

算法设计与分析

Computer Algorithm Design & Analysis

赵峰

zhaof@hust.edu.cn



Chapter 15

Dynamic Programming

动态规划

■
最优化问题：这一类问题的可行解可能有很多个。每个解都有一个值，我们希望寻找具有最优值的解（最小值或最大值）。

注：这里，我们称这个解为问题的一个最优解（an optimal solution），而不是the optimal solution，因为最优解也可能有多个。

——这种找最优解的问题通常称为**最优化问题**。

给定一个“函数” $F(X)$ ，以及“自变量” X ， X 应满足的一定条件，求 X 为怎样的值时， $F(X)$ 取得其最大值或最小值。

这里， $F(X)$ 称为“**目标函数**”， X 应满足的条件称为“**约束条件**”。
约束条件可用一个集合 D 表示为： $X \in D$ 。

求**目标函数 $F(X)$ 在约束条件 $X \in D$ 下的最小值或最大值问题**，就是一般**最优问题的数学模型**，可以用数学符号简洁地表示成：

$$\text{Min } F(X) \text{ 或 } \text{Max } F(X)$$

动态规划(Dynamic Programming)与分治法：通过组合子问题的解来求解原问题。

分治法：互不相交的子问题，递归地求解子问题。如果子问题有重叠，则递归求解中就会反复地求解这些公共子问题，造成算法效率的下降。

动态规划：有子问题重叠的情况，即不同的子问题具有公共的子子问题。动态规划算法对每个这样的子子问题只求解一次，将其解保存在一个表格中，再次碰到时，无需重新计算，只从表中找到上次计算的结果加以引用即可。

动态规划算法的步骤

1. 刻画一个最优解的结构特征；
2. 递归地定义最优解的值；
3. 计算最优解的值；
4. 利用计算出的信息，构造一个最优解。

15.1 钢条切割

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，怎么切割才能有最大的收益呢？

- 假设，切割工序本身没有成本支出。
- 假定出售一段长度为 i 英寸的钢条的价格为 p_i ($i=1,2,\dots$)。钢条的长度为 n 英寸。如下给出一个价格表 P 。

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

长度为 i 英寸的钢条可以为公司带来 p_i 美元的收益

钢条切割问题：

给定一段长度为 n 英寸的钢条和一个价格表 P ，求切割钢条方案，使得销售收益 r_n 最大。

分析：如果长度为 n 英寸的钢条的价格 p_n 足够大，则可能完全不需要切割，出售整条钢条是最好的收益。

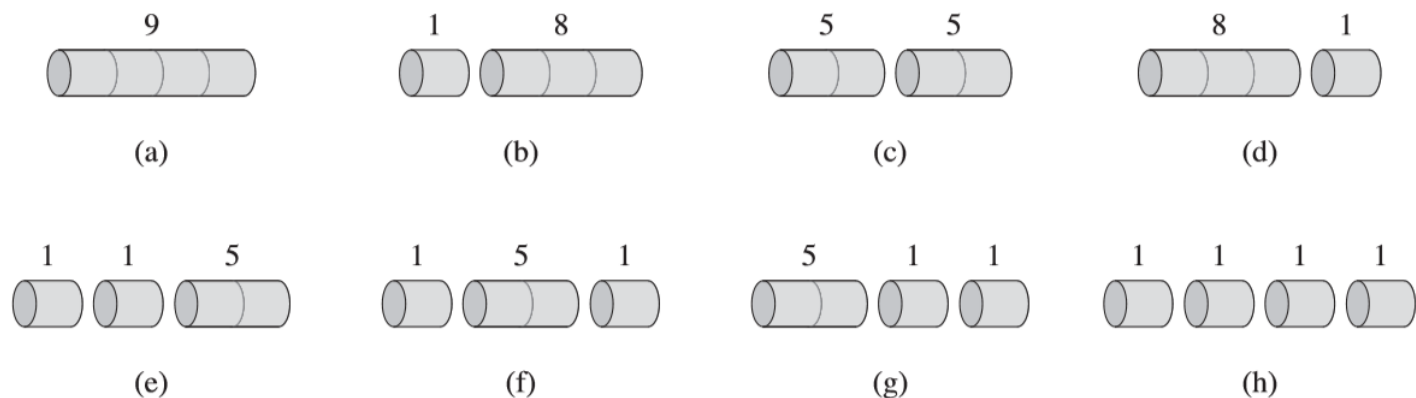
但由于每个 p_i 不同，可能切割后出售会更好一些。

考虑如下 $n=4$ 的情况。

■ 价格表：

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

■ 4英寸的钢条所有可能的切割方案。



4 英寸钢条的 8 种切割方案

最优方案：方案c，将4英寸的钢条切割为两段各长为2英寸的钢条，此时可产生的收益为10，为最优解。

- 长度为n英寸的钢条共有 2^{n-1} 中不同的切割方案。
- 如果一个最优解将总长度为n的钢条切割为k段，每段的长度为 i_j ($1 \leq j \leq k$)，则有：

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

得到的最大收益为： $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$

对于长度为 n ($n \geq 1$) 的钢条，设 r_n 是最优切割的收益

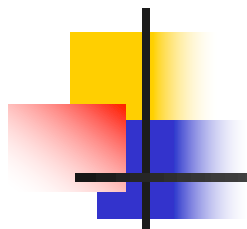
- 对**最优切割**，若其**首次切割**在位置 i ，钢条被分成长度为 i 和 $n-i$ 的两段，有：

$$r_n = r_i + r_{n-i}$$

- **一般情况**，任意切割点 j 都将钢条分为两段，长度分别为 j 和 $n-j$ ， $1 \leq j \leq n$ 。令 r_j 和 r_{n-j} 分别是**这两段的最优切割收益**，则该切割可获得的最好收益是： $r'_n = r_j + r_{n-j}$

所以有：

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



- 体现了动态规划的一个重要性质：**最优子结构性**

钢条切割问题的递归求解过程：

简化：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（递归求解）。

此时有：

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

一个自顶向下的递归求解过程可以描述如下：

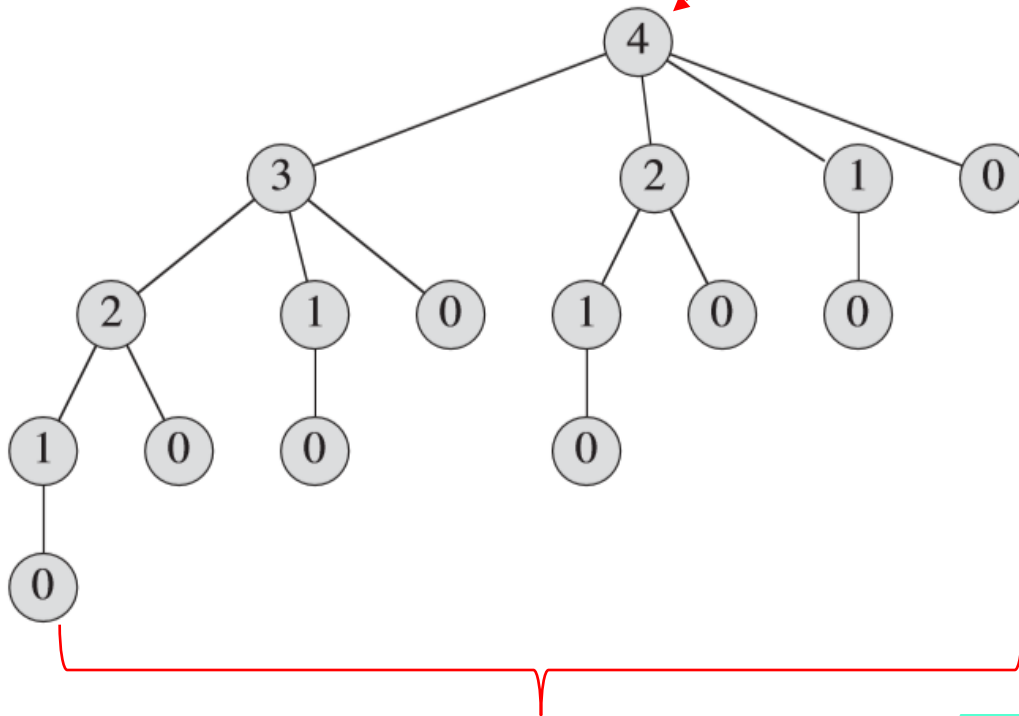
CUT-ROD(*p*, *n*)

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

其中，*p*是价格数组，*n*是钢条长度。

若*n*=0，则收益为0。

结点中的数字为对应子问题的规模



有 2^{n-1} 个叶结点。

```

CUT-ROD( $p, n$ )
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 

```



令 $T(n)$ 表示对规模为 n 的问题，CUT-ROD的调用次数，则有：

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

■ 可以证明： $T(n) = 2^n$

钢条切割问题的动态规划求解

对每个子问题只求解一次，并将结果保存下来。不必重新计算。

- 动态规划方法需要付出额外的空间保存子问题的解，是一种典型的时空权衡（time-memory trade-off）。
- 动态规划方法节省了时间：可以将一个指数时间的解转化为一个多项式时间的解。

■ 动态规划求解的两种方法

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

- **递归**编写过程，保存每个子问题的解。
- 通常保存在一个数组或散列表中。

MEMOIZED-CUT-ROD(p, n)

```
1  let  $r[0..n]$  be a new array
2  for  $i = 0$  to  $n$ 
3       $r[i] = -\infty$ 
4  return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```

(2) 自底向上法 (bottom-up method)

- 按由小到大的顺序顺次求解：当求解某个子问题时，它所依赖的更小子问题都已求解完毕。

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

```
1  let  $r[0..n]$  be a new array
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6           $q = \max(q, p[i] + r[j - i])$ 
7       $r[j] = q$ 
8  return  $r[n]$ 
```

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

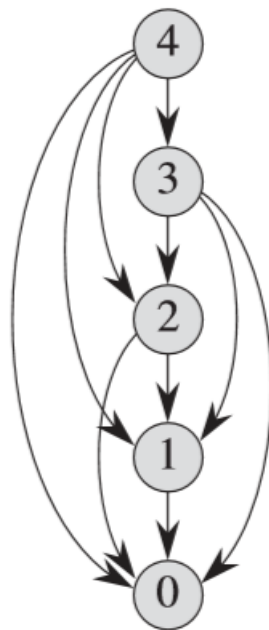
证明：略。

- 通常，自顶向下法和自底向上法具有相同的渐近运行时间。

子问题图

子问题图：用于描述子问题与子问题之间的依赖关系。

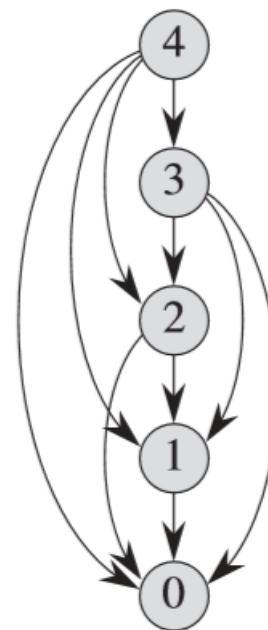
- 子问题图是一个有向图，每个顶点唯一地对应一个子问题。
- 若求子问题 x 的最优解时需要直接用到子问题 y 的最优解，则在子问题图中就会有一条从子问题 x 的顶点到子问题 y 的顶点的有向边。



- 子问题图是自顶向下递归调用树的“简化版”或“收缩版”
- 自底向上的动态规划方法处理子问题图中顶点的顺序为：

对一个给定的子问题 x ，在求解它之前先求解邻接至它的子问题。即，对于任何子问题，仅当它依赖的所有子问题都求解完成了，才会求解它。

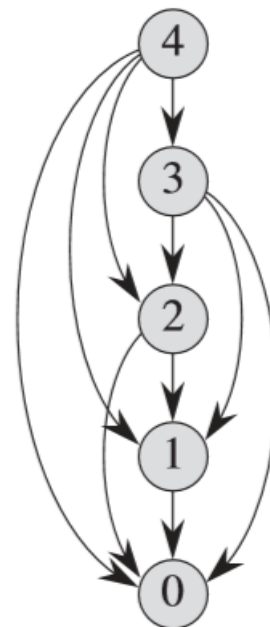
—— **逆序**进行处理。



基于子问题图 “估算” 算法的运行时间：

与子问题图中对应顶点的 “出度” 成正比，而子问题的数目等于子问题图的顶点数。

因此，一般情况下，动态规划算法的运行时间与顶点和边的数量呈线性关系。



■ 解重构

在求出最优收益之后，求出切割方案。

EXTENDED-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       $q = -\infty$ 
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  $s$ 
```

数组s用于记录对规模为j的钢条切割出的第一段钢条的长度s[j]。

- PRINT-CUT-ROD-SOLUTION输出完整的最优切割方案：

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

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

实例：

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

- PRINT-CUT-ROD-SOLUTION($p, 10$) : 10
- PRINT-CUT-ROD-SOLUTION($p, 7$) : 1 , 6

15.2 矩阵链乘法

两个矩阵的乘积：

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，

$$c_{ij} = \sum_{1 \leq k \leq r} a_{ik} b_{kj}, \quad i = 1, 2, \dots, p, \quad j = 1, 2, \dots, q$$

计算C共需要 pqr 次标量乘法运算。

矩阵链相乘

n个要连续相乘的矩阵构成一个矩阵链 $\langle A_1, A_2, \dots, A_n \rangle$, 计算矩阵链乘积。

- 矩阵链乘满足结合律。
- 相当于在矩阵之间加适当的括号。

如，已知四个矩阵 A_1, A_2, A_3, A_4 ，乘积 $A_1 A_2 A_3 A_4$ 可用五种不同的加括号方式完成：

$$\begin{aligned} & (A_1(A_2(A_3A_4))) & (A_1((A_2A_3)A_4)) \\ & ((A_1A_2)(A_3A_4)) & ((A_1(A_2A_3))A_4) \\ & (((A_1A_2)A_3)A_4) \end{aligned}$$



MATRIX-MULTIPLY(A, B)

```
1  if  $A.columns \neq B.rows$ 
2      error “incompatible dimensions”
3  else let  $C$  be a new  $A.rows \times B.columns$  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                   $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
9      return  $C$ 
```

- 矩阵“相容” (compatible)

问题： 不同的加括号方式带来不同的计算模式，代价不同

如，设有三个矩阵的链 $\langle A_1, A_2, A_3 \rangle$ ，假设维数分别为 10×100 ， 100×5 ， 5×50 。

1) $((A_1 A_2) A_3)$: **7500次**标量乘法运算。

2) $(A_1 (A_2 A_3))$: **75000次**标量乘法运算。

可见，**10倍**！

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

给定 n 个矩阵的链 $\langle A_1, A_2, \dots, A_n \rangle$ ，其中 $i=1, 2, \dots, n$ ，矩阵 A_i 的维数为 $p_{i-1} \times p_i$ 。求一个完全“**括号化方案**”，使得计算乘积 $A_1 A_2 \dots A_n$ 所需的**标量乘法次数最小**。

■令 $p(n)$ 表示 n 个矩阵的链相乘时，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： **$P(n) = \Omega(2^n)$**

1) 最优括号化方案的结构特征

用记号 $A_{i,j}$ 表示 $A_i A_{i+1} \dots A_j$ 通过加括号后得到的一个**最优计算模式**，且恰好在 A_k 与 A_{k+1} 之间分开。

则“前缀”子链 $A_i A_{i+1} \dots A_k$ 必是一个最优的括号化子方案，记为 $A_{i,k}$ ；同理“后缀”子链 $A_{k+1} A_{k+2} \dots A_j$ 也必是一个最优的括号化子方案，记为 $A_{k+1,j}$ 。

- 证明：如若不然，设 $A'_{i,k}$ 是 $\langle A_i, A_{i+1}, \dots, A_k \rangle$ 一个代价更小的计算模式，则由 $A'_{i,k}$ 和 $A_{k+1,j}$ 构造计算过程 $A'_{i,j}$ ，代价将比 $A_{i,j}$ 小，这与 $A_{i,j}$ 是最优链乘模式相矛盾。

对 $A_{k+1,j}$ 亦然。

——**最优子结构性。**

2. 递归求解方案

(1) 递推关系式

令 $m[i,j]$ 为计算矩阵链 A_i, \dots, A_j 所需的标量乘法运算次数的最小值。则

含义：

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

- 对 $m[i,j]$ 和任意的 k 所分开的矩阵链乘 $\langle A_i, A_{i+1}, \dots, A_j \rangle$ ， $m[i,j]$ 等于计算子乘积 $A_{i,k}$ 最小代价 $m[i,k]$ 加计算子乘积 $A_{k+1,j}$ 的最小代价 $m[k+1,j]$ ，再加上这两个子矩阵相乘的代价 $p_{i-1}p_kp_j$ 。
- $m[1,n]$ 是计算 $A_{1,n}$ 的最小代价。



$s[i,j]$ 记录使 $m[i,j]$ 取最小值的 k 。

下述过程**MATRIX-CHAIN-ORDER**采用**自底向上表格法**计算 n 个矩阵链乘的最优模式。

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           $m[i, j] = \infty$ 
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                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

自底向上完成 $m[i,j]$ 的计算：在 $m[i,i]=0$ 的基础上，求出所有 $m[i,j]$ 。最后算出 $m[1,n]$ 。

A diamond-shaped lattice diagram representing a 2D hexagonal lattice. The top node is labeled 'm'. The left and right boundaries are labeled 'j' and 'i' respectively. The bottom nodes are labeled $A_1, A_2, A_3, A_4, A_5, A_6$. The lattice contains numerical values in its cells, with some highlighted in red (2625) and blue (7875). A horizontal line is drawn across the middle of the lattice.

A diamond-shaped grid of 15 cells, each containing a number from 1 to 5. The grid is oriented with a single cell at the top and a row of five cells at the bottom. The numbers are distributed as follows:

- Row 1 (top): 1 cell with 's'.
- Row 2: 2 cells with 6 and 1.
- Row 3: 3 cells with 5, 3, and 2. A red 'j' is to the left of the 5, and a red 'i' is to the right of the 2.
- Row 4: 4 cells with 4, 3, 3, and 3.
- Row 5: 5 cells with 3, 3, 3, 3, and 4.
- Row 6: 6 cells with 2, 1, 3, 3, 5, and 5.
- Row 7 (bottom): 5 cells with 1, 2, 3, 4, and 5.

时间复杂度分析

算法的主体由一个三层循环构成。MATRIX-CHAIN-ORDER的算法复杂度是 $\Omega(n^3)$ 。

另，算法需要 $\Theta(n^2)$ 的空间保存m和s。

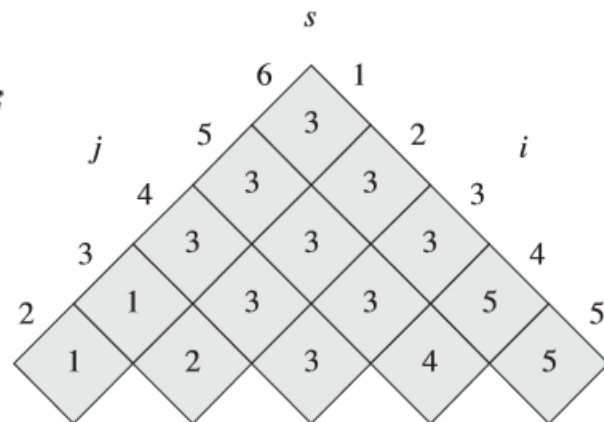
```
MATRIX-CHAIN-ORDER(p)
1  n = p.length - 1
2  let m[1..n, 1..n] and s[1..n - 1, 2..n] be new tables
3  for i = 1 to n
4      m[i, i] = 0
5  for l = 2 to n           // l is the chain length
6      for i = 1 to n - l + 1
7          j = i + l - 1
8          m[i, j] = ∞
9          for k = i to j - 1
10             q = m[i, k] + m[k + 1, j] + pi-1pkpj
11             if q < m[i, j]
12                 m[i, j] = q
13                 s[i, j] = k
14  return m and s
```

(4) 构造最优解

- $S[i,j]$ 记录了 $A_i A_{i+1} \dots A_j$ 的最优括号化方案的“最后一个”分割点 k 。
- $A_{1\dots n}$ 的最优方案中最后一次矩阵乘运算是 $A_{1\dots s[1,n]} A_{s[1,n]+1\dots n}$ 。

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 ")"
```

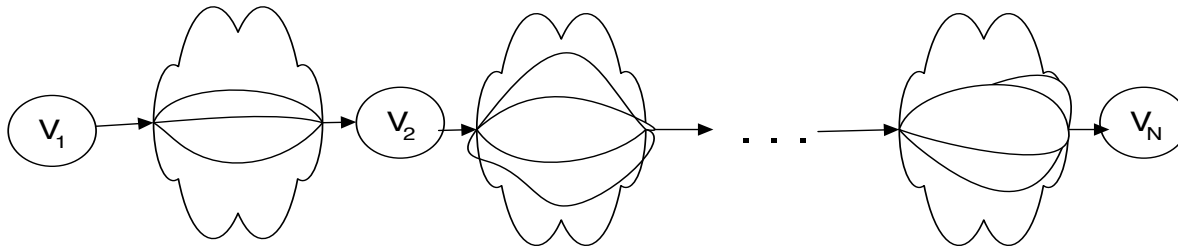


例: PRINT-OPTIMAL-PARENS($s, 1, 6$) $\Rightarrow ((A1(A2A3))((A4A5)A6))$

15.3 动态规划的一般方法

动态规划(dynamic programming)是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法。

1. 多阶段决策过程



假设事件在初始状态后需要经过 n 个这样的阶段。一般情况下，从 i 阶段发展到 $i+1$ 阶段（ $0 \leq i < n$ ）可能有多种不同的途径，而事件必须从中选择一条途径往前进展。使过程从一个状态演变到下一状态。

决策：在一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择称为决策。也就是在两个阶段间选择发展途径的行为。

决策变量：描述决策的变量称决策变量，用一个数或一组数表示。不同的决策对应着不同的数值。

决策序列：事件的发展过程之中需要经历 n 个阶段，需要做 n 次“决策”，这些“决策”就构成了事件整个发展过程的一个决策序列。

多阶段决策过程：具备上述性质的过程称为多阶段决策过程 (multistep decision process)，求解多阶段决策过程问题就是求取事件发展的决策序列。

无后效性：对任意的阶段 i ，阶段 i 以后的行为仅依赖于 i 阶段的状态，而与 i 阶段之前过程是如何达到这种状态的方式无关，这种性质称为**无后效性**。

状态的这个性质意味着过程的历史只能通过当前的状态去影响它的未来的发展，而不能直接作用于未来的发展。



状态转移方程：第 $i+1$ 阶段的状态变量 $x(i+1)$ 的值随 $x(i)$ 和第 i 阶段的决策 $u(i)$ 的值变化而变化，可以把这一关系看成 $(x(i), u(i))$ 与 $x(i+1)$ 的函数对应关系，用

$$x(i+1) = T_i(x(i), u(i))$$

表示。

这种从 i 阶段到 $i+1$ 阶段的状态转移规律称为**状态转移方程**。

最优化问题：

多阶段决策过程的最优化问题就是：求能够获得问题最优解的决策序列——最优决策序列。

2. 多阶段决策过程的求解策略

问题的决策序列表示为： (x_1, x_2, \dots, x_n) ，其中， x_i 表示第*i*阶段的决策：

$$S_0 \xrightarrow{x_1} S_1 \xrightarrow{x_2} S_2 \xrightarrow{x_3 \dots} \dots \xrightarrow{x_n} S_n$$

1) 枚举法

若问题的决策序列由*n*次决策构成，每一阶段分别有 p_1 、 p_2 、...、 p_n 选择，则可能的决策序列将有 $p_1 p_2 \dots p_n$ 个。

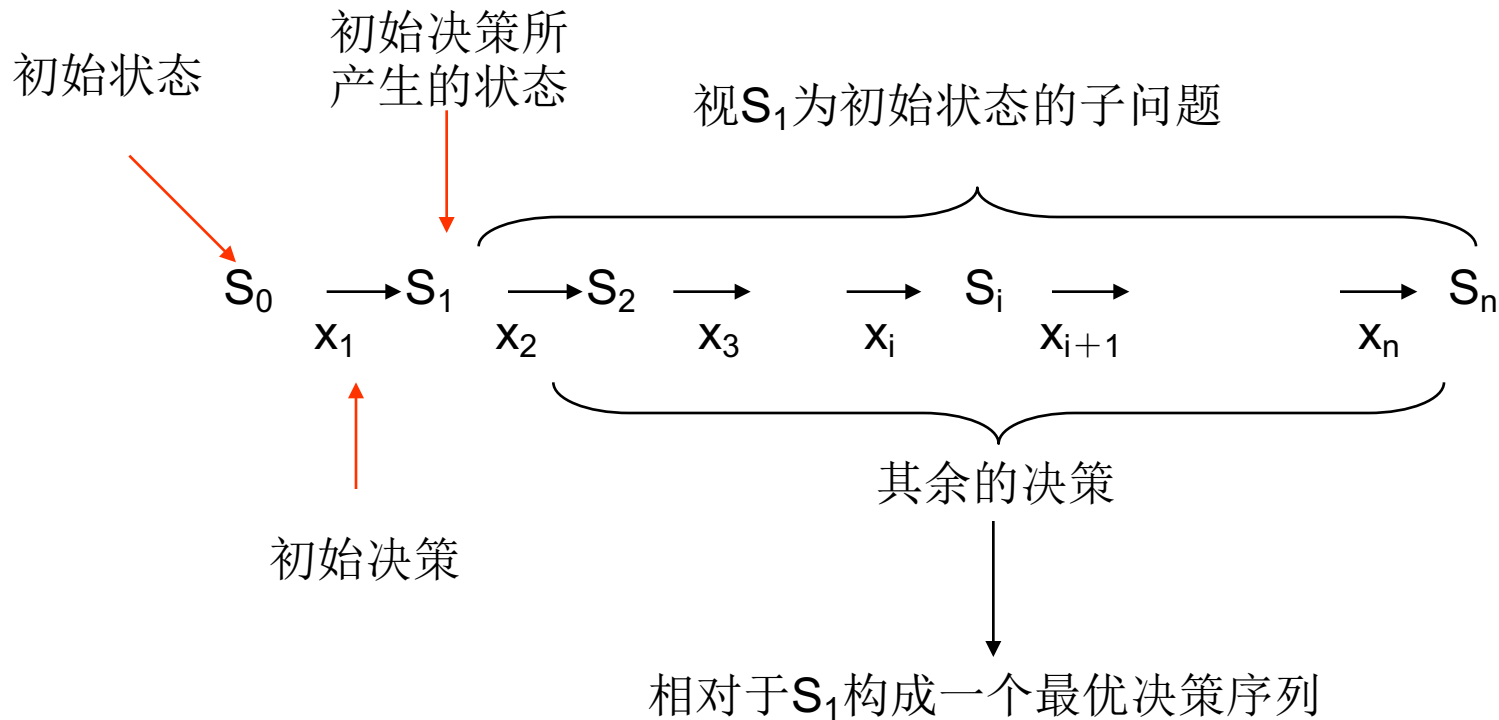
2) 动态规划

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，创立了解决这类过程优化问题的新方法——**动态规划**。

动态规划(dynamic programming)是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法。

3. 最优化原理(Principle of Optimality)

过程的最优决策序列具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。



例：最短路径

若 $v_1v_2v_3\cdots v_n$ 是从节点 v_1 到节点 v_n 的最短路径。则：

- ▶ $v_2v_3\cdots v_n$ 是从 v_2 到 v_n 的最短子路径；
- ▶ $v_3\cdots v_n$ 是从 v_3 到 v_n 的最短子路径；

.....

推广：

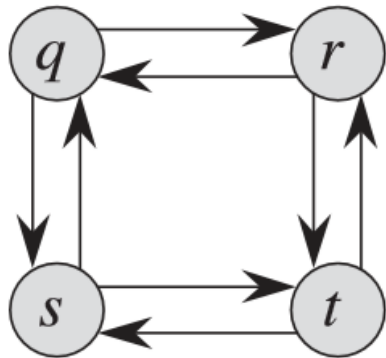
对 $v_1v_2v_3\cdots v_n$ 中的任意一段子路径：

$$v_pv_{p+1}\cdots v_q \ (p \leq q, 1 \leq p, q \leq n),$$

将代表从 v_p 至 v_q 的最短子路径。

■ 最长简单路径问题

- 最短路径问题具有最优子结构性
- 最长简单路径问题不具有最优子结构性



此例显示了无权有向图最长简单路径问题不具有最优子结构性质。路径 $q \rightarrow r \rightarrow t$ 是从 q 到 t 的一条最长简单路径，但 $q \rightarrow r$ 不是从 q 到 r 的一条最长简单路径， $r \rightarrow t$ 同样不是从 r 到 t 的一条最长简单路径

利用动态规划求解问题的方法：

第一步：证明问题满足最优性原理

第二步：获得问题状态的递推关系式（即状态转移方程）

能否求得各阶段间状态变换的递推关系式是解决问题的关键。

$$f(x_1, x_2, \dots, x_i) \rightarrow x_{i+1} \quad \text{向后递推}$$

$$\text{或} \quad f(x_i, x_{i+1}, \dots, x_n) \rightarrow x_{i-1} \quad \text{向前递推}$$

回顾：

1) 不管是钢管切割问题还是矩阵链乘问题，

证明问题的最优解满足最优子结构性

2) 状态转移方程

➤ 钢管切割问题：
$$r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$$

➤ 矩阵链乘问题：
$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

对动态规划带来的改进的理解

- 若问题的决策序列由 n 次决策构成，而每次决策有 p 种选择，若采用枚举法，则可能的决策序列将有 p^n 个。而**利用动态规划策略的求解过程中仅保存了所有子问题的最优解**，而舍去了所有不能导致问题最优解的次优决策序列，可能有**多项式**的计算复杂度。

子问题无关性

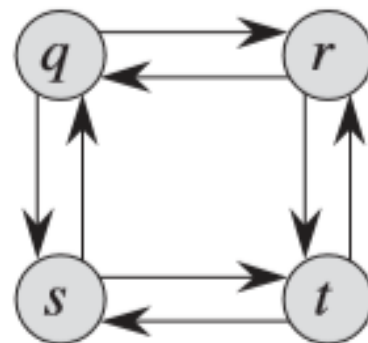
能够用动态规划策略求解的问题，构成最优解的子问题间必须是无关的

■最长简单路径问题

- 子问题间相关，不能用动态规划策略求解。

■最短路径问题

- 子问题不相关，满足最优子结构性，可以用动态规划问题求解。



“剪切-粘贴” (cut-and-paste) 技术：

一般用反证法证明：假定子问题的解不是其自身的最优解，而存在“**更优的解**”，那么我们可以从原问题的解中“剪切”掉这些“最优解”的部分，而将“更优的解”粘贴进去，从而得到原问题的一个“更优”解，这与最初的解是原问题最优解的前提假设相矛盾。因此，不可能存在“更优的解”。反之，原问题的子问题的解应是其自身的最优解——最优子结构性成立。

关于动态规划算法时间的一般讨论

求原问题最优解的代价通常就是子问题最优解的代价再加上由此次选择直接产生的代价。

最优解的构造

通常定义一个表，记录每个子问题所做的选择。当求出最优解后，利用该表回推求取最优方案。

- 如矩阵链乘法中的表 $s[1...n]$ 。

备忘（查表）

15.4 最长公共子序列



一个应用背景：**基因序列比对**。

用英文单词的首字母来代表四种碱基，这样一个DNA螺旋被表示为有穷字符集{A,C,G,T}上的一个串。

如：两个有机体的DNA分别为

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$
$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$


图 4-4 DNA 分子的结构和碱基配对

“基因序列比对”：对两个由A、C、G、T组成的字符串的比较。

■度量DNA的相似性：


➤ *Edit distance 15-5*

➤ 在 S_1 和 S_2 中找出第三个螺旋 S_3 ，使得 S_3 中的基以同样的顺序出现在 S_1 和 S_2 中，但不一定连续。

然后视 S_3 的长度，确定 S_1 和 S_2 的相似度。 S_3 越长， S_1 和 S_2 的相似度越大，反之越小。

➤ 如上面的两个DNA串中，最长的公共螺旋是

$S_3 = \text{GTCGTCGGAAGCCGGCCGAA}。$



怎么找最长的公共螺旋——两个字符串的公共非连续子串，
称为最长公共子序列。

1、最长公共子序列的定义

(1) 子序列

给定两个序列 $X = \langle x_1, x_2, \dots, x_n \rangle$ 和序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$, 若存在 X 的一个严格递增下标序列 $\langle i_1, i_2, \dots, i_k \rangle$, 使得对所有 $j = 1, 2, \dots, k$, 有 $x_{i_j} = z_j$, 则称 Z 是 X 的子序列。

如： $Z = \langle B, C, D, B \rangle$ 是 $X = \langle A, B, C, B, D, A, B \rangle$ 的一个子序列，
相应下标序列为 $\langle 2, 3, 5, 7 \rangle$ 。

2) 公共子序列

对给定的两个序列X和Y，若序列Z既是X的子序列，也是Y的子序列，则称Z是X和Y的**公共子序列**。

如， $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$ ，则序列 $\langle B, C, A \rangle$ 是X和Y的一个公共子序列。

3) 最长公共子序列

两个序列的长度最大的公共子序列称为它们的最长公共子序列。

如， $\langle B, C, A \rangle$ 是上面X和Y的一个公共子序列，但不是X和Y的最长公共子序列。最长公共子序列是 $\langle B, C, B, A \rangle$ 。

怎么求最长公共子序列？

2、最长公共子序列问题 (Longest-Common-Subsequence, **LCS**)

——求 (两个) 序列的最长公共子序列

前缀：给定一个序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ ，对于 $i = 0, 1, \dots, m$ ，定义 X 的第 i 个前缀为 $X_i = \langle x_1, x_2, \dots, x_i \rangle$ ，即前 i 个元素构成的子序列。

如， $X = \langle A, B, C, B, D, A, B \rangle$ ，则

$$X_4 = \langle A, B, C, B \rangle。$$

$$X_0 = \Phi。$$

1) LCS问题的最优子结构性

定理6.2 设有序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ ，并设序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ 为 X 和 Y 的任意一个**LCS**。

- (1) 若 $x_m = y_n$ ，则 $z_k = x_m = y_n$ ，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS。
- (2) 若 $x_m \neq y_n$ ，则 $z_k \neq x_m$ 蕴含 Z 是 X_{m-1} 和 Y 的一个LCS。
- (3) 若 $x_m \neq y_n$ ，则 $z_k \neq y_n$ 蕴含 Z 是 X 和 Y_{n-1} 的一个LCS。

证明：

(1) 如果 $z_k \neq x_m$ ，则可以添加 x_m （也即 y_n ）到 Z 中，从而可以得到 X 和 Y 的一个长度为 $k+1$ 的公共子序列。这与 Z 是 X 和 Y 的最长公共子序列的假设相矛盾，故必有 $z_k = x_m = y_n$ 。

假设 X_{m-1} 和 Y_{n-1} 有一个长度大于 $k-1$ 的公共子序列 W ，则将 x_m 添加到 W 上就会产生一个长度大于 k 的公共子序列，与 Z 是 X 和 Y 的最长公共子序列的假设相矛盾，故 Z_{k-1} 必是 X_{m-1} 和 Y_{n-1} 的LCS。

(2) 若 $z_k \neq x_m$ ，设 X_{m-1} 和 Y 有一个长度大于 k 的公共子序列 W ，

则 W 也应该是 X_m 和 Y 的一个公共子序列。这与 Z 是 X 和 Y 的一个LCS的假设相矛盾。故 Z 是 X_{m-1} 和 Y 的一个LCS。

(3) 同 (2) 。

(证毕)

上述定理说明，两个序列的一个LCS也包含了两个序列的前缀的一个LCS，即LCS问题具有最优子结构性质。

2) 递推关系式

记, $c[i,j]$ 为前缀序列 X_i 和 Y_j 的一个 LCS 的 **长度**。则有

$$c[i, j] = \begin{cases} 0 & \text{如果 } i = 0 \text{ 或 } j = 0 \\ c[i-1, j-1] + 1 & \text{如果 } i, j > 0 \text{ 且 } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{如果 } i, j > 0 \text{ 且 } x_i \neq y_j \end{cases}$$


注： 以上情况涵盖了 X_m 和 Y_n 的 LCS 的所有情况。

3) LCS的求解

过程 **LCS-LENGTH(X,Y)** 求序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 的LCS的长度，表 **$c[1..m, 1..n]$** 中包含每一阶段的LCS长度， **$c[m,n]$** 等于X和Y的LCS的长度。

同时，设置表 **$b[1..m, 1..n]$** ，记录当前 **$c[i,j]$** 的计值情况，以此来构造该LCS。

LCS-LENGTH(X,Y)



```
m ← length[X], n ← length[Y]
for i ← 1 to m do c[i,0] ← 0 repeat
for j ← 1 to n do c[0,j] ← 0 repeat
for i ← 1 to m do
  for j ← 1 to n do
    if  $x_i = y_j$  then
       $c[i,j] \leftarrow c[i-1,j-1] + 1$ 
       $b[i,j] \leftarrow \nwarrow$ 
    else if  $c[i-1,j] \geq c[i,j-1]$  then
       $c[i,j] \leftarrow c[i-1,j]$ 
       $b[i,j] \leftarrow \uparrow$ 
    else  $c[i,j] \leftarrow c[i,j-1]$ 
       $b[i,j] \leftarrow \leftarrow$ 
    endif
  endif
endif
repeat
repeat
END LCS-LENGTH
```

LCS-LENGTH的时间复杂度为 $O(mn)$

例，下图给出了在 $X = \langle A, B, C, B, D, A, B \rangle$ 和 $Y = \langle B, D, C, A, B, A \rangle$ 上运行LCS-LENGTH计算出的表。

		j	0	1	2	3	4	5	6
			y _j (B) D (C) A (B) (A)						
i	x _i								
0			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

说明：

- 1) 第 i 行和第 j 列中的方块包含了 $c[i, j]$ 的值以及 $b[i, j]$ 记录的箭头。
- 2) 对于 $i, j > 0$ ，项 $c[i, j]$ 仅依赖于是否有 $x_i = y_j$ 及项 $c[i-1, j]$ ， $c[i, j-1]$ 和 $c[i-1, j-1]$ 的值。
- 3) 为了重构一个LCS，从右下角开始跟踪 $b[i, j]$ 箭头即可，即如图所示中的阴影。
- 4) 图中， $c[7, 6] = 4$ ，

$LCS(X, Y) = \langle B, C, B, A \rangle$

构造一个LCS

表b用来构造序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 的一个LCS：按照反序，从 $b[m, n]$ 处开始，沿箭头在表格中向上跟踪。
每当在表项 $b[i, j]$ 中：

- 遇到一个“↖”时，意味着 $x_i = y_j$ 是LCS的一个元素，下一步继续在 $b[i-1, j-1]$ 中寻找上一个元素；
- 遇到“←”时，下一步继续到 $b[i, j-1]$ 中寻找上一个元素；
- 遇到“↑”时，下一步继续到 $b[i-1, j]$ 中寻找上一个元素。

过程PRINT-LCS按照上述规则输出X和Y的LCS

PRINT-LCS(b,X,i,j)

if $i=0$ or $j=0$ then return;

if $b[i,j]="\nwarrow"$ then

PRINT-LCS(b,X,i-1,j-1)

print x_i

else if $b[i,j]="\uparrow"$ then

PRINT-LCS(b,X,i-1,j)

else PRINT-LCS(b,X,i,j-1)

endif

endif

END PRINT-LCS

4) 算法的改进

(1) 可以去掉表b，直接基于c求LCS。

思考：如何做到这一点？

有何改进？

(2) 算法中，每个 $c[i,j]$ 的计算仅需c的两行的数据：
正在被计算的一行和前面的一行。

故可以仅用表c中的 $2 * \min(m, n)$ 项以及 $O(1)$ 的
额外空间即可完成整个计算。

思考：如何做到这一点

15.5 最优二叉搜索树

- **场景**：语言翻译，从英语到法语。
- **方法**：创建一棵**二叉搜索树**，以n个英语单词作为关键字（对应地可以找到其法语单词）构建。
- **思路**：频繁使用的单词，如the，尽可能靠近根，而不经常出现的单词可以离根远一些。
 - 思考：如果反之会怎样？
- **引入新的因素**：使用频率

■ 最优二叉搜索树

(1) 二叉搜索树 (二分检索树)

二叉搜索树 T 是一棵二元树，它或者为空，或者其每个结点含有一个可以比较大小的数据元素，且有：

- T 的左子树的所有元素比根结点中的元素小；
- T 的右子树的所有元素比根结点中的元素大；
- T 的左子树和右子树也是二叉搜索树。

注：

二分检索树要求树中所有结点的元素值互异

二叉搜索树的检索算法

算法 SEARCH(T, X, i)

//在二叉搜索树 T 中检索 X 。如果 X 不在 T 中，则置 $i=0$ ；否则 i 有 $IDENT(i)=X$ //

$i \leftarrow T$

while $i \neq 0$ do

case

: $X < IDENT(i)$: $i \leftarrow LCHILD(i)$ //若 X 小于 $IDENT(i)$ ，则在左子树中继续查找 //

: $X = IDENT(i)$: return // X 等于 $IDENT(i)$ ，则返回 //

: $X > IDENT(i)$: $i \leftarrow RCHILD(i)$ //若 X 大于 $IDENT(i)$ ，则在右子树中继续查找 //

endcase

repeat

end SEARCH

(2) 最优二叉搜索树

给定一个 n 个关键字的已排序的序列 $K = \langle k_1, k_2, \dots, k_n \rangle$ (不失一般性, 设 $k_1 < k_2 < \dots < k_n$), 对每个关键字 k_i , 都有一个概率 p_i 表示其搜索频率。根据 k_i 和 p_i 构建一个二叉搜索树 T , 每个 k_i 对应树中的一个结点。

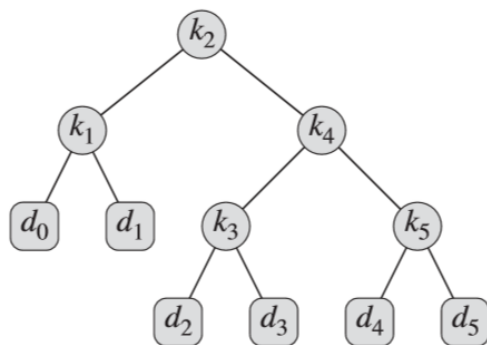
- 若搜索对象 x 等于某个 k_i , 则一定可以在 T 中找到结点 k_i ;
- 若 $x < k_1$ 或 $x > k_n$ 或 $k_i < x < k_{i+1} (1 \leq i < n)$, 则在 T 中将搜索失败。
 - 为此引入外部结点 d_0, d_1, \dots, d_n , 用来表示不在 K 中的值, 称为伪结点。
 - 这里每个 d_i 代表一个区间, d_0 表示所有小于 k_1 的值, d_n 表示所有大于 k_n 的值, 对于 $i=1, 2, \dots, n-1$, d_i 表示所有在 k_i 和 k_{i+1} 之间的值。
 - 同时每个 d_i 也有一个概率 q_i 表示搜索对象 x 恰好落入区间 d_i 的频率。

例：设有 $n=5$ 个关键字的集合，每个 k_i 的概率 p_i 和 d_i 的概率 q_i 如表所示：

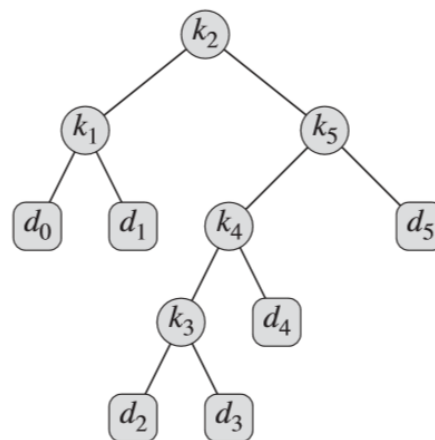
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

这里有： $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$ 。

基于该集合，两棵可能的二叉搜索树如下所示。



(a)



(b)

二叉搜索树的期望搜索代价

- 代价等于从根结点开始访问结点的数量。

- 从根结点开始访问结点的数量等于结点在T中的深度+1；

- 二叉搜索树T的期望代价

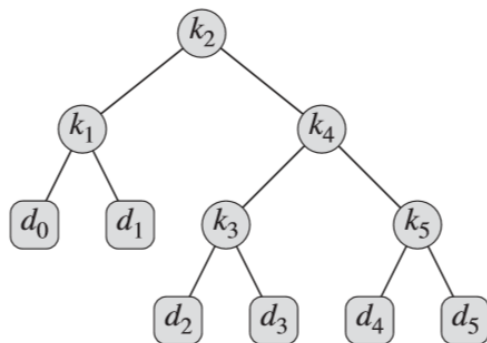
- 记 $\text{depth}_T(i)$ 为结点i在T中的深度，则T搜索代价的期望为：

$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i, \end{aligned}$$

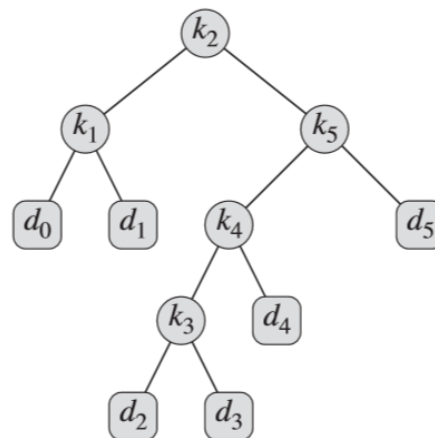
$n=5$,

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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



(a)



(b)

➤ (a)的期望搜索代价为2.80。

➤ (b)的期望搜索代价为2.75。

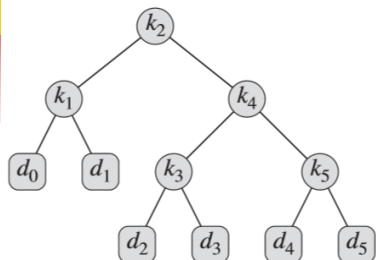
$$\begin{aligned}
 E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\
 &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i ,
 \end{aligned}$$

➤ 树b是当前问题实例的一棵最优二叉搜索树

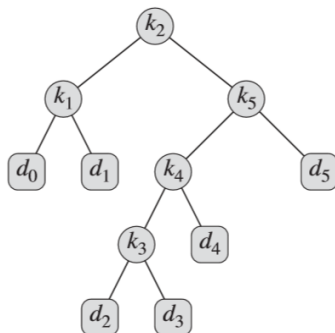
$n=5$,

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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



(a)



(b)

逐结点计算期望搜索代价如表所示：

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

最优二叉搜索树的定义：

对给定的概率集合，期望搜索代价最小的二叉搜索树称为**最优二叉搜索树**。

- 对给定的概率集合，怎么构造最优二叉搜索树？
- 关键问题：**谁是树根**？

■ 用动态规划构造最优二叉搜索树

(1) 最优二叉搜索树的结构

最优二叉搜索树的**最优子结构**：如果一棵**最优二叉搜索树** T 有一棵包含关键字 k_i, \dots, k_j 的子树 T' ，则 T' 必然是包含关键字 k_i, \dots, k_j 和伪关键字 d_{i-1}, \dots, d_j 的子问题的最优解。

证明：用剪切-粘贴法证明

对关键字 k_i, \dots, k_j 和伪关键字 d_{i-1}, \dots, d_j ，如果存在子树 T'' ，其期望搜索代价比 T' 低，那么将 T' 从 T 中删除，将 T'' 粘贴到相应位置，则可以得到一棵比 T 期望搜索代价更低的二叉搜索树，与 T 是最优的假设矛盾。

(1) 构造最优二叉搜索树

- 利用最优二叉搜索树的最优子结构性来构造最优二叉搜索树。

分析：

对给定的关键字 k_i, \dots, k_j ，若其最优二叉搜索树的根结点是 k_r ($i \leq r \leq j$)，则 k_r 的左子树中包含关键字 k_i, \dots, k_{r-1} 及伪关键字 d_{i-1}, \dots, d_{r-1} ，右子树中将含关键字 k_{i+1}, \dots, k_j 及伪关键字 d_r, \dots, d_j 。

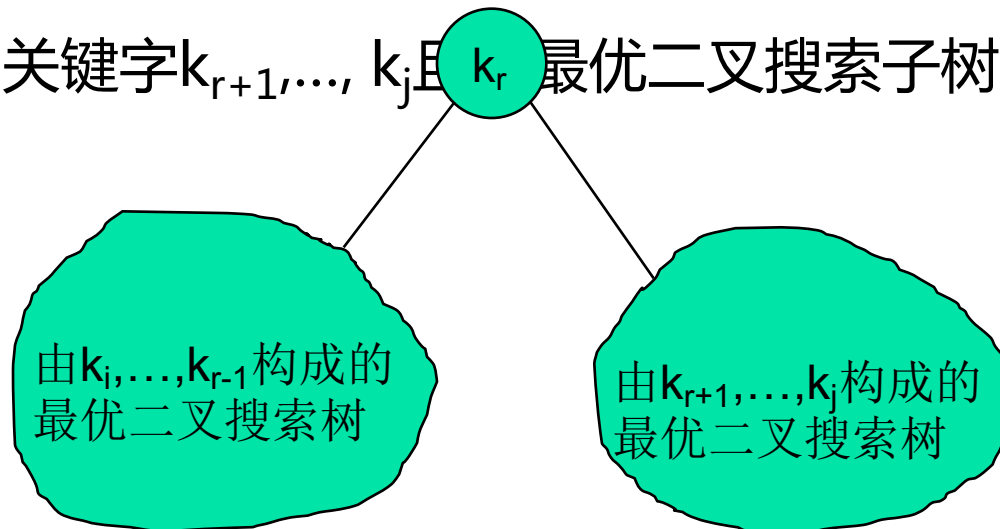
策略：检查所有可能的根结点 k_r ($i \leq r \leq j$)，如果事先分别求出包含关键字 k_i, \dots, k_{r-1} 和关键字 k_{r+1}, \dots, k_j 的最优二叉搜索子树，则可保证找到原问题的最优解。

计算过程：求解包含关键字 k_i, \dots, k_j 的最优二叉搜索树，其中，
 $i \geq 1, j \leq n$ 且 $j \geq i-1$ 。

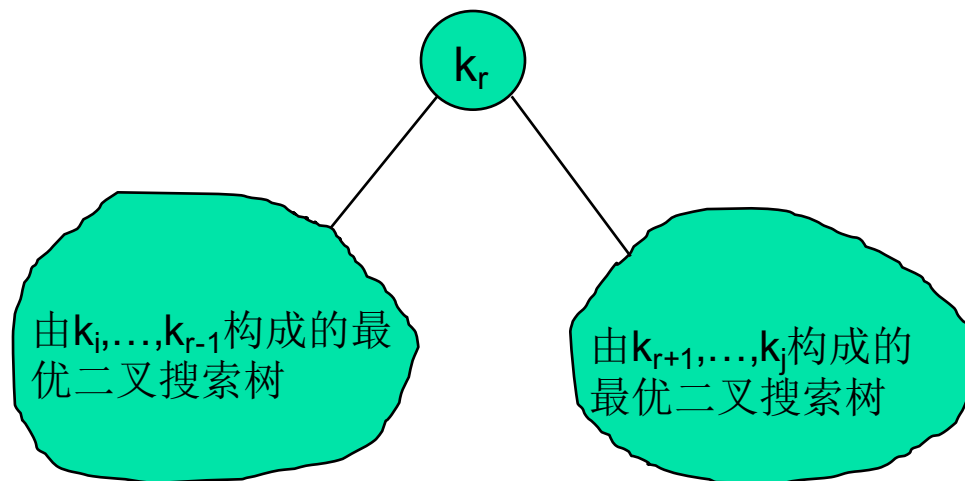
定义 $e[i, j]$ ：为包含关键字 k_i, \dots, k_j 的最优二叉搜索树的期望搜索代价。

➤ $e[1, n]$ 为问题的最终解。

1) 当 $j \geq i$ 时，我们需要从 k_i, \dots, k_j 中选择一个根结点 k_r ，其左子树包含关键字 k_i, \dots, k_{r-1} 且是最优二叉搜索子树，其右子树包含关键字 k_{r+1}, \dots, k_j 且是最优二叉搜索子树。



而当一棵子树成为一个结点的子树时，其期望搜索代价有以下变化：



- 子树的每个结点的深度增加1，根据二叉搜索树的期望搜索代价计算公式，这棵子树对根为 k_r 的树的期望搜索代价的贡献是其期望搜索代价+其所含所有结点的概率之和：

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l .$$

若 k_r 为包含关键字 k_i, \dots, k_j 的最优二叉搜索树(树根), 则其期望搜索代价与左、右子树的期望搜索代价 $e[i, r-1]$ 和 $e[r+1, j]$ 有如下递推关系:

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) .$$

其中,

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j)$$

故有:

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) .$$

求 k_r 的递归公式为:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 , \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j . \end{cases}$$

(2) 边界条件： $j=i-1$

- 存在 $e[i, i-1]$ 和 $e[j+1, j]$ 的边界情况。
- 此时，子树不包含实际的关键字，而只包含伪关键字 d_{i-1} ，其期望搜索代价为： $e[i, i-1] = q_{i-1}$ 。

(3) 构造最优二叉搜索树

定义 $\text{root}[i, j]$ ，保存计算 $e[i, j]$ 时使 $e[i, j]$ 取得最小值的 r 。在求出 $e[1, n]$ 后，利用 root 的记录构造出整棵最优二叉搜索树。

计算最优二叉搜索树的期望搜索代价

- $e[1..n+1, 0..n]$: 用于记录所有 $e[i, j]$ 的值。

- 注 : $e[n+1, n]$ 对应伪关键字 d_n 的子树 ;

- $e[1, 0]$ 对应伪关键字 d_0 的子树。

- $root[1..n]$: 用于记录所有最优二叉搜索子树的根结点 ,
包括整棵最优二叉搜索树的根。

- $w[1..n+1, 0..n]$: 用于子树保存增加的期望搜索代价 , 且有

$$w[i, j] = w[i, j - 1] + p_j + q_j .$$

这样 , 对于 $\Theta(n^2)$ 个 $w[i, j]$, 每个的计算时间仅为 $\Theta(1)$ 。

- 过程OPTIMAL-BST利用概率列表 p 和 q ，对 n 个关键字，计算最优二叉搜索树的表 e 和 $root$ 。

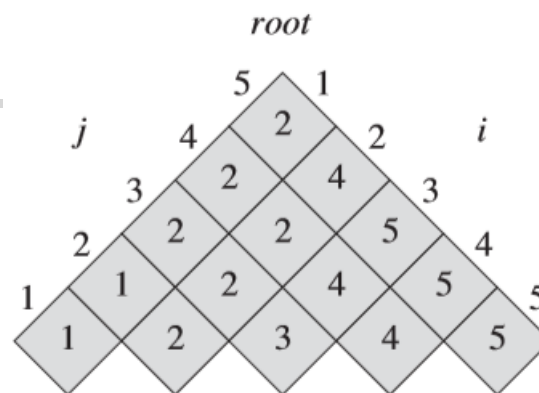
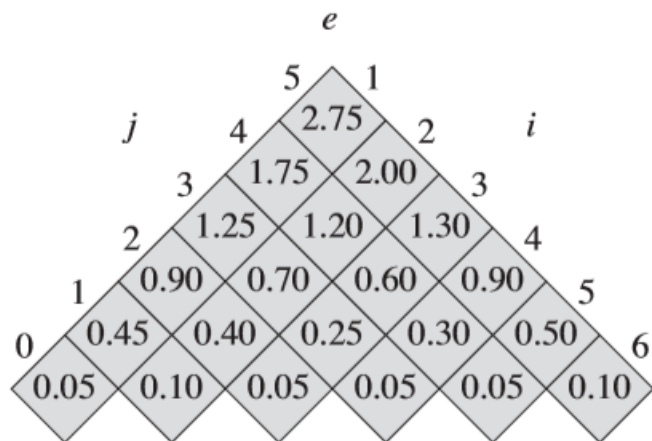
OPTIMAL-BST(p, q, n)

```
1  let  $e[1..n+1, 0..n]$ ,  $w[1..n+1, 0..n]$ ,  
    and  $root[1..n, 1..n]$  be new tables  
2  for  $i = 1$  to  $n + 1$   
3       $e[i, i - 1] = q_{i-1}$   
4       $w[i, i - 1] = q_{i-1}$   
5  for  $l = 1$  to  $n$   
6      for  $i = 1$  to  $n - l + 1$   
7           $j = i + l - 1$   
8           $e[i, j] = \infty$   
9           $w[i, j] = w[i, j - 1] + p_j + q_j$   
10         for  $r = i$  to  $j$   
11              $t = e[i, r - 1] + e[r + 1, j] + w[i, j]$   
12             if  $t < e[i, j]$   
13                  $e[i, j] = t$   
14                  $root[i, j] = r$   
15  return  $e$  and  $root$ 
```

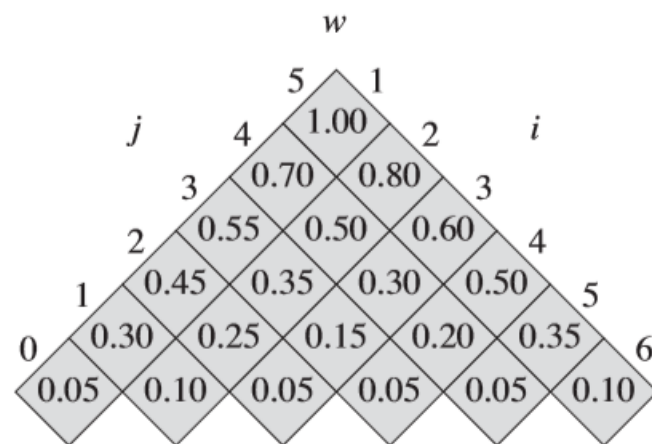
时间复杂度 $\Theta(n^3)$

■ 例：n=5，

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j. \end{cases}$$



$$w[i, j] = w[i, j-1] + p_j + q_j.$$

