روش (برنامه‌ریزی) پویا در طراحی الگوریتم‌ها

ویژگی‌ها راه‌حل برنامه‌ریزی پویا (Dynamic Programming)

- مسئله معمولاً بهینه‌سازی است.
- برای راه‌حل بهینه هم زیرمسئله‌ها هم باید بهینه حل شوند.
- حل باروش تقسیم‌وحل، موجب تکرار حل زیر مسئله‌های مشابه می‌شود.

 © محمد قدسی

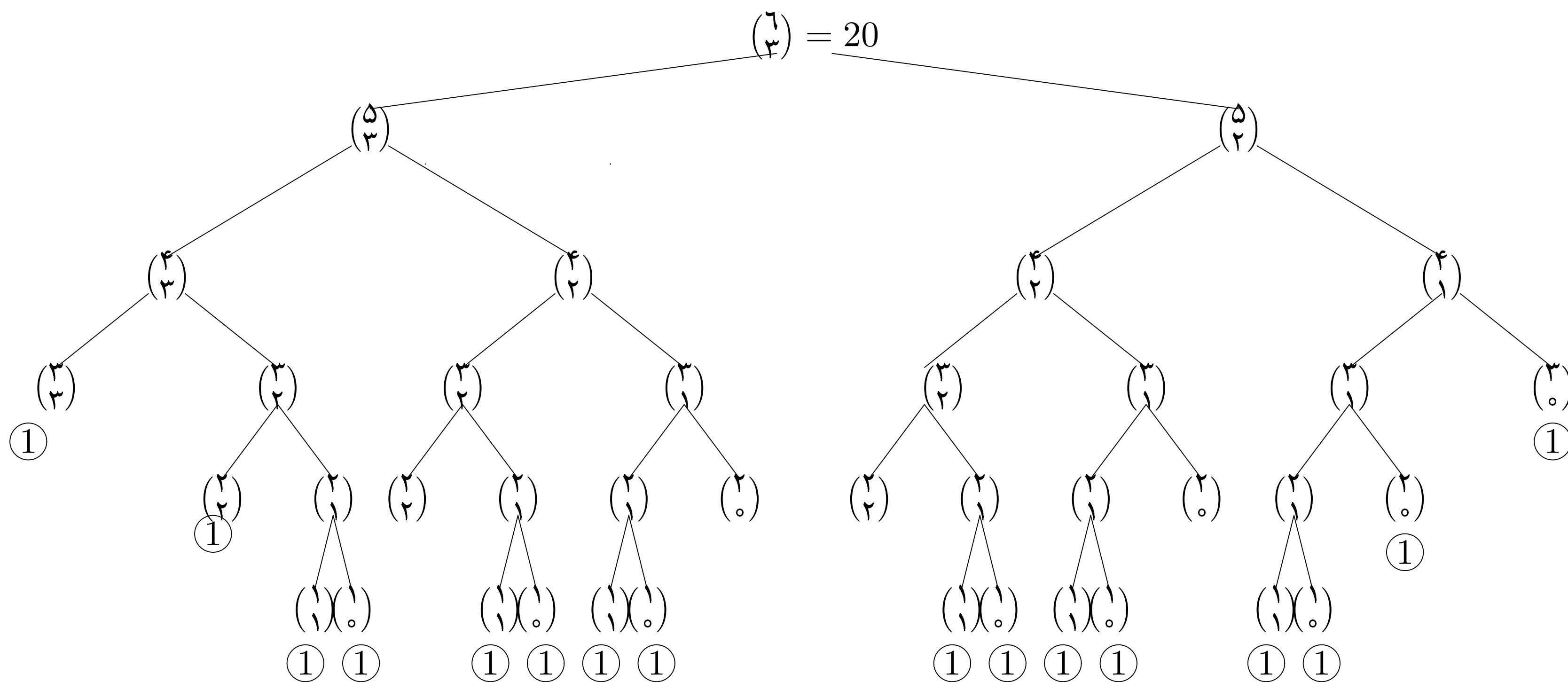
ترکیب m از n

می‌خواهیم حاصل $\binom{n}{m}$ را تنها با عمل جمع انجام دهیم:

$$\binom{n}{m} = \begin{cases} ۱, & \text{اگر } m = ۰ \text{ یا } n = m \\ \binom{n-۱}{m} + \binom{n-۱}{m-۱}, & \text{در غیر این صورت} \end{cases}$$

روش تقسیم‌وحل

```
COMBINATION(n, m)
        ▷ Computes (n choose m) using '+' only
  1  if  n = m  or  m = 0
  2     then return  1
  3     else
  4           return  COMBINATION(n−1, m)+COMBINATION(n−1, m−1)
```



درخت بازگشت برای محاسبه‌ی $\binom{6}{3}$

تحلیل

اگر $T(n,m)$ تعداد اعمال جمع باشد:

$$T(n,m) = \begin{cases} ۰, & n=m \quad \text{یا} \quad m=۰ \quad \text{اگر} \\ T(n-۱,m)+T(n-۱,m-۱)+۱, & \text{در غیر این صورت} \end{cases}$$

می‌توان دید که $T(n,m) = \binom{n}{m} - ۱$

(راه‌حل راحت‌تر؟!)

استفاده از ماتریس برای ذخیره‌ی اعداد میانی

از یک ماتریس به ابعاد $(m+۱) \times (n+۱)$ به نام C استفاده می‌کنیم. $C(i,j) = \binom{i}{j}$

```
     0   1   2   3   4   5   .   .   m
     ---------------------------------
0    1
1    1   1
2    1   2   1
3    1   3   3   1
4    1   4   6   4   1
5    1   5  10  10   5   1
.
.
.
.
.
n                                    C
```

محاسبه با استفاده از یک آرایه

<u>COMBINATION</u> (n, m)

$\triangleright$ A is an array of $[0..100]$

1 $A[0] \leftarrow 1$
2 **for** $i \leftarrow 1$ **to** n
3   **do if** $i \leq m$
4   **then** $A[i] \leftarrow 1$
5   **for** $j \leftarrow \min\{i-1, m\}$ **downto** 1
6   **do** $A[j] \leftarrow A[j] + A[j-1]$
7 **return** $A[m]$

مثال

	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	9	10	5	1	
6	1	6	15	19	15	6	1
7	1	7	21	34	34	21	7
8	1	8	28	35	68	55	28
9	1	9	36	63	103	123	83
10	1	10	45	99	166	226	206

تحلیل

تعداد اعمال جمع در این الگوریتم برابر است با

$$T(n,m) = ۱ + ۲ + ... + m - ۱ + m(n-m) = m(۲n - m - ۱)/۲$$

که برای برخی مقادیر n و m کوچک‌تر از $\binom{n}{m-۱}$ ولی برای مقادیر نزدیک m و n بزرگتر از آن است.

روش به‌تر

```
    0 1 2 3 4 . . . m
0   1
1   1 1
2   1 x 1
3   1 x x 1
4     x x x 1
.       x x x 1
.         x x x 1
.           x x x 1
.             x x x 1
.               x x x
                  x x
n                   x
```

الگوریتم به‌تر

```
COMBINATION()
1  for i ← 1 to n
2      do if i ≤ m
3            then A[i] ← 1
4         for j ← min(i − 1, m) downto max(1, m − n + i)
5             do A[j] ← A[j] + A[j − 1]
6
```

تعداد اعمال جمع $= (n-m)m$

با توجه به این که $\binom{n}{m} = \binom{n}{n-m}$ **این تعداد کمینه است و داریم** $(n-m)m \leq \binom{n}{m} - 1$.

برش چوب

- یک قطعه‌چوب به‌اندازه‌ی n را می‌خواهیم به قطعاتی تقسیم کنیم و بفروشیم.
- قیمت هر قطعه معلوم است. (اندازه‌ها اعداد صحیح)
- برش هزینه‌ای ندارد.
- برش به‌چه صورتی باشد تا بیش‌ترین سود را ببریم؟

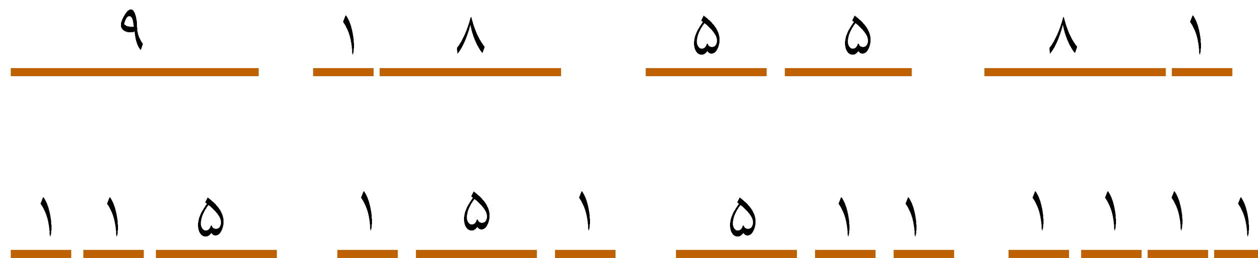
مثال

قیمت قطعات

طول قطعه‌ی i	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
ارزش p_i	۱	۵	۸	۹	۱۰	۱۷	۱۷	۲۰	۲۴	۳۰

چوب به‌اندازه‌ی n را در $۲^{n-۱}$ حالت می‌توان برید.

اگر چوب به‌اندازه‌ی ۴ باشد



راه‌حل بازگشتی

اگر p_i (برای $1 \leq i \leq n$) قیمت فروش قطعه‌ای به‌طول i باشد و

r_n بیش‌ترین سود از برش چوب به‌طول n باشد،

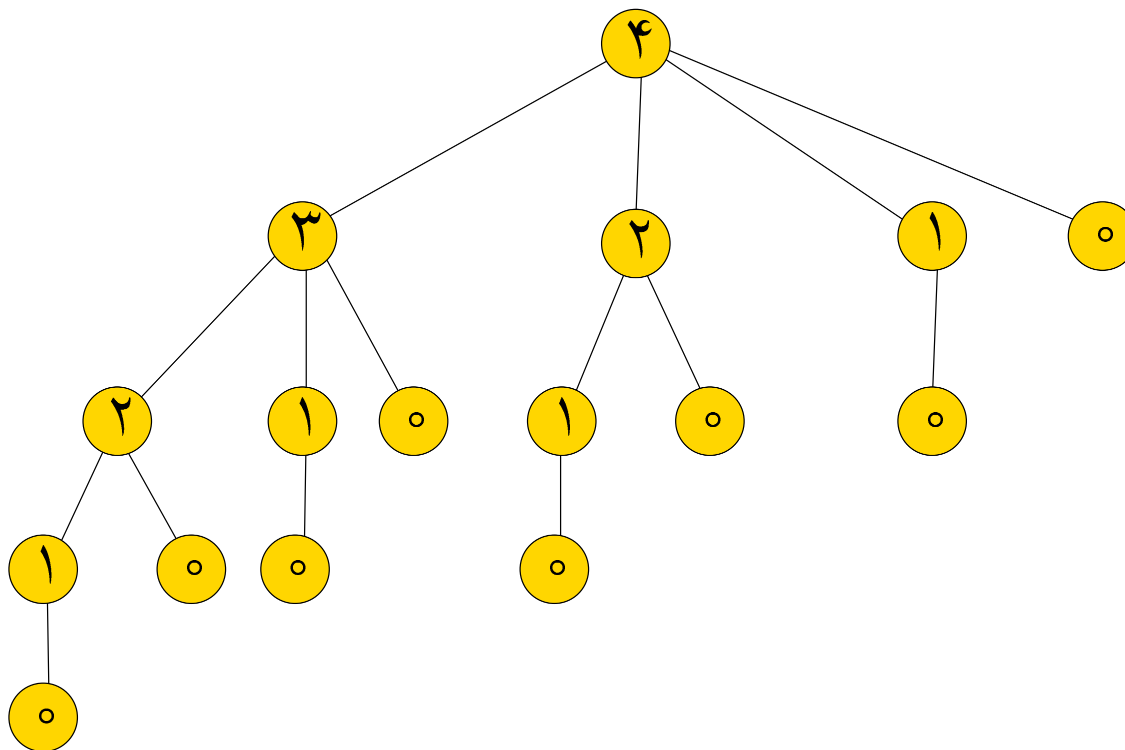
$$r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$$

راه‌حل بازگشتی از بالا-به-پایین

```
TOP-DOWN-CUT-ROD(p, n)
1  if n = 0
2     then return 0
3  q ← −∞
4  for i ← 1 to n
5       do q ← max(q, p[i] + TOP-DOWN-CUT-ROD(p, n − i))
6  return q
```

تحلیل

درخت بازگشت



زیرمسئله‌های تکراری

تحلیل (ادامه)

$$T(1) = 1$$

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

که جواب آن 2^n است.

راه‌حل از بالا-به-پایین (Memoized)

```
MEMOIZED-CUT-ROD (p, n)
        ▷ r[0..n] is a new array
  1  for i ← 0 to n
  2      do r[i] ← −∞
  3  return MEMOIZED-CUT-ROD-AUX(p, n, r))
```

```
MEMOIZED-CUT-ROD-AUX (p, n, r)
1  if r[n] ≥ 0
2     then return r[n]
3  if n = 0
4     then q ← 0
5     else q ← −∞
6           for i ← 1 to n
7               do q ←
                    max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8           r[n] ← q
9  return q
```

زمان اجرا $= \mathcal{O}(n^2)$

راه‌حل پویا (از پایین-به-بالا)

```
DYNAMIC-CUT-ROD(p, n)
      ▷ r[0..n] is a new array
  1  r[0] ← 0
  2  for j ← 1 to n
  3      do q ← −∞
  4          for i ← 1 to j
  5              do q ← max(q, p[i] + r[j − i])
  6          r[j] ← q
  7  return r[n]
```

زمان اجرا $= \mathcal{O}(n^۲)$

برای نوشتن جواب

```
DYNAMIC-CUT-ROD-EXTENDED(p, n)
      ▷ r[0..n] and s[0..n] are new arrays
  1  r[0] ← 0
  2  for j ← 1 to n
  3      do q ← −∞
  4          for i ← 1 to j
  5              do if q < p[i] + r[j − i]
  6                    then q ← p[i] + r[j − i]
  7                         s[j] ← i
  8          r[j] ← q
  9  return r and s
```

نوشتن جواب

```
PRINT-CUT-ROD-SOLUTION (p, n)
1  r, s ← DYNAMIC-CUT-ROD-EXTENDED(p, n)
2  while n > 0
3        do print s[n]
4           n ← n − s[n]
```

مسئله‌ی چوب‌بری

- یک قطعه چوب به‌اندازه‌ی ℓ داده شده
- آن را باید از همه‌ی نقاط $x_۱$ تا $x_n = \ell$ (از چپ به‌راست) ببریم.
- هزینه‌ی برش یک چوب به طول $n \leq r \leq ۱$ از هر نقطه‌ی آن r ریال است.
- به‌چه ترتیبی قطعه‌چوب داده شده را ببریم تا هزینه‌ی برش کمینه شود

زیرمسئله ؟

P_{ij} **برش بهینه‌ی قطعه چوب** $[x_i..x_j]$ **از نقاط** x_{i+1} **تا** x_{j-1}

C_{ij} **هزینه‌ی بهینه برای مسئله‌ی** P_{ij}.

$$C_{ij} = \min_{i<k<j} \{x_j - x_i + C_{ik} + C_{kj}\}$$

ضرب ماتریس‌ها

n ماتریس را می‌خواهیم درهم ضرب کنیم:

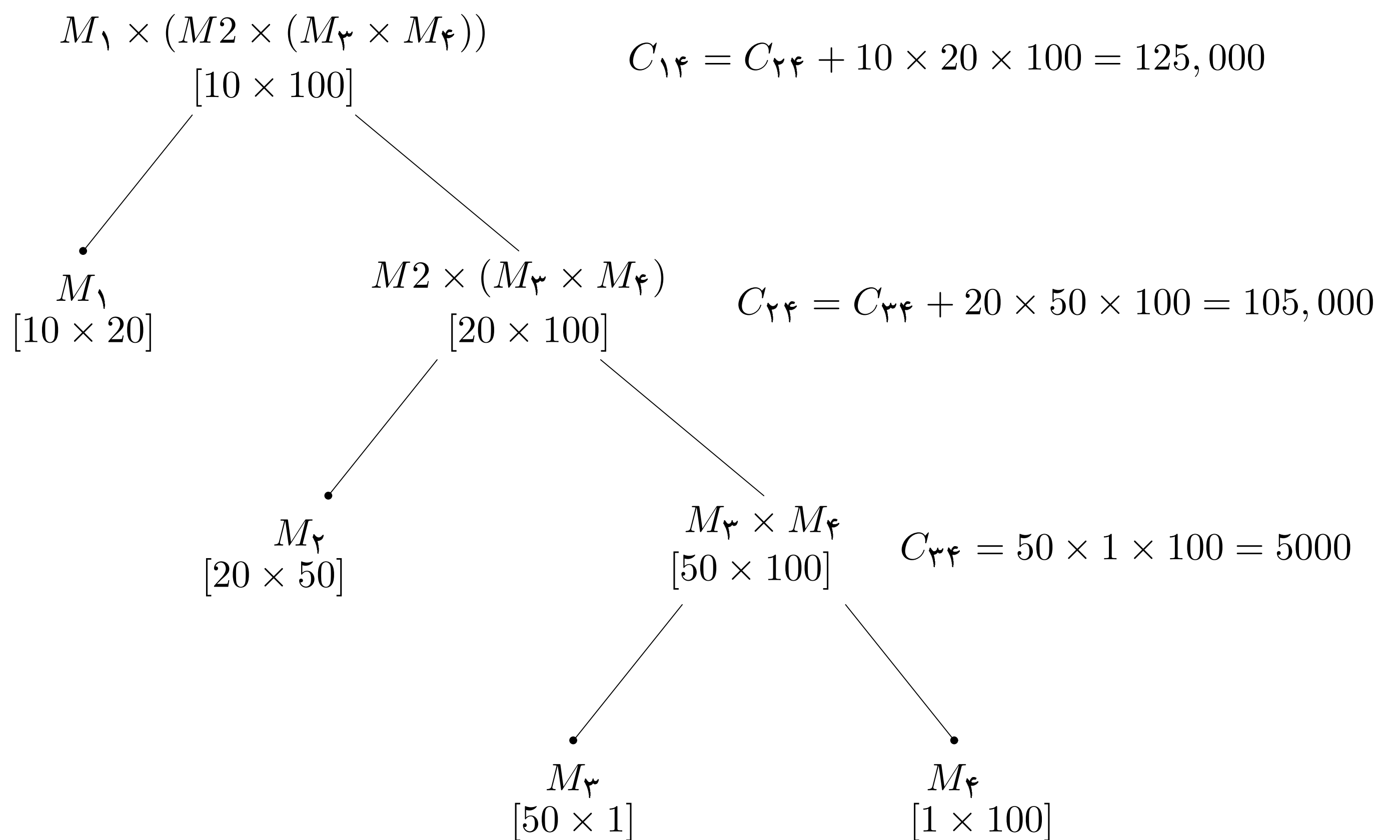
$$M_۱ \times M_۲ \times \cdots \times M_n$$

ابعاد M_i برابر است با: $d_{i-۱} \times d_i$

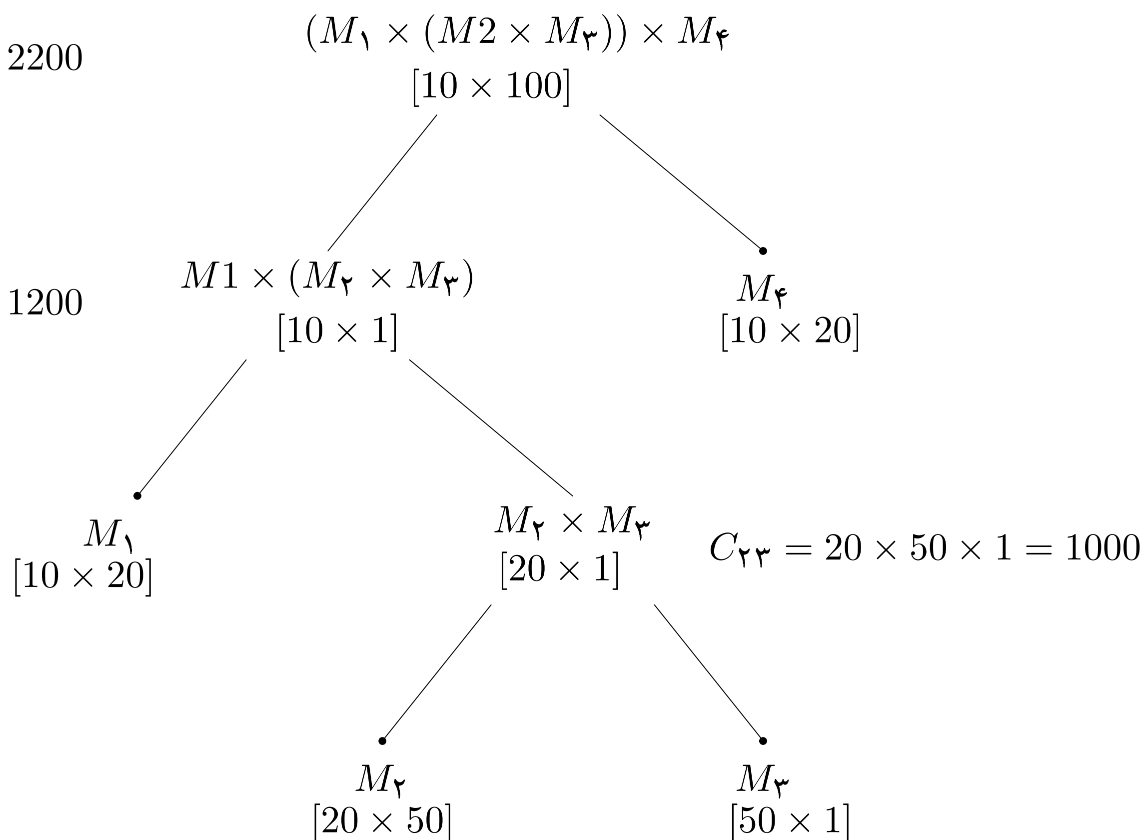
می‌خواهیم این ضرب‌ها را به‌ترتیبی انجام دهیم که هزینه‌ی کل کمینه شود.

مثال

$$M = \underset{[10 \times 20]}{M_1} \times \underset{[20 \times 50]}{M_2} \times \underset{[50 \times 1]}{M_3} \times \underset{[1 \times 100]}{M_4}$$



ترتیب و هزینه‌ی ضرب ماتریس‌ها در محاسبه‌ی $M_۱ \times (M_۲ \times (M_۳ \times M_۴))$



$C_{۱۴} = C_{۱۳} + 10 \times 1 \times 100 = 2200$

$C_{۲۴} = C_{۲۳} + 10 \times 20 \times 1 = 1200$

$C_{۲۳} = 20 \times 50 \times 1 = 1000$

ترتیب و هزینه‌ی ضرب ماتریس‌ها در محاسبه‌ی $(M_۱ \times (M_۲ \times M_۳)) \times M_۴$.

راه‌حل بازگشتی

زیر مسئله:

$$M_{ij} = M_i \times M_{i+۱} \times \cdots \times M_j$$

هزینه‌ی بهینه برای M_{ij} : C_{ij}

$$C_{ij} = \begin{cases} ۰ & if \;\; i = j \\ \min\limits_{i \le k < j} \{C_{ik} + C_{k+۱,j} + d_{i-۱} d_k d_j\} & if \;\; i < j \end{cases}$$

راه‌حل بازگشتی

```
RECURSIVE-MATRIX-MULTIPLICATION(d, i, j)
        ▷ Computes C[i, j], the optimal cost of M_i × M_{i+1} × ··· × M_j
  1  if i = j
  2    then return 0
  3  C[i, j] ← ∞
  4  for k ← i to j − 1
  5       do q ← RECURSIVE-MATRIX-MULTIPLICATION(d, i, k) +
                 RECURSIVE-MATRIX-MULTIPLICATION(d, k + 1, j) +
d_{i−1} d_k d_j
  6           if q < C[i, j]
  7             then C[i, j] ← q
  8  return C[i, j]
```

تحلیل

$T(n)$ زمان اجرای $\textsc{Recursive-Matrix-Multiplication}(d, 1, n)$

$$T(1) \geq 1$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1}[T(k) + T(n-k) + 1], \text{ for } n > 1$$

در این رابطه، هر $T(i)$ دو بار تکرار می‌شود، پس

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n$$

از روش جای‌گذاری و تکرار اثبات می‌کنیم که $T(n) = \Omega(2^n)$

نشان می‌دهیم که برای $n \geq 1$ داریم $T(n) \geq 2^{n-1}$.

پایه‌ی استقرا روشن است.

برای گام استقرا داریم،

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\geq 2^{n-1} \end{aligned}$$

از طرفی دیگر، این الگوریتم در واقع همه‌ی حالات ضرب n ماتریس را بررسی می‌کند و بین آن‌ها حالت بهینه را به‌دست می‌آورد.

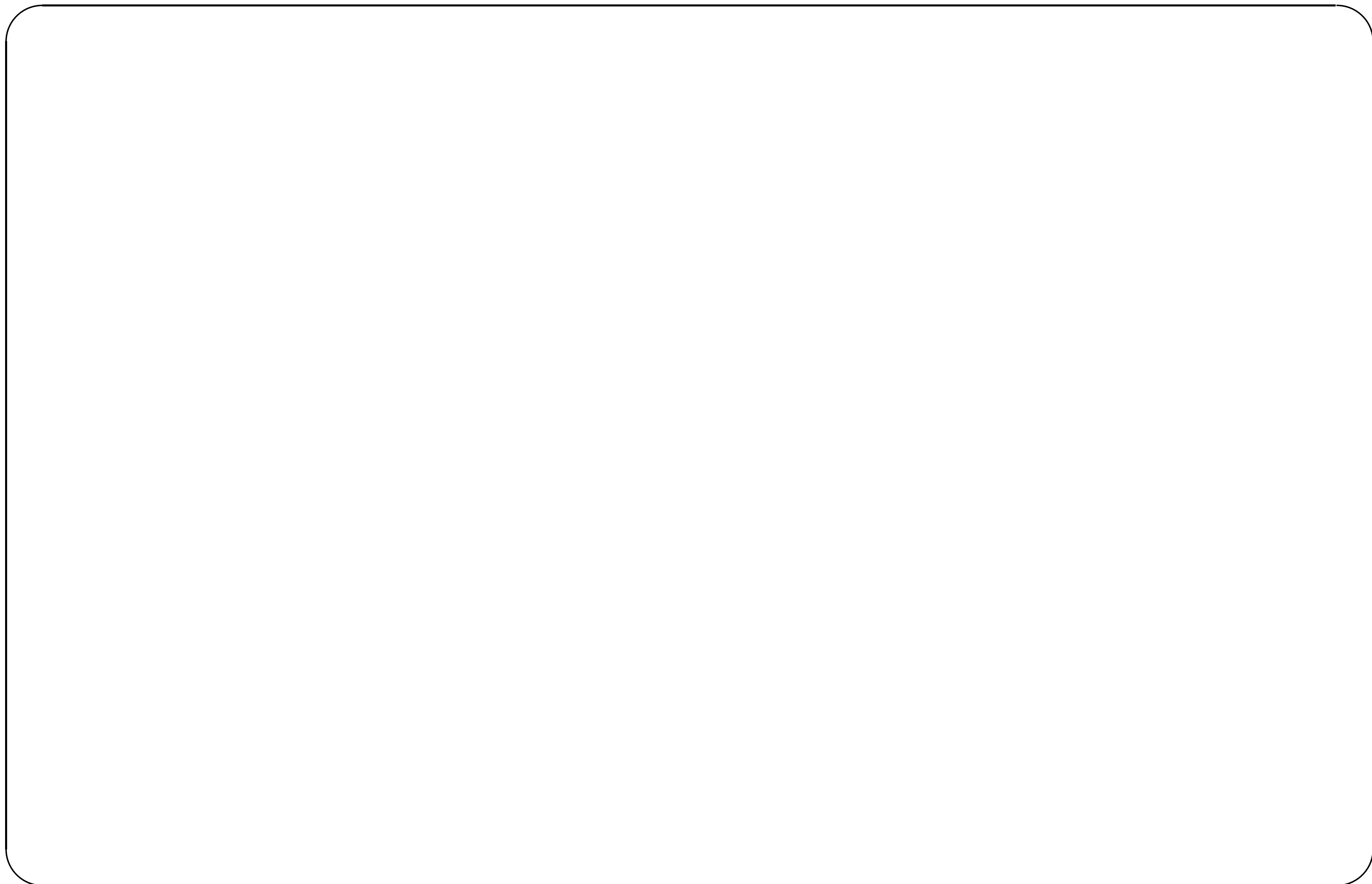
هزینه‌ی آن متناسب با تعداد حالات ممکن ضرب و یا پرانتزگذاری ضرب این ماتریس‌هاست که آن‌را با $T(n)$ نمایش می‌دهیم.

داریم،

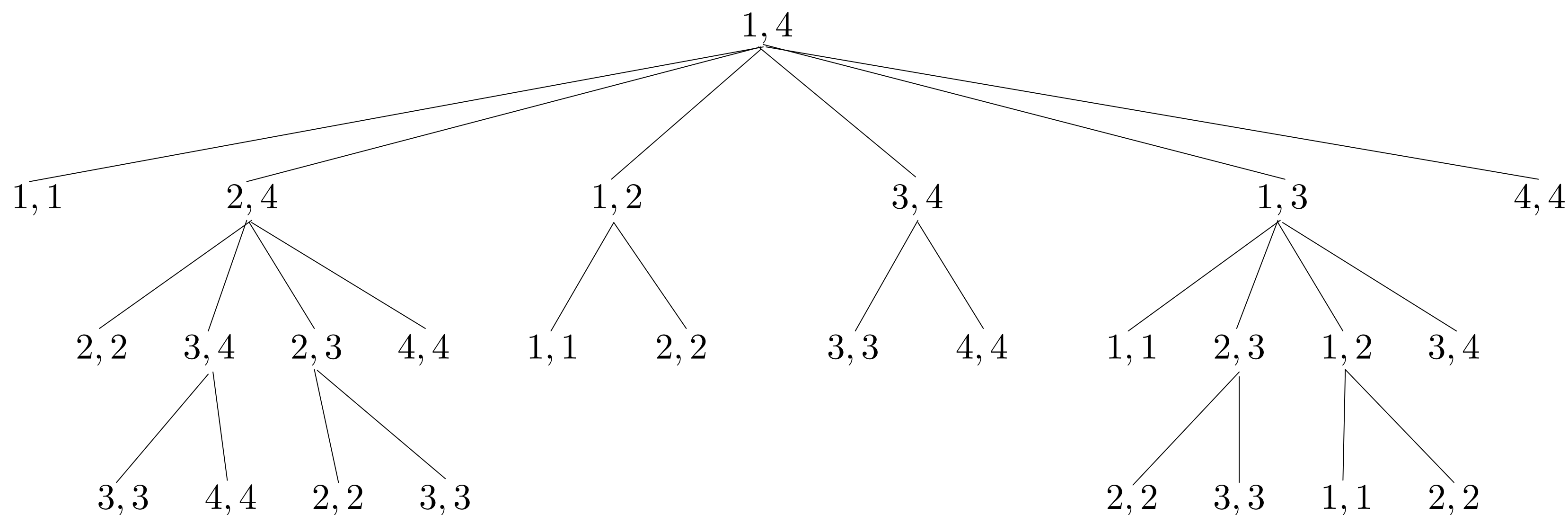
$$T(n) = \begin{cases} ۱ & n = ۱ \\ \sum_{i=۱}^{n-۱} T(i)T(n-i) & n > ۱ \end{cases}$$

می‌توان نشان داد که $T(n) = C(n-۱)$ که $C(n)$ nامین عدد کاتالان و برابر است با

$$C(n) = \frac{۱}{n+۱}\binom{۲n}{n} = \Omega(۴^n/n^{\frac{۳}{۲}})$$



زیرمسئله‌های تکراری



درخت فراخوانی‌ها برای $\textsc{Recursive-Matrix-Multiplication}(d, ۱, ۴)$

راه‌حل پویا

مسئله راه‌حل پویا دارد، چرا؟

زیرمسئله:

$$M_{ij} = M_i \times M_{i+1} \times \cdots \times M_j$$

بدیهی است که برای حل بهینه‌ی مسئله همه‌ی زیرمسئله‌ها هم باید بهینه حل شوند!

اگر C_{ij} هزینه‌ی بهینه‌ی M_{ij} باشد، داریم: $C_{ii} = 0$ و

$$C_{ij} = \min_{i \le k < j} \{C_{ik} + C_{k+1,j} + d_{i-1} d_k d_j\}$$

الگوریتم پویا

```
Dybnamic-Matrix-Multiplicationi(d)
1  for i ← 1 to n
2      do C[i,i] ← 0
3  for l ← 2 to n
4      do
           ▷ l = number of multiplied matrices
5          for i ← 1 to n − l + 1
6              do j ← i + l − 1
                   ▷ solving M_ij
7                  for k ← i to j − 1
8                      do C[i,j] ← min{C[i,k] + C[k+1,j]} + d[i−1] * d[k] * d[j]
9                         R[i,j] ← the value of k that makes the minimum
```

نوشتن پرانتزگذاری بهینه

```
PrintResults (i, j)
1  if i ≠ j
2    then k ← R[i, j]
3          Print '('
4          PrintResults(i, k)
5          Print '×'
6          PrintResults (k + 1, j)
7          Print ')'
```

تحلیل: زمان $\mathcal{O}(n^3)$ و حافظه‌ی $\mathcal{O}(n^2)$

مثال ۱

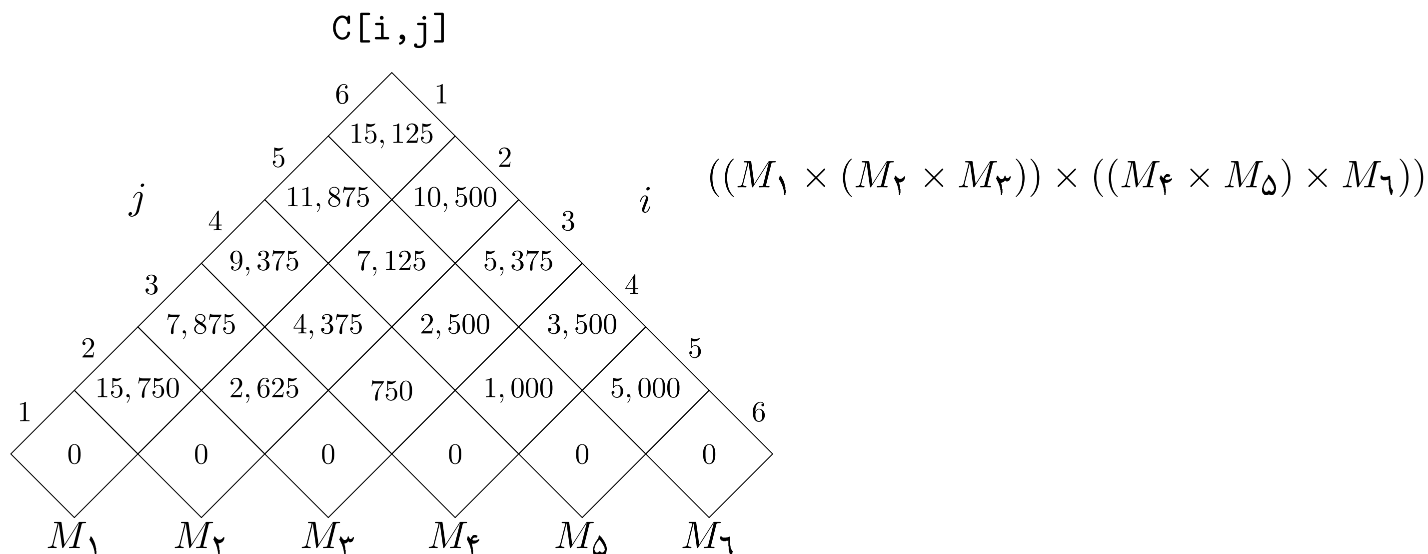
$$M = \underset{[10 \times 20]}{M_1} \times \underset{[20 \times 50]}{M_2} \times \underset{[50 \times 1]}{M_3} \times \underset{[1 \times 100]}{M_4}$$

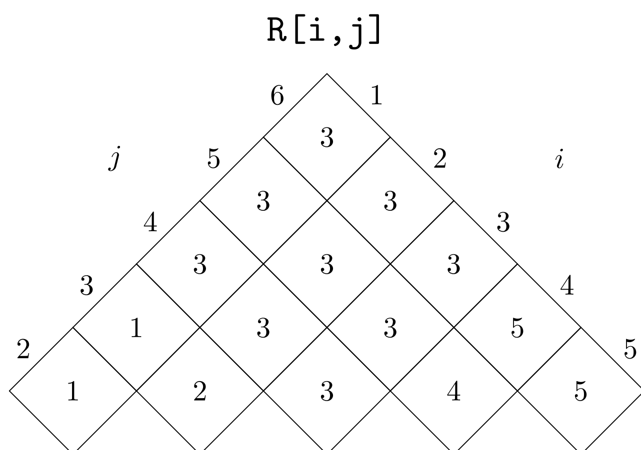
	۱	۲	۳	۴
۱	۰	۱۰۰۰۰, ۱	۱۲۰۰, ۱	۲۲۰۰, ۳
۲		۰	۱۰۰۰, ۲	۳۰۰۰, ۳
۳			۰	۵۰۰۰, ۳
۴				۰

مثال

ضرب شش ماتریس با اندازه‌های زیر:

$$\underset{[\text{۳۰} \times \text{۳۵}]}{M_{۱}} \times \underset{[\text{۳۵} \times \text{۱۵}]}{M_{۲}} \times \underset{[\text{۱۵} \times \text{۵}]}{M_{۳}} \times \underset{[\text{۵} \times \text{۱۰}]}{M_{۴}} \times \underset{[\text{۱۰} \times \text{۲۰}]}{M_{۵}} \times \underset{[\text{۲۰} \times \text{۲۵}]}{M_{۶}}$$





$$
\begin{aligned}
C[2,5] &= \min \begin{cases} C[2,2] + C[3,5] + d_1 d_2 d_5 = 0 + 2500 + 35 * 15 * 20 & = 13000, \\ C[2,3] + C[4,5] + d_1 d_3 d_5 = 2625 + 1000 + 35 * 5 * 20 & = 7125, \\ C[2,4] + C[5,5] + d_1 d_4 d_5 = 4375 + 0 + 35 * 10 * 20 & = 11375 \end{cases} \\
&= 7125.
\end{aligned}
$$

روش به‌خاطر سپاری (Memoization)

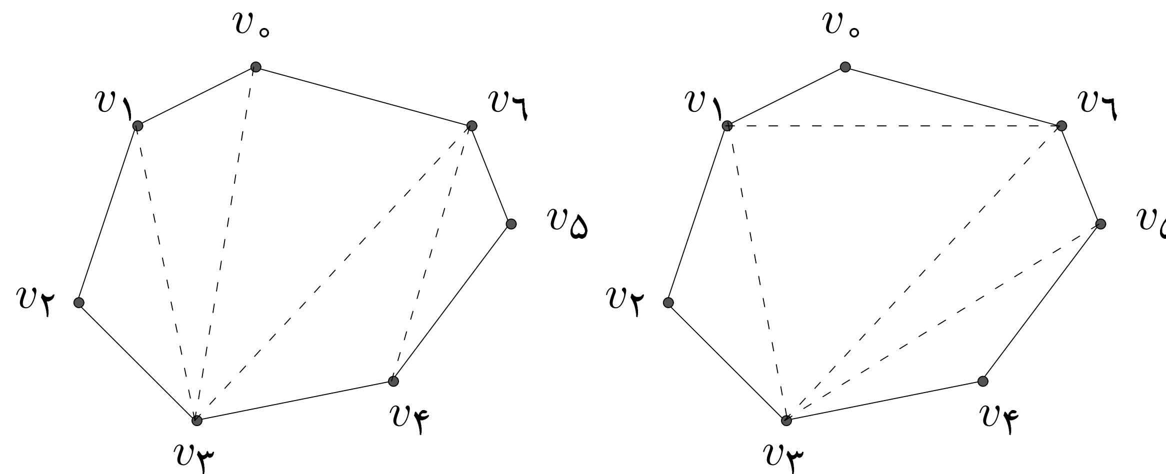
- بازگشتی : از بالا به پایین و کورکورانه
- پویا : از پایین به بالا و ذخیره‌ی حاصل زیرمسئله‌ها
- به‌خاطرسپاری : از بالا به پایین ولی انجام ندادن کار تکراری

```
MEMOIZED-MATRIX-MULTIPLICATION(d)
1  n ← length[d] − 1
2  for i ← 1 to n
3      do for j ← i to n
4             do C[i, j] ← ∞
```

```
LOOKUP-MATRIX(p, 1, n)
1  if C[i, j] < ∞
2    then return  C[i, j]
3  if i = j
4    then C[i, j] ← 0
5    elsefor k ← i to j − 1
6               do q ← LOOKUP-MATRIX(d, i, k) +
                      LOOKUP-MATRIX(d, k + 1, j) + d_{i−1} d_k d_j
7                  if q < C[i, j]
8                    then C[i, j] ← q
9  return  C[i, j]
```

این روش نیز مانند روش پویا از مرتبه‌ی $\Theta(n^3)$ است، چرا که $\Theta(n^2)$ درایه‌ی ماتریس C هر یک فقط یک‌بار و هر بار با مرتبه‌ی $\Theta(n)$ حساب می‌شود.

مثلث‌بندی بهینه‌ی چندضلعی محدب



یک چندضلعی محدب و دو نوع مثلث‌بندی آن

فرمول‌بندی مسئله

- n -ضلعی محدب $< v_0, v_1, \ldots, v_{n-1} >$ = $P_{0,n-1}$ داده‌شده است
- v_i مختصات رأس iام
- $< v_i, v_{i+1} >$ و $< v_i, v_{i-1} >$ اضلاع P هستند
- $< v_i, v_j >$ که ضلع نباشد، قطر است. قطرها در داخل P قرار می‌گیرند
- تعداد مثلث‌های هر مثلث‌بندی $n-2$ و تعداد قطرها $n-3$ است. چرا؟ $(3(n-2) = n + 2(n-3))$
- هر ضلع در یک مثلث قرار می‌گیرد

ورودی مسئله: مختصات رئوس $v_0, v_1, \ldots, v_{n-1}$ **یک چندضلعی محدب**

خروجی: قطرهایی که چندضلعی را مثلث‌بندی کنند و طولشان کمینه باشد.

می‌توان تابع وزن w **بر روی ضلع‌ها، قطرها یا مثلث‌های حاصل تعریف کرد.**

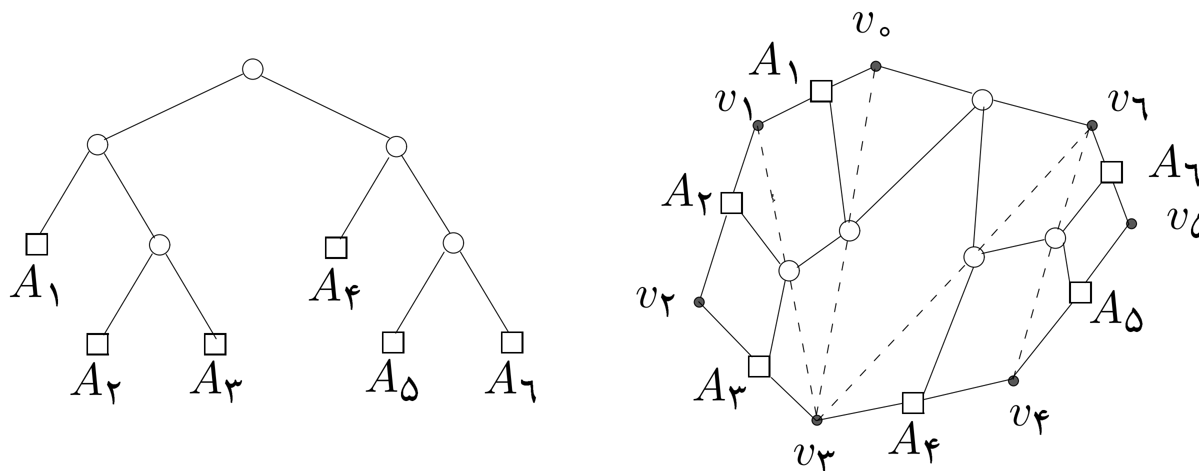
مثلث‌بندی بهینه: **مثلث‌بندی‌ای که در آن حاصل جمع وزن‌ها کمینه شود**

طول قطرها می‌تواند یک تابع وزن باشد

وزن مثلث‌ها بر روی هر مثلث به صورت زیر تعریف می‌شود:

$$w(\Delta_{v_i v_j v_k}) = \overline{v_i v_j} + \overline{v_j v_k} + \overline{v_i v_k}$$

$\overline{v_i v_j}$ **فاصله‌ی اقلیدسی بین** v_i **و** v_j **است.**



تناظر مسئله‌های ضرب ماتریس‌ها و مثلث‌بندی یک چندضلعی محدب.

متناظر با: $((A_1(A_2A_3))(A_4(A_5A_6)))$

تبدیل مسئله‌ی ضرب ماتریس‌ها به مثلث‌بندی بهینه

ورودی: $A_1 \times A_2 \times \ldots \times A_n$ با اندازه‌های $d_0, d_1, \ldots, d_n$

- یک $n+1$ ضلعی محدب با رئوس $v_0, v_1, \ldots, v_n$ می‌سازیم و آن‌را مثلث‌بندی بهینه می‌کنیم
- ماتریس A_i متناظر با ضلع $< v_{i-1} v_i >$
- وزن مثلث $w(\Delta_{v_i v_j v_k}) = d_i d_j d_k$ محاسبه می‌شود $A_1 \times A_2 \times \ldots \times A_n$
- قطر $< v_i, v_j > (i < j)$ متناظر است با ضرب ماتریس‌های $A_{i+1} \times A_{i+1} \times \ldots \times A_j$
- مثلث‌بندی بهینه چندضلعی P با در نظر گرفتن تابع فوق درخت پارس برای پرانتزگذاری بهینه‌ی $A_1 A_2 \ldots A_n$ را ایجاد می‌کند.

استفاده از کد ضرب ماتریس‌ها برای حل مسئله‌ی مثلث‌بندی بهینه

ورودی: n ضلعی P با رئوس $v_0, v_1, \ldots, v_{n-1}$

- ضرب ماتریس‌های $A_1 \times A_2 \times \ldots \times A_{n-1}$ را در نظر می‌گیریم.
- ماتریس A_i متناظر است با ضلع $< v_{i-1}v_i >$ و وزن آن‌را $w(A_i) = \overline{v_{i-1}v_i}$ در نظر می‌گیریم.
- هزینه‌ی ضرب دو ماتریس $A_i \times A_{i+1}$ برابر $w(A_i) + w(A_{i+1})$ محاسبه می‌کنیم. و $w(A_i \times A_{i+1}) = \overline{v_{i-1}v_{i+1}}$ (طول قطر) می‌گیریم.
- ماتریس $A_i \times A_{i+1}$ متناظر است با ضلع $< v_{i-1}v_{i+1} >$
- قطر $< v_i, v_j >$ $(i < j)$ متناظر است با ضرب ماتریس‌های $A_j \times \ldots \times A_{i+1} \times A_{i+1}$

- کد ضرب ماتریس‌ها را با این تغییرات اجرا می‌کنیم.

راه‌حل مستقیم

- زیرمسئله: $P_{i,j} =< v_i, v_{i+1}, \ldots, v_j >$ برای $i \leq j + 2$
- $< v_i, v_j >$ قطر است (مگر برای $P_{1,n}$)
- C_{ij} هزینه‌ی بهینه‌ی P_{ij}
- v_k رأس مقابل مثلثی به ضلع $< v_i, v_j >$. $(i < k < j)$

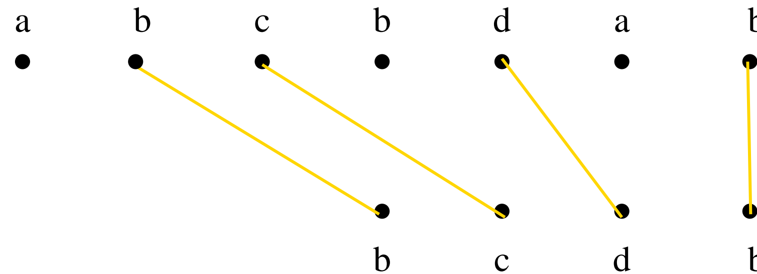
داریم:

$$C_{i,j} = \begin{cases} ۰ & i \leq j \leq i + ۲ \\ \min_{i<k<j}\{C_{ik} + C_{kj} + d_{ik} + d_{kj}\}, & j > i + ۲ \end{cases}$$

d_{ij} برابر اندازه‌ی قطر $< v_i, v_j >$ است. اگر $< v_i, v_j >$ ضلع باشد، $d_{ij} = ۰$.

الگوریتم پویا با $\mathcal{O}(n^۳)$ بدیهی است.

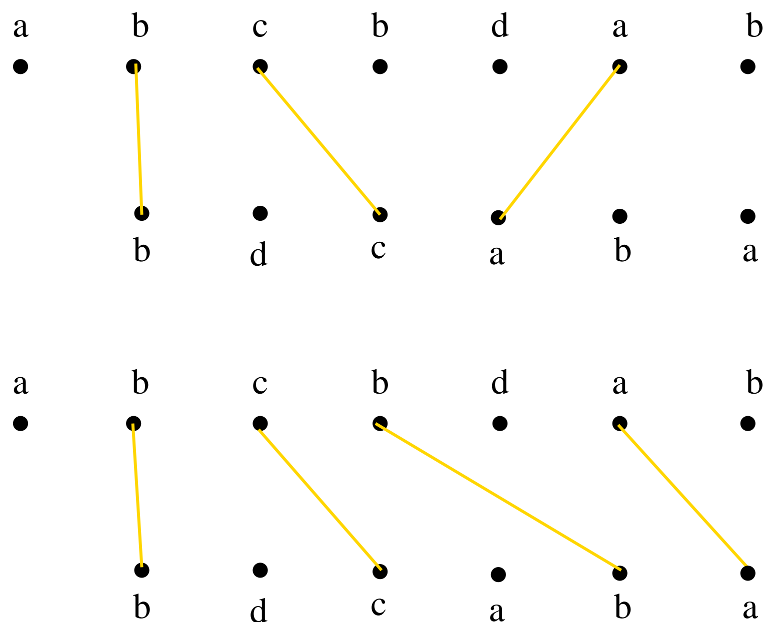
بزرگ‌ترین زیردنباله‌ی مشترک



$Z =< z_۱, z_۲, \cdots, z_k >$ زیردنباله‌ی $X =< x_۱, x_۲, \cdots, x_m >$ است اگر دنباله‌ی اکیداً صعودی $< i_۱, i_۲, \cdots, i_k >$ از اندیس‌های عناصر X وجود داشته باشد به‌طوری‌که برای $j = ۱, \ldots, k$ داشته باشیم: $z_i = x_{i_j}$.

مثلاً $Z =< b, c, d, b >$ یک زیردنباله از $X =< a, b, c, b, d, a, b >$ است که دنباله‌ی اندیس‌های مربوط $< ۲, ۳, ۵, ۷ >$ می‌باشد.

مسئله: پیدا کردن Z «**بزرگ‌ترین زیردنباله‌ی مشترک**» (Longest Common Subsequence (LCS)) **برای دو دنباله‌ی داده شده‌ی** X **و** Y **است.**

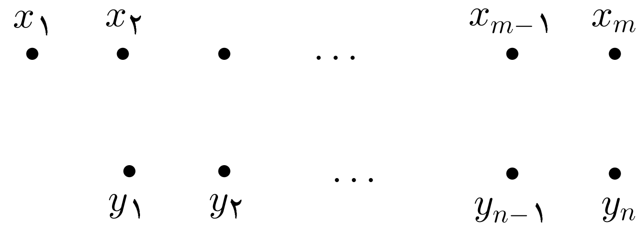


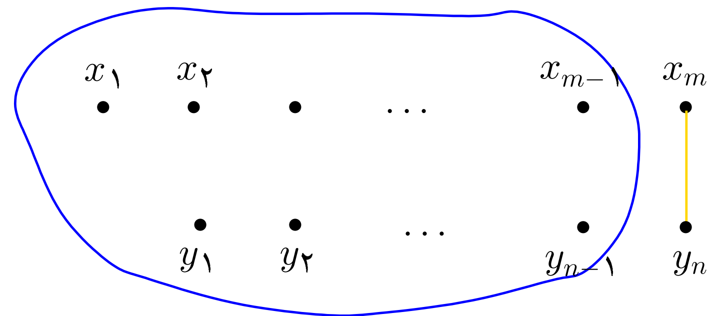
ورودی: دو دنباله‌ی $X = < x_۱, x_۲, \cdots, x_m >$ **و** $Y = < y_۱, y_۲, \cdots, y_n >$

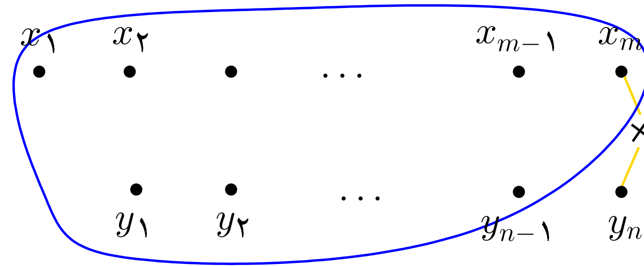
خروجی: LCS **این دو دنباله.**

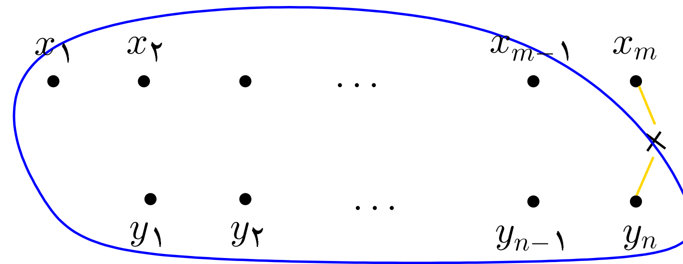
کاربرد: `diff file1 file2`

زیرساختار بهینه






$x_۱$
$x_۲$
$x_{m-۱}$
x_m
$y_۱$
$y_۲$
$y_{n-۱}$
y_n


$x_۱$
$x_۲$
$x_{m-۱}$
x_m
$y_۱$
$y_۲$
$y_{n-۱}$
y_n

یک دنباله: $X =< x_1, x_2, \cdots, x_m >$

i **امین پیشوند (prefix)** X **برای** $i = 1..m$: $X_i \equiv< x_1, x_2, \cdots, x_m >$

مثلاً برای $X =< A, B, C, B, D, A, B >$، **داریم:** $X_4 =< A, B, C, B >$.

قضیه‌ی ۱ **اگر** $X =< x_۱, x_۲, \cdots, x_m >$ **و** $Y =< y_۱, y_۲, \cdots, y_n >$ **دنباله‌های ورودی باشند و** LCS **این دو** $Z =< z_۱, z_۲, \cdots, z_k >$ **باشد داریم:**

۱. **اگر** $x_m = y_n$، **داریم** $z_k = x_m = y_n$ **و** $Z_{k-۱}$ **برابر** LCS $X_{m-۱}$ **و** $Y_{n-۱}$ **است.**

۲. **اگر** $x_m \neq y_n$، **آن‌گاه از** $z_k \neq x_m$ **نتیجه می‌گیریم که** Z **برابر** LCS $X_{m-۱}$ **و** Y **است.**

۳. **اگر** $x_m \neq y_n$، **آن‌گاه از** $z_k \neq y_n$ **نتیجه می‌گیریم که** Z **برابر** LCS $Y_{n-۱}$ **و** X **است.**

راه‌حل بازگشتی

راه‌حل بازگشتی منجر به حل تکراری زیرمسئله‌های مختلف می‌شود.

راه‌حل پویا

اگر $c[i,j]$ طول LCS برای X_i و Y_j باشد،

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

```
for i := 1 to m do c[i,0] := 0;
for j := 1 to m do c[0,j] := 0;
for i := 1 to m do
    for j := 1 to n do
        if x_i = y_j then begin
            c[i,j] := c[i-1,j-1] + 1;
            b[i,j] := '↖'
        end
        else if c[i-1,j] ≥ c[i,j-1]
            then begin
                c[i,j] := c[i-1,j];
                b[i,j] := '↑'
            end
            else begin
                c[i,j] := c[i,j-1];
                b[i,j] := '←'
            end
```

الگوریتم ارائه شده از مرتبه‌ی زمانی $\mathcal{O}(mn)$ و میزان حافظه‌ی مصرفی آن نیز $\mathcal{O}(mn)$

است.

مثال: $X =< A, B, C, B, D, A, B >$ و $Y =< B, D, C, A, B, A >$

i	j	0	1	2	3	4	5	6
		y_j	b	d	c	b	b	a
0	x_i	0	0	0	0	0	0	0
1	a	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	b	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	c	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	b	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	d	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	a	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	b	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

ساختن LCS

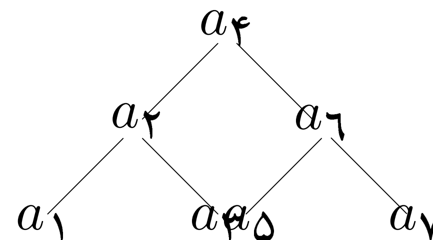
با توجه به اطلاعات ذخیره‌شده در ماتریس b می‌توان LCS را با فراخوانی $\text{Print-LCS}(b, X, m, n)$ ساخت.

```
PRINT-LCS(b, X, i, j)
if (i = 0 or j = 0)
   then return;
if b[i, j] = '↖'
   then begin
          PRINT-LCS (b, X, i − 1, j − 1);
          print x_i
   end
   else if b[i, j] = '↑'
         then PRINT-LCS (b, X, i − 1, j)
         else PRINT-LCS (b, X, i − 1, j)
```

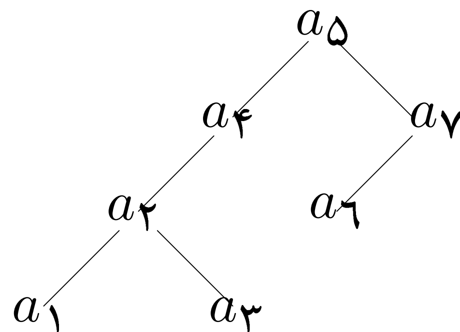
این الگوریتم از مرتبه‌ی $\mathcal{O}(m+n)$ است.

درخت دودویی جست‌وجوی بهینه (Optimal BST)

درخت شامل ۷ عنصر $a_1 < a_2 < \cdots < a_7$ و احتمال جست‌وجو برای این عناصر به‌ترتیب برابر $\frac{2}{24}$، $\frac{1}{24}$، $\frac{3}{24}$، $\frac{4}{24}$، $\frac{5}{24}$، $\frac{3}{24}$ $\frac{6}{24}$ باشد

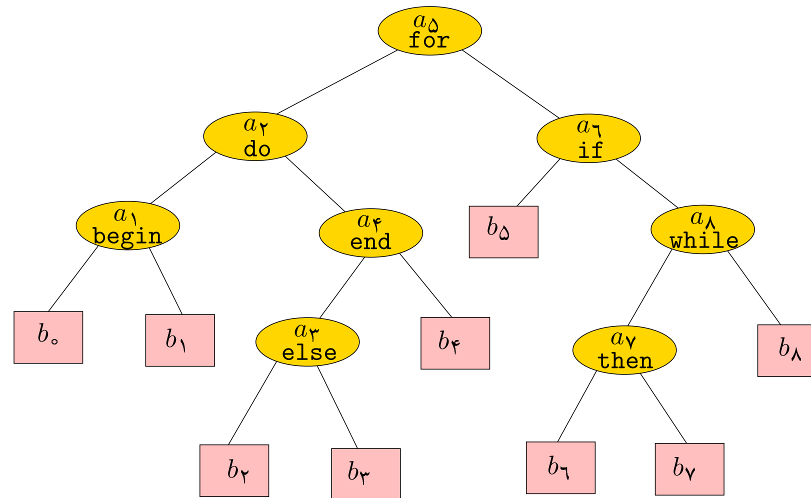


درخت متوازن با میانگین زمان جست‌وجوی $\frac{64}{26}$



درخت نامتوازن با میانگین زمان جست‌وجوی $\frac{58}{26}$

مثال‌ها: کتاب‌خانه، جدول نمادها



یک درخت دودویی جست‌وجو برای جدول نمادها

مسئله در حالت کلی

جست‌وجوی موفق و ناموفق (a_i و b_i): عنصر داخلی و خارجی

$$b_0 < a_1 < b_2 < \cdots < a_{i-1} < b_i < a_i < \cdots < b_{n-1} < a_n < b_n$$

لم. تعداد عناصر خارجی، برای یک درخت دودویی جست‌وجو با n عنصر برابر $n+1$ است.

تعریف دقیق ورودی مسئله

- $a_1 < a_2 < \cdots < a_n$ عناصر درخت
- p_i احتمال جست‌وجو موفق برای a_i $(i = 1..n)$
- q_i احتمال جست‌وجوی ناموفق برای b_i، اگر $a_i < b_i < a_{i+1}$ $(i = 1..n-1)$
- q_0 احتمال جست‌وجوی ناموفق برای $b_0 < a_1$
- q_n احتمال جست‌وجوی ناموفق برای $a_n < b_n$

خروجی مسئله

با داشتن p_i ها و q_i ها یک درخت دودویی جست‌وجوی بهینه بسازید که متوسط زمان جست‌وجو (اعم ازموفق یا نا موفق) در آن کمینه شود

این مقدار

$$\sum_{i=1}^{n} p_i(1 + depth(a_i)) + \sum_{i=0}^{n} q_i(depth(b_i))$$

که $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$

تعریف زیرمسئله

- زیر مسئله‌ی T_{ij}
درخت OBST برای $a_{j} < \cdots < a_{i+۱}$ با ورودی $q_i, p_{i+۱}, q_{i+۱}, \ldots, p_j, q_j$
$T_{\circ n}$ مسئله‌ی اصلی است.
- C_{ij} : هزینه‌ی T_{ij}

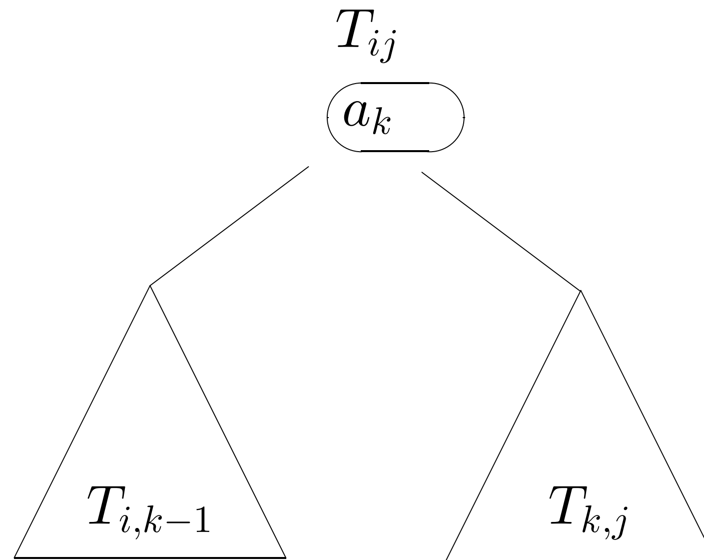
$$C_{ij} = \sum_{r=i+۱}^{j} p_r[depht(a_r) + ۱] + \sum_{r=\circ}^{j} q_r[depth(b_r)]$$

- $C_{ij}^{(k)}$: C_{ij} اگر a_k ریشه‌ی T_{ij} باشد.
- w_{ij} : وزن درخت T_{ij}، برابر مجموع احتمال‌های p_i و q_i در T_{ij}:

$$w_{ij} = q_i + \sum_{r=i+1}^{j} (p_r + q_r)$$

- r_{ij} ریشه‌ی T_{ij}

زیر درخت T_{ij} برای $a_{i+1} < \cdots < a_j$



$i < k \leq j$

حل زیر مسئله‌ی T_{ij}

اگر a_k (برای $i < k \leq j$) ریشه باشد، داریم:

$$\begin{aligned} C_{ij}^{(k)} &= (C_{i,k-1} + w_{i,k-1}) + (C_{kj} + w_{kj}) + p_k \\ &= C_{i,k-1} + C_{kj} + w_{ij} \end{aligned}$$

و داریم:

$$C_{ij} = \min_{i<k\leq j} C_{ij}^{(k)}$$

در شروع برای T_{ii} داریم $\Longleftarrow$ $w_{ii} = q_i$ و $c_{ii} = 0$

الگوریتم

$\underline{\text{OBST}}(p_1, \ldots, p_n, q_0, \ldots, q_n)$

1 **for** $i \leftarrow 0$ **to** n
2   **do** $w_{ii} \leftarrow q_i$
3         $c_{ii} \leftarrow 0$
4 **for** $l \leftarrow 1$ **to** n
5   **do for** $i \leftarrow 0$ **to** $n - l$
6   **do** $j \leftarrow i + l$
7             $w_{ij} \leftarrow w_{i,j-1} + p_j + q_j$
8             $c_{ij} \leftarrow \min\limits_{i<k\leq j}\{c_{i,k-1} + c_{kj} + w_{ij}\}$
9             $r_{ij} \leftarrow$ the k for which above is mininimum

تحلیل

الگوریتم فوق از $\Theta(n^۳)$ است.

می‌توان این الگوریتم را در $\mathcal{O}(n^۲)$ حل کرد

اگر r_{ij} شماره‌ی ریشه‌ی جواب بهینه‌ی زیر مسئله‌ی T_{ij} باشد،

$$r_{i,j-۱} \leq r_{ij} \leq r_{i+۱,j}$$

استفاده از رابطه‌ی فوق در الگوریتم پویا راه‌حل را $\mathcal{O}(n^۲)$ می‌کند.

مثال

$$n = 4$$

$$a_1 < a_2 < a_3 < a_4$$

$$p_1 = \frac{1}{4}$$

$$p_2 = \frac{1}{8}$$

$$p_3 = p_4 = \frac{1}{16}$$

$$q_0 = \frac{1}{8}$$

$$q_1 = \frac{3}{16}$$

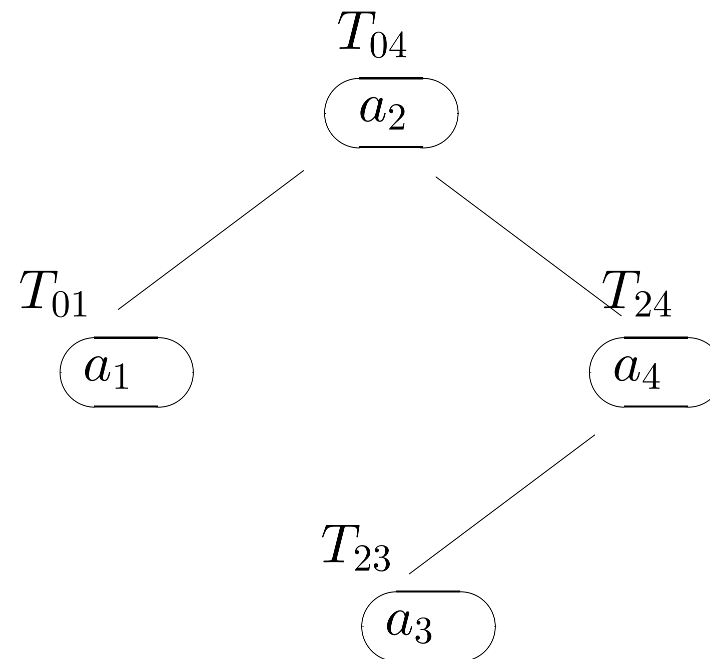
$$q_2 = q_3 = q_4 = \frac{1}{16}$$

جدول‌ها

مراحل مختلف الگوریتم گفته‌شده در جدول زیر می‌آید:

	۰	۱	۲	۳	۴
۰	$c_{00} = 0$ $w_{00} = 2$	$c_{01} = 9$ $w_{01} = 9$ $r_{01} = a_1$	$c_{02} = 18$ $w_{02} = 12$ $r_{02} = a_1$	$c_{03} = 25$ $w_{03} = 14$ $r_{03} = a_1$	$c_{04} = 33$ $w_{04} = 16$ $r_{04} = a_2$
۱		$c_{11} = 0$ $w_{11} = 3$	$c_{12} = 6$ $w_{12} = 6$ $r_{12} = a_2$	$c_{13} = 11$ $w_{13} = 8$ $r_{13} = a_2$	$c_{14} = 18$ $w_{14} = 10$ $r_{14} = a_2$
۲			$c_{22} = 0$ $w_{22} = 1$	$c_{23} = 3$ $w_{23} = 3$ $r_{23} = a_3$	$c_{24} = 8$ $w_{24} = 5$ $r_{24} = a_4$
۳				$c_{33} = 0$ $w_{33} = 1$	$c_{34} = 3$ $w_{34} = 3$ $r_{14} = a_4$
۴					$c_{44} = 0$ $w_{44} = 1$

درخت مثال



مسئله‌های کوله‌پشتی (Knapsack)

ورودی:

- یک کوله‌پشتی با توانایی حمل M واحد وزن بار
- N نوع بار
- تعداد بار نوع i برابر N_i
- وزن بار نوع i ام W_i
- ارزش بار نوع i ام C_i

فرض: M و W_i اعداد صحیح هستند.

می خواهیم کوله‌پشتی را با این بارها کاملاً یا تا حد امکان پر کنیم به‌طوری‌که اگر بارها ارزش داشته باشند، مجموع ارزش بارها بیشینه شود.

حالت اول: $C_i = ۰$، $N_i = ۱$، و کوله‌پشتی پر شود

این مسئله به 0-1-Knapsack مشهور است که یک مسئله‌ی NP-Complete است.

راه‌حل پس‌گرد (Backtracking)

پر کردن کوله‌پشتی با اندازه‌ی M از بارهای i تا N

```
KNAPSACK(M, i)
      ▷ returns true if M can be filled from loads i to N
  1  if M = 0
  2    then return  true
  3  if  M < 0 or i > N
  4    then return  false
      ▷ load i is a candidate
  5  if KNAPSACK (M − W_i, i + 1)
  6    then PRINT i, W_i
  7          return  true
  8    elsereturn  KNAPSACK (M, i + 1)
```

بدترین حالت هنگامی است که مسئله جواب نداشته باشد.

$$T(n) = ۲T(n-۱) + \Theta(۱) \Rightarrow T(n) = \Theta(۲^n)$$

راه حل کورکورانه

```
BlindKnapSack()
1  for i ← 0 to 2^N − 1
2      do find the N-bit binary representation of i
3         find the loads with 1's in binary form of i
4         if the sum of weights of the selected loads is equal to M
5           then this is one solution
```

در بدترین حالت این دو الگوریتم مانند هم عمل می‌کنند.

راه‌حل پویا

زیرمسئله: $S[i, j]$، پر کردن کوله‌پشتی j از بارهای ۱ تا i

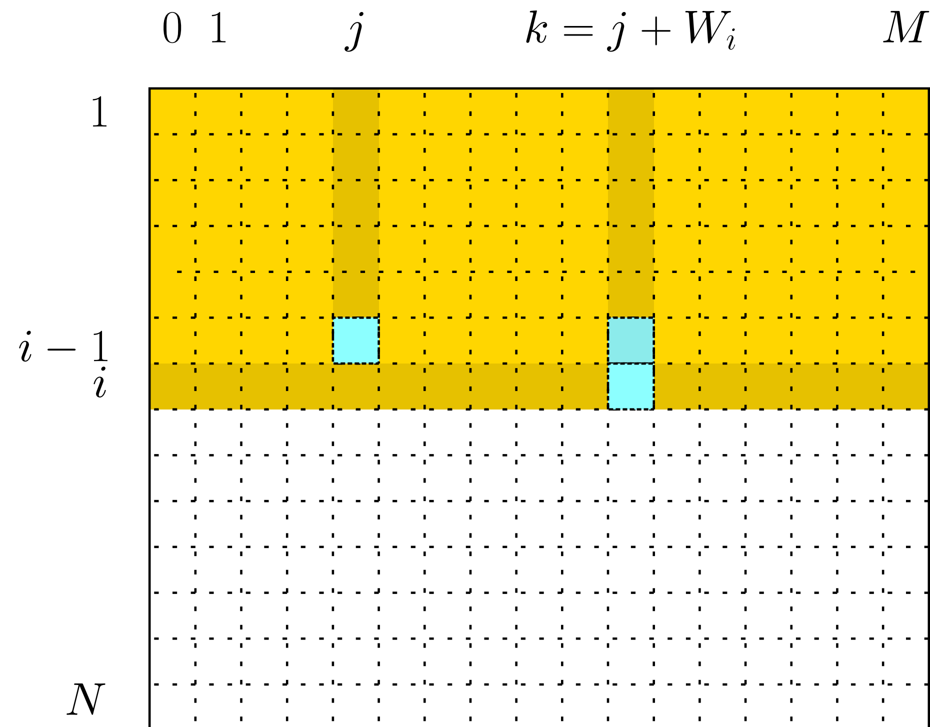
آیا همه‌ی زیرمسئله‌ها باید بهینه حل شوند؟

جواب: بله. اگر مسئله بهینه حل شده باشد، یک بار از جواب برداریم، آن زیرمسئله هم باید بهینه حل شود.

فرض: $W_i \leq M$

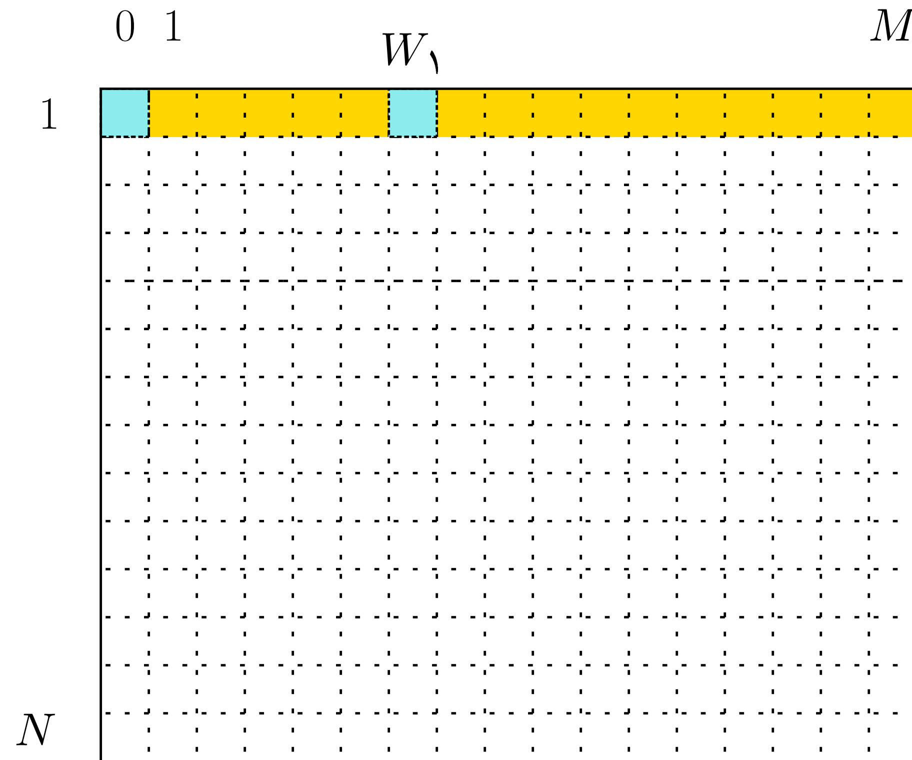
ماتریس $S_{M \times N}$:

$S[i, j]$: اگر کوله‌پشتی j را نتوان پر کرد، این مقدار صفر است و گرنه برابر $k > 0$ است که k شماره‌ی آخرین باری است که در کوله‌پشتی قرار می‌دهیم.

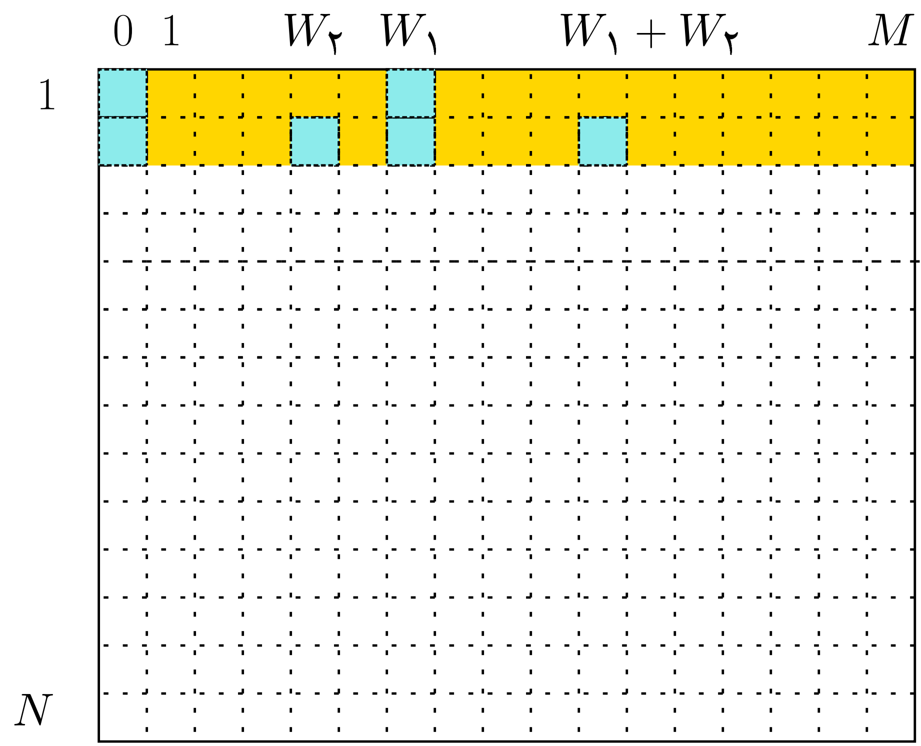


$$S[i,k] = S[i-1,k] \vee S[i-1,j]$$

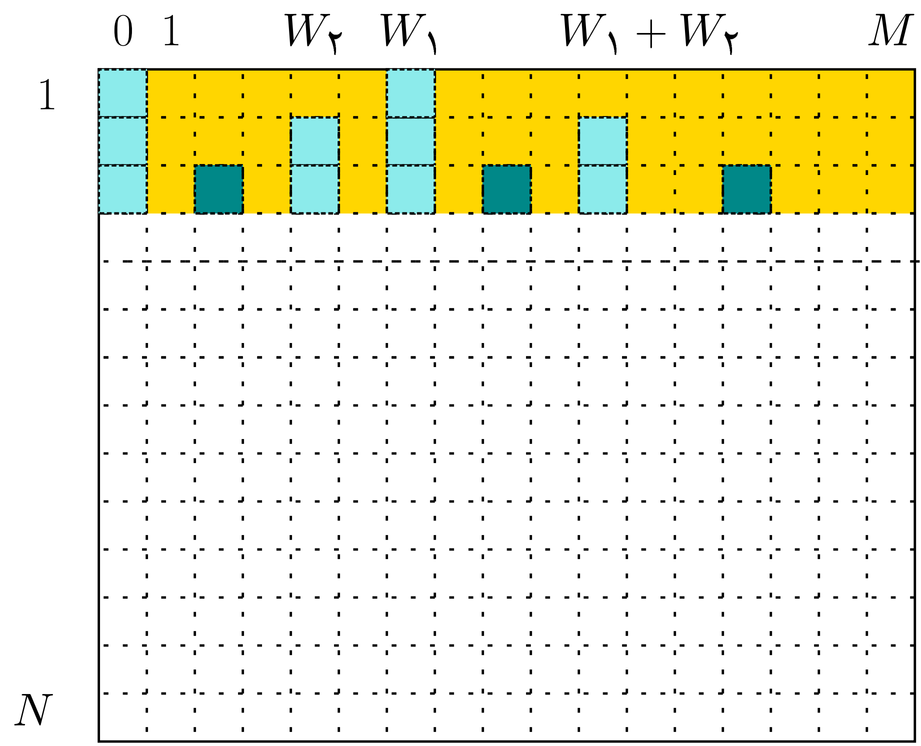
$S[i,j]$: آیا می‌توان کوله‌پشتی با اندازه‌ی j را با بارهای از ۱ تا i پر کرد؟



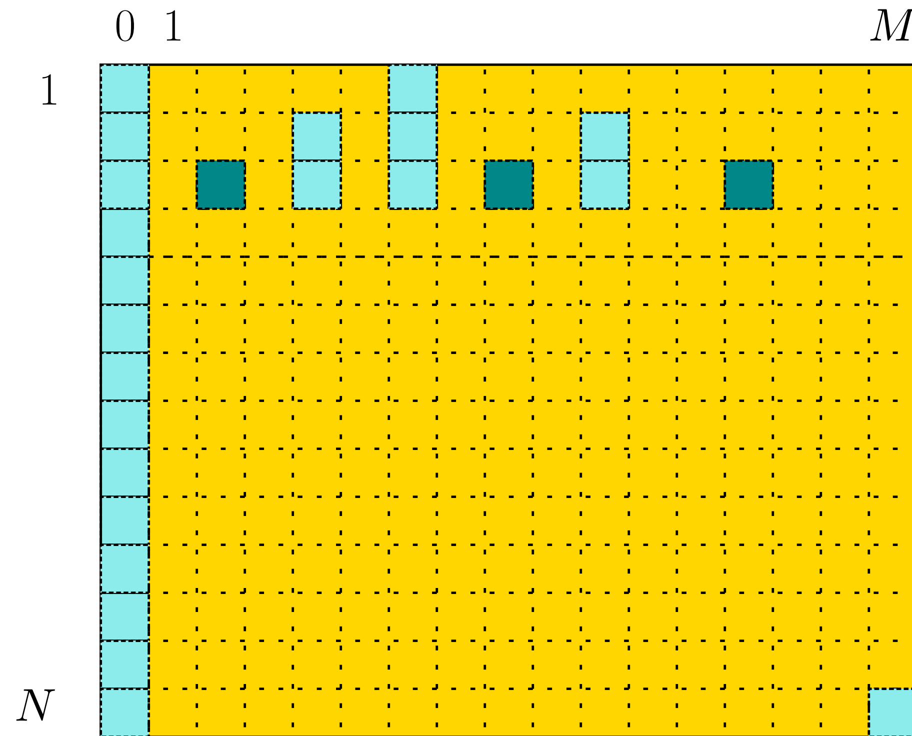
شروع، سطر اول



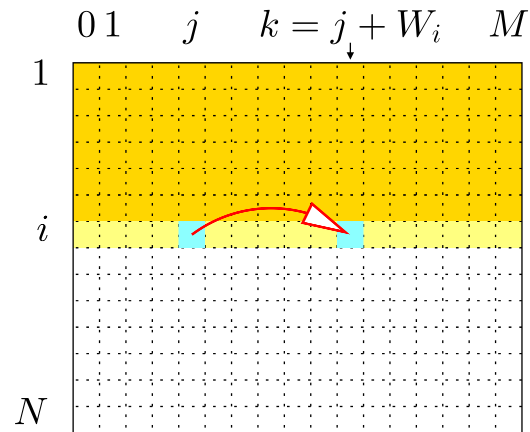
سطر دوم



سطر سوم



سطر آخر



نوع دیگر پر کردن:

اگر $S[i,k]$ **مقدار نداشته باشد،** $S[i,k] \leftarrow i$

پر کردن ماتریس

شرح الگوریتم: (فرض $k = j + W_i$)

- **در سطر اول داریم: $S[1, W_1] = 1$**
- **سطر i بر اساس سطر $i - 1$ پر می‌شود.**
- **ابتدا همه‌ی مقادیر سطر $i - 1$ در سطر i کپی می‌شود.**
- **از $S[i, j]$، $S[i, k]$ را پر می‌کنیم، اگر $S[i, k]$ توسط بارهای قبلی پر نشده باشد.**
- **اگر $j = 0$، یا $i <> S[i, j]$، $S[i, k]$ را فقط با یک عدد بار i پر می‌کنیم.**

```
KnapSack-1 (M, Weight)
1  N ← length[Weight]
2  for i ← 1 to N
3      do for j ← 0 to M − W_i
4             do S[i, j] ← S[i − 1, j]
5                k ← j + W_i
6                if (j = 0) or
                    ((S[i, j] > 0) and (S[i, j] <> i) and (S[i − 1, k] =
0))
7                  then ▷ we can fill S[i, k], putting load i last
8                       S[i, k] ← i
```

با یک آرایه همین کار را می‌توان انجام داد.

KnapSack-1 $(M, Weight)$
1 $N \leftarrow length[Weight]$
2 **for** $i \leftarrow 1$ **to** N
3   **do for** $j \leftarrow 0$ **to** $M - W_i$
4   **do** $k \leftarrow j + W_i$
5   **if** $(j = 0)$ **or**
                $((S[j] > 0)$ **and** $(S[j] <> i)$ **and** $(S[k] = 0))$
6   **then** ▷ we can fill $S[i, k]$, putting load i last
7                   $S[k] \leftarrow i$

```
PRINT-ONE-RESULT (S, j)
      ▷ print the one solution for knapsack of size j
  1  if j = 0
  2    then return
  3  k ← S[j]
  4  if k > 0
  5    then PRINT k
  6         PRINT-ONE-RESULT (j − W_k)
```

به‌دلیل این که اگر مسئله‌ی $S[i,j]$ بیش از یک جواب داشته باشد (یکی با استفاده از بار i و دیگری با بارهای کم‌تر از آن) ما اولویت را به جواب $S[i-1,j]$ می‌دهیم، راه‌حل فوق درست است. یعنی اگر $S[j]=k$، حتماً $S[j-k]<k$ یا $j=k$.

حالت دوم (کلی): $C_i = ۰$، $۱ < N_i < \infty$، و کاملاً پر

(مهم) زیرمسئله: $S[i,j]$. برای حل بهینه‌ی $S[i,j]$، زیرمسئله‌های $S[i-۱,j]$ و $S[i-۱,j-W_i]$ هم باید بهینه حل شده باشند.

پس مسئله راه‌حل پویا دارد و آن‌را با یک آرایه هم می‌شود حل کرد.

در این حالت درایه‌ی $S[j]$ شامل دو مولفه است:

- $last(S[j])$: شماره‌ی آخرین باری که در کوله‌پشتی j قرار می‌گیرد. اگر این کوله‌پشتی را نتوان پر کرد، این مولفه برابر صفر است.
- $Num(S[j])$: تعداد استفاده شده از بار $last(S[j])$ در این کوله‌پشتی.

N_i: تعداد بار از نوع i.

شرح الگوریتم

برای هر i سطر iام ماتریس را پر می‌کنیم. در این صورت

- برای $j = 1..M - W_i$، و با فرض $k = j + W_i$، از $S[j]$ تحت شرایطی $S[k]$ را با قرار دادن یک عدد دیگر از بار iام پر می‌کنیم.
- $S[k]$ فقط در صورتی که قبلاً پر نشده باشد، پر می‌شود.
- بار i برای اولین بار استفاده می‌شود، اگر $j = 0$، یا $S[j]$ پر باشد ولی آخرین بار استفاده شده در آن i نباشد.
- بار i مجدداً استفاده می‌شود، اگر به‌تعداد کافی موجود باشد.

توجه: بارها به‌ترتیب شماره‌اشان در کوله‌پشتی قرار می‌گیرند.

```
KnapSack-2 (M, Weight)
 1  N ← length[Weight]
 2  for j ← 1 to M
 3      do last(S[j]) ← 0
 4  for i ← 1 to N
 5      do for j ← 1 to M − W_i
 6              do k ← j + W_i
 7                  if last(S[k]) = 0
 8                    then  ▷ trying to fill S[k]
 9                          if (j = 0) or ((last(S[j]) > 0)
10                            then if last(S[j]) ≠ i))
11                                    then ▷ use load i for the first time
12                                         last(S[k]) ← i
13                                         Num(S[k]) ← 1
14                                    elseif Num(S[j]) < N_i
15                                         then last(S[k]) ← i
16                                              Num(S[k]) ← Num(S[j]) + 1
```

```
PRINT-ONE-RESULT(S, j)
      ▷ print the one solution for knapsack of size j
  1  if j = 0
  2    then return
  3  k ← last(S[j])
  4  if k > 0
  5    then PRINT k
  6         PRINT-ONE-RESULT (j − W_k)
```

حالت سوم: $C_i = \circ$، $N_i = \infty$، و کاملاً پر

در این حالت بار نوع i می‌تواند بیش از یک بار استفاده شود.

```
KnapSack-3 (M, Weight)
1  N ← length[Weight]
2  S ← 0
3  for i ← 1 to N
4      do for j ← 0 to M − W_i
5             do k ← j + W_i
6                if j = 0 or S[j] ≠ 0
7                  then S[k] ← i
```

رویه‌ی Print-One-Result در این حالت نیز درست کار می‌کند.

مسئله‌ی خرد کردن پول (اگر نخواهیم تعداد سکه‌ها کمینه باشد) یک مسئله‌ی کوله‌پشتی با $n_i = \infty$ است.

حالت چهارم: $N_i = ۱$، $C_i > ۰$ و کاملاً پر

جواب بهینه‌ی مسئله‌ی $S[i, j]$ برابر ماکزیمم جواب‌های زیرمسئله‌های $S[i - ۱, j]$ و $S[i - ۱, j - W_i]$

در این دو زیرمسئله از بار i استفاده نمی‌شود و باید بهینه حل شوند.

زیرمسئله‌ی بهینه: راه‌حل پویا

$value(S[i, j])$: **بیشترین ارزش بارهای از نوع ۱ تا i برای پرکردن کوله‌پشتی j.**

$last(S[i, j])$: **آخرین بار در کوله‌پشتی فوق**

```
Knapsack-4(M, Weight)
 1  N ← length[Weight]
 2  for i ← 1 to N
 3      do for j ← 0 to M
 4              do Value(S[i, j]) ← 0
 5  for i ← 1 to N
 6      do for j ← 1 to M − W_i
 7              do k ← j + W_i
 8                 if j = 0 or Value(S[i − 1, j]) > 0
 9                   then ▷ we may be able to fill S[i, k]
10                        if Value(S[i − 1, k]) < Value(S[i − 1, j]) + C_i
11                          then last(S[i, k]) ← i
12                               Value(S[i, k]) ← Value(S[i − 1, j]) + C_i
13                          else S[i, k] ← S[i − 1, k]
```

اگر با ماتریس حل کرده باشیم، جواب را نیز می‌توان نوشت:

```
PRINT-ONE-RESULT (i, j)
1  if j = 0
2    then return
3  k ← last(S[i, j])     ▷ k has to be > 0
4  PRINT k
5  PRINT-ONE-RESULT (k − 1, j − W_k)
```

این مسئله را می‌توان با یک آرایه هم حل کرد. ولی بارهای استفاده شده را نمی‌توان به‌دست آورد. چرا؟

آرایه‌ی $S[0..M]$ همان ارزش است.

```
KnapSack-4(M, Weight)
1  N ← length[Weight]
2  for j ← 0 to M
3      do S[j] ← 0
4  S[W_1] ← C_1
5  for i ← 2 to N
6      do for k ← M downto W_i
7              do j ← k − W_i
8                  if S[k] < S[j] + C_i
9                     then S[k] ← S[j] + C_i
```

این مسئله را می‌توان با یک آرایه هم حل کرد. ولی بارهای استفاده شده را نمی‌توان به‌دست آورد. چرا؟

برای این‌که اگر k آخرین بار برای پرکردن کوله‌ی j باشد، ممکن است برای پرکردن بهینه‌ی $j - W_k$ هم از بار k استفاده شده باشد.

حالت پنجم: $n_i < \infty$، $C_i > ۰$ کاملاً پر

در این حالت زیر مسئله‌ی $S[i,j]$ از جواب سه زیرمسئله‌ی $S[i-۱,j]$ (اگر در جواب بهینه‌ی آن اصلاً بار نوع i استفاده نشده باشد)، و یا $S[i-۱,j-W_i]$ (اگر در جواب بهینه فقط یک عدد از بار نوع i استفاده شده باشد) یا از $S[i,j-W_i]$ (اگر در جواب بهینه بیش از یک عدد از بار نوع i استفاده شده باشد، به‌شرط آن‌که به‌تعداد کافی از این بار موجود باشد) به‌دست می‌آید. حالتی که بیش‌ترین ارزش را به‌دست دهد بهترین جواب است.

متأسفانه لزومی ندارد جواب $S[i,j-W_i]$ بهینه باشد.

پس مسئله راه‌حل پویای دو بعدی ندارد.

حل پویای سه‌بعدی

$S[i,j,k]$ با ارزش‌ترین جواب برای پر کردن کوله‌پشتی j از بارهای ۱ تا i که بار i حداکثر k عدد استفاده شده باشد.

برای پیدا کردن با ارزش‌ترین جواب ممکن است کوله‌پشتی کاملاً پر نشود.

در این صورت در سطر N ام دنبال بزرگ‌ترین ارزش می‌گردیم.