Д И Н А М И Ч Н О П Р О Г Р А М И Р А Н Е
(ПРИМЕРНО КОНТРОЛНО 4 ПО ДАА ЗА КН 1 — СУ, ФМИ, 2017 / 2018)

Задача 1. Съставете алгоритъм, който по зададено цяло положително число n намира най-малкия брой точни квадрати със сбор n. (30 точки)
Разширете алгоритъма така, че да отпечатва представянето на n като сбор на най-малък брой точни квадрати. (20 точки)
Демонстрирайте алгоритъма при $n = 10$. Анализирайте сложността по време и по памет за $\forall n$.

Решение: Таблицата на динамичното програмиране представлява едномерен масив $\mathrm{dyn}[0 \ldots n]$ от цели числа, където $\mathrm{dyn}[k]$ е най-малкият брой точни квадрати със сбор k. Масивът се попълва с помощта на началното условие $\mathrm{dyn}[0] = 0$ и рекурентната формула $\mathrm{dyn}[k] = 1 + \min\limits_{i} \left\{ \mathrm{dyn}\left[k - i^2\right] \right\}$, $k > 0$, където минимумът е по всички $i = 1, 2, 3, \ldots, \lfloor \sqrt{k} \rfloor$.

Разширеният вариант на алгоритъма (който отпечатва и събираемите) използва допълнителен масив $\mathrm{addend}[1 \ldots n]$ от цели числа, във всяка клетка на който се пази най-малкото събираемо (за подробности вж. в псевдокода).

```
SQUARES(n: positive integer)
 1  dyn[0...n]: array of positive integers;
 2  // dyn[k] = най-малкият брой точни квадрати със сбор k.
 3  addend[1...n]: array of positive integers;
 4  // addend[k] = най-малкото събираемо в представянето на k
 5  //                 като сбор на минимален брой точни квадрати.
 6  dyn[0] ← 0
 7  for k ← 1 to n
 8      dyn[k] ← k + 1
 9      i ← 1
10      j ← 1
11      while j ≤ k do
12          if dyn[k − j] < dyn[k]
13              dyn[k] ← dyn[k − j]
14              addend[k] ← j
15          i ← i + 1
16          j ← i × i
17      dyn[k] ← dyn[k] + 1
18  k ← n
19  while k > 0 do
20      j ← addend[k]
21      print j
22      k ← k − j
23  return dyn[n]
```

Демонстрация на работата на алгоритъма при $n = 10$:

k	0	1	2	3	4	5	6	7	8	9	10
dyn	0	1	2	3	1	2	3	4	2	1	2
addend		1	1	1	4	1	1	1	4	9	1

Таблицата се попълва отляво надясно. Например последната колонка е получена тъй:
$\mathrm{dyn}\big[10\big] = 1 + \min\Big\{\mathrm{dyn}\left[10 - 1^{\,2}\right]\ ;\ \mathrm{dyn}\left[10 - 2^{\,2}\right]\ ;\ \mathrm{dyn}\left[10 - 3^{\,2}\right]\Big\} =$
$= 1 + \min\Big\{\mathrm{dyn}\big[9\big]\ ;\ \mathrm{dyn}\big[6\big]\ ;\ \mathrm{dyn}\big[1\big]\Big\} = 1 + \min\Big\{1\ ;\ 3\ ;\ 1\Big\} = 1 + 1 = 2,$
което означава, че за числото 10 са нужни два квадрата.
Най-малкият елемент на мултимножеството $\Big\{1\ ;\ 3\ ;\ 1\Big\}$ е числото 1, което се среща два пъти и съответства на събираемите $1^{\,2}$ и $3^{\,2}$. Без значение е кое от събираемите ще вземем. Така, както алгоритъмът е оформен по-горе (със строго неравенство на ред № 12), той взима по-малкото събираемо, затова $\mathrm{addend}\big[10\big] = 1^{\,2} = 1$. (Ако на ред № 12 има нестрого неравенство, тогава в addend ще се пази по-голямото събираемо. В такъв случай алгоритъмът ще работи вярно, но по-бавно: редове № 13 и № 14 ще се изпълняват излишно.)

Самото представяне на 10 като сбор от точни квадрати се получава с помощта на масива addend. Тръгваме от колонката $k = 10$ и намираме addend $= 1$, което е най-малкото събираемо в сбора. Изваждаме това събираемо от сумата и остава $10 - 1 = 9$. От колонката $k = 9$ намираме следващото събираемо: addend $= 9$. Изваждаме това събираемо от оставащата сума: $9 - 9 = 0$. Не остава нищо, така че алгоритъмът приключва работа. Намерено е представянето $10 = 1 + 9$.

Анализ на сложността на алгоритъма:

Най-голямо количество допълнителна памет изразходват масивите dyn и addend, затова те определят сложността по памет: $\Theta(n)$.

Сложността по време зависи от броя итерации на циклите. Цикълът на редове № 11 – № 16 се изпълнява, докато $j = i^{\,2} \le k$, т.е. за $i = 1,\ 2,\ 3,\ \ldots\ ,\ \big\lfloor\sqrt{k}\,\big\rfloor$, което прави $\big\lfloor\sqrt{k}\,\big\rfloor$ итерации, а това по порядък е равно на $\sqrt{k}$. Следователно сложността на цикъла на редове № 7 – № 17 е равна по порядък на $\displaystyle\sum_{k=1}^{n}\sqrt{k} \;=\; \sum_{k=1}^{n} k^{\,0,5} \;=\; \Theta\left(n^{\,1,5}\right) = \Theta\left(n\sqrt{n}\right)$.

Тялото на цикъла на редове № 19 – № 22 се изпълнява $\mathrm{O}(n)$ пъти, защото на всяка итерация стойността на k намалява поне с една единица. Останалите команди (ред № 6 и ред № 23) изразходват константно време, затова не влияят на порядъка на сложността.

Времето на алгоритъма е сбор от времената на частите му: $\Theta\left(n\sqrt{n}\right) + \mathrm{O}(n) = \Theta\left(n\sqrt{n}\right)$. Окончателно, времевата сложност на алгоритъма е $\Theta\left(n\sqrt{n}\right)$.

Задача 2. Крадец влиза в картинна галерия с n картини с цени съответно $a_1\ ,\ a_2\ ,\ \ldots\ ,\ a_n$ лева. Картините са подредени последователно и крадецът знае за повреда в алармената система, която се активира, ако бъдат откраднати две съседни картини, но не се активира, ако бъде откраднатата картина, без да се пипат съседните.

Предложете бърз алгоритъм, изчисляващ цената на най-скъпата колекция, която крадецът може да отмъкне безнаказано. (30 точки)

Разширете алгоритъма така, че да изчисли кои картини да вземе крадецът. (20 точки)

Решение:

```
THIEF(a[1...n]: array of positive integers)
 1  dyn[1...n]: array of non-negative integers;
 2  // dyn[k] = максималната печалба, която крадецът щеше да получи,
 3  //          ако в галерията бяха само първите k картини.
 4  dyn[1] ← a[1]
 5  dyn[2] ← max(a[1] , a[2])
 6  for k ← 3 to n
 7      dyn[k] ← max(dyn[k − 1] , dyn[k − 2] + a[k])
 8  return dyn[n]
```

Идея на алгоритъма: При изчисляването на $dyn[k]$ има две възможности: крадецът да вземе или да не вземе k-ата картина. Ако той реши да не вземе k-ата картина, тогава максималната му печалба е колкото от първите $k-1$ картини, т.е. $dyn[k-1]$. Ако крадецът вземе k-ата картина, то максималната му печалба е колкото от първите $k-2$ картини, т.е. $dyn[k-2]$, плюс стойността на k-ата картина, т.е. $a[k]$. Тук имаме $k-2$, а не $k-1$, тъй като, щом крадецът е решил да вземе k-ата картина, той трябва да се откаже от $(k-1)$-ата.

Количеството допълнителна памет, използвано от алгоритъма, може да се оптимизира: елементите на масива dyn не са нужни през цялото време на изпълнение на алгоритъма; достатъчно е да пазим три последователни елемента, а именно $dyn[k-2]$, $dyn[k-1]$ и $dyn[k]$. Това намалява на порядък сложността по памет.

Ако допълнително поискаме от алгоритъма да намира оптималната колекция от картини, а не само максималната възможна печалба, то трябва да внесем промени в кода:

```
THIEF(a[1...n]: array of positive integers)
 1  dyn[1...n]: array of non-negative integers;
 2  // dyn[k] = максималната печалба, която крадецът щеше да получи,
 3  //          ако в галерията бяха само първите k картини.
 4  taken[1...n]: array of boolean;
 5  // taken[k] = true ⇔ k-тата картина би трябвало да бъде взета,
 6  //                   ако в галерията бяха само първите k картини.
 7  dyn[1] ← a[1]
 8  taken[1] ← true
 9  dyn[2] ← max(a[1] , a[2])
10  taken[2] ← (a[2] > a[1])
11  for k ← 3 to n
12      dyn[k] ← max(dyn[k − 1] , dyn[k − 2] + a[k])
13      taken[k] ← (dyn[k − 2] + a[k] > dyn[k − 1])
14  k ← n
15  while k > 0 do
16      if taken[k]
17          print k // Крадецът взима k-тата картина.
18          k ← k − 2
19      else k ← k − 1
20  return dyn[n]
```

Оптимизацията по памет не може да се приложи към масива $taken$: той е нужен през цялото време на изпълнение на алгоритъма.