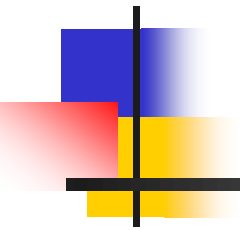
算法设计与分析

Computer Algorithm Design & Analysis

2025.11

王多强

QQ：1097412466



Chapter 15
Dynamic Programming

动态规划

本章起研究最优化问题：

这类问题通常**有一定的条件**，在满足条件的**多个解中找最好解**。通常表示为： **Min F(X)** 或 **Max F(X)**，其中，

- 问题需要满足的条件称为**约束条件**。
- 满足约束条件的解称为**可行解**。可行解不唯一。
- **X** 表示问题的输入/可行解。
- **F** 是**目标函数**，**F(X)** 表示对可行解的评价。
- **Min/Max**表示求**F(X)**的**最小/最大值**。
- 能够使**F(X)**达到极值的可行解称为问题的**最优解**（optimal solution）。
- 一般情况下，最优解也可能不唯一，但它们的极值是唯一的——能够使**F(X)**取得极值的可行解都是问题的最优解。

根据问题的**约束条件**和**目标函数数学模型**，以及**求解问题策略**的不同，问题可分为：**线性规划问题**、**整数规划问题**、**非线性规划问题**、**动态规划问题**等一些具体类型。

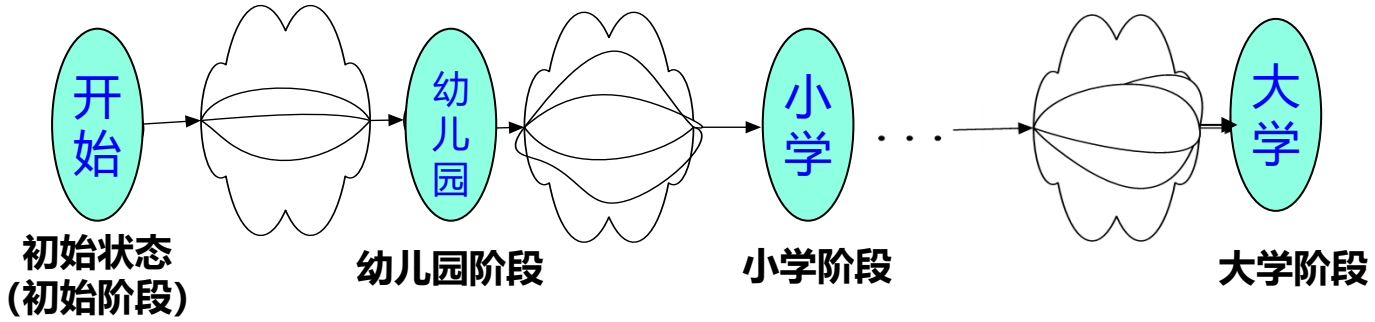
研究解决这一类问题的科学统称为**最优化理论和方法**。

本章研究解决这一类问题的一种策略 ——**动态规划策略**。

动态规划是20世纪50年代初美国数学家***R.E.Bellman***等人在研究**多阶段决策过程**的优化问题时创立的新方法，是求解**决策过程最优化**的一种数学方法。

多阶段决策过程

通常，一个事件的发展过程分为若干个相互联系的**阶段**。事件的发展从**初始状态**开始，依次经过第一阶段、第二阶段、…，直至最后一个阶段结束。



一般情况下，从 i 阶段发展到 $i+1$ 阶段（$0 \le i < n$）可能有**多种不同的途径**，而事件必须从中选择一条途径向前进展，从而使事件的发展过程从一个状态演变到下一状态。途径的选择称为**决策**。这里，用 x_i 表示 i 阶段的决策，用 D_i 表示 i 阶段的约束（可选路径集），$x_i \in D_i$ 的决策称为**合法的决策**。

如果事件的发展需要经历 n 个阶段，就需要做 n 次决策，这些决策构成了事件整个发展过程的一个**决策序列**，用 $x_1\ x_2\ \cdots\ x_n$ 表示 —— **多阶段决策过程问题就是求事件发展的决策序列**。

- **最优决策序列：**

设从阶段 i 到阶段 $i+1$ 有 p_i 种不同的选择，则从初始阶段至最后一个阶段就有 $p_1\ p_2\ \cdots\ p_n$ 种可能的不同发展途径，每个途径对应一个决策序列。

每一决策会附有一定的**成本**或**效益**，一个决策序列的成本或效益是序列中所有决策的成本或效益之和。

成本最小或**效益最大**的决策序列称为问题的**最优决策序列**。

对**多阶段决策过程**，

- **可行解**：从问题的开始阶段到最后阶段每一个**合法**的决策序列都是问题的一个**可行解**。

  合法决策序列仅指由各阶段的**合法决策**构成的序列。

- **目标函数**：用来衡量**可行解优劣**的函数。

- **最优解**：能够使目标函数取得**极值**的可行解是问题的**最优解**。

- **多阶段决策过程的最优化问题**：就是求能够使问题获得**最优解**的决策序列 —— 称为**最优决策序列**。

动态规划就是求决策过程最优化问题的一种数学方法。

动态规划，即**Dynamic Programming**，其中的 ***Programming*** 是什么意思？ —— **造表**

动态规划与分治法有点相似：都是通过**组合子问题的解**来求解原始问题，但对于子问题的要求不同：

分治法要求将问题划分为一些**互不相交**的子问题，然后递归求解这些子问题。如果**子问题有重叠**（指有公共子问题），递归过程中就会反复求解这些公共子问题，造成算法效率下降。

动态规划相反，适用于有**子问题重叠**的情况，即不同的子问题间具有公共的**子子问题**。动态规划算法只对公共子问题求解一次，并将其解保存到一个**表格**中。如果再次碰到相同子问题时，只需从表中找到以前的结果引用即可，从而**避免对公共子问题的重复计算，加快计算速度**。

用动态规划求解问题的步骤

（1）刻画问题最优解的一个结构特征（**最优子结构性质**）；

（2）构建计算**最优效益**的方法，得到一个递推关系式（称为**状态转移方程**）；

（3）实施递推计算，找到**最优效益**；

（4）利用计算过程中得到的信息**构造最优解**。

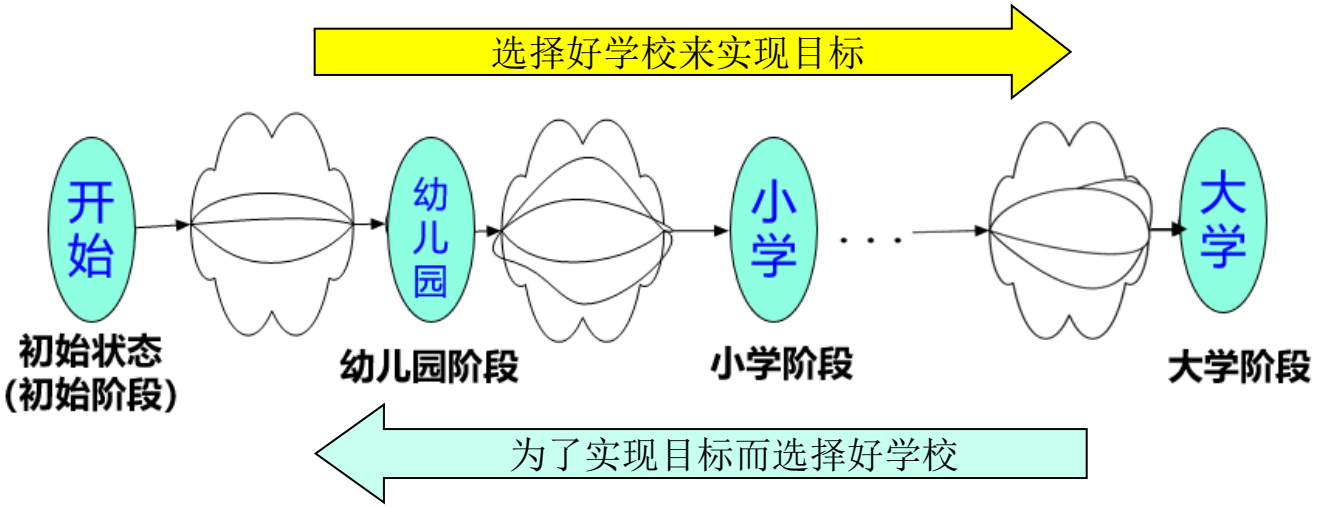
注：**最优解**是指最优决策序列，**最优效益**是指最优决策序列带来的效益。基本策略是：先计算出最优效益，然后再推导带来该效益的最优决策序列。

注：这里的已知条件和求解目标是

一般情况下，问题的**阶段划分已知**（或**容易构造**）、阶段间的**可供选择的决策已知**（或**容易找到**），而问题的求解目标是找出其中的最优决策序列。

所以（记决策序列为：$x_1 x_2 \ldots x_n$），

- 可**正向推导**：　$x_1 \to x_2 \to \ldots \to x_n$
- 也可**反向推导**：$x_n \to x_{n-1} \to \ldots \to x_1$



15.1 钢条切割问题

Serling公司购买长钢条，将其**切割为短钢条**出售。**不同的切割方案，收益是不同的**。问怎么切割才能有最大的收益呢?

这里用一个**价格表** P 给出不同长度钢条的价格，i 英寸钢条的价格记为 p_i，$i=1, 2, \ldots$。如下所示:

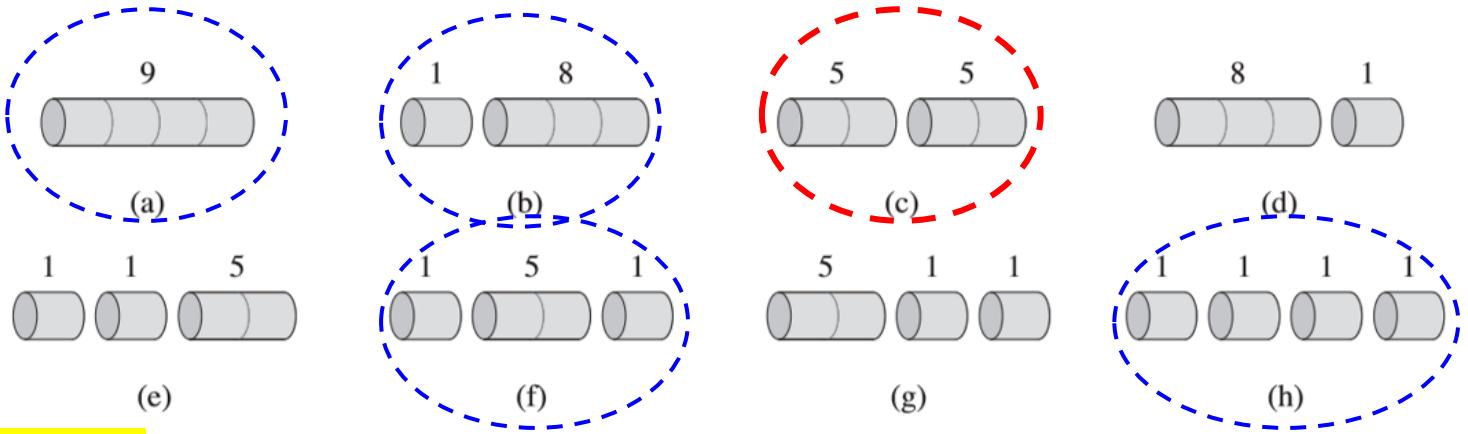
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

注：不同长度的钢条售价不一样，**平均每英寸钢条的价格不一**，所以将长钢条切割成若干短钢条出售可能会获得更大的收益。

这里不计切割成本

钢条切割问题：给定一段长度为 n 英寸的长钢条和一个价格表 P，求切割为短钢条的**方案**，使得销售收益最大（**假设都能卖出去**）。

举例说明：考虑 $n = 4$ 的情况。4英寸的钢条有以下八种切割方案：



根据等效关系，
有效方案只有5种

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

显然，对价格表P，**最优方案**是 c：将4英寸钢条切割为**两段**，每段长2英寸，每段售价5，可产生的收益为10，是最优解。

- 对长度为 n 英寸的长钢条，如何切割才能获得最大收益呢？

由于**每一英寸都可切割**，所以长度为 n 英寸的长钢条共有 2^{n-1} 种不同的切割方案，其中能够获得最大收益的方案称为**最优切割方案**。

记出售长度为 i 的钢条（整段出售或切割成多段出售）得到的**最大收益**为 r_i，则 n 英寸的长钢条的**最优切割方案**的**收益**为 r_n。

设**最优切割方案**下将长度为 n 的长钢条切割为 k 段，每段的长度为 i_j（$1 \le j \le k$），则有：

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

$$r_n = r_{i1} + r_{i2} + \ldots + r_{ik}$$

每段作为一个整体考虑，卖出它的整体最好价格

- **该如何切割才能获得最优方案呢?**

不失一般性，由于是**整英寸**切割，所以可以用 $1\sim n$ 间的整数标识切割点的位置。在 $1\sim n$ 间整数位置切割的任一方案都是问题的一个**可行解**。**分析如下：**

对于任意可行解，其中任意一个切割点 j，$1 \le j < n$，都将钢条分为长度分别为 j 和 $n-j$ 的两段，然后分别考虑**“整体”**出售。

令 r_j 和 r_{n-j} 分别是“整体”出售这样的两段钢条获得的**最好收益**，则切割点 j 能带来的最好收益是：$r_j + r_{n-j}$。

对于最优切割亦然：对最优切割中的任意切割点 i，也将钢条分成长度为 i 和 $n-i$ 的两段，令 r_i 和 r_{n-i} 分别是这两段的最优收益，则该情况下的收益 $r_i + r_{n-i}$ **就应该等于** r_n。

也就是，**最优方案里的切割点可用以下方法寻找**：

对 1~n 范围内的**每个切割点** j 都计算一下 $r_j + r_{n-j}$，$1 \le j < n$。比较它们各自带来的收益，收益最好的就是最佳切割点：

$$r_n = \max_j\{r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_j + r_{n-j}, \ldots, r_{n-1} + r_1, p_n\}$$

但完成这样测试的前提是要先知道所有的 $r_1 \sim r_{n-1}$，即长度为 1 ~ n-1**各种小段子钢条**的最佳收益 $r_1 \sim r_{n-1}$。**怎么办?**

对此，可视为一个**递归问题**：将原始的求长度为 n 的长钢条最佳收益问题**转换为**求长度为 1 ~ n-1 的小段子钢条最佳收益的子问题。

获得这些子问题的解后，就可完成对原始问题的求解。

思考两个问题：

（1）就算按照上述方式（$r_n = \max_j\{r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_j + r_{n-j}, \ldots, r_{n-1} + r_1, p_n\}$）求出一个最佳位置 i，会不会有其它位置更好呢？**不会！**

因为上述测试是对**每个位置**都做的测试，如果有更好的位置，就不会找 i。反之，既然找到 i，就一定不会有更好的其它位置。

（但可能存在能带来同样“最好收益”的其它位置，我们只求其一，i 就是当前最好的选择）。

（2）$r_n = r_i + r_{n-i}$ 的含义是：最佳收益 r_n 是两个子问题最佳收益 r_i 和 r_{n-i} 之和，会不会用**子问题的其它收益**求 r_n 会比用 r_i 和 r_{n-i} 更好呢？ **显然不会！**

r_n 是值最大的收益，如果不用子问题的最大收益求这个和，和的值也不会最大。而 r_i 和 r_{n-i} 就是两个子问题的最佳收益，没有比这更好的。

与以上问题相关的是动态规划的一个重要性质：最优子结构性

最优子结构性：如果一个决策序列是问题的**最优决策序列**，那么该最优决策序列中的任何子序列也必是该子序列对应子问题的**最优决策子序列**。

简单地说：全局最优则局部最优

以钢条切割问题为例说明：

对钢条切割问题，**决定在哪个位置切割就是一次决策**，所以切割方案里的**切割点序列**就构成钢条切割问题的**决策序列**。

设$l_1 l_2 \dots l_x\, i\, c_1 c_2 \dots c_k\, j\, h_1 h_2 \dots h_y$是一钢条切割问题的**最优决策序列**，$c_1 c_2 \dots c_k$代表该最优决策序列中从 i 到 j 这一段“**子钢条**”（视为**子问题**）的决策子序列，则 $c_1 c_2 \dots c_k$ 必是 $i \sim j$ 这一子问题的**最优决策子序列**，即相对该段“子钢条”的**最优切割方案**。

为什么?

证明：剪切-粘贴法（反证法）

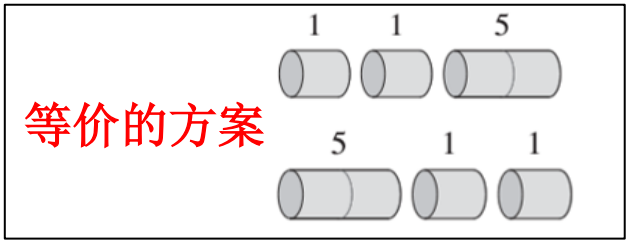
假设 $c_1 c_2 \dots c_k$ 不是 $i \sim j$ 这一子问题的**最优决策子序列**，即按照这样的切割方案无法获得 $i \sim j$ 段“子钢条”的最好收益。

但对 $i \sim j$ 这段子钢条必存在“**另外一种**”可以获得更好收益的切割方案，记为 $d_1 d_2 \dots d_m$。那么就可以在原来的**最优决策序列**中“**剪切**”掉 $c_1 c_2 \dots c_k$，而“**粘贴**”进 $d_1 d_2 \dots d_m$，从而得到一个新决策序列：

$$\underbrace{l_1 l_2 \dots l_x i}_{\text{收益没变}} \underbrace{d_1 d_2 \dots d_m}_{\text{收益变大}} \underbrace{j h_1 h_2 \dots h_y}_{\text{收益没变}}$$

由于其它切割点没变，所以新序列就能够获得比原来**最优决策序列“还好”的收益**，这与 $l_1 l_2 \dots l_x i c_1 c_2 \dots c_k j h_1 h_2 \dots h_y$ 是问题的最优决策序列相矛盾。所以**假设不成立**。即 $d_1 d_2 \dots d_m$ 这样“更好的子序列”是不存在的，而 $c_1 c_2 \dots c_k$ 就是相对 $i \sim j$ 段子钢条的**最优决策子序列**。

钢条切割问题的算法设计



算法1：朴素的递归算法

根据**切割的对称性**将切割方式简化为：将钢条从左边切割下长度为 i 的一段，然后只对右边剩下的长度为 $n-i$ 的一段继续进行切割（视为**递归求解**）。对左边长为 i 的那段不再进行切割。

此时有“递推式”：

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

其中 r_{n-i} 代表对右边 $n-i$ 段的**递归求解**。这样，原问题的每个最优解中只涉及**一个后续子问题**（r_{n-i}）。

一个**自顶向下的递归求解过程**可以描述如下：

```
CUT-ROD ( p, n)
1   if n == 0
2         return 0
3   q = -∞
4   for i =1 to n
5         q = max(q, p[i] + CUT-RUD( p, n – i ))
6   return q
```

边界条件：若 $n = 0$，则收益为 0。

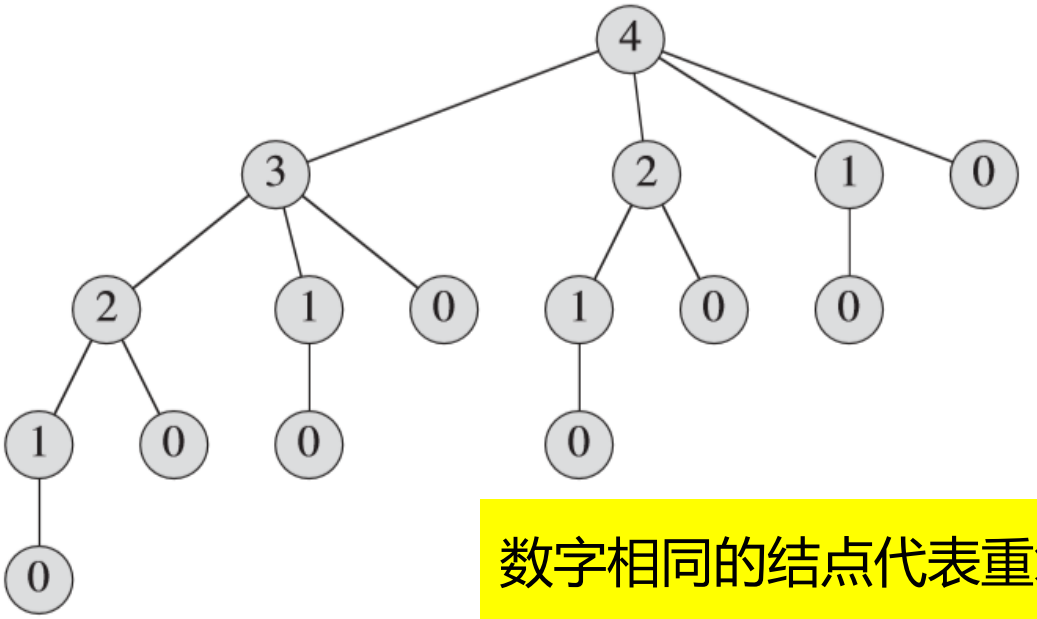
递归求解子问题

上述算法存在的问题：效率很差

存在一些**相同长度的子问题**，在CUT-ROD的执行过程中（主要是递归过程）**反复地被计算**。

这是由简单递归造成的**冗余计算**，严重降低了算法执行效率。

如 $n = 4$， CUT-ROD的递归执行过程可以用**递归调用树**表示为：



结点中的数字是该结点对应的子问题的规模。

数字相同的结点代表重复子问题。它们本质相同，但在递归过程中被重复计算了，花费了大量时间。

- 从父结点 s 到子结点 t 的边表示从钢条左端切下长度为 $num_s - num_t$ 的一段后，继续递归求解剩余规模为 num_t 的子问题。
- 从根结点到某个叶结点的路径对应一种切割方案。
- 该树有 2^{n-1} 个**叶结点**，对应 2^{n-1} 种切割方案。

```
CUT-ROD (p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i =1 to n
5       q = max(q, p[i] + CUT-RUD( p, n –i ))
6  return q
```

CUT-ROD的时间分析：

令 $T(n)$ 表示当问题规模为 n 时CUT-ROD的执行时间，则有：

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

可以证明： $T(n) = 2^n$

n 代表从 n 个子问题对应的解中找出最大的那个。

```
CUT-ROD (p, n)
1  if n == 0
2        return 0
3  q = -∞
4  for i =1 to n
5        q = max(q, p[i] + CUT-RUD( p, n – i ))
6  return q
```

CUT-ROD是一个没有经过任何优化的简单递归计算过程。

算法2：动态规划算法

动态规划策略中将仔细安排“**求解顺序**”，对每个子问题只求解一次，并**将子问题的结果保存下来**；之后，如果再次碰到该子问题，只需查找以前的计算结果即可获得其解，而不必重新计算。因此，

① **空间需求一般比较大**：动态规划算法一般需要**额外的空间**保存子问题的解。

② **节省了时间**：避免对重复子问题进行重复计算。

因此，**动态规划是一种用空间换时间的策略**。如果子问题的数量是 n 的多项式函数，而且可以在多项式时间内求解每个子问题，则动态规划方法的总运行时间一般可以达到**多项式阶**。

动态规划求解的两种方法

(1) 带备忘表的自顶向下法（top-down with memoization）

形式上还是**递归**（自顶向下求解），但处理过程中会**保存每个子问题的解**。

在此过程中，每次处理前，首先检查当前子问题是否已经被计算并保存过子问题的解。

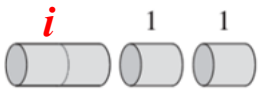
- 如果保存过，则直接返回以前保存的值；
- 如果没保存过，再进行计算并保存结果。

备忘表：用于保存之前已经计算出来的结果的表。具体形式可以是一个数组或者散列表。

带备忘表的钢条切割问题动态规划算法

```
MEMOIZED-CUT-ROD ( p, n )
1   let r [0 .. n] be a new array   //这里 r 就是用作备忘表的辅助数组
2   for i = 0 to n
3         r[i] = -∞   // r 中的元素都初始化为-∞
4   return MEMOIZED-CUT-ROD-AUX (p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX ( p, n, r )
1   if r[n] ≥ 0      // 查备忘表：r[n] ≥ 0 表示有保存的解，直接引用 r[n]
2      return r[n]
3   if n == 0        // 否则，实施后续计算
4      q = 0
5   else  q = -∞
6      for i = 1 to n
7         q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX( p, n - i, r ))
8   r[n] = q    // r[n]保存当前结果
9   return q
```



只切割后面的部分，前面的不再切割

递归计算

（2）自底向上法（bottom-up method）

上述过程**存在递归**，开销还是较大。一种**自底向上**的计算策略可以避免递归：

因为**较大子问题的解是通过组合较小子问题的解而得到的**，因此可**从最小子问题开始**，按照**最小子问题**、**较小子问题**、…、**较大子问题**的顺序依次求解，直到得到原始问题的答案。

自底向上的求解策略：先求解较小的子问题，当求解一个较大子问题时，它所**依赖**的更小子问题都已求解完毕，结果已经保存，所以可以直接**引用**更小子问题的解并组合出它自己的解。这个过程不涉及子问题的递归求解。

钢条切割问题的自底向上动态规划算法

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0 .. n] be a new array //这里 r 依然被用作记录子问题的解
2  r[0] = 0
3  for j = 1 to n          // 对长度为1~n 的所有子问题都求解
4      q = -∞
5      for i = 1 to j      // 对长度为 j 的子问题求解
6          q = max(q, p[i] + r[j - i])   // 这里不再判断 r[j - i] 是否求过解，因为从
                                         // “短到长”计算，r[j - i] 之前肯定被计算
                                         // 过，这里直接引用即可。
7      r[j]= q   // “从短到长”计算，
                 // j 是一段更长的钢条，之前没计算过，
                 // 这里直接保存结果
8  return r[n]
```

注：r 中记录了长度 $0 \sim n$ 的所有子问题的解。最后只回答 $r[n]$ 即可得到长度为 n 的钢条切割问题的解。

对比：带备忘的自顶向下法 *Vs* 自底向上法

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

递归处理

自顶向下

```
BOTTOM-UP-CUT-ROD(p,n)
1   let r[0 .. n] be a new array //这里 r 亦然被用作记录子问题的解
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           q = max(q, p[i] + r[j - i])
7       r[j]= q
8    return r[n]
```

迭代处理

自底向上

- **MEMOIZED-CUT-ROD和BOTTOM-UP-CUT-ROD具有相同的渐近运行时间：$\Theta(n^2)$。**

  证明：略，见P208.

- 通常，**自顶向下法和自底向上法具有相同的渐近运行时间。**
  - 但由于自底向上法**没有频繁的递归函数调用**的开销，所以自底向上法的时间复杂性函数通常具有**更小的系数**。
  - 但在某些特殊情况下，自顶向下法可能没有递归处理**所有可能的子问题**（**存在剪枝**）从而有更少的工作量。相反自底向上法要计算所有子问题的解。

重构解

上述算法中，最后 r **表**中记录了各种长度的钢条售卖所能获得的**最优收益**。但**如何切割的呢**，即各段的**切割点**都在哪里？

为此，扩展上述算法，再引入一个 s **表**（也是一个一维数组）：

对**长度为** j 的钢条，若求出**在其中** i **位置切割可以获得最优收益**，则令 $s[j] = i$，即记录下这个可以获得最优收益的位置。

当完成 r 表的计算后，利用 s 表就可以求出**切割方案**。

引入 *s* 表后的算法称为EXTENDED-BOTTOM-UP-CUT-ROD

```
EXTENDED-BOTTOM-UP-CUT-ROD ( p, n )
1   let r[0 .. n] and s[0 .. n] be new arrays    // r 记录子问题的最优收益,
                                                 // s 记录最优收益下的切割点位置
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 ]    // 有更好的收益
8               s[j] = i                 // 在位置 i 切割可获得更优收益时，记下该位置
9       r[j] = q
10  return r and s
```

对 **for** $i = 1$ **to** j
$q = \max(q, p[i] + r[j - i])$
的具体实现。

注： i 仅是相对于 j 的 **“第一刀”**

◆ 输出完整的最优切割方案

若求长度为 n 的钢条的最优售卖方案(包括最优收益和切割点的位置)，先调用**EXTENDED-BOTTOM-UP-CUT-ROD**，得到 r 和 s 表：

① $r[n]$ 就是售卖长度为 n 的钢条能够获得的最优收益。

② $s[n]$ 就是获得 $r[n]$ 收益的切割点——可视为**第一个切割点**，即切割下来的**第一段钢条长** $s[n]$。此后，

③ **剩余钢条的长度是** $n - s[n]$，这段钢条又是如何切割的呢？

继续查 s 表：令 $n \leftarrow n - s[n]$，则**新的** n 对应的 $s[n]$ 就是剩余钢条的切割点——可视为**第二个切割点**，即切下来的**第二段钢条的长度**就是现在的 $s[n]$。

重复上述过程，直到剩余长度为 0 结束。

以上输出的切割点序列就构成最优收益的最优切割方案。

◆ 输出完整最优切割方案的程序描述

```
PRINT-CUT-ROD-SOLUTION ( p, n )
1    (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)  // 先求 r 和 s
2    while n > 0                  // 利用 s 表，反推切割点
3        print s[n]               // 输出当前切割点
4        n = n- s[n]              // 修改 n 为剩余长度，以继续循环输出后面的切割点
```

例：对 $n = 10$ 的钢条计算最优切割方案，得到的 r 表和 s 表如下：

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

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

执行：PRINT-CUT-ROD-SOLUTION(p, **10**)

输出切割点：**print s[10]** ➡ **10**

执行：PRINT-CUT-ROD-SOLUTION(p, **7**)

输出切割点：**print s[7]** ➡ **1**

n = 7-s[7] = 6

print s[6] ➡ **6**

15.2 矩阵链乘法问题

1、两个矩阵的简单相乘

已知A为 $p\times r$ 的矩阵，B为 $r\times q$ 的矩阵，则A与B的乘积是一个 $p\times q$ 的矩阵，记为 C:

$$C = A_{p\times r} \times B_{r\times q} = (c_{ij})_{p\times q}$$

其中，

$$c_{ij} = \sum_{1\le k\le r} a_{ik}b_{kj}\ , \qquad i = 1,2,..p,\ \ j = 1,2,...,q$$

如:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 8 \end{bmatrix}, B = \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}$$

$$C = AB = \begin{bmatrix} 2 & 3 \\ 5 & 8 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}$$

$$= \begin{bmatrix} 2*1+3*6 & 2*4+3*7 \\ 5*1+8*6 & 5*4+8*7 \end{bmatrix}$$

$$= \begin{bmatrix} 20 & 29 \\ 53 & 76 \end{bmatrix}$$

$$c_{ij} = \sum_{1 \le k \le r} a_{ik} b_{kj}$$

直接相乘共需要8次元素乘和4次元素加

一个标准的两矩阵乘算法如下：

```
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
```

只有两个矩阵“相容”才能相乘，否则报错

三重循环结构

两个矩阵简单相乘的时间复杂度：

$$A=(a_{ij})_{p\times r},\quad B=(b_{ij})_{r\times q},$$

$$C=A_{p\times r}\times B_{r\times q}=(c_{ij})_{p\times q}$$

其中，$c_{ij}=\sum_{1\le k\le r} a_{ik}b_{kj}$，　　$i=1,2,..p,\ \ j=1,2,...,q$

每个 c_{ij} 的计算需要 r **次乘法**（另有 $r-1$ 次加法，这里仅考虑元素的**标量乘法**）。C 中共有 $p\times q$ 个元素，所以计算结果矩阵 C 共需要 pqr 次标量乘法运算。

若 $p=r=q=n$，则两个矩阵简单相乘的时间复杂度就是 $O(n^3)$。

2、矩阵链乘

n 个矩阵**连续相乘**称为**矩阵链乘**，记为乘积： $A_1 A_2 \ldots A_n$。

- **矩阵链乘不满足交换律**：$A_1 A_2 A_3 \neq A_3 A_2 A_1$
- **矩阵链乘满足结合律**： $(A_1 A_2) A_3 = A_1 (A_2 A_3)$

矩阵链乘可以通过在矩阵之间**加括号**而导出不同的**计算模式**。

如，已知四个矩阵A_1, A_2, A_3, A_4，根据不同的**加括号方式**，乘积 $A_1 A_2 A_3 A_4$ 有五种不同的计算模式：

$(A_1(A_2(A_3 A_4)))$　　$(A_1((A_2 A_3)A_4))$

$((A_1 A_2)(A_3 A_4))$　　$((A_1(A_2 A_3))A_4)$

$(((A_1 A_2)A_3)A_4)$

不同的计算模式最后得到的结果是一样的

但尽管不同的计算模式最后得到的结果是一样的，计算过程中产生的**代价却不同** —— **标量乘法的次数有多有少。**

如，设有三个矩阵的链 $<A_1, A_2, A_3>$，维数分别为**10×100, 100×5, 5×50**。分别按 $((A_1 A_2)A_3)$ 和 $(A_1(A_2 A_3))$ 两种模式计算。

$((A_1A_2)A_3)$：A_1与A_2乘需要**10×100×5=5000**次标量乘法运算，得到一个**10×5**的中间结果矩阵，再与A_3相乘，需要**10×5×50=2500**次标量乘法运算，总共为**7500次**标量乘法运算。

$(A_1(A_2A_3))$：A_2与A_3乘需要**100×5×50=25000**次标量乘法运算，得到一个**100×50**的中间结果矩阵，A_1与之相乘，再需要**10×100×50=50000**次标量乘法运算，总共为**75000次**标量乘法运算。

以上两种计算模式的计算量相差10倍!

3、矩阵链乘法问题 (matrix-chain multiplication problem)

给定 n 个矩阵的链，记为 $<A_1, A_2, \ldots, A_n>$，其中，矩阵 A_i 的维数为 $p_{i-1} \times p_i$，$i = 1, 2, \ldots, n$。**矩阵链乘法问题**就是求一个**“完全括号化方案”**，使得计算 $A_1 A_2 \ldots A_n$ 乘积的**标量乘法次数最少**。

令 $p(n)$ 表示 n 个矩阵链乘时可供选择的**括号化方案的数量**。

则有：$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

可以证明：$P(n) = \Omega(2^n)$

$$\overbrace{(\underbrace{A_i A_{i+1} \ldots A_k}_{p(k)})(\underbrace{A_{k+1} A_{k+2} \ldots A_j}_{p(n-k)})}^{p(n)}$$

显然，**穷举**所有可能的括号化方案是“不可行”的。

(1) 最优括号化方案的结构特征—— 寻找**最优子结构**

用记号 $A_{i,j}$ 表示**子问题** $A_iA_{i+1}\cdots A_j$ 通过加括号后得到的一个**最优计算模式**，且该计算模式下的**“最大区间”**恰好在 A_k 与 A_{k+1} 之间分开，如下所示：

$$(\overline{A_iA_{i+1}\cdots A_k})\,(\overline{A_{k+1}A_{k+2}\cdots A_j})$$

本问题中，**加括号就是“决策”**，加括号的位置就构成该问题的决策序列。子序列上面加**横线**表示还有**内部子模式**

则必须有：$(\overline{A_iA_{i+1}\cdots A_k})$ 必是**“前缀子链”** $A_iA_{i+1}\cdots A_k$ 的一个最优的括号化子方案，记为 $A_{i,k}$；同理 $(\overline{A_{k+1}A_{k+2}\cdots A_j})$ 也必是**“后缀子链”** $A_{k+1}\ A_{k+2}\ \cdots\ A_j$ 的一个最优的括号化子方案，记为 $A_{k+1,j}$。

这里子链就是子问题

证明：**反证法**

如若不然，设 $A'_{i,k}$ 是 $<A_i, A_{i+1}, \ldots, A_k>$ 一个代价更小的计算模式，也就是 $<A_i, A_{i+1}, \ldots, A_k>$ 间有不同于前面的**另一加括号方案**，而该方案的计算代价更小。则由 $A'_{i,k}$ 和 $A_{k+1,j}$ 构造一个新的计算模式 $A'_{i,j}$，总代价将比 $A_{i,j}$ 小：

$$\underbrace{(\overline{A_i A_{i+1} \ldots A_k})}_{A'_{i,k}} \underbrace{(\overline{A_{k+1} A_{k+2} \ldots A_j})}_{A_{k+1,j}}$$

这与 $A_{i,j}$ 是最优链乘模式相矛盾。

对 $A_{k+1,j}$ 亦然。

所以矩阵链乘法问题具有最优子结构性质。

即，若 $A_{i,j}$ 是子问题 $A_i A_{i+1} \ldots A_j$ 的最优的计算模式，则其中的 $A_{i,k}$、$A_{k+1,j}$ 也都是相应子问题 $A_i A_{i+1} \ldots A_k$ 和 $A_{k+1} A_{k+2} \ldots A_j$ 的最优计算模式。——**全局最优则局部最优**。

- **为什么用动态规划策略求解问题前要证明问题满足最优子结构性?**

最优子结构性告诉我们：问题的**最优方案**内包含的**子方案也是最优的**，或者说：**最优方案由最优子方案组合而成**。

该性质给了我们求解最优方案的一个方向指引：

先找子问题的最优子方案，然后组合这些最优子方案来生成完整的最优方案。

而子问题的“**非优**”方案就不用考虑了，从而可以大大节约计算量。

动态规划就是通过用子问题的最优子方案组合生成问题最优方案的一种策略，因此**满足最优子结构性是实施动态规划进行问题求解的前提**。

- 只有问题满足最优子结构性，才能进行下一步动态规划算法的具体设计，否则就要换其它方法求解。
- 如何证明一个问题满足最优子结构性？

  ——“**剪切-粘贴**”法（本质上就是**反证法**）。

注：证明问题满足最优子结构性并不是先求出最优解，再证明该解满足最优子结构性，而是**通过最优解的结构特征直接推导来实现证明，是一种形式化的证明过程**（回顾钢条切割问题和矩阵链乘法问题的相关证明过程）。

动态规划算法设计的第二步：构造**状态转移方程**。

状态转移方程是一个**递推关系式**，用以**计算最优（子）解的成本/收益值**，以**数学函数式**的形式体现。

—— 我们把决策序列的一组具体取值称为问题求解过程中的一个**状态**。通过该**递推函数**的计算就可以逐步获得最优决策序列的取值，体现了问题状态的连续变化过程，所以称为**状态转移方程**。

状态转移方程一般以**问题最优解的成本值或收益值**的**计算表达式**给出。

(2) 矩阵链乘法问题的状态转移方程（递推关系式）

- 定义矩阵链乘法的**最优成本**：

令 $m[i,j]$ 表示计算 $A_i \cdots A_j$ **最优计算模式** $A_{i,j}$ 所需的**最小标量乘法运算次数**。则 $m[i,j]$ 的递推关系式（**状态转移方程**）如下：

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

理解 $m[i,j]$ 的递推关系式：

① 对矩阵链乘模式 $A_{i,j}$，无论内部如何加括号，总有一个**最大区间括号**所在的位置 k，而 k 将 $A_{i,j}$ 分成两个“**最大子链**”：$A_{i,k}$ 和 $A_{k+1,j}$：　$(A_i A_{i+1} \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j)$

↑ 在这个位置后面加左侧的右括号

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

② **根据最优子结构性**：

无论 k 在哪个位置，它对 $A_{i,j}$ 带来的**最优计算量**等于其左侧子链 $A_{i,k}$ 的最优计算量 ($m[i,k]$) 加上右侧子链 $A_{k+1,j}$ 的最优计算量 ($m[k+1,j]$)，再加上**左侧子链计算出来的中间结果矩阵与右侧子链计算出来的中间结果矩阵在本级相乘所带来的计算量：**

根据矩阵链乘的规则，左侧子链<A_i，A_{i+1}，…，A_k>计算出来的中间结果矩阵的大小是 $p_{i-1}\times p_k$，右侧子链<A_{k+1},A_{k+2},…,A_j>计算出来的中间结果矩阵的大小是 $p_k\times p_j$，所以两个**中间结果矩阵再乘**所带来的本级计算量是 $p_{i-1}p_kp_j$。

综上，对位置k，总计算量就是：$m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

③ 对 i~j 范围的**最优矩阵链乘模式** $A_{i,j}$，这个能带来最优计算量的 k **在哪里呢**?

理论上，对不同的问题实例，k 可以在任何位置。需要根据实例情况进行“搜索”（“**动态**”的一种体现），即测试 $i\sim j$ 范围内的每个位置，看哪个位置的计算量最小，那么该位置就是要求的 k。

所以按：

$$\min_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$$

测试每个可能位置，最后找到 k。

注：k **的搜寻范围是** $i\sim j-1$。这是因为在位置 j 处加左侧“大”括号等效于整个矩阵链 $A_{i,j}$，所以不需要对位置 j 进行测试。

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

④ **边界条件**：$i==j$，表示矩阵链 $A_{i,j}$ 中实际只有一个矩阵 (A_i)，不发生“乘运算”。所以边界条件时 $m[i,j]=0$。

再下面就是具体计算 $m[i,j]$

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

（3）状态转移方程计算的具体实施：

状态转移方程形式上是一个**递推关系式**，计算过程采用**递推方式**具体实现（**自底向上**）。

- **以 $m[i,j]$ 的计算为例进行说明下：**

第一步：先计算**最小子问题**的解，即最小矩阵链的 $m[i,j]$。

最小矩阵链就是长度为1，即 $i==j$ 的矩阵链。

此时“链”中只有一个矩阵，根据 $m[i,j]$ 的边界定义，$m[i,i]$ 等于 0（也可以是 $m[j,j]$ 等于 0）。

注意：这样的**最小矩阵链**有 n **个**，是指所有的 $A_{1,1}$、$A_{2,2}$、…、$A_{n,n}$ 对应的子链，要对所有的 $A_{i,i}$ 都计算出它们的 $m[i,i]$ 值，$1 \le i \le n$（虽然事实上这些 $m[i,i]$ 的值都等于0，但是都要记下来——造出相应的表）。

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

第二步：在第一步（计算出长度为1的最小矩阵链子问题解）的基础上，再计算稍大一点的子问题——**长度为 2 的矩阵链的解**，即所有 $j-i+1=2$ 时的 $m[i,j]$，$1\le i,j\le n$。

注意：长度为 2 的**矩阵链**有 ***n*-1 个**：$A_{1,2}$、$A_{2,3}$、$A_{3,4}$、…、$A_{n-1,n}$，都计算出来并记在表中。

第三步：在第一步和第二步（计算出长度为1和2的小矩阵链子问题解）的基础上，再计算更大一点的子问题——**长度为 3 的矩阵链的解**，即所有 $j-i+1=3$ 时的 $m[i,j]$，$1\le i,j\le n$。

同理：长度为 3 的**矩阵链**有 ***n*-2 个**：$A_{1,3}$、$A_{2,4}$、$A_{3,5}$、…、$A_{n-2,n}$，都计算出来并记在表中。

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

上述过程继续进行，再依次计算出长度为 4、5、… 等所有更长矩阵链对应的子问题的解，一直持续到长度为 n 的矩阵链，即原始矩阵链，此时 $j - i + 1 = n$：

原始矩阵链是 $<A_1, A_2, \ldots, A_n>$，最终计算结果是 $m[1, n]$。

进一步讨论：$m[1, n]$ 给出了**最优括号化方案**下的**最优计算量**。但怎么加的括号呢?

再造一个表：在上述计算过程中再造一个表 $s[1..n, 1..n]$，在每个 $m[i, j]$ 计算出来后，$s[i, j]$ 记录使 $m[i, j]$ 取得最小值的 k，即矩阵链 $A_{i,j}$ 内的加“**最大区间括号**”的**最佳位置**。

首先完成上述的计算。在得到 $m[1, n]$ 和 $s[1, n]$ 后，按照下面的步骤“**反向推导**”出的最优链乘模式：

① 在求出 $m[1, n]$ 后， $s[1, n]$ 记录了一个使 $m[1, n]$ 取得最小值的 k，它就是原始矩阵链 $<A_1, A_2, \ldots, A_n>$ 中“**最大区间**”**括号**所加的位置：

$$(\ <A_1, A_2, \ldots, A_k>\ )(\ <A_{k+1}, A_{k+2}, \ldots, A_n>\ )$$

② 在加了①中“**最大括号**”后，子矩阵链 $<A_1, A_2, \ldots, A_k>$ 和 $<A_{k+1}, A_{k+2}, \ldots, A_n>$中的括号又该怎么加呢？

因为 $A_{1,k}$ 和 $A_{k+1,n}$ 是**短子矩阵链**，所以在这之前就分别计算过 $m[1, k]$ 和 $m[k+1, n]$，同时也记录了 $s[1, k]$、$s[k+1, n]$（就是它们各自的“**最佳括号位置**”），所以此时只要在 s 中查一下记录就可以推导出这两个“大子链”中各自的“**最大区间**”**括号**所在的位置。

③ 第②步之后，就相当于对于原始矩阵链 $<A_1, A_2, \ldots, A_n>$ 加了**两重括号**，分成了四个子矩阵链。

$$((<A_1, .., A_{k'}>)(<A_{k'+1}, \ldots, A_k>))((<A_{k+1}, \ldots, A_{k''}>)(<A_{k''+1}, \ldots, A_n>))$$

同样的方法，在所有还有内部结构（长度大于2）的子矩阵链中继续查找计算子链的 m 值时记录的 s 值，就可逐步推导出所有加括号的位置，从而最终得到原始问题 $A_{1,n}$ 的**完整加括号方案**，得到矩阵链乘的**最优计算模式**。

◆ 这里需要认识和理解的是：

m **表、** s **表**的计算是从**最小区间**（所有的$[i, i]$，$1 \le i \le n$）向**最大区间** $[1, n]$ **“正向计算”**完成的。而加括号方案的推导是在 m 表和 s 表计算完成后，**“反向推导”**得到的。

造表（*Programming*）：

对矩阵链乘，m 和 s 就是两个表，分别用于记录每个可能的子矩阵链的最佳计算量和内部加“最大区间括号”的最佳位置。

由于这两个表中，每个元素（子问题区间）需要 i、j **两个下标**表示其所代表的区间（ $m[i, j]$ 或 $s[i, j]$ ，且 $0 \le i$、$j \le n$ ），所以实际实现时用两个 $n \times n$ **二维数组**表示。

计算过程的程序描述如下：

注：输入 $p=\langle p_0, p_1, \ldots, p_n\rangle$，是$n$个矩阵的维数表示，矩阵 A_i 的维数是 $p_{i-1} \times p_i$，$i=1,2,\ldots,n$。

```
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      定义表 m 和表 s
3   for i= 1 to n
4         m[i, i] = 0          //对长度为1的矩阵链直接设定 m[i, i]=0
5   for l = 2 to n             // l 代表子矩阵链长度，对长度为 2~n 的矩阵链依次计算
6       for i = 1 to n – l + 1     // i 是一个长度为 l 的矩阵子链的开始下标
7           j = i + l - 1          // j 是该矩阵子链的结束下标，当前子链区间是 i ~ j
8           m[i, j] = ∞
9           for k = i to j -1
10              q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
11              if q < m[i, j]
12                    m[i, j] = q
13                    s[i, j] = k
14  return m and s
```

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

计算 $m[i,j]$，并记录 $s[i,j]$

算法复杂度分析

- 算法的主体由一个**三层循环**构成。
  - 最外层循环执行n-1次；
  - 内层两个循环都至多执行n-1次；所以MATRIX-CHAIN-ORDER的时间复杂度是 $\Omega(n^3)$。
- 算法需要 $\Theta(n^2)$ 的空间保存 m 和 s。

```
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          //对长度为1的矩阵链直接设定 m[i,i]=0
5   for l = 2 to n           //对长度为2~n的矩阵链依次计算
6       for i = 1to n – l + 1    // i 是矩阵子链的开始下标
7           j = i + l - 1        // j 是矩阵子链的结束下标
8           m[i, j] = ∞          // 当前子链是A_{i,j}
9           for k = i to j -1
10              q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
11              if q < m[i, j]
12                  m[i, j]= q
13                  s[i, j] = k
14  return m and s
```

例，设 p = <30, 35, 15, 5, 10, 20, 25>，即矩阵链中有**6个矩阵**，每个矩阵的维度如表，求其最佳链乘模式。

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

① 计算长度为1的最小矩阵链

长度为1的子矩阵链区间是 **[*i*, *i*]**，$1 \le i \le 6$，且 $m[i, i]=0$。这样的“区间”有6个：

$m[1,1]=0$

$m[2,2]=0$

$m[3,3]=0$

$m[4,4]=0$

$m[5,5]=0$

$m[6,6]=0$

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j.\end{cases}$$

② 计算长度为2的子矩阵链

长度为2的子矩阵链区间是 $[i, i+1]$，$1 \le i \le 6-1$，

所以有：$\boldsymbol{m[i,i+1]} = \min\limits_{i \le k < i+1} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$

$= \min\{\boldsymbol{m[i,i]}, \boldsymbol{m[i+1,i+1]} + \boldsymbol{p_{i-1}p_i p_{i+1}}\}$

$= \boldsymbol{p_{i-1}p_i p_{i+1}}$

$\boldsymbol{m[1,2]} = \min\limits_{1 \le k < 2} \{m[1,k] + m[k+1,2] + p_0 p_k p_2\}$

$= p_0 p_1 p_2$

$= \mathbf{30 \times 35 \times 15}$

$= 15750$

$\boldsymbol{s[1,2]} = \mathbf{1}$

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

共5个，其它长度为2的区间：

$m[2,3] = 2625$，$s[2,3] = 2$

$m[3,4] = 750$，$s[3,4] = 3$

$m[4,5] = 1000$，$s[4,5] = 4$

$m[5,6] = 5000$，$s[5,6] = 5$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

③ 计算长度为3的子矩阵链

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

长度为3的子矩阵链区间是 $[i, i+2]$，$1 \le i \le 6-2$，

所以有：$m[i,i+2] = \min\limits_{i \le k < i+2} \{m[i,k] + m[k+1,i+2] + p_{i-1}p_k p_{i+2}\}$

$= \min\{\boldsymbol{m[i,i]} + \boldsymbol{m[i+1,i+2]} + \boldsymbol{p_{i-1}p_i p_{i+2}},$

$\boldsymbol{m[i,i+1]} + \boldsymbol{m[i+2,i+2]} + \boldsymbol{p_{i-1}p_{i+1}p_{i+2}}\}$

$\boldsymbol{m[1,3]} = \min\limits_{1 \le k < 3} \{m[1,k] + m[k+1,3] + p_0 p_k p_3\}$

$= \min\{m[1,1] + m[2,3] + p_0 p_1 p_3,\ m[1,2] + m[3,3] + p_0 p_2 p_3\}$

$= \min\{\mathbf{0 + 2625 + 5250},\ 15750 + 0 + 2250\}$

$= 7875$

$\boldsymbol{s[1,3]} = \mathbf{1}$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

共4个，其它长度为3的区间：

$m[2,4] = 4375,\ s[2,4] = 3$

$m[3,5] = 2500,\ s[3,5] = 3$

$m[4,6] = 3500,\ s[4,6] = 5$

④ 计算长度为4的子矩阵链

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

长度为4的子矩阵链区间是 $[i, i+3]$，$1 \le i \le 6-3$，

所以有：$m[i,i+3] = \min\limits_{i \le k < i+3} \{m[i,k] + m[k+1,i+3] + p_{i-1}p_k p_{i+3}\}$

$= \min\{\boldsymbol{m[i,i] + m[i+1,i+3] + p_{i-1}p_i p_{i+3}}$，

$\boldsymbol{m[i,i+1] + m[i+2,i+3] + p_{i-1}p_{i+1}p_{i+3}}$，

$\boldsymbol{m[i,i+2] + m[i+3,i+3] + p_{i-1}p_{i+2}p_{i+3}}\}$

$\boldsymbol{m[1,4]} = \min\limits_{1 \le k < 4} \{m[1,k] + m[k+1,4] + p_0 p_k p_4\}$

$= \min\{m[1,1] + m[2,4] + p_0p_1p_4, m[1,2] + m[3,4] + p_0p_2p_4, m[1,3] + m[4,4] + p_0p_3p_4\}$

$= \min\{0 + 4375 + 10500, 15750 + 750 + 4500, \mathbf{7875 + 0 + 1500}\}$

$= 9375$

$\boldsymbol{s[1,4]} = \mathbf{3}$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

共3个，其它长度为3的区间：

$m[2,5] = 7125$，$s[2,5] = 3$

$m[3,6] = 5375$，$s[3,6] = 3$

⑤ 计算长度为5的子矩阵链

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

长度为5的矩阵链区间是$[i, i+4]$，$1 \le i \le 6-4$，

所以有：$m[i,i+4] = \min\limits_{i \le k < i+4} \{m[i,k] + m[k+1,i+4] + p_{i-1}p_k p_{i+4}\}$

k 的取值有 i、$i+1$、$i+2$、$i+3$，所以花括号中有**4个值**比较大小

$$m[1,5] = \min_{1 \le k < 5} \{m[1,k] + m[k+1,5] + p_0 p_k p_5\}$$

$$= \min\{m[1,1] + m[2,5] + p_0p_1p_5,\ m[1,2] + m[3,5] + p_0p_2p_5,$$

$$m[1,3] + m[4,5] + p_0p_3p_5,\ m[1,4] + m[5,5] + p_0p_4p_5\}$$

$$= \min\{0 + 7125 + 21000, 15750 + 2500 + 9000, 7875 + 1000 + 3000, 9375 + 0 + 6000\}$$

$$= 11875$$

$$s[1,5] = 3$$

共2个，另一个长度为3的区间：

$m[2,6] = 10500$，$s[2,6] = 3$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

⑥ 计算长度为6的子矩阵链

长度为6的矩阵链区间只有一个：**[1, 6]**，

所以有：$m[1,6] = \min_{1\le k<6}\{m[1,k] + m[k+1,6] + p_0p_kp_6\}$

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

k 的取值有 1、2、3、4、5，所以花括号中有**5个值**比较大小

$$\boldsymbol{m[1,6]} = \min_{1\le k<6}\{m[1,k] + m[k+1,6] + p_0p_kp_6\}$$

$$= \min\{m[1,1] + m[2,6] + p_0p_1p_6, m[1,2] + m[3,6] + p_0p_2p_6,$$

$$m[1,3] + m[4,6] + p_0p_3p_6, m[1,4] + m[5,6] + p_0p_4p_6, m[1,5] + m[6,6] + p_0p_5p_6\}$$

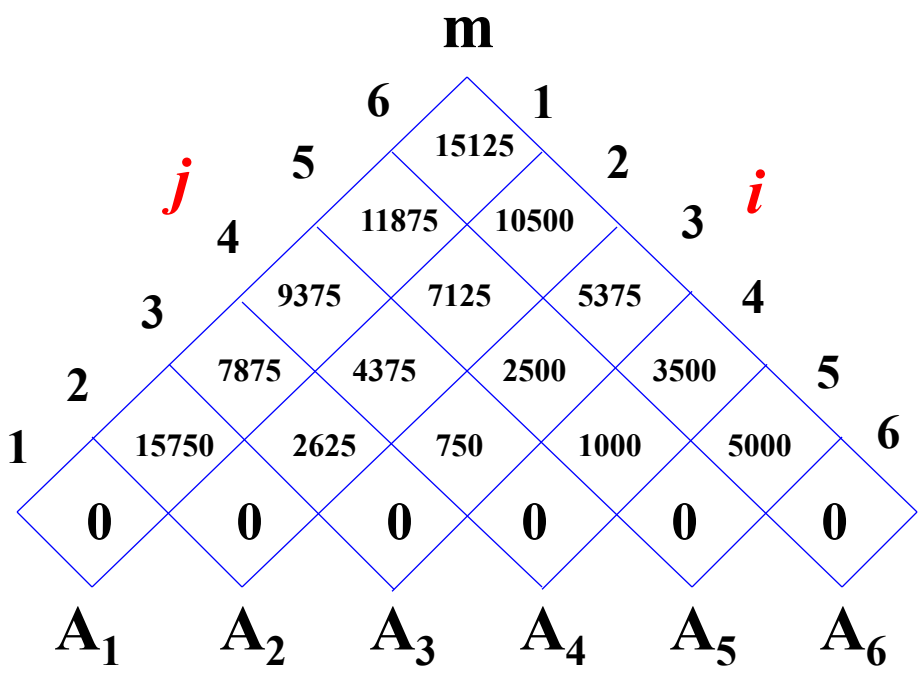
$$= \min\{0 + 10500 + 26250, 15750 + 5375 + 11250, \mathbf{7875 + 3500 + 3750}, 9375 + 5000 + 7500, 11875 + 0 + 15000\}$$

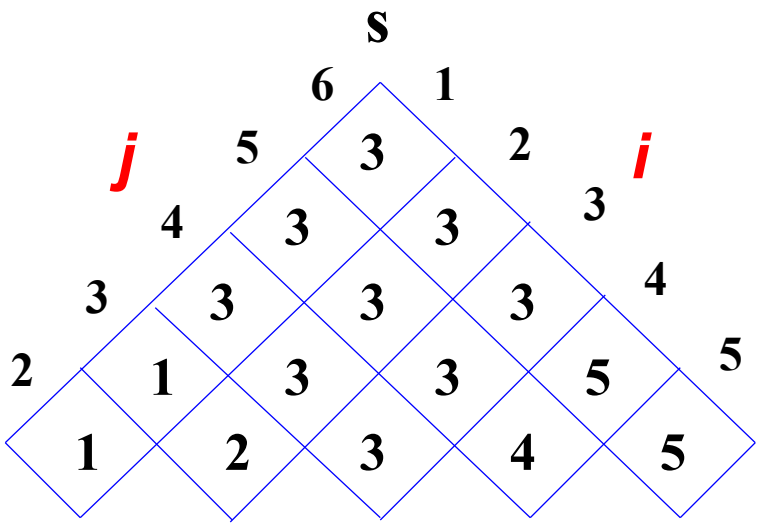
$$= 15125$$

$$\boldsymbol{s[1,6]} = \mathbf{3}$$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i\le k<j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

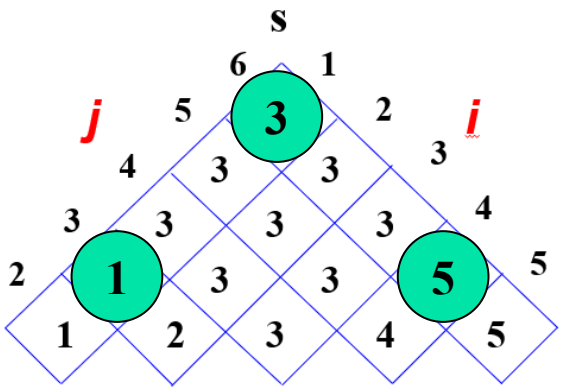
上述所有计算结果可以用如下所示的“表格”记录：





注意图中 *i*、*j* 坐标的标注

构造最优解（推导最优计算模式）



根据 s 表推导上述实例的最优计算模式：

第一重括号：　　$s[1, 6]=3$

$$(A_1\ A_2\ A_3)(A_4\ A_5\ A_6)$$

第二重括号：　　$s[1, 3]=1$　　$s[4, 6]=5$

$$(A_1\ (A_2\ A_3))((A_4\ A_5)\ A_6)$$

下面的**递归过程**以 s 表为输入，输出一个矩阵链的最优括号化方案。

根据 s 输出矩阵链乘的最优计算模式：

```
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 ")"
```

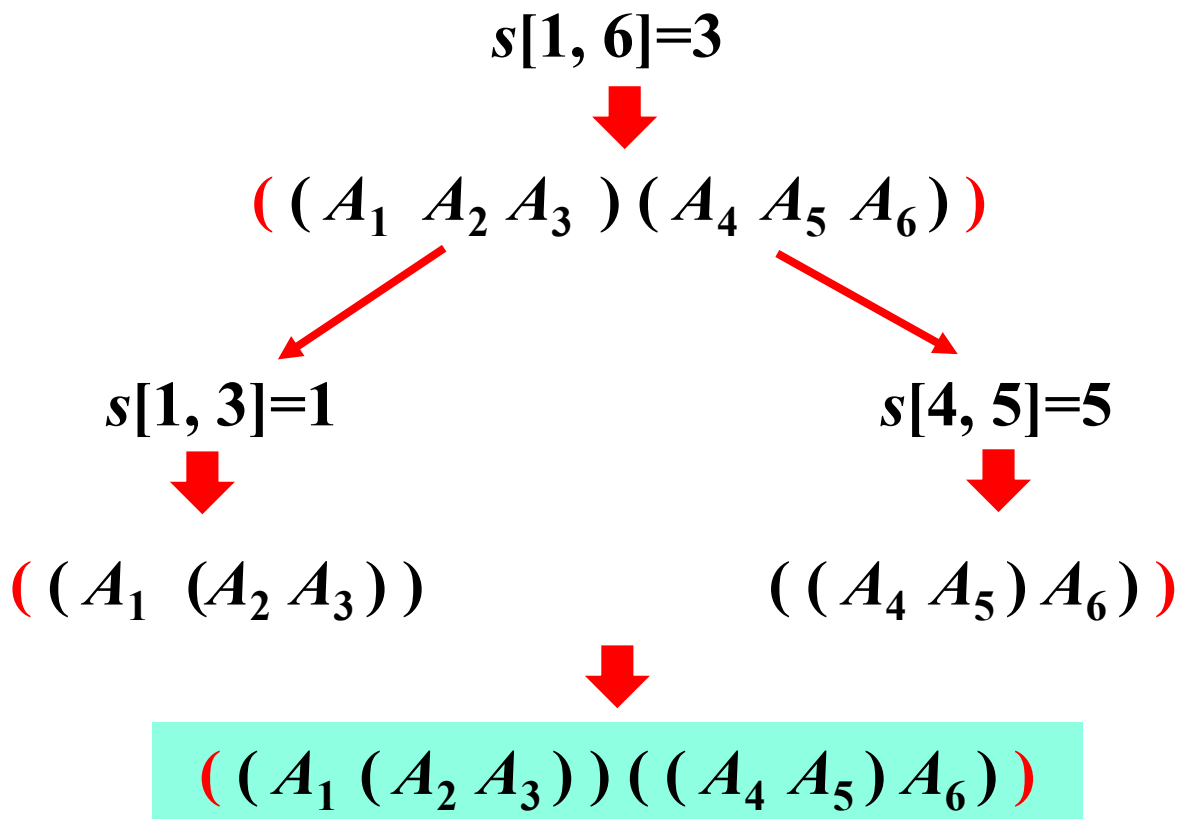
初始调用：PRINT-OPTIMAL-PARENS(**s**,**1**,**n**)

如：PRINT-OPTIMAL-PARENS(**s**,**1**,**6**)

输出：$((A_1\ (A_2\ A_3))\ ((A_4\ A_5)\ A_6))$

该算法在最外层多加了一层括号

PRINT-OPTIMAL-PARENS(*s*,1,6):



小结：用动态规划求解问题的步骤

第一步：证明问题满足最优子结构性

问题满足最优子结构性，就可以用动态规划策略求解，

否则，就不能用动态规划策略求解。

证明方法：**形式化推导**，**剪切-粘贴技术**（反证法）

第二步：推导状态转移方程

能否列出状态转移方程是解决问题的关键。**列不出状态转移方程就无法具体求解。**

第三步：计算

完成计算，**构造最优解**。

满足最优子结构性的一个典型实例：最短路径问题

例如，设在一个图中，结点序列 $v_1\, v_2\, v_3 \ldots v_n$ 是从结点 v_1到结点 v_n 的最短路径（边数最少或权值最小）。则该结点序列中的任意**子序列**都是该子序列**起始结点**到该子序列**终止结点**的**最短子路径**。如：

- $v_2\, v_3 \ldots v_n$ 是从 v_2 到 v_n 的最短子路径；
- $v_3\, v_4 \ldots v_n$ 是从 v_3 到 v_n 的最短子路径；
- 一般情况下， 若 $v_p\, v_{p+1} \ldots v_q$ 是 $v_1\, v_2\, v_3 \ldots v_n$ 中的一个连续子序列 $(p \le q,\ 1 \le p, q \le n)$，则该子序列是从 v_p 至 v_q 的最短子路径。

一般情况下，最短路径问题满足最优子结构性

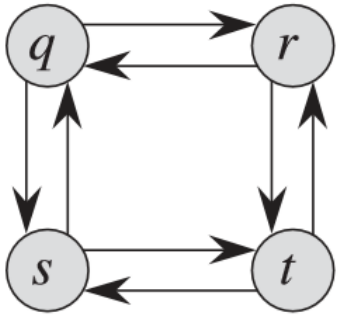
不满足最优子结构性质的两个例子

(1) 最长简单路径问题

在图中找一条从结点 u 到结点 v 的**边数最多**的**简单路径**。

- **最长简单路径问题不具有最优子结构性。**

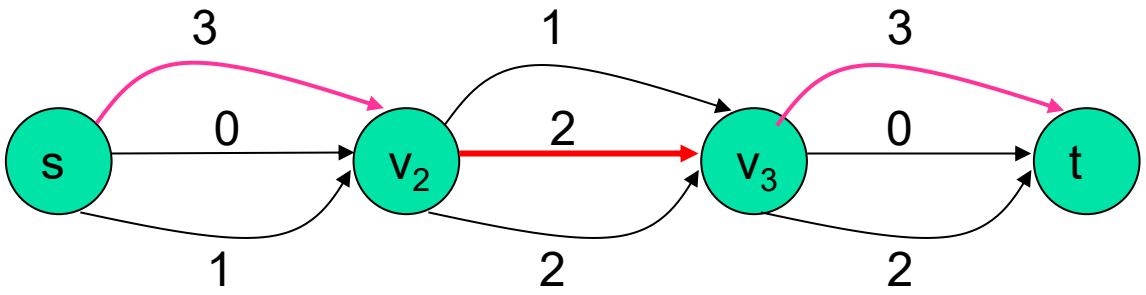
如图：



路径 $q \to r \to t$ 是从 q 到 t 的一条最长简单路径，但该路径中的子路径 $q \to r$ 不是从 q 到 r 的一条最长简单路径，$r \to t$ 也不是从 r 到 t 的一条最长简单路径。所以这个最长简单路径问题不具有最优子结构性质。

(2) 模4最优路径问题

如下图，求由 s 至 t 的一条路径，使得该路径的长度**模4**的余数最小。



通过观察可以看到，以下结点序列是从 s 到 t 的一条**模4最小**的最优路径：

$$s \xrightarrow{3} v_2 \xrightarrow{2} v_3 \xrightarrow{3} t$$

但显然该路径不具有最优子结构：

如 s 到 v_2 的最优路径是 $s \xrightarrow{0} v_2$，而不是上述最优路径里的子路径 $s \xrightarrow{3} v_2$。也就是全局最优，但局部不最优。

其它相关性质：

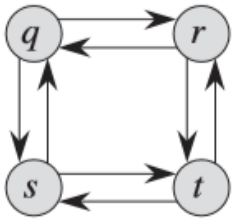
（1）子问题无关性

同一个问题的子问题，除非子问题是**重叠**或**包含**的，否则**子问题的解应是各不相关的**，可以各自独立求解并互不影响。

- **动态规划策略要求构成最优解的子问题间必须是无关的，否则不能使用动态规划策略求解。**

如：最长简单路径问题：**子问题相关**，不能用动态规划策略求解。如 $q \xrightarrow{\text{最长}} r$：$q\ s\ t\ r$

$r \xrightarrow{\text{最长}} t$：$r\ q\ s\ t$



子问题间互相交叉

最短路径问题：子问题不相关，满足最优子结构性。

$q \xrightarrow{\text{最短}} t$：　$q \xrightarrow{\text{最短}} r \xrightarrow{\text{最短}} t$

(2) 重叠子问题性质

就是有**重复子问题。**

用动态规划求解时，用表保存已解决子问题的答案，并在需要时通过查表得到子问题的答案——避免了大量的重复计算，提高了效率高。而如果没有重叠子问题，就失去了用动态规划求解问题的意义了。

总之，能用动态规划求解的问题，需满足**最优子结构性**、**子问题无关性**，并具有**重叠子问题性**。

动态规划带来的改进

(1) 因为**重叠子问题性质**，并造表保存子问题的解，避免对大量重复子问题的重复计算，从而节省了时间。

(2) 因为**最优子结构性**，所以只需用“小”子问题的最优解构造“大”子问题和原始问题的最优解，**舍去了对大量次优解的计算**，大大节省了计算量，因此可能有多项式的计算复杂度。

(3) **但依然要认识到：**

- **动态规划算法的计算量一般都比较大**：因为虽有以上改进，但还是要枚举所有可能的子问题，计算过程复杂；
- **空间需求大**：不管子问题以后是否会再被用到，只要它被计算过，都要将其结果填入表中备用，因而需要大量空间存储表。