1. Същност на алгоритмичната схема „динамично програмиране” – свеждане на задача със зададен размер към задачи от същия вид с по-малки размери и „мемоизация”.

И при пресмятането на минималното време за минаване на опашката през касата, и при пресмятането на числата на Fibonacci срещаме един и същи феномен: най-общо казано, задачата се свежда до *подзадачи*.

Конвенция 13: Задачи, подзадачи, подподзадачи

В Лекция 12 често се говори за някаква *задача* и нейните *подзадачи* и *подподзадачи*. Това са директни преводи на често срещаните английски термини *problem*, *subproblems* и *subsubproblems*, които се ползват често за описание на същността на схемата **Динамично Програмиране**.
Тази терминология обаче е формално несъвместима с Определение 1, според което “задача” е двуместна релация $\Pi \subseteq \mathcal{I} \times \mathcal{S}$. Формално, “подзачачи” би трябвало да са някакви подмножества на Π. А те не са.
Всъщност, когато говорим за *задача* Π, нейни *подзадачи* и техни *подподзадачи* в Лекция 12, имаме предвид съответно екземпляр $\mathbf{x}$ на $\mathtt{Pi}$, други екземпляри $\mathbf{y}_1, \ldots, \mathbf{y}_k$, които са конституенти на $\mathbf{x}$ в някакъв смисъл (това са подзадачите) и за всеки $\mathbf{y}_i$, екземпляри $\mathbf{z}_{i,1}, \ldots, \mathbf{z}_{i,t_i}$, които са конституенти на $\mathbf{y}_i$ в някакъв смисъл (това са подподзадачите).

Същността на схемата **Динамично Програмиране** е ефикасно пресмятане на рекурсия. Всяка рекурсия има свое дърво на рекурсията. Да мислим за дървото на рекурсията като анти-арборесценция. Всеки връх на дървото е асоциран със *състояние*, което е съответният вход на алгоритъма[↑]. Ако идентифицираме върховете с еднакви състояния, получаваме даг, чиито източници са върховете с началните условия. Този даг има размер, който е драстично по-малък от размера на дървото. Или поне в примерите, които ще разгледаме, е точно така. Алгоритмичното решение за дадения вход се състои в топо-сортиране на дага и генериране на решение при обхождане на дага в реда на топо-сортировката. Разбира се, ние няма да строим дага експлицитно, нито ще го топо-сортираме експлицитно, но е много удобна абстракция да мислим за изчислението по този начин.

Последният параграф от цитата заслужава особено внимание. Skiena казва, че **Динамично Програмиране** е приложимо, когато има някакви обекти, подредени в линейна наредба, и тази наредба не се мени, а е фиксирана. Читателят ще забележи, че това е в сила за всички примери за удачно ползване на **Динамично Програмиране**, които ще разгледаме. Ако става дума за коренови дървета (за които пише и Skiena), линейната наредба е обхождането на върховете в postorder. Ако става дума за множество, което няма иманентна наредба, наредбата е произволна линейна наредба върху това множество (задачата KNAPSACK, примерно).

- мемоизация

алгоритмични идеи. Понятието е въведено от британския компютърен учен Donald Michie, работил през 60-те години на 20 век в областта на изкуствения интелект. В *статия от 1968 г.* той описва метод, с който "компютрите се учат от опита си". В секцията "Recursively Defined Functions", Michie описва изчисление на рекурсивни функции, при което резултатите от виканията на дадена функция се съхраняват в таблица. При всяко викане на рекурсивната функция първо се проверява дали желаната стойност вече е била изчислена.

- Ако да, стойността се взема от таблицата.
- Ако не, започва изчисляването на стойността, но при викания върху по-малки стойности на аргумента отново се проверява дали за тях стойността е била вече изчислена.

Лесно се вижда, че на фундаментално ниво идеята е същата като идеята на схемата **Динамично Програмиране**: подзадачите имат общи подподзадачи, и резултатите от работата върху подподзадачите се съхраняват, за да не бъдат пресмятани отново и отново и отново. Съществена разлика между мемоизацията и схемата **Динамично Програмиране** е посоката на изчислението. Мемоизацията е истинска рекурсия с "ход надолу" и "ход нагоре", само че резултатите от викания се пазят. Динамичното програмиране е само втората част от рекурсията, "ходът нагоре".

2. Принцип за оптималност и конструиране на решението на задачата от решенията на подзадачите.

Определение 86: Принцип на оптималността

Дадена оптимизационна задача, по отношение на някаква характеристика, удовлетворява *принципа на оптималността*, ако подрешенията, по отношение на характеристиката, на всяко оптимално решение на свой ред са оптимални решения за съответните подекземпляри.

3. Задачи с линейна таблица на подзадачите (най-дълга растяща подредица).

Определение 91: Поредица и растяща поредица

Нека е дадена редица $S = (a_1, a_2, \ldots, a_n)$. *Поредица в* S е всяка редица $(a_{i_1}, a_{i_2}, \ldots, a_{i_k})$, такава че $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$. *Растяща поредица в* S е всяка поредица $(a_{i_1}, a_{i_2}, \ldots, a_{i_k})$, такава че $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$. *Максимална растяща поредица в* S е растяща поредица в S с максимална дължина.

Изч. Задача 41: Longest Increasing Sequence

екземпляр: Редица $S = (a_1, a_2, , \ldots, a_n)$.
решение: Максимална растяща поредица в S.

В олекотения вариант, задачата е само да се намери дължината на максимална растяща поредица.

Забележете, че "поредица в S" е редица от елементи, които **не са непременно съседни** в S. Поредица, чиито елементи са съседни—това е частен случай на поредица—наричаме *непрекъсната*, на английски *contiguous*; ако цялата редица е стринг, непрекъсната подредица в нея е всеки непразен подстринг. Разговорно казано, елементите на поредицата в общия случай са пръснати в S. Ето пример.

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Поредица в S е, примерно, $(9, -5, 3, 19, 10)$. Ето я като част от S:

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Непрекъсната поредица в S е, примерно, $(11, -2, 0, 3, -4)$. Ето я като част от S:

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Растяща поредица в S, максимална при това, е $(-5, -2, 0, 3, 6, 8, 10)$. Ето я като част от S:

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Максимална непрекъсната растяща в S е $(-2, 0, 3)$. Ето я като част от S:

$$S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$$

Намирането на максимална непрекъсната растяща поредица е тривиална задача. Има очевидно решение в линейно време с едно премитане, но дори решението с груба сила не е невъзможно бавно, понеже непрекъснатите поредици са $\Theta(n^2)$ на брой. Намирането на максимална растяща поредица обаче е нетривиална задача. Поредиците без ограничението за

непрекъснатост са много повече—а именно, $2^n - 1$, ако не броим празната поредица—и подходът с груба сила е безнадежден. В примера със синята поредица, когато започнем с -5 и отидем надясно, как да разберем, че трябва да прескочим 11, но да не прескочим -2?

12.8.1.1 Дин. Прогр. със сложност $\Theta(n^2)$

Първо решаваме задачата в олекотения вариант, в който се иска само стойността на оптимално решение. Изглежда смислено да характеризираме решение-поредица чрез най-десния елемент. Това е a_k за някое $k \in \{1, \ldots, n\}$.

Нека да мислим за рекурсивна декомпозиция за тази характеризация. Нека

$$\begin{aligned} S^1 &= (a_1) \\ S^2 &= (a_1, a_2) \\ S^3 &= (a_1, a_2, a_3) \\ &\ldots \\ S^n &= (a_1, a_2, \ldots, a_n) = S \end{aligned}$$

Нека ℓ_k е дължината на максимална растяща поредица в S^k, завършваща на a_k, за $1 \leqslant k \leqslant n$. Ако имаме тези ℓ_k, то ние сме готови! Тогава отговорът е $\max\{\ell_k \mid 1 \leqslant k \leqslant n\}$, тъй като всяка максимална растяща поредица завършва на някой елемент.

Как да получим ℓ_k от $\ell_{k-1}, \ldots, \ell_1$? Само числата $\ell_{k-1}, \ldots, \ell_1$ не са достатъчни, но двойките $(\ell_{k-1}, a_{k-1}), \ldots, (\ell_1, a_1)$ са достатъчни. Интересуват ни само тези a_j, ако има такива, за които $a_j < a_k$.

- Ако има такива, то за всяко такова a_j, a_k разширява растяща поредица, завършваща на a_j. За всяко такова a_j има смисъл да вземем максимална растяща поредица, завършваща на a_j, така че принципа на оптималността е в сила. Добавяйки a_k в десния край, ние разширяваме с точно един елемент. Ерго, ℓ_k е максимумът от ℓ-стойностите на въпросните a_j, плюс единица.

- Ако няма такива, то a_k не разширява поредица, а започва нова поредица, така че ℓ_k е единица.

Нека

$$\forall k \in \{1, \ldots, n\} : R_k = \{j \in \{1, \ldots, k-1\} \mid a_j < a_k\}$$

Тогава, за $1 \leqslant k \leqslant n$:

$$\ell_k = \begin{cases} \max\{\ell_j \mid j \in R_k\} + 1, & \text{ако } R_k \neq \varnothing \\ 1, & \text{ако } R_k = \varnothing \end{cases}$$

Превръщането на рекурсивната декомпозиция в алгоритъм е тривиално. Забележете, че $R_1 = \varnothing$, така че $\ell_1 = 1$ като начално условие.

Сега да помислим за по-тежкия вариант на задачата, в който се иска не просто дължината на максимална растяща поредица, а и някоя максимална растяща поредица. За всяка позиция $k \in \{1, \ldots, n\}$ записваме позицията π_k на елемента—ако има такъв—който предшества a_k в максимална растяща поредица, завършваща на a_k. Ако няма такъв, нека $\pi_k = \text{nil}$. Ето алгоритъм, който реализира тази идея.

LONGEST INCREASING SUBSEQUENCE$((a_1, a_2, \ldots, a_n))$

```
1  ℓ[1] ← 1
2  π[1] ← nil
3  for k ← 2 to n
4      ℓ[k] ← 1
5      π[k] ← nil
6      for j ← 1 to k − 1
7          if ℓ[k] < ℓ[j] + 1 and a[k] > a[j]
8              ℓ[k] ← ℓ[j] + 1
9              π[k] ← j
10 return (max_{1⩽k⩽n} {ℓ[k]}, π)
```

Коректността е очевидна. Сложността е $\Theta(n^2)$. Както ще видим, има по-ефикасен алгоритъм за тази задача, но квадратичната сложност по време не е непоносима. Ето примерна работа на алгоритъма върху $S = (14, 9, -5, 11, -2, 0, 3, -4, 12, 19, 6, 8, 22, 10)$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[k]	14	9	−5	11	−2	0	3	−4	12	19	6	8	22	10
ℓ[k]	1	1	1	2	2	3	4	2	5	6	5	6	7	7
π[k]	nil	nil	nil	3	3	5	6	3	7	9	7	11	10	12

Едно оптимално решение е маркирано в синьо.

4. Задачи с триъгълна таблица на подзадачите (оптимално разбиване на редица).

В тази секция ще разгледаме задачи, във всяка от които присъства някаква редица, за която търсим оптимална декомпозиция, **условно** наречена "скобуване". Редицата не се мени по време на работата на алгоритъма.

Да си припомним, че според цитата на Skiena на стр. 476, "Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order". Наистина, всички тези задачи се решават ефикасно с алгоритми по схемата **Динамично Програмиране** с двумерна таблица. В таблицата обаче се ползва (горе-долу) само половината над главния диагонал. Поради това понякога тези алгоритми се наричат и *алгоритми с триъгълна таблица*.

Съществото за тази секция е, че се иска оптимално скобуване. Този израз съдържа много условност, защото, примерно, при триангулацията на многоъгълниците, скобуване няма, поне не буквално. Винаги има обаче "главен срез" в редицата. Главният срез е или една "празнина" между два елемента на редицата, или един елемент от редицата, спрямо който се извършва последното действие (каквото и да е то). Скобуването се състои в това, че цялата редица се факторизира на две непразни подредици, те на свой ред се факторизират на по два непразни подредици, и така нататък, докато не бъдат достигнати началните условия.

12.5.1 Matrix-Chain Multiplication

Започваме с пример. Трябва да извършим матричното умножение $A \cdot B \cdot C \cdot D$, използвайки стандартния алгоритъм за умножение на матрици:

$$\underbrace{\begin{bmatrix} 1 & -2 & 2 & -1 & 3 \\ 4 & 5 & 4 & -1 & -3 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} 12 & -3 \\ -1 & 1 \\ 8 & 0 \\ -6 & 9 \\ 5 & 11 \end{bmatrix}}_{B} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_{C} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{D} \tag{12.16}$$

Един възможен начин да извършим това е:

$$A \cdot B \cdot C \cdot D = \underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 19 & -3 & 0 & 0 & -15 & 8 \\ -17 & -9 & 1 & 5 & 12 & -3 \end{bmatrix}}_{C} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{D} =$$

$$\underbrace{\begin{bmatrix} 646 & -324 & 19 & 95 & -537 & 351 \\ 2087 & 243 & -49 & -245 & -1578 & 675 \end{bmatrix}}_{((A \cdot B) \cdot C)} \cdot \underbrace{\begin{bmatrix} 4 & -9 \\ 3 & 21 \\ 6 & -1 \\ 0 & -11 \\ 3 & 2 \\ 1 & -8 \end{bmatrix}}_{D} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{(((A \cdot B) \cdot C) \cdot D)}$$

Да видим колко скаларни умножения са извършени при това матрично умножение. Ако умножим матрици $M_{x \times y} \cdot N_{y \times z}$, ние извършваме $x \cdot y \cdot z$ скаларни умножение. В случая, A е 2×5, B е 5×2, така че матричното умножение $(A \cdot B)$ "струва" $2 \cdot 5 \cdot 2 = 20$ скаларни умножения. $(A \cdot B)$ е 2×2, а C е 2×6, така че матричното умножение $((A \cdot B) \cdot C)$ "струва" $2 \cdot 2 \cdot 6 = 24$ скаларни умножения. $((A \cdot B) \cdot C)$ е 2×6, а D е 6×2, така че матричното умножение $(((A \cdot B) \cdot C) \cdot D)$ "струва" $2 \cdot 6 \cdot 2 = 24$ скларни умножения. Като цяло, умножението на четирите матрици по този начин "струва" $20 + 24 + 24 = 68$ скаларни умножения.

Друг начин да умножим матриците е

$$\underbrace{\begin{bmatrix} 51 & 19 \\ 66 & -49 \end{bmatrix}}_{(A \cdot B)} \cdot \underbrace{\begin{bmatrix} 30 & -328 \\ -56 & -44 \end{bmatrix}}_{(C \cdot D)} = \underbrace{\begin{bmatrix} 466 & -17564 \\ 4724 & -19492 \end{bmatrix}}_{((A \cdot B) \cdot (C \cdot D))}$$

Крайният резултат е същият. Това не е изненада, понеже умножението на матрици е асоциативно. Но да видим колко скаларни умножения ни струва този начин на умножение на матриците. $(A \cdot B)$ струва, както вече видяхме, 20 скаларни умножение. $(C \cdot D)$ струва $2 \cdot 6 \cdot 2 = 24$ скаларни умножения. $((A \cdot B) \cdot (C \cdot D))$ струва $2 \cdot 2 \cdot 2 = 8$ скаларни умножения. Като цяло, броят на скаларните умножения е $20 + 24 + 8 = 52$. Което е по-малко от 68.

Трети начин да умножим матриците е:

$$\underbrace{\begin{bmatrix}1 & -2 & 2 & -1 & 3\\ 4 & 5 & 4 & -1 & -3\end{bmatrix}}_{A}\cdot\underbrace{\begin{bmatrix}279 & -9 & -3 & -15 & -216 & 105\\ -36 & -6 & 1 & 5 & 27 & -11\\ 152 & -24 & 0 & 0 & -120 & 64\\ -267 & -63 & 9 & 45 & 198 & -75\\ -92 & -114 & 11 & 55 & 57 & 7\end{bmatrix}}_{(B\cdot C)}\cdot\underbrace{\begin{bmatrix}4 & -9\\ 3 & 21\\ 6 & -1\\ 0 & -11\\ 3 & 2\\ 1 & -8\end{bmatrix}}_{D}=$$

$$\underbrace{\begin{bmatrix}646 & -324 & 19 & 95 & -537 & 351\\ 2087 & 243 & -49 & -245 & -1578 & 675\end{bmatrix}}_{(A\cdot(B\cdot C))}\cdot\underbrace{\begin{bmatrix}4 & -9\\ 3 & 21\\ 6 & -1\\ 0 & -11\\ 3 & 2\\ 1 & -8\end{bmatrix}}_{D}=\underbrace{\begin{bmatrix}466 & -17564\\ 4724 & -19492\end{bmatrix}}_{((A\cdot(B\cdot C))\cdot D)}$$

$$M[i,j]=\begin{cases}0, & \text{ако } i=j,\\ \min_{i\leqslant k\leqslant j-1}\left(M[i,k]+M[k+1,j]+d_{i-1}d_kd_j\right), & \text{ако } i<j\end{cases}\tag{12.19}$$

```
Alg Matrix-Chain Multiplication(⟨d_0, d_1, ..., d_n⟩: цели положителни числа)
 1  създай M[1 .. n, 1 .. n] от числа
 2  for i ← 1 to n
 3      M[i, i] ← 0
 4  for diag ← 2 to n
 5      for i ← 1 to n − diag + 1
 6          j ← i + diag − 1
 7          M[i, j] ← M[i, i] + M[i + 1, j] + d_{i−1} d_i d_j
 8          for k ← i + 1 to j − 1
 9              M[i, j] ← min(M[i, j], M[i, k] + M[k + 1, j] + d_{i−1} d_k d_j)
10  return M[1, n]
```

$M[i,i]$ на ред 7, разбира се, е 0. Написано е по този начин, за да следва буквално (12.19) и да има прилика с ред 9.

Да видим пример, взет от [31, стр. 376]. Нека $n=6$ и $\langle d_0,\ldots,d_6\rangle=\langle 30,35,15,5,10,20,25\rangle$. Двумерният масив е 6×6. Клетките под главния диагонал не се ползват. Слагаме нули върху главния диагонал.

Да видим колко е $M[1,2]$. Съгласно (12.19),

$$\begin{aligned}M[1,2]&=\min_{1\leqslant k\leqslant 1}\left(M[1,k]+M[k+1,2]+d_{1-1}d_kd_2\right)\\&=M[1,1]+M[2,2]+d_0d_1d_2=0+0+30\cdot 35\cdot 15=15\,750\end{aligned}$$

	1	2	3	4	5	6
1	0	15 750				
2		0				
3			0			
4				0		
5					0	
6						0

Аналогично,

$$M[2,3] = M[2,2] + M[3,3] + d_1 d_2 d_3 = 0 + 0 + 35 \cdot 15 \cdot 5 = 2\,625$$
$$M[3,4] = M[3,3] + M[4,4] + d_2 d_3 d_4 = 0 + 0 + 15 \cdot 5 \cdot 10 = 750$$
$$M[4,5] = M[4,4] + M[5,5] + d_3 d_4 d_5 = 0 + 0 + 5 \cdot 10 \cdot 20 = 1\,000$$
$$M[5,6] = M[5,5] + M[6,6] + d_4 d_5 d_6 = 0 + 0 + 10 \cdot 20 \cdot 25 = 5\,000$$

Запълваме докрая диагонала над главния диагонал:

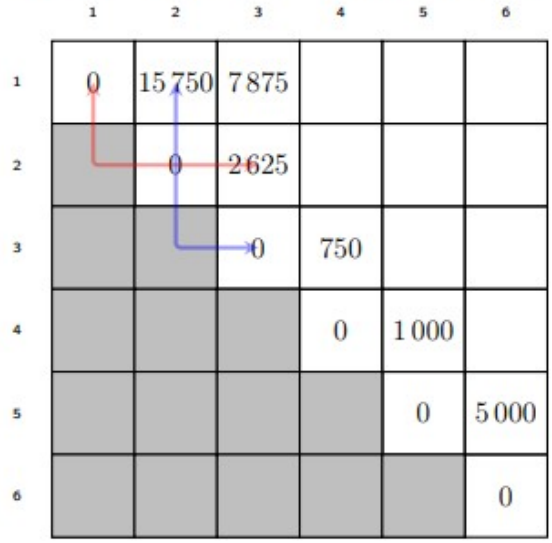
	1	2	3	4	5	6
1	0	15 750				
2		0	2 625			
3			0	750		
4				0	1 000	
5					0	5 000
6						0

Да видим колко е $M[1,3]$. Съгласно (12.19),

$$\begin{aligned} M[1,3] &= \min_{1 \leqslant k \leqslant 2} (M[1,k] + M[k+1,3] + d_{1-1} d_k d_3) \\ &= \min (M[1,1] + M[2,3] + d_0 d_1 d_3, M[1,2] + M[3,3] + d_0 d_2 d_3) \\ &= \min (0 + 2\,625 + 5\,250, 15\,750 + 0 + 2\,250) \\ &= \min (7\,875, 18\,000) = 7\,875 \end{aligned}$$

Следната фигура показва запълнената клетка $M[1,3]$, а с цветни линии е показано кои двой-

ки елементи от предишни диагонали участват в минимума.



Да кажем, че главният диагонал е *първият диагонал*, този над него е *вторият диагонал*, а следващият—който запълваме в момента—е *третият диагонал*. Изобщо, номерът на диагонала е номерът на колоната, на която той "излиза" на първия ред. Ето запълването и на третия диагонал докрая.

$$\begin{aligned}
\mathrm{M}[2,4] &= \min\left(\mathrm{M}[2,2] + \mathrm{M}[3,4] + \mathrm{d}_1\mathrm{d}_2\mathrm{d}_4, \mathrm{M}[2,3] + \mathrm{M}[4,4] + \mathrm{d}_1\mathrm{d}_3\mathrm{d}_4\right) \\
&= \min\left(0 + 750 + 5\,250, 2\,625 + 0 + 1\,750\right) \\
&= \min\left(6\,000, 4\,375\right) = 4\,375 \\
\mathrm{M}[3,5] &= \min\left(\mathrm{M}[3,3] + \mathrm{M}[4,5] + \mathrm{d}_2\mathrm{d}_3\mathrm{d}_5, \mathrm{M}[3,4] + \mathrm{M}[5,5] + \mathrm{d}_2\mathrm{d}_4\mathrm{d}_5\right) \\
&= \min\left(0 + 1\,000 + 1\,500, 750 + 0 + 3\,000\right) \\
&= \min\left(2\,500, 3\,750\right) = 2\,500 \\
\mathrm{M}[4,6] &= \min\left(\mathrm{M}[4,4] + \mathrm{M}[5,6] + \mathrm{d}_3\mathrm{d}_4\mathrm{d}_6, \mathrm{M}[4,5] + \mathrm{M}[6,6] + \mathrm{d}_3\mathrm{d}_5\mathrm{d}_6\right) \\
&= \min\left(0 + 5\,000 + 1\,250, 1\,000 + 0 + 2\,500\right) \\
&= \min\left(6\,250, 3\,500\right) = 3\,500
\end{aligned}$$

	1	2	3	4	5	6
1	0	15 750	7 875			
2		0	2 625	4 375		
3			0	750	2 500	
4				0	1 000	3 500
5					0	5 000
6						0

Ето запълването на целия масив без шестия диагонал (състоящ се само от $M[1,6]$).

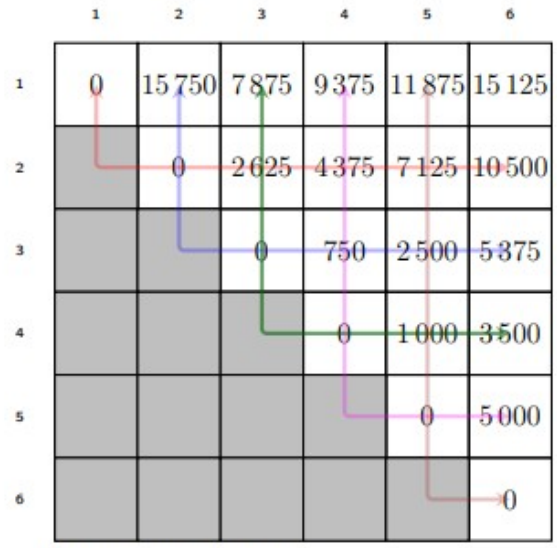
	1	2	3	4	5	6
1	0	15 750	7 875	9 375	11 875	
2		0	2 625	4 375	7 125	10 500
3			0	750	2 500	5 375
4				0	1 000	3 500
5					0	5 000
6						0

$M[1,6]$ се изчислява така:

$$\begin{aligned}
M[1,6] = \min(\, & M[1,1] + M[2,6] + d_0 d_1 d_6, \\
& M[1,2] + M[3,6] + d_0 d_2 d_6, \\
& \mathbf{M[1,3] + M[4,6] + d_0 d_3 d_6}, \\
& M[1,4] + M[5,6] + d_0 d_4 d_6, \\
& M[1,5] + M[6,6] + d_0 d_5 d_6) \\
= \min(\, & 0 + 10\,500 + 26\,250, \\
& 15\,750 + 5\,375 + 11\,250, \\
& \mathbf{7\,875 + 3\,500 + 3\,750}, \\
& 9\,375 + 5\,000 + 7\,500, \\
& 11\,875 + 0 + 15\,000) = \min(36\,750, 32\,375, \mathbf{15\,125}, 21\,875, 26\,875) = \mathbf{15\,125}
\end{aligned}$$

На следната фигура с цветни линии в пет различни цвята е показано кои двойки елементи

от предишни диагонали участват в пресмятането на минимума **15 125**.



Това е и краят на примера.

Да направим няколко пояснения върху псевдокода на Alg Matrix-Chain Multiplication. **for**-цикълът на редове 4–9 е с управляваща променлива, наречена не случайно "diag". Тя има смисъл на диагонал в направление горе-ляво – долу-дясно, чийто номер отговаря на номера на колоната, в която този диагонал пресича първия ред. Диагонал 1 е главният диагонал, който запълваме с нули на редове 2–3.

Клетките $M[i, j]$ в диагонал номер diag се отличават с това, че $j - i = \text{diag} - 1$, тоест, $j = i + \text{diag} - 1$. Примерно,

- за диагонал 2, разликата между j и i е 1: той се състои от $M[1, 2], M[2, 3], \ldots, M[n-1, n]$;
- за диагонал 3, разликата между j и i е 2: той се състои от $M[1, 3], M[2, 4], \ldots, M[n-2, n]$;
- и така нататък.

И наистина, на ред 6, на j се присвоява точно $i + \text{diag} - 1$.

За дадена стойност на diag, алгоритъмът първо пресмята $M[1, \text{diag}]$, после $M[2, \text{diag} + 1]$, после $M[3, \text{diag} + 2]$, и така нататък, и най-накрая пресмята $M[n - \text{diag} + 1, n]$.

Ние запълваме всеки диагонал в посока от горе-ляво към долу-дясно. Не е задължително да е така. За всяка клетка от даден диагонал, стойността се изчислява чрез стойности, които се вземат само от диагоналите с **по-малки номера** (както и от d стойностите от входа). Ерго, можем да запълваме клетките на даден диагонал в какъвто ред искаме. Тъй като сме избрали да запълваме всеки диагонал в посока от горе-ляво към долу-дясно, има смисъл да инициализираме i с 1 и да го инкрементираме, докато стане $n - (\text{diag} - 1) = n - \text{diag} + 1$, защото $\text{diag} - 1$ е броят на редовете под последния ред, съдържащ елемент на диагонал номер diag.

Коректността на алгоритъма е очевидна за всеки, който е убеден в коректността на рекурсивната декомпозиция, основаната на нея схема за изчисление и на техническите подробности на кода. Сложността по време е $\Theta(n^3)$.

5. Задачи с правоъгълна таблица на подзадачите (най-дълга обща подредица на две редици, задача за раницата).

Изч. Задача 42: Longest Common Sequence

екземпляр: Стрингове $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$ над азбука Σ.
решение: Максимална поредица, обща за X и Y.

Олекотеният вариант. Ето идея за ефикасно решение по схемата **Динамично Програмиране**, което връща цената на оптимално решение. Разглеждаме всички префикси на $X = x_1 \cdots x_n$ и $Y = y_1 \cdots y_m$, включително и празните. Нека $M(i, j)$, за $0 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant m$, означава максималната дължина на обща поредица в префиксите $x_1 \cdots x_i$ и $y_1 \cdots y_j$. Очевидно численото решение е $M(n, m)$.

Ако поне единият префикс е празен, дължината на максимална обща поредица е нула. Да допуснем, че и двата префикса са непразни. Нека

$$X^{i-1} = x_1 \cdots x_{i-1}, \quad X^i = x_1 \cdots x_i, \quad Y^{j-1} = y_1 \cdots y_{j-1}, \quad Y^j = y_1 \cdots y_j$$

за някакво $i \in \{1, \ldots, n\}$ и някакво $j \in \{1, \ldots, m\}$. Нека Z е максимална обща поредица за X^{i-1} и Y^{j-1}; ерго, $M(i-1, j-1) = |Z|$. Разглеждаме следните две възможности, взаимно изключващи се и изчерпателни.

① Нека $x_i = y_j$ и тази буква е $a \in \Sigma$. Тогава $X^i = X^{i-1}a$, $Y^j = Y^{j-1}a$, и очевидно Za се явява максимална обща поредица за X^i и Y^j, като $|Za| = |Z|+1$. Ерго, $M(i, j) = M(i-1, j-1)+1$ в този случай.

② Нека $x_i \neq y_j$. Дали Z е максимална обща поредица за X^i и Y^j? Не непременно. Ето пример:

$$X^i = \texttt{cdef}$$
$$Y^j = \texttt{ghic}$$

Очевидно $X^{i-1} = \texttt{cde}$ и $Y^{j-1} = \texttt{ghi}$, така че $Z = \epsilon$. А максималната обща поредица за X^i и Y^j е $\texttt{c}$, като $\texttt{c}$ се явява максимална обща поредица за X^{i-1} и Y^j.

Съществото е това, че не може максимална обща поредица за X^i и Y^j да има повече от една буква от максимална обща поредица за X^{i-1} и Y^j, понеже разликата между X^{i-1} и X^i е една буква. От съображения за симетрия, не може максимална обща поредица за X^i и Y^j да има повече от една буква от максимална обща поредица за X^i и Y^{j-1}. Следователно, ако $x_i \neq y_j$, $M(i,j) = \max(M(i-1,j), M(i,j-1))$.

Нещо повече. Не може максимална обща поредица за X^{i-1} и Y^j да има повече от една буква от максимална обща поредица за X^{i-1} и Y^{j-1}. От съображения за симетрия, не може максимална обща поредица за X^i и Y^{j-1} да има повече от една буква от максимална обща поредица за X^{i-1} и Y^{j-1}. Следователно, $\max(M(i-1,j), M(i,j-1))$ в случай ② не може да надхвърля $M(i-1,j-1)+1$ от случай ①.

И така,

$$M(i,j) = \begin{cases} 0, & \text{ако } i = 0 \text{ или } j = 0, \\ M(i-1,j-1)+1, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i = y_j \\ \max\{M(i,j-1), M(i-1,j)\}, & \text{ако } i > 0 \text{ и } j > 0 \text{ и } x_i \neq y_j \end{cases} \tag{12.38}$$

Таблицата $M[0\,..\,n, 0\,..\,m]$ е правоъгълна с размери $(n+1) \times (m+1)$. Съгласно началните условия, колона 0 и ред 0 са запълнени с нули. После попълваме отгоре надолу и отляво надясно, като $M[i,j]$ се изчислява чрез $M[i-1,j-1]$, $M[i-1,j]$ и $M[i,j-1]$. Търсеният числен отговор е $M[n,m]$.

Изч. Задача 37: Unbounded Knapsack

екземпляр: Множество от видове предмети $A = \{a_1, \ldots, a_n\}$, като има неограничено много предмети от всеки вид. За всеки вид a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.
решение: $(x_1, \ldots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^{n} x_i w(a_i) \leqslant C \tag{12.27}$$

$$\sum_{i=1}^{n} x_i v(a_i) \text{ е максимална} \tag{12.28}$$

Лесно можем да преведем това формално описание в терминологията на крадеца с раницата. C е капацитетът на раницата. Предметите за крадене са от n вида, като видовете са a_1, …, a_n, а от всеки вид има неограничено много[†] предмети за крадене. Предметите от всеки вид a_i са неразличими помежду си и имат една и съща стойност $v(a_i)$ и едно и също тегло $w(a_i)$. Наредената n-орка от естествени числа $(x_1, \ldots, x_n)$ има смисъл на това, по колко броя от всеки вид ще бъдат взети в раницата. (12.27) и (12.28) казват, че трябва да се вземат по толкова предмети от всеки вид, че общото тегло да не надхвърли капацитета, като тази подборка трябва да има максимална сумарна стойност.

Олекотеният вариант. Да направим характеризация на някое оптимално решение X. Очевидно X е непразно и се побира в раница с капацитет C. *// дотук е тривиално ...*
Продължаваме: решението X се получава от някое подрешение Y, като Y може и да е празно, чрез добавяне на един предмет от някой вид a_i. *// е, и?*
Продължаваме: Y се побира в раница с капацитет $C - w(a_i)$. *// това вече е нещо.*
Продължаваме: оптималното решение X е мултимножество от предмети. То се получава от някое подмултимножество Y с добавяне на един предмет от някой вид a_i, като при това сумарното тегло не надхвърля C, а сумарната цена е максимална. Очевидно Y е оптимално решение за раница с капацитет $C - w(a_i)$. *// аха, това е!*

Забележете, че характеристиката е капацитетът. Принципът на оптималността (Определение 86) е спазен, понеже оптимално решение за характеристика-капацитет C се получава от оптимално решение за характеристика-капацитет C', където $C' < C$.

Да помислим за рекурсивна декомпозиция. Да кажем, че числата $0, 1, \ldots, C-1$ са *редуцираните капацитети*. Ако за всеки редуциран капацитет D знаем някое оптимално решение $Y_D{}^{\uparrow}$, можем да намерим оптимално решение за пълния капацитет C, като за всяко D и всеки вид предмет a_i опитаме да добавим предмет от вид a_i към Y_D. Трябва да игнорираме получените решения, чието тегло надхвърля C, а от останалите да вземем решението с максимална цена. Това е по всички редуцирани капацитети и по всички видове предмети.

Следното решение на задачата е добре известно. Нека $M[j]$ е максималната стойност на предмети, които можем да сложим в раница с капацитет j, за всички $j \in \{0, \ldots, C\}$. Ако можем да пресмятаме $M[j]$ чрез $M[j-1], \ldots, M[0]$, ще получим решението $M[C]$. Схемата за изчисление е следната.

$$M[j] = \begin{cases} 0, & \text{ако } j = 0 \\ \max\left(M[j-1], \max_{i \in \{1,\ldots,n\} \text{ и } w(a_i) \leqslant j} \left(M[j - w(a_i)] + v(a_i)\right)\right), & \text{ако } j > 0 \end{cases}$$

Идеята е ясна. Ако раницата има нулев капацитет, в нея не можем да сложим нищо. Ако позволим капацитет $j > 0$, в нея можем да сложим най-много:

- или това, което можем да сложим при по-малкия капацитет $j-1$, с други думи не се възползваме от целия капацитет j,
- или, възползвайки от целия капацитет j, опитваме последователно да сложим един предмет a_i от всеки вид, чието тегло не надхвърля j, слагаме този a_i, който максимизира сумарната цена. Но, за да сложим предмет от вид a_i в раница с капацитет j, трябва да има свободен капацитет $j - w(a_i)$. Максимумът, който можем да постигнем, слагайки предмет от вид a_i, е максимумът за раница с капацитет $j - w(a_i)$ плюс стойността $v(a_i)$ на предмета.

Ако искаме пълно решение на задачата, което ни дава не просто числен отгово, а освен това и по колко предмети от всеки вид да сложим в раницата, можем да записваме за всяко j, $1 \leqslant j \leqslant C$, дали $M[j]$ е получено като $M[j-1]$ или по другия начин. Ако е по другия начин, а именно като максимум по всички i, такива че $w(a_i) \leqslant j$, записваме i, за което се постига максимална стойност на $M[j - w(a_i)] + v(a_i)$. От тази информация може лесно да генерираме пълно решение във време $O(C)$.

Сложността по време е $\Theta(nC)$, а по памет е $\Theta(C)$. Това е поредният псевдополиомиален алгоритъм, който разглеждаме. Ако числата $w(a_i)$ са малки, масивът M е малък и алгоритъмът е ефикасен.

Изч. Задача 38: 0-1 Knapsack

екземпляр: Множество от предмети $A = \{a_1, \ldots, a_n\}$. За всяко a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$ и неговото тегло $w(a_i) \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.

решение: Подмножество $X \subseteq A$, такова че:

$$\sum_{a \in X} w(a) \leqslant C$$

$$\sum_{a \in X} v(a) \text{ е максимална}$$

Бихме могли да дефинираме и тази версия на задачата чрез вектор $(x_1, \ldots, x_n)$, само че булев, а не от естествени числа. Това би бил характеристичен вектор, а знаем, че характеристичните вектори определят подмножества. Така че дори с такава формулировка, това, което се има предвид, е подмножество с максимална обща стойност и сумарен размер, не по-голям от капацитета.

В тази версия на задачата—също както и в предишната—не е добра идея да правим рекурсивна декомпозиция само по подмножествата от първите i елемента на A (има се предвид $\{a_1, \ldots, a_i\}$). Читателят лесно може да измисли пример, в който оптимално решение за, да кажем, $\{a_1, \ldots, a_6\}$, не ползва оптимално решение за $\{a_1, \ldots, a_5\}$. Ще направим двумерна рекурсивна декомпозиция, подобна на решението на 2-Partition. Нека $M[i, j]$ е (стойността на) оптимално решение за раница с капацитет j, ползващо предмети $a_1, \ldots, a_i$, където $0 \leqslant i \leqslant n$ и $0 \leqslant j \leqslant C$. Решението е $M[n, C]$. Схемата за изчисление е следната.

$$\begin{aligned}
&M[i, 0] = 0, \text{ за } 1 \leqslant i \leqslant n \quad // \text{ нулев капацитет} \\
&M[0, j] = 0, \text{ за } 1 \leqslant j \leqslant C \quad // \text{ няма предмети за слагане} \\
&M[i, j] = M[i-1, j], \text{ ако } i > 0 \text{ и } j > 0 \text{ и } w(a_i) > j \quad // \ a_i \text{ не се побира при капацитет } j \\
&M[i, j] = \max(\underbrace{M[i-1, j]}_{\text{не вземаме } a_i}, \underbrace{M[i-1, j - w(a_i)] + v(a_i)}_{\text{вземаме } a_i}), \text{ ако } i, j > 0 \text{ и } w(a_i) \leqslant j \qquad (12.29)
\end{aligned}$$

Случай (12.29) е основата на декомпозицията и може би единственият, чиято обосновка не е напълно очевидна. Имаме две възможности.

- Да не вземем предмета a_i. Стойността на такова решение ни е известна и тя е $M[i-1, j]$.
- Да вземем предмета a_i. При това обаче трябва да има капацитет за него $w(a_i)$, така че гледаме колко най-много можем да сложим, оставяйки си свободен капацитет $w(a_i)$, и какво ще получим при това. Тази стойност се пресмята тривиално и тя е $M[i-1, j-w(a_i)] + v(a_i)$.

Избираме по-добрата от двете възможности, тоест, максимума.

Алгоритъм, реализиращ това изчисление, е следният.

```
0-1 KNAPSACK(A = {a_1, ..., a_n}, v : A → ℝ^+, w : A → ℕ^+, C ∈ ℕ^+)
 1  for i ← 0 to n
 2      M[i, 0] ← 0
 3  for j ← 0 to C
 4      M[0, j] ← 0
 5  for i ← 1 to n
 6      for j ← 1 to C
 7          if w(a_i) > j
 8              M[i, j] ← M[i − 1, j]
 9          else
10              M[i, j] ← max(M[i − 1, j], M[i − 1, j − w(a_i)] + v(a_i))
11  return M[n, C]
```

Коректността на алгоритъма е очевидна. Сложността както по време, така и по памет, е $\Theta(nC)$.

Сега са дадени определен брой копия от всеки предмет. Тази версия на задачата е обобщение на 0-1 KNAPSACK, където беше дадено точно по един предмет от всеки вид. Тук ще разгледаме решение от книгата на Martello и Toth [103].

Изч. Задача 39: BOUNDED KNAPSACK

екземпляр: Множество от видове предмети $A = \{a_1, \ldots, a_n\}$. За всеки вид a_i, $1 \leqslant i \leqslant n$, са дадени неговата стойност $v(a_i) \in \mathbb{R}^+$, неговото тегло $w(a_i) \in \mathbb{N}^+$ и броят предмети от него $b_i \in \mathbb{N}^+$. Даден е капацитет $C \in \mathbb{N}^+$.
решение: $(x_1, \ldots, x_n) \in \mathbb{N}^n$, такава че

$$\sum_{i=1}^{n} x_i w(a_i) \leqslant C \tag{12.30}$$

$$0 \leqslant x_i \leqslant b_i, \text{ за } 1 \leqslant i \leqslant n \tag{12.31}$$

$$\sum_{i=1}^{n} x_i v(a_i) \text{ е максимална} \tag{12.32}$$

БОО, нека $\sum_{i=1}^{n} b_i w(a_i) > C$, защото в противен случай имаме тривиално решение $x_i = b_i$ за $1 \leqslant i \leqslant n$.

Задачата се свежда до 0-1 KNAPSACK със следната трансформация. Нека е даден екземпляр Π_B на BOUNDED KNAPSACK:

$$\langle \{a_1, \ldots, a_n\}, (v_1, \ldots, v_n), (w_1, \ldots, w_n), (b_1, \ldots, b_n), C \rangle$$

На него съответства екземпляр $\Pi_{0,1}$ на 0-1 KNAPSACK:

$$\langle \{\hat{a}_1, \ldots, \hat{a}_m\}, (\hat{v}_1, \ldots, \hat{v}_m), (\hat{w}_1, \ldots, \hat{w}_m), C \rangle$$

За всеки вид a_i, $1 \leqslant i \leqslant n$, конструираме $\lceil \log_2(b_i + 1) \rceil$ предмети в $\Pi_{0,1}$ така:

- първо, $\lfloor \log_2 b_i \rfloor$ предмети, чиито наредени двойки (стойност, тегло) са

$$(v_i, w_i), (2v_i, 2w_i), (4v_i, 4w_i), \ldots, (\lfloor \log_2 b_i \rfloor v_i, \lfloor \log_2 b_i \rfloor w_i)$$

- второ, ако b_i не е число от вида "точна степен на двойката минус единица", още един предмет със стойност

$$b_i v_i - (v_i + 2v_i + 4v_i + \cdots \lfloor \log_2 b_i \rfloor v_i)$$

и размер

$$b_i w_i - (w_i + 2w_i + 4w_i + \cdots + \lfloor \log_2 b_i \rfloor w_i)$$

Очевидно броят на предметите в $\Pi_{0,1}$, съответни на a_i, е $\lceil \log_2(b_i + 1) \rceil$, като сумата от стойностите им е $b_i v_i$ и сумата от размерите им е $b_i w_i$. Общо броят на предметите в $\Pi_{0,1}$ е $\sum_{i=1}^{n} \lceil \log_2(b_i + 1) \rceil$. Накратко ще записваме това число като m

m е и броят на булевите променливи в решението на $\Pi_{0,1}$. Ерго, на всяка променлива-естествено число x_i от решението Π_B съответстват булеви променливи $\hat{x}_{i,1}, \ldots, \hat{x}_{i,\lceil \log_2(b_i+1) \rceil}$ от решението на $\Pi_{0,1}$. Нещо повече, за $1 \leqslant h \leqslant \lceil \log_2(b_i + 1) \rceil$, булевата променлива $\hat{x}_{i,h}$ "дава" n_h предмети от Π_B, където

$$n_h = \begin{cases} 2^{h-1}, & \text{ако } h < \lceil \log_2(b_i + 1) \rceil \\ b_i - \sum_{j=1}^{\lceil \log_2(b_i+1) \rceil - 1} 2^{j-1}, & \text{ако } h = \lceil \log_2(b_i + 1) \rceil \end{cases}$$

Следователно $x_i = \sum_{h=1}^{\lceil \log_2(b_i+1) \rceil} n_h \hat{x}_{i,h}$ може да приеме всяка стойност от $\{0, \ldots, b_i\}$.

Сложността на алгоритъма е сложността на трансформацията плюс сложността на 0-1 Knapsack върху конструирания екземпляр. Сложността на трансформацията е $\Theta(m)$, а след това 0-1 Knapsack работи във време $\Theta(mC)$, така че общата сложност е $\Theta(mC)$. Да се направи подробно формално доказателство на коректността би било доста дълго, така че ще разгледаме примера за трансформацията от [103, стр. 83].

Като пример, да разгледаме следния екземпляр на Bounded Knapsack, в който $n = 3$ и $C = 10$.

	a_1	a_2	a_3
v	10	15	11
w	1	3	5
b	6	4	2

На око се вижда, че оптималното решение е $(x_1, x_2, x_3) = (6, 1, 0)$ с обща стойност

$$6 \cdot 10 + 1 \cdot 15 + 0 \cdot 11 = 75$$

и общо тегло

$$6 \cdot 1 + 1 \cdot 3 + 0 \cdot 5 = 9$$

което очевидно не нарушава капацитета 10.

Да приложим трансформацията. Имаме

$$\lceil \log_2(b_1 + 1) \rceil = \lceil \log_2(6 + 1) \rceil = 3$$
$$\lceil \log_2(b_2 + 1) \rceil = \lceil \log_2(4 + 1) \rceil = 3$$
$$\lceil \log_2(b_3 + 1) \rceil = \lceil \log_2(2 + 1) \rceil = 2$$

Ерго, $m = 3 + 3 + 2 = 8$. Нека трите предмета, съответстващи на a_1, са a_1', a_1'' и a_1''', трите предмета, съответстващи на a_2, са a_2', a_2'' и a_2''', и двата предмета, съответстващи на a_3, са a_3' и a_3''. Това са осемте предмета в екземпляра на 0-1 Knapsack, който строим. Стойностите и теглата са следните.

	a_1'	a_1''	a_1'''	a_2'	a_2''	a_2'''	a_3'	a_3''
v	10	20	30	15	30	15	11	11
w	1	2	3	3	6	3	5	5

Новият капацитет съвпада с оригиналния и е 10.

$a_1' = 1 \cdot 10 = 10$, $a_1'' = 2 \cdot 10 = 20$, а $a_1''' = b(a_1) \cdot v(a_1) - (a_1' + a_1'') = 60 - 30 = 30$. Останалите стойности и теглата се получават аналогично. Ако пуснем този екземпляр на 0-1 Knapsack в knapsack solver с $C = 10$, получаваме оптимално решение с цена 75 и тегло 9 при

$$(x_1', x_1'', x_1''', x_2', x_2'', x_2''', x_3', x_3'') = (1\,1\,1\,1\,0\,0\,0\,0)$$

Имената на променливите x съответстват по очевиден начин на предметите. Решението $(1\,1\,1\,1\,0\,0\,0\,0)$ означава вземане на следните предмети и само тях.

- Предмети a_1', a_1'' и a_1'''. Тоест, $1 + 2 + 3 = 6$ предмети от вида a_1.
- Предмет a_2'. Тоест, 1 предмет от вида a_2.

Точно както в решението на око.