1 \leq i \leq n$   
 Следният код реализира рекурсивната декомпозиция:  $\left( \begin{array}{l} +\infty, \text{ ако дадената сума} \\ \text{не е възможна да се} \\ \text{реализира с дадените} \\ \text{монети или } \min(1 + f(x - C[i])) \\ \text{(минимума по всяка възможна} \\ \text{монета)} \end{array} \right)$

1.  $M[0..S] \leftarrow \{+\infty, \dots, +\infty\}$

2.  $M[0] \leftarrow 0$

3. for  $x \leftarrow 1$  to  $S$  // конструираме всяка сума

4. for  $i \leftarrow 1$  to  $n$  // проверяваме с всички монети

5. if  $x \geq C[i]$  // за да не излезем извън масива, като се "връщаме" назад

6.  $M[x] \leftarrow \min(M[x], 1 + M[x - C[i]])$

7. return  $M[S]$

Очевидно сложността на алгоритма е  $\Theta(n \cdot S)$ , но това означава ли, че той е линеен?

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

Сами можете да забележите, че ако търсената сума  $S$  е 2 и ако е  $10^{10}$ , то времето за изпълнение ще е коренно различно.

Същото е и ако проверяваме дали  $2^8 - 1$  е просто и дали  $2^{999} - 1$  е просто число.

Зад. 3/ По даден уложен масив да се намери най-големата нарастваща подредица в масива (елементите не са непременно едни до друг и всеки елемент сам по себе си е нарастваща редица)

Пример: 3, 6, -4, 5, 8, 1, 40. Тъй най-дългата нарастваща подредица е с дължина 4 (например 3, 6, 8, 40 или -4, 5, 8, 40). Нека сведем задачата до по-проста и разглеждаме най-дълга нарастваща редица не в масива  $A[1..n]$ , а в подмасива  $A[1..i]$ . За  $i=1$  отговорът е 1, понеже всеки елемент сам по себе си е нарастваща редица. За  $i=2$  имаме два случая. Ако  $A[2] > A[1]$ , то отговорът е 2, в противен случай отговорът е 1, понеже всеки елемент сам по себе си е нарастваща редица. За  $i=3$  може 1)  $A[3] > A[1]$  и  $A[2] > A[3]$ , тогава отговорът е 2 (подредицата от  $A[1], A[3]$ ); 2)  $A[3] > A[2]$  и  $A[2] > A[1]$  тогава отговорът е 3; 3)  $A[3] > A[2]$  и  $A[3] < A[1]$ , тогава отговорът е 2; и 4)  $A[3] < A[1]$  и  $A[3] < A[2]$ , тогава отговорът зависи от това, дали  $A[1] < A[2]$ .



**Решение:** Нема разгледаме най-дългата нарастваща подредица завършваща в  $A[i]$ . Нев  $i$  получаваме, като 1, ако  $A[i]$  е най-малък елемент в подмасива  $A[1..i]$  или  $\max(1 + M[j])$  за  $j=1, \dots, i-1$ , ако  $A[i] > A[j]$ , където  $M[j]$  е най-дългата нарастваща подредица завършваща в  $A[j]$ .

SOLUTION LIS( $A[1..n]$ ): масив от цели числа

1.  $LIS[1..n] \leftarrow \{1, \dots, 1\}$

2. for  $j \leftarrow 2$  to  $n$

3. for  $j \leftarrow 1$  to  $i$

4. if  $A[i] > A[j]$

5.  $LIS[i] \leftarrow \max(LIS[i], 1 + LIS[j])$

6. return  $\max(LIS[1..n])$

Ако входният масив беше  $A[1..7] = \{3, 6, -4, 5, 8, 1, 40\}$ , то масивът LIS би изглеждал  $\{1, 2, 1, 2, 3, 2, 4\}$ . Ако трябва да ~~вземем~~ вземем ~~една~~ една най-дълга нарастваща подредица, то не можем да я възстановим от масива LIS ~~възвръщан~~ отзад напред. Например тъй търсим първо къде е 4, после първата 3 на, тогава се елемент на индекса  $i$  е по-малък от елемента на индекса на 4-тата, после първата такава 2 на и така до 1.

зад. 4/ На една стена има  $n$  картини, като всяка от картините има някаква цена  $c_1, \dots, c_n$ . Искаме така да вземем <sup>тези</sup> картините, така те сумата от цените им е максимална, но нямаме право да взимаме две съседни картини.

**Решение:** Задачата не може да се реши чрез сумиране на цените на четните и нечетните позиции и след това взимане на така, защото ако цените бяха 300, 1, 3, 1000, то ще е най-добре да вземем първата и четвъртата картина, а не втората и четвъртата. Идеята тук е за  $i$ -тата картина в подмасива е да вземем + ~~или~~ оптималното решение за  $A[1..i-2]$  или да не вземем и оптималното решение за  $A[1..i-1]$ . Тук рекурсивната декомпозиция е  $M[i] = \max(M[i-1], C[i] + M[i-2])$  където  $C[i]$  е цената на  $i$ -тата картина, а  $M[i]$  е оптималното решение за подмасива  $A[1..i]$ . Алгоритъмът на следващата страница реализира този идеал.



SOLUTION( $C[1..n]$ : массив с цените на картинките)

1.  $M[1..n]$  //  $M[i]$  съдържа оптималното решение за  $C[1..i]$
2.  $M[1] \leftarrow C[1]$
3.  $M[2] \leftarrow \max(C[1], C[2])$
4. for  $i \leftarrow 3$  to  $n$
5.      $M[i] \leftarrow \max(C[i] + M[i-2], M[i-1])$
6. return  $M[n]$