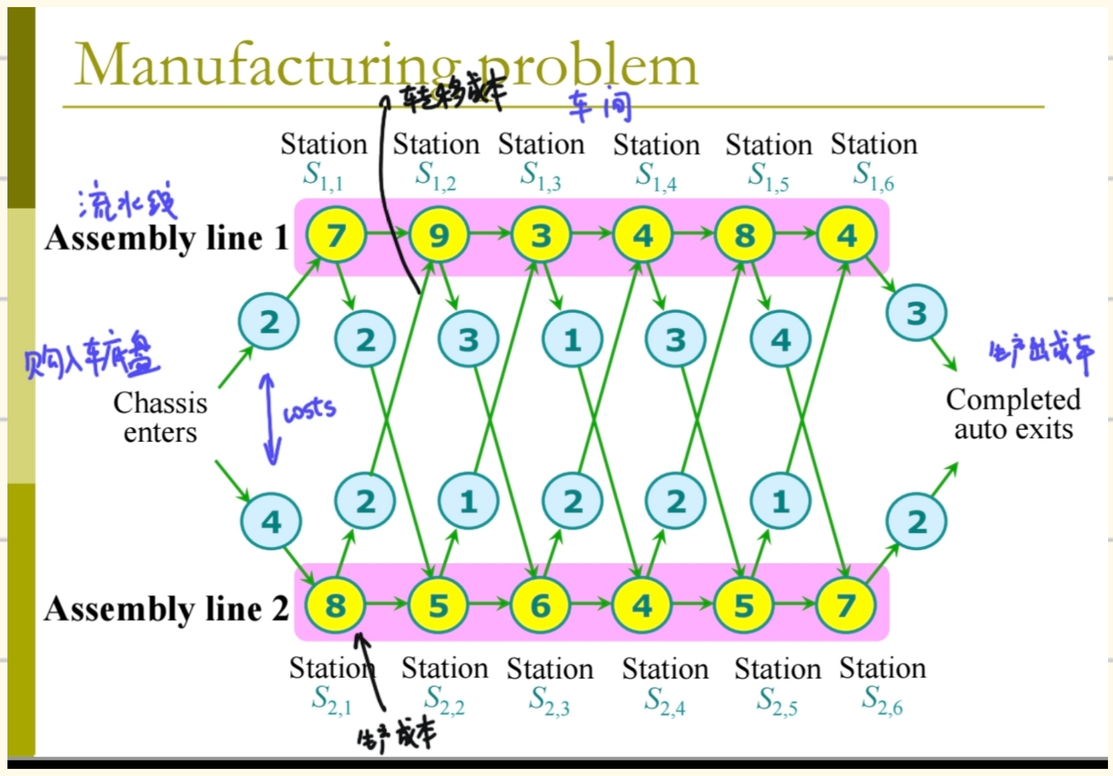
Data Structure Note10 Dynamic Programming

Example 1



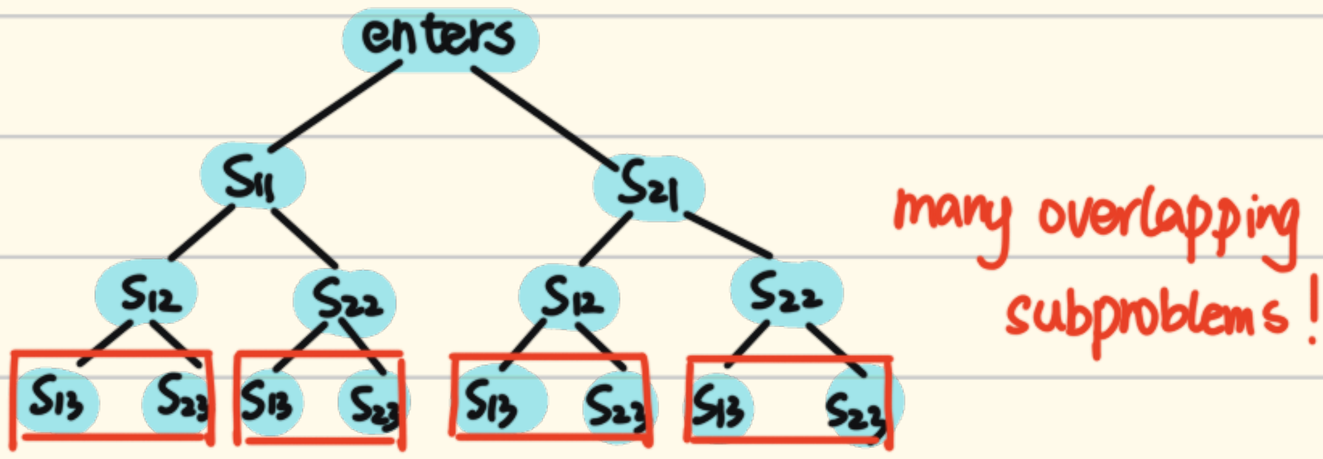
1. Bruteforce Algorithm

Check every way through a factory and choose the fastest way.

Analysis

- Checking = $O(n)$ time per way.
- 2^n possible ways to choose stations.
- Worst-case running time = $O(n2^n)$
  = exponential time.

It is infeasible!



many overlapping subproblems!

2. Dynamic Programming

① Some Notations:

$f_i[j]$ denote the fastest possible time to get a chassis from the starting point through station S_{ij}.

e_i denote an entry time for the chassis to enter assembly line i.

x_i denote an exit time for the completed auto to exit assembly line i.

$a_{i,j}$ denote the assembly time required at station S_{ij}.

$t_{i,j}$ denote the time to transfer a chassis away from assembly line i after through station S_{ij}.

Our ***ultimate goal*** is:

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$.

② Recursive Functions:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j=1 \\ \min\{f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}\} & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j=1 \\ \min\{f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}\} & \text{if } j \geq 2 \end{cases}$$

③ Memory to store the results of subproblems and solutions to the best answer

- Memory to store the results of subproblems:

  The best answer of the subproblem

- Solutions to the best answer

  The last choice we make to solve the subproblem, help to rebuild the solution

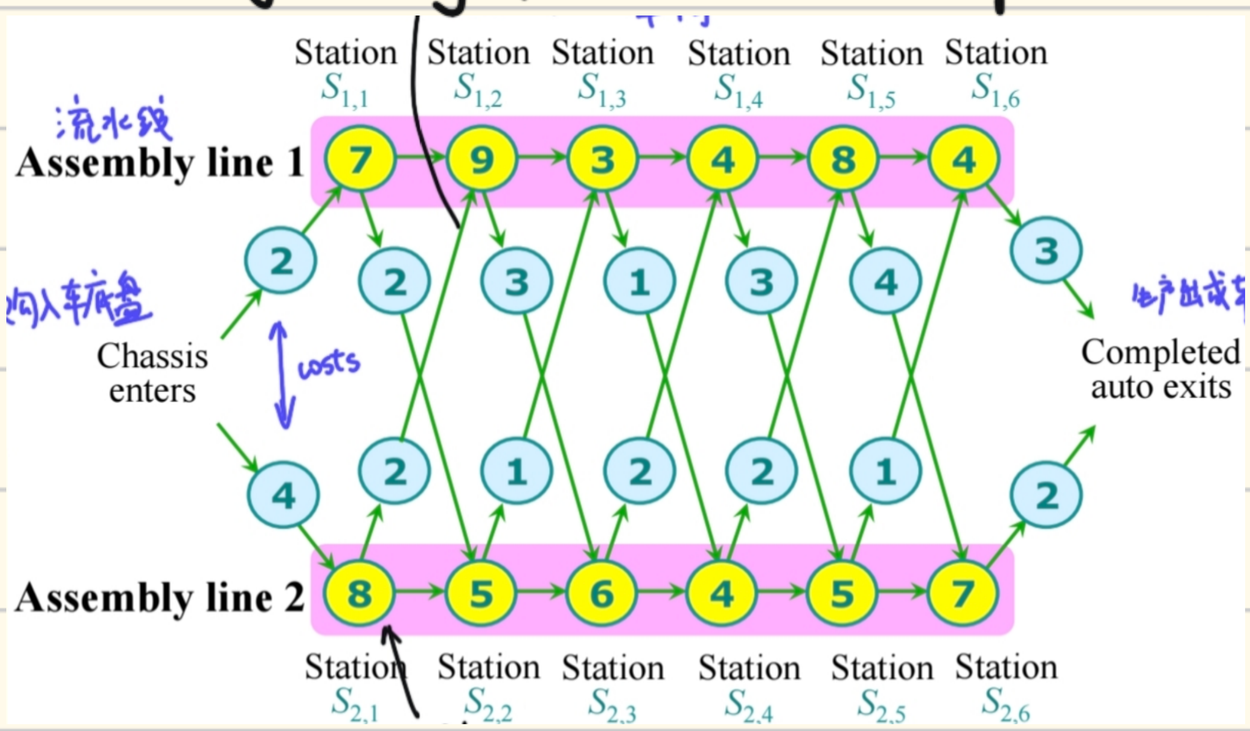
j	1	2	3	4	5	6
$f_1[j]$						
$f_2[j]$						

↑ The best answer of the subproblem

j	2	3	4	5	6
$l_1[j]$					
$l_2[j]$					

↑ The last choice we make to solve the subproblem

④ Dynamic Programming Process (Bottom up)



j	1	2	3	4	5	6
$f_1[j]$	9					
$f_2[j]$	12					

j	2	3	4	5	6
$l_1[j]$					
$l_2[j]$					

j	1	2	3	4	5	6
$f_1[j]$	9	18				
$f_2[j]$	12	16				

j	2	3	4	5	6
$l_1[j]$	1				
$l_2[j]$	1				

j	1	2	3	4	5	6
$f_1[j]$	9	18	20			
$f_2[j]$	12	16	22			

j	2	3	4	5	6
$l_1[j]$	1	2			
$l_2[j]$	1	2			

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24		
$f_2[j]$	12	16	22	25		

j	2	3	4	5	6
$l_1[j]$	1	2	1		
$l_2[j]$	1	2	1		

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	
$f_2[j]$	12	16	22	25	30	

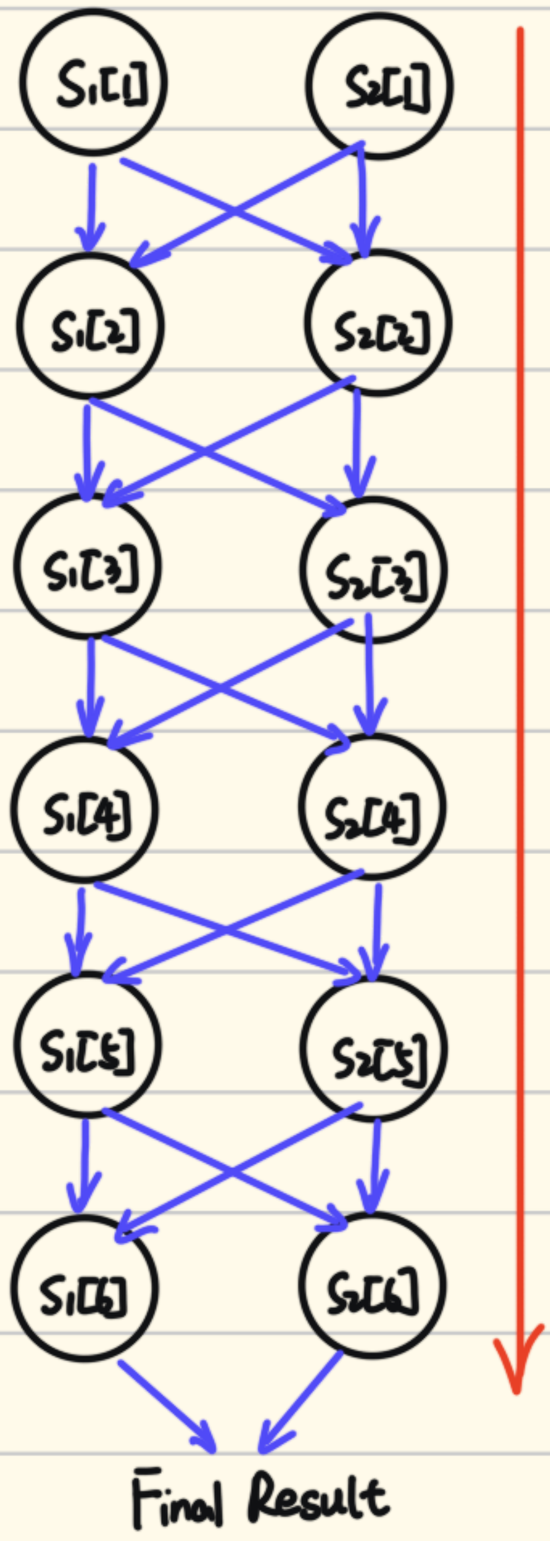
j	2	3	4	5	6
$l_1[j]$	1	2	1	1	
$l_2[j]$	1	2	1	2	

$35+3 < 37+2 \Rightarrow l^* = 1$

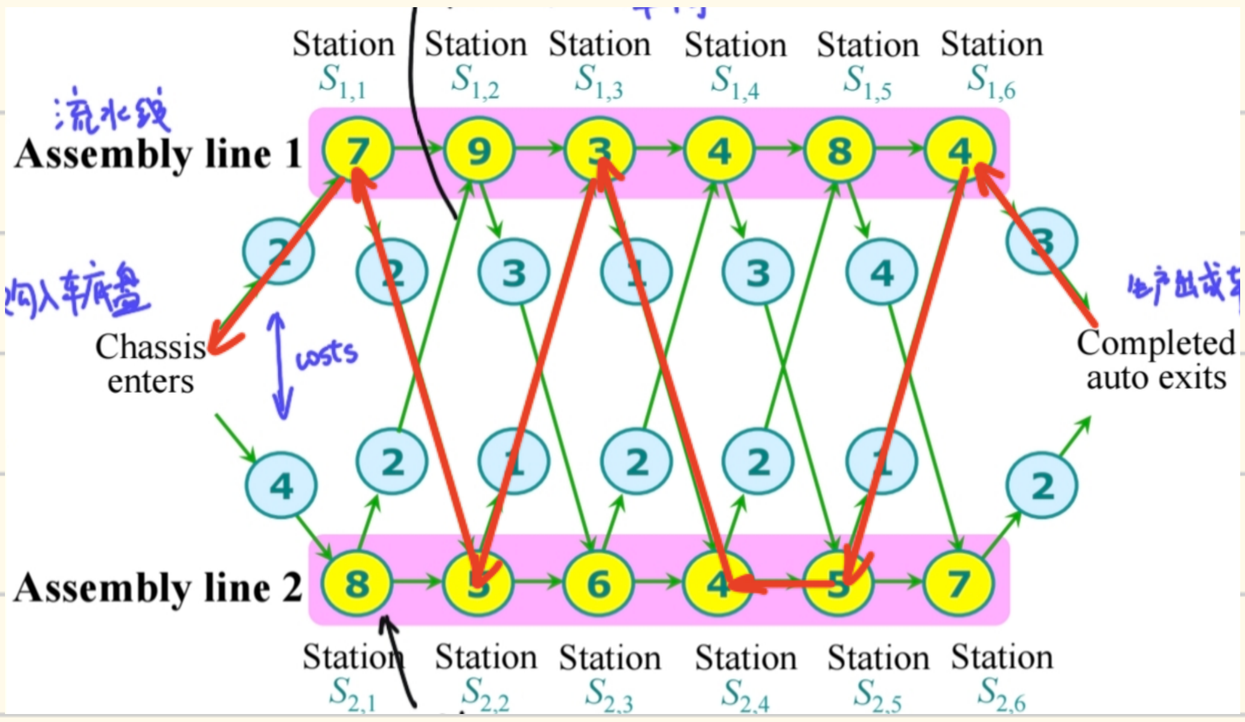
j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

Subproblem $S_i[j]$



Rebuild the solution:



Example 2

Rod-cutting problem

Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

$n = i_1 + i_2 + \cdots + i_k$

$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

We need to get the largest r

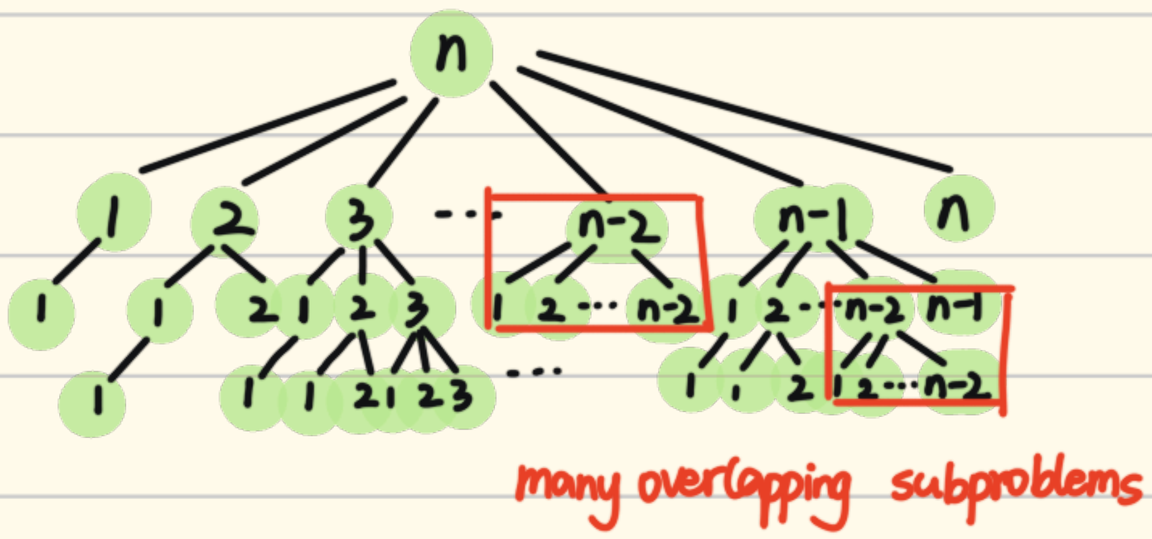
① Bruteforce Algorithm

Check every possible ways of cutting up a rod and choose the best one.

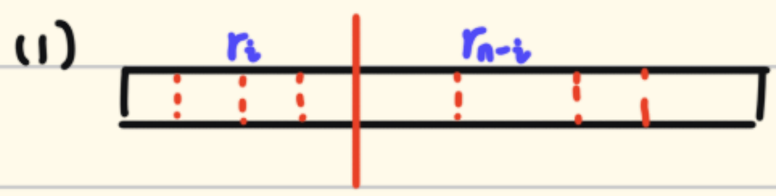
Analysis

- Checking $= O(n)$ time per way.
- 2^{n-1} possible ways to cut up a rod.
- Worst-case running time $= O(n2^{n-1})$
  $=$ exponential time.

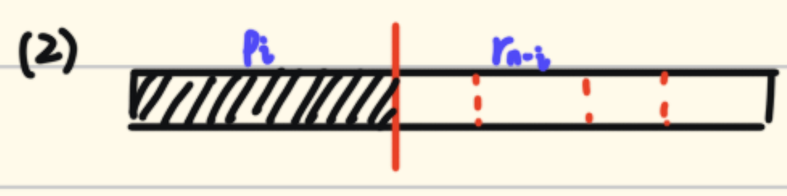
It is infeasible!



② Recursion Function



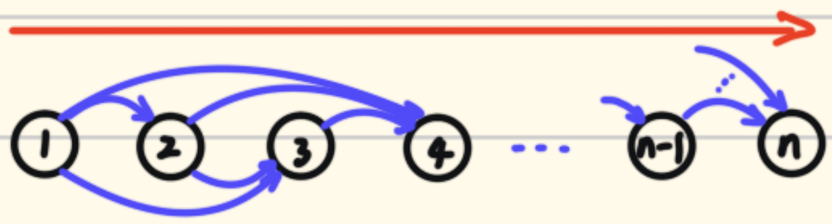
$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$



$r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$

→ This is better

③ Dynamic Programming Process (Bottom-up)



BOTTOM-UP-CUT-ROD(p, n)

```
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

record the result of the subproblem

record the last choice (length of the last piece) to rebuild the solution.

Rebuild the solution

PRINT-CUT-ROD-SOLUTION(p, n)

```
1  (r, s) = BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
4      Print s[n]
4      n = n - s[n]
```

r[7]

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Dynamic : give result of all length and then query. 以空间换时间

Example 3 : Matrix Multiplication

Calculate : $A_1A_2\cdots A_n$

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

对矩阵链加括号的方式会对乘积运算的代价产生巨大影响。我们先来分析两个矩阵相乘的代价。下面的伪代码给出了两个矩阵相乘的标准算法，它是 4.2 节 SQUARE-MATRIX-MULTIPLY 过程的推广。属性 *rows* 和 *columns* 是矩阵的行数和列数。

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2        error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4        for i = 1 to A.rows
5            for j = 1 to B.columns
6                c_ij = 0
7                for k = 1 to A.columns
8                    c_ij = c_ij + a_ik · b_kj
9  return C
```

两个矩阵 A 和 B 只有**相容**(compatible)，即 A 的列数等于 B 的行数时，才能相乘。如果 A 是 $p\times q$ 的矩阵，B 是 $q\times r$ 的矩阵，那么乘积 C 是 $p\times r$ 的矩阵。计算 C 所需时间由第 8 行的标量乘法的次数决定，即 pqr。下文中我们将用标量乘法的次数来表示计算代价。

我们以矩阵链$\langle A_1, A_2, A_3\rangle$相乘为例，来说明不同的加括号方式会导致不同的计算代价。假设三个矩阵的规模分别为 10×100、100×5 和 5×50。如果按$((A_1A_2)A_3)$的顺序计算，为计算 A_1A_2(规模 10×5)，需要做 10 · 100 · 5=5 000 次标量乘法，再与 A_3 相乘又需要做 10 · 5 · 50=2 500 次标量乘法，共需 7 500 次标量乘法。如果按$(A_1(A_2A_3))$的顺序，计算 A_2A_3(规模 100×50)，需 100 · 5 · 50=25 000 次标量乘法，A_1 再与之相乘又需 10 · 100 · 50=50 000 次标量乘法，共需 75 000 次标量乘法。因此，按第一种顺序计算矩阵链乘积要比第二种顺序快 10 倍。

矩阵链乘法问题(matrix-chain multiplication problem)可描述如下：给定 n 个矩阵的链$\langle A_1, A_2, \cdots, A_n\rangle$，矩阵 A_i 的规模为 $p_{i-1}\times p_i(1\leqslant i\leqslant n)$，求完全括号化方案，使得计算乘积 $A_1A_2\cdots A_n$ 所需标量乘法次数最少。 371

注意，求解矩阵链乘法问题并不是要真正进行矩阵相乘运算，我们的目标只是确定代价最低的计算顺序。确定最优计算顺序所花费的时间通常要比随后真正进行矩阵相乘所节省的时间(例如仅进行 7 500 次标量乘法而不是 75 000 次)要少。

计算括号化方案的数量

在用动态规划方法求解矩阵链乘法问题之前，我们先来说服自己——穷举所有可能的括号化方案不会产生一个高效的算法。对一个 n 个矩阵的链，令 $P(n)$表示可供选择的括号化方案的数量。当 $n=1$ 时，由于只有一个矩阵，因此只有一种完全括号化方案。当 $n\geqslant 2$ 时，完全括号化的矩阵乘积可描述为两个完全括号化的部分积相乘的形式，而两个部分积的划分点在第 k 个矩阵和第 $k+1$ 个矩阵之间，k 为 1，2，…，$n-1$ 中的任意一个值。因此，我们可以得到如下递归公式：

$$P(n)=\begin{cases}1 & \text{如果 } n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{如果 } n\geqslant 2\end{cases} \tag{15.6}$$

$P(n) \sim \Omega(4^n/n^{\frac{3}{2}})$

思考题 12-4 要求证明一个相似的递归公式产生的序列为**卡塔兰数**(Catalan numbers)，这个序列的增长速度为 $\Omega(4^n/n^{3/2})$。练习 15.2-3 要求证明递归公式(15.6)的结果为 $\Omega(2^n)$。因此，括号化方案的数量与 n 呈指数关系，通过暴力搜索穷尽所有可能的括号化方案来寻找最优方案，是一个糟糕的策略。

应用动态规划方法

下面用动态规划方法来求解矩阵链的最优括号化方案，我们还是按照本章开头提出的 4 个步骤进行：

1. 刻画一个最优解的结构特征。
2. 递归地定义最优解的值。

372

3. 计算最优解的值，通常采用自底向上的方法。
4. 利用计算出的信息构造一个最优解。

我们按顺序进行这几个步骤，清楚地展示针对本问题每个步骤应如何做。

步骤 1：最优括号化方案的结构特征

动态规划方法的第一步是寻找最优子结构，然后就可以利用这种子结构从子问题的最优解构造出原问题的最优解。在矩阵链乘法问题中，此步骤的做法如下所述。为方便起见，我们用符号 $A_{i..j}(i \leqslant j)$表示 $A_iA_{i+1}\cdots A_j$ 乘积的结果矩阵。可以看出，如果问题是非平凡的，即 $i<j$，那么为了对 $A_iA_{i+1}\cdots A_j$ 进行括号化，我们就必须在某个 A_k 和 A_{k+1}之间将矩阵链划分开(k 为 $i \leqslant k<j$ 间的整数)。也就是说，对某个整数 k，我们首先计算矩阵 $A_{i..k}$和 $A_{k+1..j}$，然后再计算它们的乘积得到最终结果 $A_{i..j}$。此方案的计算代价等于矩阵 $A_{i..k}$的计算代价，加上矩阵 $A_{k+1..j}$的计算代价，再加上两者相乘的计算代价。

下面我们给出本问题的最优子结构。假设 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案的分割点在 A_k 和 A_{k+1}之间。那么，继续对“前缀”子链 $A_iA_{i+1}\cdots A_k$ 进行括号化时，我们应该直接采用独立求解它时所得的最优方案。这样做的原因是什么呢？如果不采用独立求解 $A_iA_{i+1}\cdots A_k$ 所得的最优方案来对它进行括号化，那么可以将此最优解代入 $A_iA_{i+1}\cdots A_j$ 的最优解中，代替原来对子链 $A_iA_{i+1}\cdots A_k$进行括号化的方案(比 $A_iA_{i+1}\cdots A_k$ 最优解的代价更高)，显然，这样得到的解比 $A_iA_{i+1}\cdots A_j$原来的“最优解”代价更低：产生矛盾。对子链 $A_{k+1}A_{k+2}\cdots A_j$，我们有相似的结论：在原问题 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案中，对子链 $A_{k+1}A_{k+2}\cdots A_j$ 进行括号化的方法，就是它自身的最优括号化方案。

现在我们展示如何利用最优子结构性质从子问题的最优解构造原问题的最优解。我们已经看到，一个非平凡的矩阵链乘法问题实例的任何解都需要划分链，而任何最优解都是由子问题实例的最优解构成的。因此，为了构造一个矩阵链乘法问题实例的最优解，我们可以将问题划分为两个子问题($A_iA_{i+1}\cdots A_k$ 和 $A_{k+1}A_{k+2}\cdots A_j$ 的最优括号化问题)，求出子问题实例的最优解，然后将子问题的最优解组合起来。我们必须保证在确定分割点时，已经考察了所有可能的划分点，

373 这样就可以保证不会遗漏最优解。

步骤 2：一个递归求解方案

下面用子问题的最优解来递归地定义原问题最优解的代价。对矩阵链乘法问题，我们可以将对所有 $1 \leqslant i \leqslant j \leqslant n$ 确定 $A_iA_{i+1}\cdots A_j$ 的最小代价括号化方案作为子问题。令 $m[i, j]$表示计算矩阵 $A_{i..j}$所需标量乘法次数的最小值，那么，原问题的最优解——计算 $A_{1..n}$所需的最低代价就是$m[1, n]$。

我们可以递归定义 $m[i, j]$如下。对于 $i=j$ 时的平凡问题，矩阵链只包含唯一的矩阵$A_{i..i}=A_i$，因此不需要做任何标量乘法运算。所以，对所有 $i=1, 2, \cdots, n$，$m[i, i]=0$。若$i<j$，我们利用步骤 1 中得到的最优子结构来计算 $m[i, j]$。我们假设 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案的分

割点在矩阵 A_k 和 A_{k+1} 之间，其中 $i \leqslant k < j$。那么，$m[i, j]$ 就等于计算 $A_{i..k}$ 和 $A_{k+1..j}$ 的代价加上两者相乘的代价的最小值。由于矩阵 A_i 的大小为 $p_{i-1} \times p_i$，易知 $A_{i..k}$ 与 $A_{k+1..j}$ 相乘的代价为 $p_{i-1} p_k p_j$ 次标量乘法运算。因此，我们得到

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

此递归公式假定最优分割点 k 是已知的，但实际上我们是不知道的。不过，k 只有 $j-i$ 种可能的取值，即 $k=i, i+1, \cdots, j-1$。由于最优分割点必在其中，我们只需检查所有可能情况，找到最优者即可。因此，$A_i A_{i+1} \cdots A_j$ 最小代价括号化方案的递归求解公式变为：

$$m[i,j] = \begin{cases} 0 & \text{如果 } i = j \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{如果 } i < j \end{cases} \tag{15.7}$$

$m[i, j]$ 的值给出了子问题最优解的代价，但它并未提供足够的信息来构造最优解。为此，我们用 $s[i, j]$ 保存 $A_i A_{i+1} \cdots A_j$ 最优括号化方案的分割点位置 k，即使得 $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ 成立的 k 值。

步骤 3：计算最优代价

现在，我们可以很容易地基于递归公式(15.7)写出一个递归算法，来计算 $A_1 A_2 \cdots A_n$ 相乘的最小代价 $m[1, n]$。像我们在钢条切割问题一节中所看到的，以及即将在 15.3 节中看到的那样，此递归算法是指数时间的，并不比检查所有括号化方案的暴力搜索方法更好。 374

注意到，我们需要求解的不同子问题的数目是相对较少的：每对满足 $1 \leqslant i \leqslant j \leqslant n$ 的 i 和 j 对应一个唯一的子问题，共有 $\binom{n}{2} + n = \Theta(n^2)$ 个。递归算法会在递归调用树的不同分支中多次遇到同一个子问题。这种子问题重叠的性质是应用动态规划的另一个标识(第一个标识是最优子结构)。

我们采用自底向上表格法代替基于公式(15.7)的递归算法来计算最优代价(我们将在 15.3 节中给出对应的带备忘的自顶向下方法)。下面给出的过程 MATRIX-CHAIN-ORDER 实现了自底向上表格法。此过程假定矩阵 A_i 的规模为 $p_{i-1} \times p_i (i = 1, 2, \cdots, n)$。它的输入是一个序列 $p = \langle p_0, p_1, \cdots, p_n \rangle$，其长度为 $p.length = n+1$。过程用一个辅助表 $m[1..n, 1..n]$ 来保存代价 $m[i, j]$，用另一个辅助表 $s[1..n-1, 2..n]$ 记录最优值 $m[i, j]$ 对应的分割点 k。我们就可以利用表 s 构造最优解。

为了实现自底向上方法，我们必须确定计算 $m[i, j]$ 时需要访问哪些其他表项。公式(15.7)显示，$j-i+1$ 个矩阵链相乘的最优计算代价 $m[i, j]$ 只依赖于那些少于 $j-i+1$ 个矩阵链相乘的最优计算代价。也就是说，对 $k = i, i+1, \cdots, j-1$，矩阵 $A_{i..k}$ 是 $k-i+1 < j-i+1$ 个矩阵的积，矩阵 $A_{k+1..j}$ 是 $j-k < j-i+1$ 个矩阵的积。因此，算法应该按长度递增的顺序求解矩阵链括号化问题，并按对应的顺序填写表 m。对矩阵链 $A_i A_{i+1} \cdots A_j$ 最优括号化的子问题，我们认为其规模为链的长度 $j-i+1$。

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n−1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
```

```
10              q=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j
11              if q < m[i,j]
12                  m[i,j]=q
13                  s[i,j]=k
375 14      return m and s
```

算法首先在第 3～4 行对所有 $i=1, 2, \cdots, n$ 计算 $m[i, i]=0$(长度为 1 的链的最小计算代价)。接着在第 5～13 行 **for** 循环的第一个循环步中，利用递归公式(15.7)对所有 $i=1, 2, \cdots, n-1$计算 $m[i, i+1]$(长度 $l=2$ 的链的最小计算代价)。在第二个循环步中，算法对所有$i=1, 2, \cdots, n-2$ 计算 $m[i, i+2]$(长度 $l=3$ 的链的最小计算代价)，依此类推。在每个循环步中，第 10～13 行计算代价 $m[i, j]$时仅依赖于已经计算出的表项 $m[i, k]$和 $m[k+1, j]$。

图 15-5 展示了对一个长度为 6 的矩阵链执行此算法的过程。由于我们定义 $m[i, j]$仅在 $i \leqslant j$ 时有意义，因此表 m 只使用主对角线之上的部分。图中的表是经过旋转的，主对角线已经旋转到了水平方向。矩阵链的规模列在了图的下方。在这种布局中，我们可以看到子矩阵链 $A_i A_{i+1} \cdots A_j$ 相乘的代价 $m[i, j]$恰好位于始于 A_i 的东北至西南方向的直线与始于 A_j 的西北至东南方向的直线的交点上。表中同一行中的表项都对应长度相同的矩阵链。MATRIX-CHAIN-ORDER 按自下而上、自左至右的顺序计算所有行。当计算表项 $m[i, j]$时，会用到乘积 $p_{i-1} p_k p_j (k=i, i+1, \cdots, j-1)$，以及 $m[i, j]$西南方向和东南方向上的所有表项。

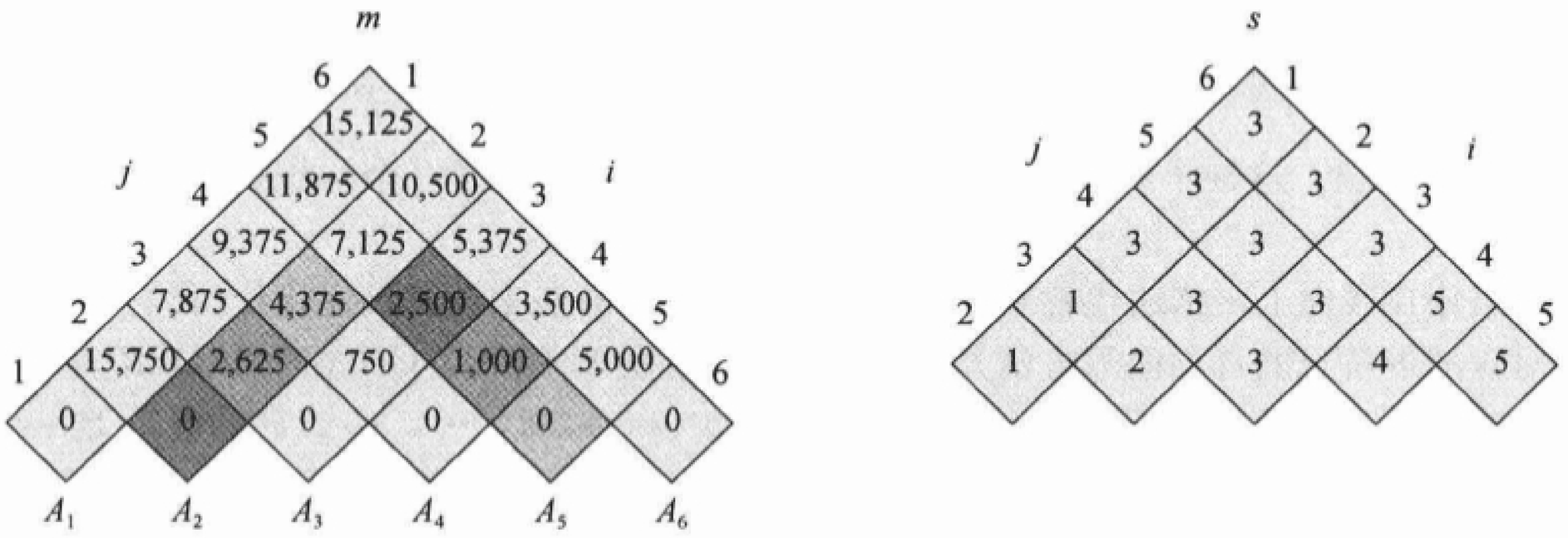


图 15-5 当 $n=6$ 和矩阵规模如下表时，MATRIX-CHAIN-ORDER 计算出的 m 表和 s 表。

矩阵	A_1	A_2	A_3	A_4	A_5	A_6
规模	30×35	35×15	15×5	5×10	10×20	20×25

我们将两个表进行了旋转，使得主对角线方向变为水平方向。表 m 只使用主对角线和上三角部分，表 s 只使用上三角部分。6 个矩阵相乘所需的最少标量乘法运算次数为 $m[1, 6]=15\,125$。表中有些表项被标记了深色阴影，相同的阴影表示过程在第 10 行中计算 $m[2, 5]$时同时访问了这些表项：

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2]+m[3,5]+p_1 p_2 p_5 = 0+2\,500+35 \cdot 15 \cdot 20 = 13\,000 \\ m[2,3]+m[4,5]+p_1 p_3 p_5 = 2\,625+1\,000+35 \cdot 5 \cdot 20 = 7\,125 \\ m[2,4]+m[5,5]+p_1 p_4 p_5 = 4\,375+0+35 \cdot 10 \cdot 20 = 11\,375 \end{cases} \\
&= 7\,125
\end{aligned}
$$

简单分析 MATRIX-CHAIN-ORDER 的嵌套循环结构，可以看到算法的运行时间为 $O(n^3)$。循环嵌套的深度为三层，每层的循环变量(l、i 和 k)最多取 $n-1$ 个值。练习 15.2-5 要求证明此算法的运行时间实际上是 $\Omega(n^3)$。算法还需要 $\Theta(n^2)$的内存空间来保存表 m 和 s。因此，MATRIX-CHAIN-ORDER 比起穷举所有可能的括号化方案来寻找最优解的指数阶算法要高效得多。

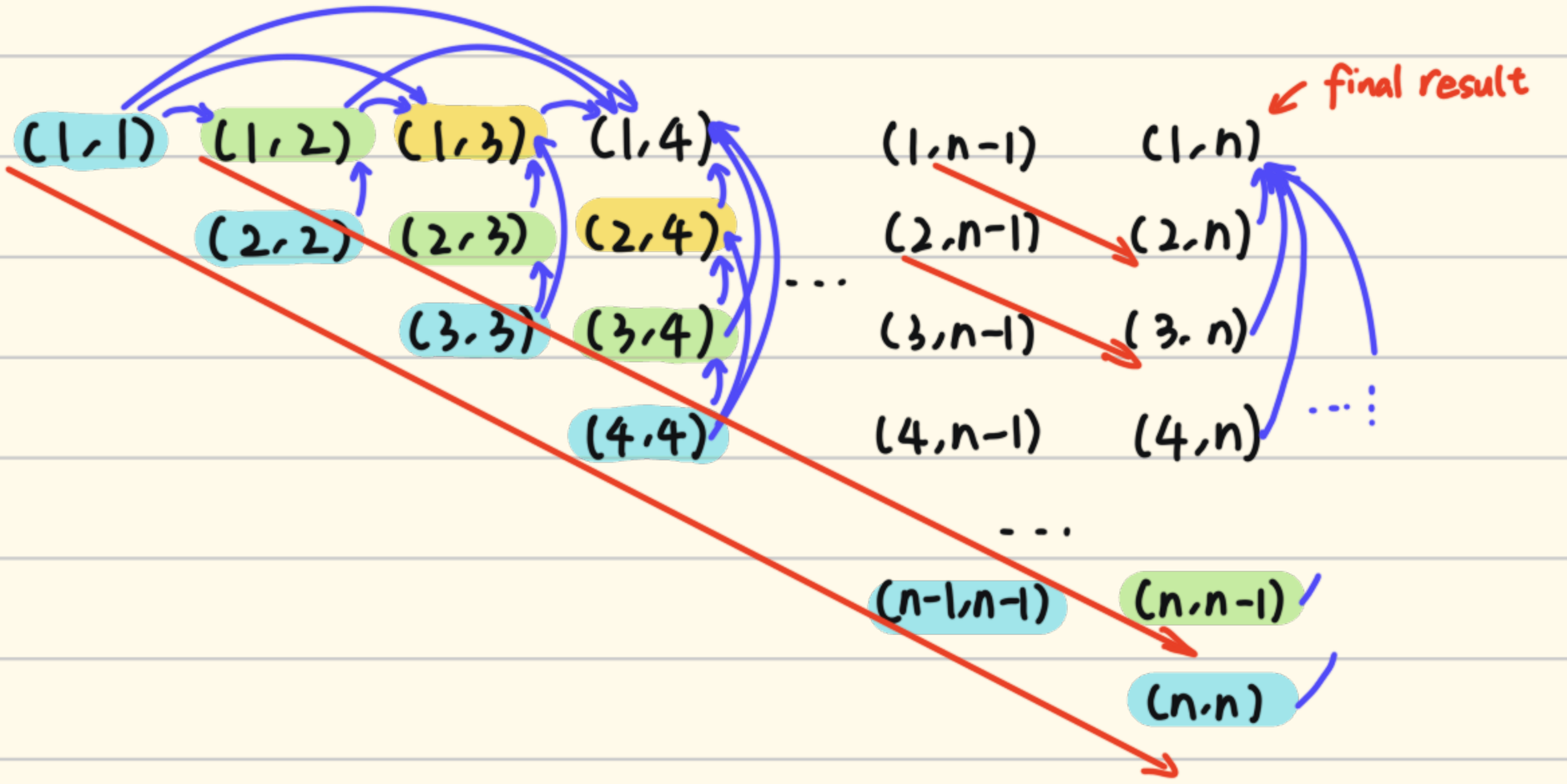
步骤 4：构造最优解

虽然 MATRIX-CHAIN-ORDER 求出了计算矩阵链乘积所需的最少标量乘法运算次数，但它并未直接指出如何进行这种最优代价的矩阵链乘法计算。表 $s[1..n-1, 2..n]$记录了构造最优解所需的信息。每个表项 $s[i, j]$记录了一个 k 值，指出 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案的分割点应在 A_k 和 A_{k+1} 之间。因此，我们知道 $A_{1..n}$ 的最优计算方案中最后一次矩阵乘法运算应该是 $A_{1..s[1,n]}A_{s[1,n]+1..n}$。我们可以用相同的方法递归地求出更早的矩阵乘法的具体计算过程，因为 $s[1, s[1, n]]$指出了计算 $A_{1..s[1,n]}$时应进行的最后一次矩阵乘法运算；$s[s[1, n]+1, n]$指出了计算 $A_{s[1,n]+1..n}$时应进行的最后一次矩阵乘法运算。下面给出的递归过程可以输出$\langle A_i, A_{i+1}, \cdots, A_j\rangle$的最优括号化方案，其输入为 MATRIX-CHAIN-ORDER 得到的表 s 及下标 i 和 j。调用 PRINT-OPTIMAL-PARENS(s, 1, n)即可输出$\langle A_1, A_2, \cdots, A_n\rangle$的最优括号化方案。

```
PRINT-OPTIMAL-PARENS(s, i, j)
1   if i==j
2       print "A"_i
3   else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i,j])
5       PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
6       print ")"
```

对图 15-5 中的例子，调用 PRINT-OPTIMAL-PARENS(s, 1, 6)输出括号化方案

$$((A_1(A_2A_3))((A_4A_5)A_6))$$



动态规划 { 子问题最优
子问题重叠