



算法设计与分析

Computer Algorithm Design & Analysis

2022.03

何琨

brooklet60@hust.edu.cn

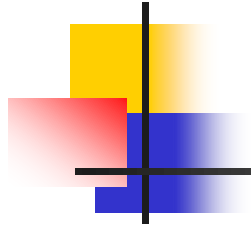
群 号 : 638562638



Chapter 15

Dynamic Programming

动态规划



Advanced Design and Analysis Techniques

▪
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**。

最优化问题的分类：

根据描述约束条件和目标函数的数学模型的特性和问题的求解方法的不同，可分为：线性规划、整数规划、非线性规划、动态规划等问题。而研究解决这些问题的科学一般就总称之为最优化理论和方法。

在运筹学领域有对最优化理论更深入的研究

本章介绍动态规划（Dynamic Programming，简称DP）

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。

注：

- 1) 前三步是动态规划算法求解问题的基础。如果仅需要一个最优解的值，而非解本身，可以忽略步骤4。
- 2) 如果确实要做步骤4，则有时需要在执行步骤3的过程中维护一些额外的信息，以便用来构造一个最优解。
- 3) 第三步通常采用**自底向上**的方法计算最优解；



Richard Bellman



Richard Bellman(1920-1984)
美国艺术与科学研究院 (1975)
美国国家工程院 (1977)
美国国家科学院 (1983)

1953年提出动态规划算法

“在生命的最后11年，虽饱受脑部手术并发症的困扰，仍发表超过100篇论文。”

Stuart Dreyfus (2003) "Richard Ernest Bellman". In: International Transactions in Operational Research. Vol 10, no. 5, pp. 543-545.

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 ($i=1, 2, \dots$)，下面是一个价格表 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。

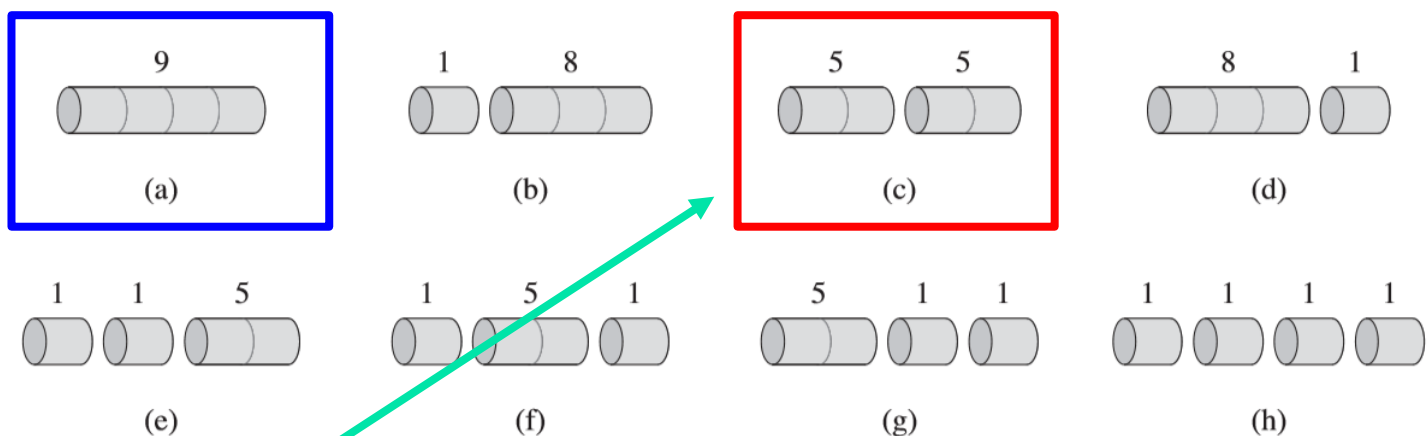
思考：先按 p_i 最大 i 出售，剩余的再按较小 p_j 出售，是否就可以获得最好的收益？（贪心策略）

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

考虑如下 $n=4$ 的情况。

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。
 - 每一英寸都可切割，共有n-1个切割点
- 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： $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$

如，从价格表可得以下基本方案：

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

$r_1=1$ ，切割方案1=1（无切割）

$r_2=5$ ，切割方案2=2（无切割）

$r_3=8$ ，切割方案3=3（无切割）

$r_4=10$ ，切割方案4=2+2

$r_5=13$ ，切割方案5=2+3

$r_6=17$ ，切割方案6=6（无切割）

$r_7=18$ ，切割方案7=1+6或7=2+2+3

$r_8=22$ ，切割方案8=2+6

$r_9=25$ ，切割方案9=3+6

$r_{10}=30$ ，切割方案10=10（无切割）

对于长度为 n ($n \geq 1$) 的钢条，设 r_n 是最优切割的收益，那么该如何获得该最优切割？

- 对**最优切割**，从左往右看，设**首次切割**在位置 i ，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子切割收益，则有：

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

- **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： $r'_n = r_j + r_{n-j}$

- **j 和 i 有什么关系呢？**

- 这样的j有n种选择(包括不切割)，而**最优切割是能够获得最大收益的切割方案**，所以有：

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

- 即，首次切割后，将两段钢条看成两个独立的钢条切割问题实例。若分别获得两段钢条的最优切割收益 r_j 和 r_{n-j} ，则**原问题的解就可以通过组合这两个相关子问题的最优子解获得**。
- 这里体现了该问题最优解的一个重要性质：**最优子结构性**
 - 如果 $r_n = r_i + r_{n-i}$ 是最优切割收益，则 r_i 、 r_{n-i} 是相应子问题的最优切割收益（为什么？）。

钢条切割问题的朴素递归求解过程：

对切割过程可做如下简化：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），但对左边的一段不再进行切割。

此时有：

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

即，此时，原问题的最优解只包含一个相关子问题（右端剩余部分）的解，而不是两个。

一个自顶向下的递归求解过程可以描述如下：

其中， p 是价格数组， n 是钢条长度。

若 $n=0$ ，则收益为0。

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

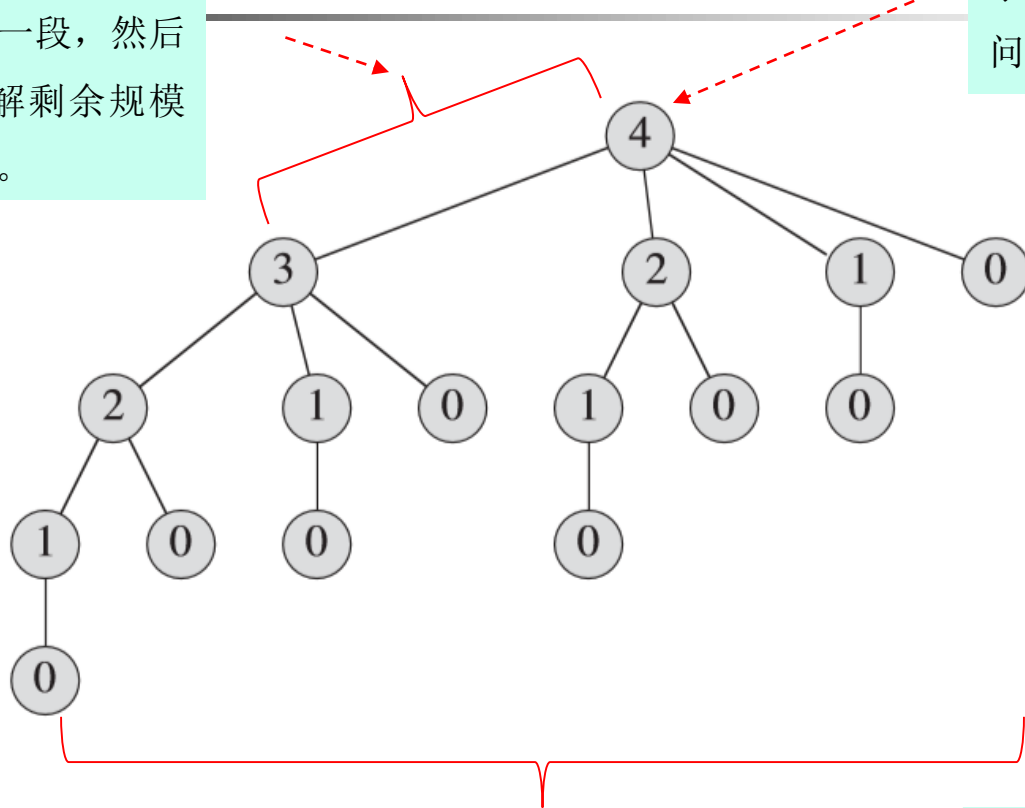
该过程的效率很差：

- 存在一些相同的子问题，CUT-ROD反复地用一些相同的参数做重复的递归调用。

如 $n=4$ ，CUT-ROD的递归执行过程可以用递归调用树表示为：

从父结点 s 到子结点 t 的边表示从钢条左端切下长度为 $s-t$ 的一段，然后继续递归求解剩余规模为 t 的子问题。

结点中的数字为对应子问题的规模



一般来说，这棵递归调用树有 2^n 个结点，其中有 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： $T(n) = 2^n$

```
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  $q$ 
```

钢条切割问题的动态规划求解

动态规划方法将**仔细安排**求解顺序，对每个子问题只求解一次，并将**结果保存下来**。如果再次需要此子问题的解，只需查找保存的结果，而不必重新计算。

- 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）。

■ 动态规划求解的两种方法

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

- 依旧按照**递归**的形式编写过程，但处理过程中会**保存每个子问题的解**。
- 当需要时，过程首先检查是否已经保存过此解。
 - ◆ 如果是，则直接返回保存的值；
 - ◆ 否则按照通常的方式计算该子问题。

带备忘：“记住”之前已经计算出来的结果。

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  $q$ 
```

辅助数组 $r[0..n]$ 用于保存子问题的结果。

- 初始化为 $-\infty$;
- 当有新的结果时, $r[n]$ 保存结果 q ;
- 当 $r[n] \geq 0$ 时, 直接引用其中已保存的值。

(2) 自底向上法 (bottom-up method)

➤ 将子问题按规模排序:

最小子问题、较小子问题、…、较大子问题、原问题。

➤ 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，结果已经保存，故可以直接引用并组合出它自身的解

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  $r[n]$ 
```

■ 这里采用子问题的自然顺序：若 $i < j$ ，则规模为 i 的子问题比规模为 j 的子问题“更小”。

■ 过程依次求解规模为 $j=0,1,\dots,n$ 的子问题。

■ 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  $r[n] \geq 0$ 
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  $q$ 
```

自顶向下



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  $r[n]$ 
```

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。
 - 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（剪枝）。
 - 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。

■ 对比简单递归和动态规划

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  $q$ 
```

递归：“硬”求解。整个过程对存在的重复子问题的重复计算，造成效率低下。

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  $q$ 
```

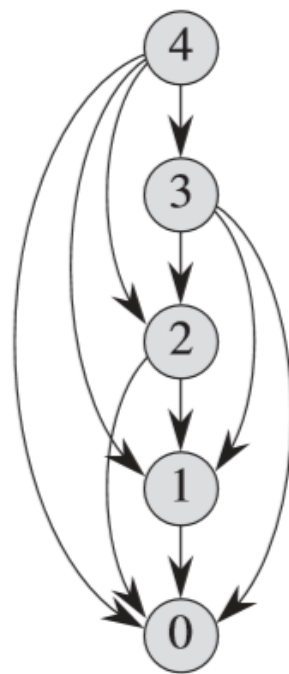
动态规划：递归只是处理的一部分。在求解的过程中记录了重复子问题的解。通过“引用”以前的计算结果，避免重复计算，提高效率。

子问题图

当思考一个动态规划问题时，应该弄清楚所涉及的子问题与子问题之间的**依赖关系**，可用**子问题图**描述：

子问题图：用于描述子问题与子问题之间的依赖关系。

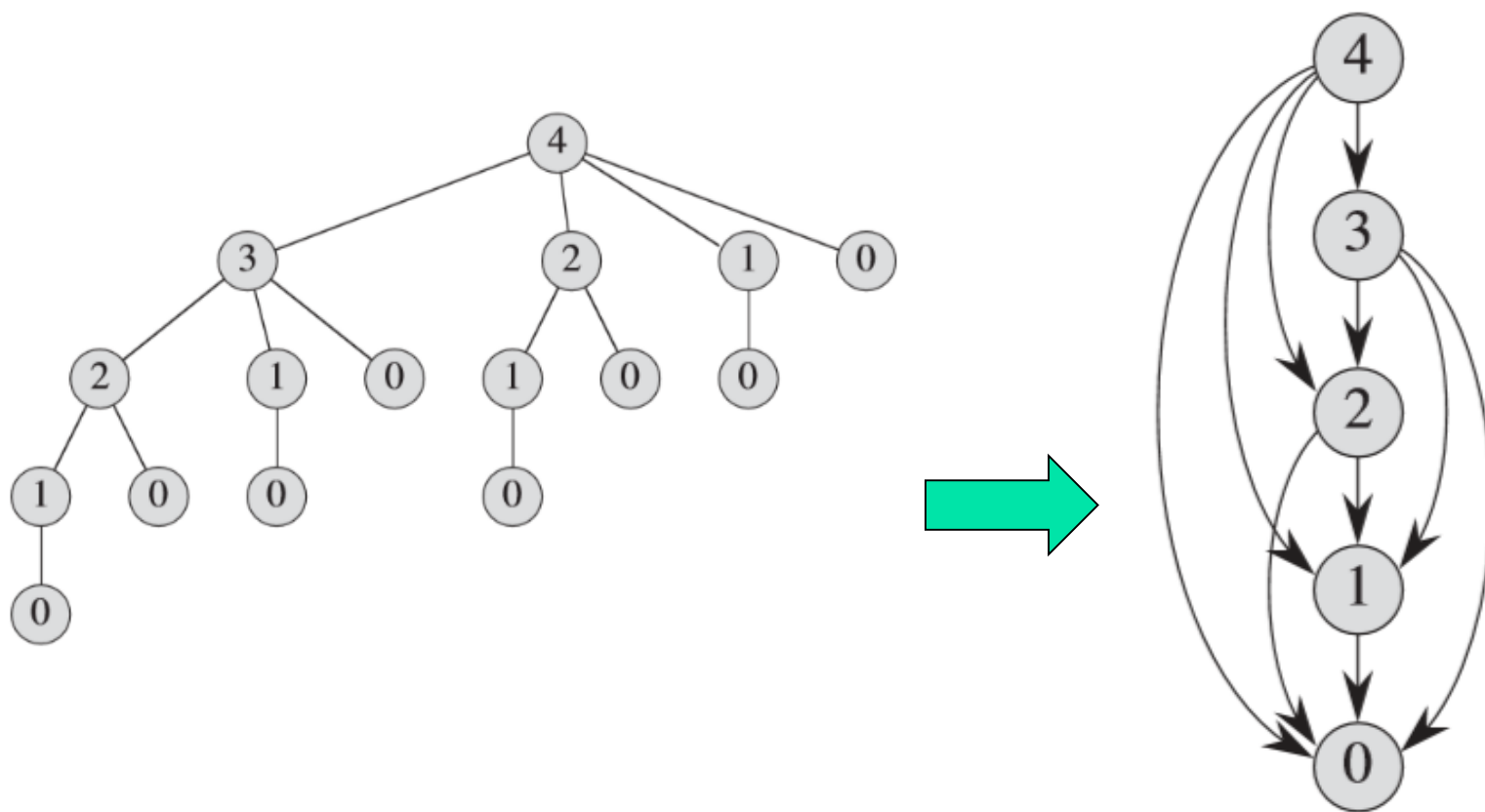
- 子问题图是一个有向图，每个顶点唯一地对应一个子问题。
- 若求子问题 x 的最优解时需要直接用到子问题 y 的最优解，则在子问题图中就会有一条从子问题 x 的顶点到子问题 y 的顶点的有向边。



$n=4$ 时，钢条切割问题的子问题图。顶点的标号给出了子问题的规模。有向边 (x,y) 表示当求解子问题 x 时需要子问题 y 的解。

- 子问题图是自顶向下递归调用树的“简化版”或“收缩版”

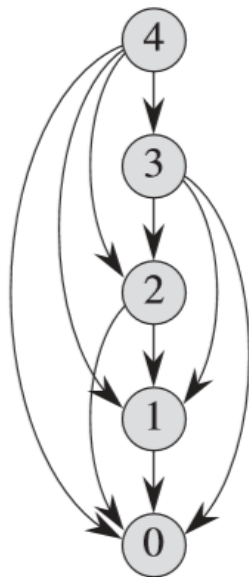
—— 递归树中所有对应相同子问题的结点合并为子问题图中的一个单一顶点，相关的边都从父结点指向子结点。



- **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。

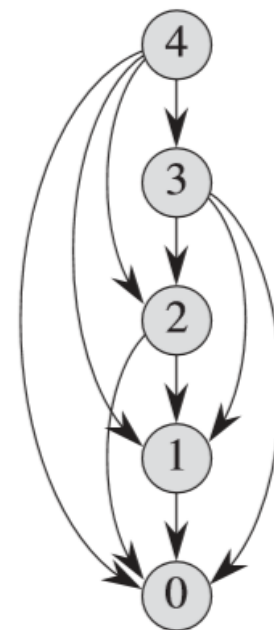
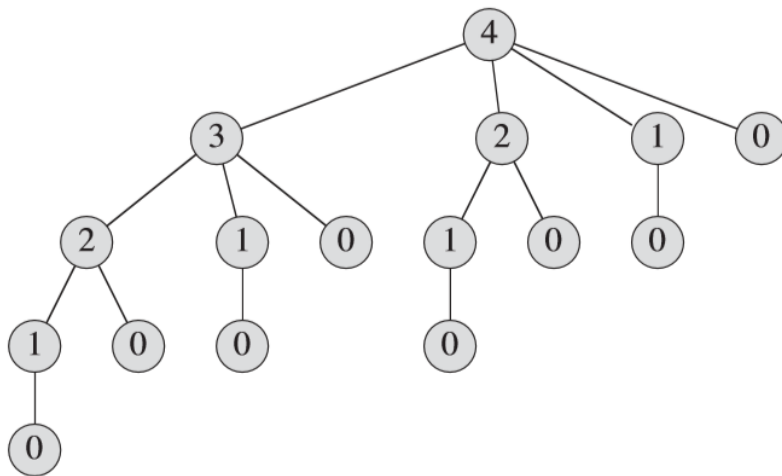
—— **逆拓扑序**，**深度优先原则**进行处理。



基于子问题图 “估算” 算法的运行时间：

算法的运行时间等于所有子问题求解的时间之和。子问题图中，子问题对应顶点，子问题的数目等于顶点数。一个子问题的求解时间与子问题图中对应顶点的“出度”成正比。

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至少呈线性关系。



■ 重构解

CUT-ROD算法给出了最优收益，但怎么切割的、切割点在哪里呢？通过扩展上述动态规划算法，在求出最优收益之后，即可求出切割方案。

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 $s[j]$ 。

扩展的BOTTOM-UP-CUT-ROD算法，对长度为 j 的钢条不仅计算出最大收益 r_j ，同时记录切割的第一段钢条的长度 s_j

■ 输出完整的最优切割方案

对已知价格表 p 和钢条长度 n ，下述过程能够计算出长度数组

$s[1..n]$ ，并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       $n = n - s[n]$ 
```

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

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($p, 7$) : 1 , 6

➤ PRINT-CUT-ROD-SOLUTION($p, 10$) : 10

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，

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

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），则计算C共需要 pqr 次标量乘法运算。

- 一个标准的两矩阵乘算法如下：

注：只有两个矩阵“相容”
(compatible)才能相乘。

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  $C$ 
```

三重循环结构

矩阵链相乘

n个要连续相乘的矩阵构成一个矩阵链 $\langle A_1, A_2, \dots, A_n \rangle$, 要计算这n个矩阵的连乘乘积: $A_1 A_2 \dots A_n$, 称为矩阵链乘问题。

- 矩阵链乘满足结合律, 不满足交换律。
- 矩阵链乘过程相当于在矩阵之间加适当的括号, 从而根据组合关系定义出矩阵链乘的计算模式。

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:

$$(A_1(A_2(A_3A_4))) \quad (A_1((A_2A_3)A_4))$$

$$((A_1A_2)(A_3A_4)) \quad ((A_1(A_2A_3))A_4)$$

$$(((A_1A_2)A_3)A_4)$$

问题： 不同的加括号方式代表不同的**计算模式**，而不同的计算模式计算矩阵链乘积的代价是不同的。

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

1) 如果按 $((A_1 A_2) A_3)$ 的次序完成乘法，则 A_1 与 A_2 乘需要 $10 \times 100 \times 5 = 5000$ 次标量乘法运算，得一 10×5 的中间结果矩阵，再继续与 A_3 相乘，又需要 $10 \times 5 \times 50 = 2500$ 次标量乘法运算，总共为**7500次**标量乘法运算。

2) 如果按 $(A_1 (A_2 A_3))$ 的次序完成乘法，则 A_2 与 A_3 乘需要 $100 \times 5 \times 50 = 25000$ 次标量乘法运算，得一 100×50 的中间结果矩阵， A_1 与之再次相乘，又需要 $10 \times 100 \times 50 = 50000$ 次标量乘法运算，总共为**75000次**标量乘法运算。

可见，上述两种方法的计算量**相差10倍**！

■ **矩阵链中的矩阵怎么两两相乘才能使总的代价最小呢？**

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

给定 n 个矩阵的链，记为 $\langle A_1, A_2, \dots, A_n \rangle$ ，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**。

- 显然，**穷举**所有可能的括号化方案是不可取的：

此时，可令 $p(n)$ 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

—— 寻找最优子结构

用记号 $A_{i,j}$ 表示子问题 $A_i A_{i+1} \cdots A_j$ 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，则有：

$$(\overline{A_i A_{i+1} \cdots A_k})(\overline{A_{k+1} \cdots A_j})$$

则必须有： $(\overline{A_i A_{i+1} \cdots A_k})$ 必是“前缀”子链 $A_i A_{i+1} \cdots A_k$ 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 $A_{k+1,j}$ 。

证明：反证法

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。

——这一性质称为（该问题的）**最优子结构性**。

2. 递归求解方案

最优子结构性告诉我们：

整体的最优括号化方案可以通过寻找使最终标量乘法次数最小的两个最优括号化子方案得到。

形如： $(A_1 A_{i+1} \dots A_k)(A_{k+1} \dots A_n)$

到哪里找这样的k，使得上述计算的标量乘法次数最小呢？

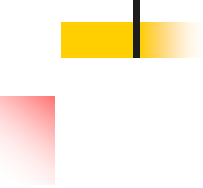
(1) 递推关系式

令 $m[i, j]$ 为计算矩阵链 $A_{i, j}$ 所需的标量乘法运算次数的最小值, 则有

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

含义:

- 对任意的 k ($i \leq k < j$) 分开的子乘积, $A_{i, k}$ 是一个 $p_{i-1} \times p_k$ 的矩阵, $A_{k+1, j}$ 是一个 $p_k \times p_j$ 的矩阵。结果矩阵 $A_{i, j}$ 是 $A_{i, k}$ 和 $A_{k+1, j}$ 最终相乘的结果。
- 对 $m[i, j]$ 和任意 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 $m[i, k]$ + 计算子乘积 $A_{k+1, j}$ 的最小代价 $m[k+1, j]$ + 这两个子矩阵最后相乘的代价 $p_{i-1}p_kp_j$, 而这样的 k 有 $j-i$ 种可能性, 取其中最小者。
- $m[1, n]$ 是计算 $A_{1, n}$ 的最小代价。



再设 $s[i, j]$ ，记录使 $m[i, j]$ 取最小值的 k ，则可以依靠 s 求出最优链乘模式。

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。

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

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
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  $s$ 
```

$m[1, n]$ 是计算 $A_{1, n}$ 的最小代价

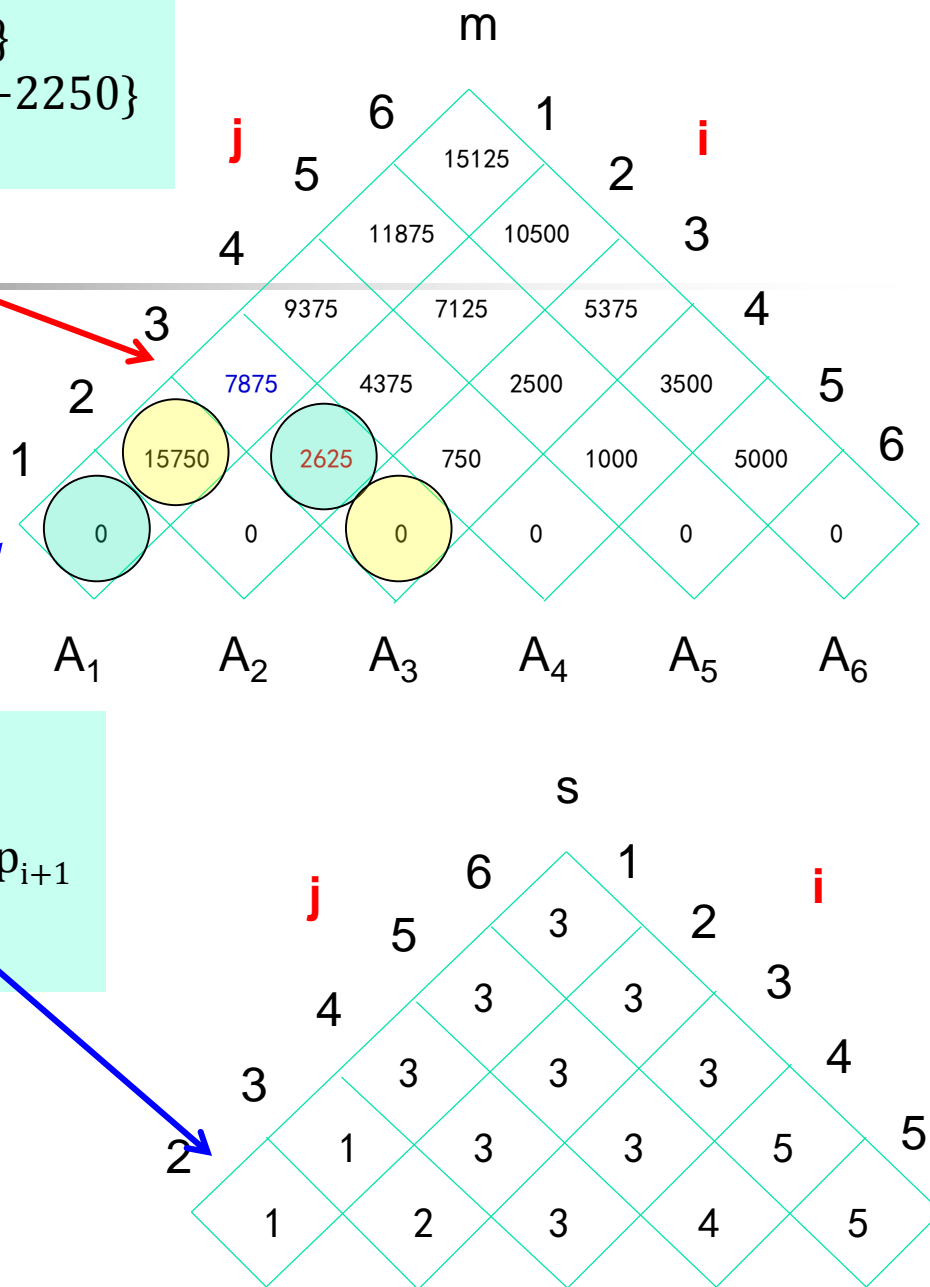
$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

$$\begin{aligned}
 m[1,3] &= \min\{m[1,1]+m[2,3]+30 \times 35 \times 5, \\
 &\quad m[1,2]+m[3,3]+30 \times 15 \times 5\} \\
 &= \min\{0+2625+5250, 15750+0+2250\} \\
 &= 7875
 \end{aligned}$$

例，设

矩阵	维数
A1	30×35
A2	35×15
A3	15×5
A4	5×10
A5	10×20
A6	20×25

$$\begin{aligned}
 m[i,i] &= 0 \\
 m[i,i+1] &= p_{i-1}p_ip_{i+1} \\
 s[i,i+1] &= i
 \end{aligned}$$



时间复杂度分析

算法的主体由一个三层循环构成。外循环执行 $n-1$ 次，内层循环至多执行 $n-1$ 次，所以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           $m[i, j] = \infty$ 
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  $s$ 
```

(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“首个”分割点 k 。

- ◆ 基于 $s[i, j]$ ，对 $A_i A_{i+1} \dots A_j$ 的括号化方案是：

$$(A_i A_{i+1} \dots A_{s[i, j]}) (A_{s[i, j]+1} \dots A_j)$$

- $A_{1\dots n}$ 的最优方案中最后一次矩阵乘运算是：

$$(A_{1\dots s[1, n]}) (A_{s[1, n]+1\dots n})$$

- 用递归的方法求出 $A_{1\dots s[1, n]}$ 、 $A_{s[1, n]+1\dots n}$ 及其它所有子问题的最优括号化方案。

根据s求出矩阵链乘的最优计算模式:

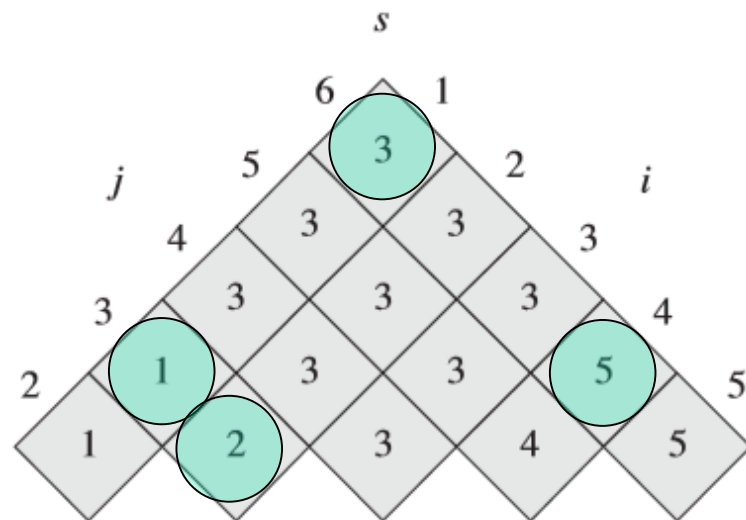
PRINT-OPTIMAL-PARENS(s, i, j)

```
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 ")"
```

例: PRINT-OPTIMAL-PARENS($s, 1, 6$)



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



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

所谓“问题满足最优性原理”即：问题的最优决策序列具有**最优子结构性**。

证明问题满足最优性原理是实施动态规划的**必要条件**：如果证明了问题满足最优性原理，则说明用动态规划方法**有可能解决该问题**。

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。

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

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

回顾：

1) 不管是钢管切割问题还是矩阵链乘问题，我们都首先讨论了问题最优解的结构特征，即证明问题的最优解具有最优子结构性：

- ▶ 钢管切割问题：若 $s(1, n)$ 为最优切割方案，则第一次切割（假定切割点在位置 k ）后得到的两段： $s(1, k)$ 和 $s(k+1, n)$ 也必是最优的子方案。
- ▶ 矩阵链乘问题：设 $A_{1, n}$ 是最优括号化方案，“最大子括号”加在 A_k 后面，则 $A_{1, k}$ 和 $A_{k+1, n}$ 必是子矩阵链的最优括号化方案。

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

动态规划实质上是一种以空间换时间的技术，它在实现的过程中，需要存储过程中产生的各种状态（中间结果），所以它的空间复杂度要大于其它的算法。

■ 详见P218~220.

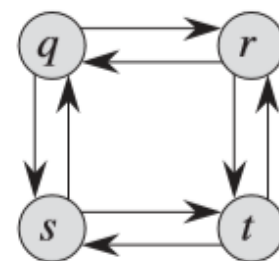
子问题无关性

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

所谓**子问题无关**，是指同一个原问题的一个子问题的解不影响另一个子问题的解，可以各自独立求解。

- **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。

- 详见P217~218.

如何证明问题的最优解满足最优子结构性呢？

即证明：作为构成原问题最优解的组成部分，对应每个子问题的部分应是该子问题的最优解。

“剪切-粘贴”（cut-and-paste）技术：

本质上是反证法证明：假定原问题最优解中对应的某个子问题的部分解不是该子问题的最优解，而存在“更优的子解”，那么我们可以从原问题的解中“剪切”掉这一部分，而将“更优的子解”粘贴进去，从而得到一个比最优解“更优”的解，这与最初的解是原问题的最优解的前提假设相矛盾。因此，不可能存在“更优的子解”。

——所以，原问题的子问题的解必须是其最优解，最优子结构性成立。

最优解的构造

通常再定义一个表，记录每个子问题所做的选择。当求出最优解的值后，利用该表回推就可以求取最优方案。

- 如矩阵链乘法中的表 $s[1\cdots n]$ 。

备忘（查表）

为了避免对重复子问题的重复计算，在递归过程中加入**备忘机制**。这样当第一次遇到子问题时，计算其解，并将结果存储在备忘表中。而其后再遇到同一个子问题时，就只简单地查表，返回其解即可，不用重复计算，节省了时间。

- 例：带有备忘的矩阵链乘法（见P220~221）

15.4 最长公共子序列

一个应用背景：**基因序列比对**。

DNA (Deoxyribonucleic Acid, 脱氧核糖核酸) 是染色体的主要组成成分。DNA 又是由腺嘌呤 (adenine)、鸟嘌呤 (guanine)、胞嘧啶 (cytosine)、胸腺嘧啶 (thymine) 等四种碱基分子 (canonical bases) 组成。用它们单词的首字母 A、C、G、T 来代表这四种碱基，

这样**一条DNA上碱基分子的排列被表示为有穷字符集{A,C,G,T}上的一个串进行表示。**

如：两个有机体的DNA分别为

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$



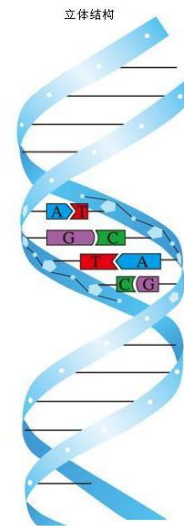
图 4-4 DNA 分子的结构和碱基配对

可以通过比较两个有机体的DNA来确定这两个有机体有多么“相似”。这在生物学上叫做“**基因序列比对**”，而用计算机的话讲就是把比较两个DNA相似性的操作看作是对两个由A、C、G、T组成的字符串的比较。

■ 度量DNA的相似性：

- 如果一个DNA螺旋是另一个DNA螺旋的子串，就说这两个DNA（串）相似。
- 当两个DNA螺旋互不为对方子串的时候，怎么度量呢？

方法一：如果将其中一个转换成另一个所需改变的工作量小，则可称其相似（参见 *Edit distance 15-5*）。



方法二：在 S_1 和 S_2 中找出第三个存在 S_3 ，使得 S_3 中的基以同样的先后顺序出现在 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，最长的公共存在是

$S_3 = \text{GTCGTCGGAAGCCGGCCGAA}。$

$S_1 = \text{ACCGGTCGAGTGCAGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAA}$

怎么找最长的公共存在——两个字符串的最长公共非连续子串，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,
相应的下标序列为 $\langle 2, 3, 5, 7 \rangle$ 。

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。

子问题的定义：从“ X_m 和 Y_n 的LCS”到“ X_{m-1} 和 Y_{n-1} 的LCS”、
“ X_{m-1} 和 Y_n 的LCS”、“ X_m 和 Y_{n-1} 的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 $z_k = x_m = y_n$ 。

同时, 如果 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 (也即 y_n) 添加到 W 上就会产生一个 X 和 Y 的长度大于 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 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

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

含义：

1) 若 $i=0$ 或 $j=0$ ，即其中一个序列的长度为零，则二者的LCS的长度为0， $LCS=\Phi$ ；

2) 若 $x_i=y_j$ ，则 X_i 和 Y_j 的LCS是在 X_{i-1} 和 Y_{j-1} 的LCS之后附加将 x_i （也即 y_j ）得到的，所以

$$c[i, j] = c[i-1, j-1] + 1;$$

3) 若 $x_i \neq y_j$ ，则 X_i 和 Y_j 的LCS的最后一个字符不会是 x_i 或 y_j （不可能同时等于两者，或与两者都不同），此时该LCS应等于 X_{i-1} 和 Y_j 的LCS与 X_i 和 Y_{j-1} 的LCS之中的长者。所以

$$c[i, j] = \max(c[i-1, j], c[i, j-1]);$$

3) LCS的求解

X_m 和 Y_n 的LCS是基于 X_{m-1} 和 Y_{n-1} 的LCS、或 X_{m-1} 和 Y_n 的LCS、或 X_m 和 Y_{n-1} 的LCS求解的。

下述过程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，还设置了一个表 $b[1..m, 1..n]$ ，记录当前 $c[i, j]$ 的计值情况，以此来构造该LCS。

LCS-LENGTH(X, Y)

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = "\nwarrow"$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = "\uparrow"$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = "\leftarrow"$ 
18  return  $c$  and  $b$ 

```

		j	0	1	2	3	4	5	6
i		y_j		(B)	D	(C)	A	(B)	(A)
		x_i							
0			0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	←1	↖
2	(B)		0	↖①	←1	←1	↑	↖	←2
3	(C)		0	↑	↑	↖②	←2	↑	↑
4	(B)		0	↖	↑	↑	↑	↖③	←3
5	D		0	↑	↖	↑	↑	↑	↑
6	(A)		0	↑	↑	↑	↖	↑	↖④
7	B		0	↖	↑	↑	↑	↖	↑

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	↑	↑	↑	↖	1	↖
2	(B)		0	↖	①	←	1	↖	2
3	(C)		0	↑	↑	↖	②	←	2
4	(B)		0	↑	↑	↑	↑	↖	③
5	D		0	↑	↖	2	↑	↑	3
6	(A)		0	↑	↑	↑	↖	3	↖
7	B		0	↑	↑	↑	↑	↖	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 $c[i-1, j]$ 、 $c[i, j-1]$ 、 $c[i-1, j-1]$ 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，即如图所示中的蓝色方块给出的轨迹。
- 4) 图中， $c[7, 6] = 4$ ，
LCS(X, Y) = $\langle B, C, B, A \rangle$

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									
0	x_i		0	0	0	0	0	0	0
1	A		0	<div>↑ 0</div>	<div>↑ 0</div>	<div>↑ 0</div>	<div>↖ 1</div>	<div>← 1</div>	<div>↖ 1</div>
2	B		0	<div>↖ ①</div>	<div>← 1</div>	<div>← 1</div>	<div>↑ 1</div>	<div>↖ 2</div>	<div>← 2</div>
3	C		0	<div>↑ 1</div>	<div>↑ 1</div>	<div>↖ ②</div>	<div>← 2</div>	<div>↑ 2</div>	<div>↑ 2</div>
4	B		0	<div>↖ 1</div>	<div>↑ 1</div>	<div>↑ 2</div>	<div>↑ 2</div>	<div>↖ ③</div>	<div>← 3</div>
5	D		0	<div>↑ 1</div>	<div>↖ 2</div>	<div>↑ 2</div>	<div>↑ 2</div>	<div>↑ 3</div>	<div>↑ 3</div>
6	A		0	<div>↑ 1</div>	<div>↑ 2</div>	<div>↑ 2</div>	<div>↖ 3</div>	<div>↑ 3</div>	<div>↖ ④</div>
7	B		0	<div>↖ 1</div>	<div>↑ 2</div>	<div>↑ 2</div>	<div>↑ 3</div>	<div>↖ 4</div>	<div>↑ 4</div>

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

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	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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 1 \quad x_1 \neq y_1$$

$$\begin{aligned} c[1, 1] &= \max\{c[1, 0], c[0, 1]\} \\ &= \max\{0, 0\} \\ &= 0 \end{aligned}$$

$$b[1, 1] = \text{'}\uparrow\text{'}$$

elseif $c[i-1, j] \geq c[i, j-1]$
 $c[i, j] = c[i-1, j]$
 $b[i, j] = \text{'}\uparrow\text{'}$

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	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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 2 \quad x_1 \neq y_2$$

$$\begin{aligned} c[1, 2] &= \max\{c[1, 1], c[0, 2]\} \\ &= \max\{0, 0\} \\ &= 0 \end{aligned}$$

$$b[1, 2] = \uparrow$$

elseif $c[i-1, j] \geq c[i, j-1]$
 $c[i, j] = c[i-1, j]$
 $b[i, j] = \uparrow$

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	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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 3 \quad x_1 \neq y_3$$

$$\begin{aligned} c[1, 3] &= \max\{c[1, 2], c[0, 3]\} \\ &= \max\{0, 0\} \\ &= 0 \end{aligned}$$

$$b[1, 3] = \text{'}\uparrow\text{'}$$

elseif $c[i-1, j] \geq c[i, j-1]$
 $c[i, j] = c[i-1, j]$
 $b[i, j] = \text{'}\uparrow\text{'}$

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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 4 \quad x_1 = y_4$$

$$\begin{aligned} c[1, 4] &= c[0, 3] + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

$$b[1, 4] = \nwarrow$$

if $x_i == y_j$
 $c[i, j] = c[i-1, j-1] + 1$
 $b[i, j] = \nwarrow$

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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 5 \quad x_1 \neq y_5$$

$$\begin{aligned} c[1, 5] &= \max\{c[1, 4], c[0, 5]\} \\ &= \max\{1, 0\} \\ &= 1 \end{aligned}$$

$$b[1, 5] = \leftarrow$$

$$\begin{aligned} \text{else } c[i, j] &= c[i, j-1] \\ b[i, j] &= \leftarrow \end{aligned}$$

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	x _i		0	0	0	0	0	0	0
1	A		0	↑	↑	↑	1	←1	1
2	(B)		0	①	←1	←1	1	2	←2
3	(C)		0	↑	↑	②	←2	2	2
4	(B)		0	↑	↑	↑	↑	③	←3
5	D		0	↑	2	↑	↑	↑	↑
6	(A)		0	↑	↑	↑	3	↑	④
7	B		0	↑	↑	↑	↑	4	↑

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 1, j = 6 \quad x_1 = y_6$$

$$\begin{aligned} c[1, 6] &= c[0, 5] + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

$$b[1, 6] = \nwarrow$$

if $x_i == y_j$
 $c[i, j] = c[i-1, j-1] + 1$
 $b[i, j] = \nwarrow$

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

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 2, j = 1 \quad x_2 = y_2$$

$$\begin{aligned} c[2, 1] &= c[1, 0] + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

$$b[2, 1] = \nwarrow$$

if $x_i == y_j$
 $c[i, j] = c[i-1, j-1] + 1$
 $b[i, j] = \nwarrow$

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	x _i		0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	↖	↖
2	(B)		0	①	←1	←1	↖	↖	↖
3	(C)		0	↑	↑	②	←2	↑	↑
4	(B)		0	↑	↑	↑	↑	③	←3
5	D		0	↑	↑	↑	↑	↑	↑
6	(A)		0	↑	↑	↑	↑	↑	④
7	B		0	↑	↑	↑	↑	↑	↑

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

$$c[0, j] = 0$$

$$c[i, 0] = 0$$

$$i = 2, j = 5 \quad x_2 = y_5$$

$$\begin{aligned} c[2, 5] &= c[1, 4] + 1 \\ &= 1 + 1 \\ &= 2 \end{aligned}$$

$$b[2, 5] = \nwarrow$$

if $x_i == y_j$
 $c[i, j] = c[i-1, j-1] + 1$
 $b[i, j] = \nwarrow$

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:

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, 下一步到 $b[i, j-1]$ 中寻找上一个元素;
- 遇到“↑”时, 下一步到 $b[i-1, j]$ 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( $b, X, i, j$ )  
1  if  $i == 0$  or  $j == 0$   
2      return  
3  if  $b[i, j] == \nwarrow$   
4      PRINT-LCS( $b, X, i - 1, j - 1$ )  
5      print  $x_i$   
6  elseif  $b[i, j] == \uparrow$   
7      PRINT-LCS( $b, X, i - 1, j$ )  
8  else PRINT-LCS( $b, X, i, j - 1$ )
```

由于每一次循环使 i 或 j 减1，最终 $m=0$ ， $n=0$ ，算法结束，所以PRINT-LCS的时间复杂度为 $O(m+n)$

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

```

1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == \nwarrow$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5  print  $x_i$ 
6  elseif  $b[i, j] == \uparrow$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )

```

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

PRINT-LCS($b, X, 7, 6$)

PRINT-LCS($b, X, 6, 6$) print A

PRINT-LCS($b, X, 5, 5$)

PRINT-LCS($b, X, 4, 5$) print B

PRINT-LCS($b, X, 3, 4$)

PRINT-LCS($b, X, 3, 3$) print C

PRINT-LCS($b, X, 2, 2$)

PRINT-LCS($b, X, 2, 1$) print B

PRINT-LCS($b, X, 1, 0$) 结束

4) 算法的改进

- (1) 可以去掉表b，直接基于c求LCS。
- (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。

15.5 最优二叉搜索树

- **场景**：语言翻译，从英语到法语，对给定的单词，在单词表里找到该词。
- **方法**：创建一棵**二叉搜索树**，以英语单词作为关键字构建树。
- **目标**：尽快地找到英语单词，使“**总**”的搜索时间尽量少。
- **思路**：频繁使用的单词，如the，应尽可能靠近根；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？

最优二叉搜索树的定义

假设所有元素互异

(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。

(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 表示其被搜索的频率。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。

对搜索对象x，在T中可能找到、也可能找不到：

- 若x等于某个 k_i ，则一定可以在T中找到结点 k_i ，称为成功搜索。
 - ◆ 成功搜索的情况一共有n种，分别是x恰好等于某个 k_i 。

- 若 $x < k_1$ 、或 $x > k_n$ 、或 $k_i < x < k_{i+1}$ ($1 \leq i < n$)，则在T中搜索x将失败，称为**失败搜索**。

- 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**伪关键字**。

- 伪关键字在T中对应**外部结点**，共有 **$n+1$** 个。

——**扩展二叉树**：内结点表示关键字 k_i ，外结点(叶子结点)表示 d_i 。

- 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 $i=1, \dots, n-1$ ， d_i 表示所有在 k_i 和 k_{i+1} 之间的值。

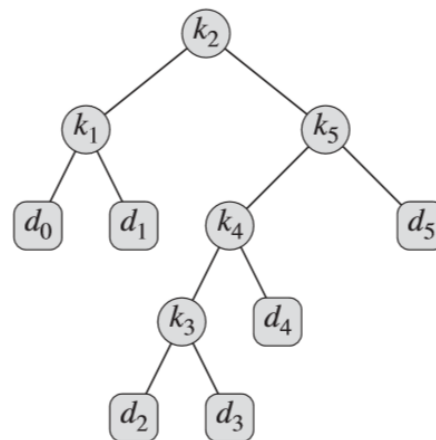
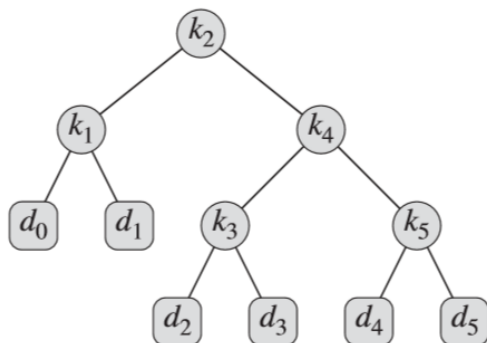
- 每个 d_i 也有一个**概率 q_i** ，表示搜索对象x恰好落入区间 d_i 的频率。

例：设有 $n=5$ 个关键字的集合，每个 k_i 的概率 p_i 和相 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。



每个 k_i 对应一个内结点，共有 n 个，用圆形结点表示，代表成功检索的位置。

每个 d_i 对应一个外部结点，有 $n+1$ 个，用矩形框表示，代表失败检索的情况。

二叉搜索树的期望搜索代价

➤ 一次搜索的代价等于从根结点开始访问结点的数量(包括外部结点)。

◆ 从根结点开始访问结点的数量等于结点在T中的深度+1;

◆ 记 $\text{depth}_T(i)$ 为结点i在T中的深度。

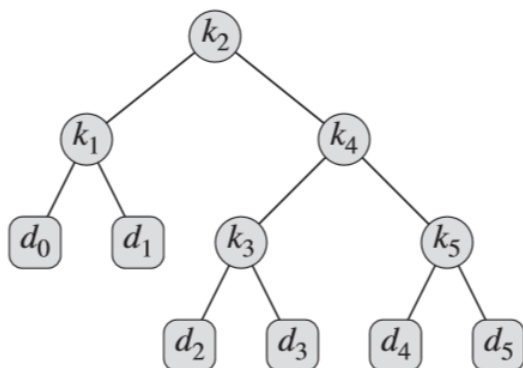
➤ 二叉搜索树T的期望代价为

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

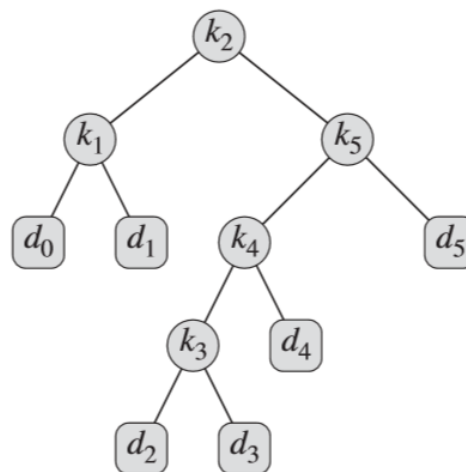
例：n=5，

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

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



(a)



(b)

➤ (a) 的期望搜索代价为2.80。

➤ (b) 的期望搜索代价为2.75。

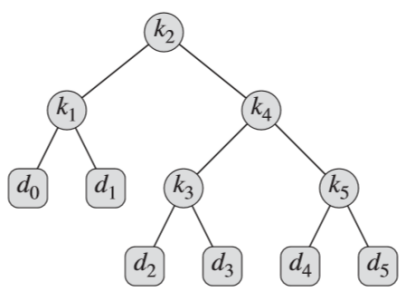
$$\begin{aligned}
 E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\
 &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i,
 \end{aligned}$$

显然，树**b**的期望代价小于树**a**。事实上，树**b**是当前问题实例的一棵最优二叉搜索树

n=5 ,

<i>i</i>	0	1	2	3	4	5
<i>p_i</i>		0.15	0.10	0.05	0.10	0.20
<i>q_i</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)

(a) 的期望搜索代价为2.80

node	depth	probability	contribution
<i>k₁</i>	1	0.15	0.30
<i>k₂</i>	0	0.10	0.10
<i>k₃</i>	2	0.05	0.15
<i>k₄</i>	1	0.10	0.20
<i>k₅</i>	2	0.20	0.60
<i>d₀</i>	2	0.05	0.15
<i>d₁</i>	2	0.10	0.30
<i>d₂</i>	3	0.05	0.20
<i>d₃</i>	3	0.05	0.20
<i>d₄</i>	3	0.05	0.20
<i>d₅</i>	3	0.10	0.40
Total			2.80

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

最优二叉搜索树的定义：

对于给定的关键字及其概率集合，期望搜索代价最小的二叉搜索树称为其**最优二叉搜索树**。

- 对给定的关键字和概率集合，怎么构造最优二叉搜索树？
- 关键问题：确定**谁是树根**
 - 树根不一定是概率最高的关键字；
 - 树也不一定是最矮的树；
 - 但该树的期望搜索代价必须是最小的。

用动态规划策略构造最优二叉搜索树

(1) 证明最优二叉搜索树的最优子结构

如果 T 是一棵相对于关键字 k_1, \dots, k_n 和伪关键字 d_0, \dots, d_n 的最优二叉搜索树，则 T 中一棵包含关键字 k_i, \dots, k_j 的子树 T' 必然是相对于关键字 k_i, \dots, k_j (和伪关键字 d_{i-1}, \dots, d_j) 的最优二叉搜索子树。

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'' 粘贴到相应位置上，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。

(2) 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（子）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 。

谁可能是这个根呢？

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

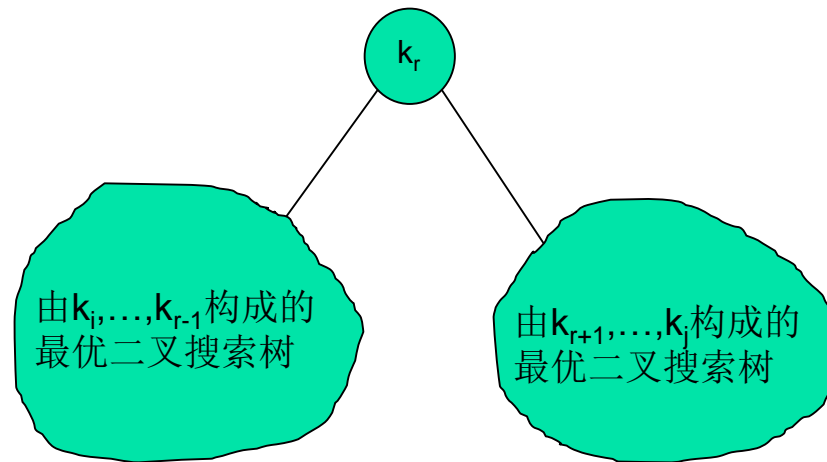
$$i \geq 1, j \leq n \text{ 且 } j \geq i-1。$$

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

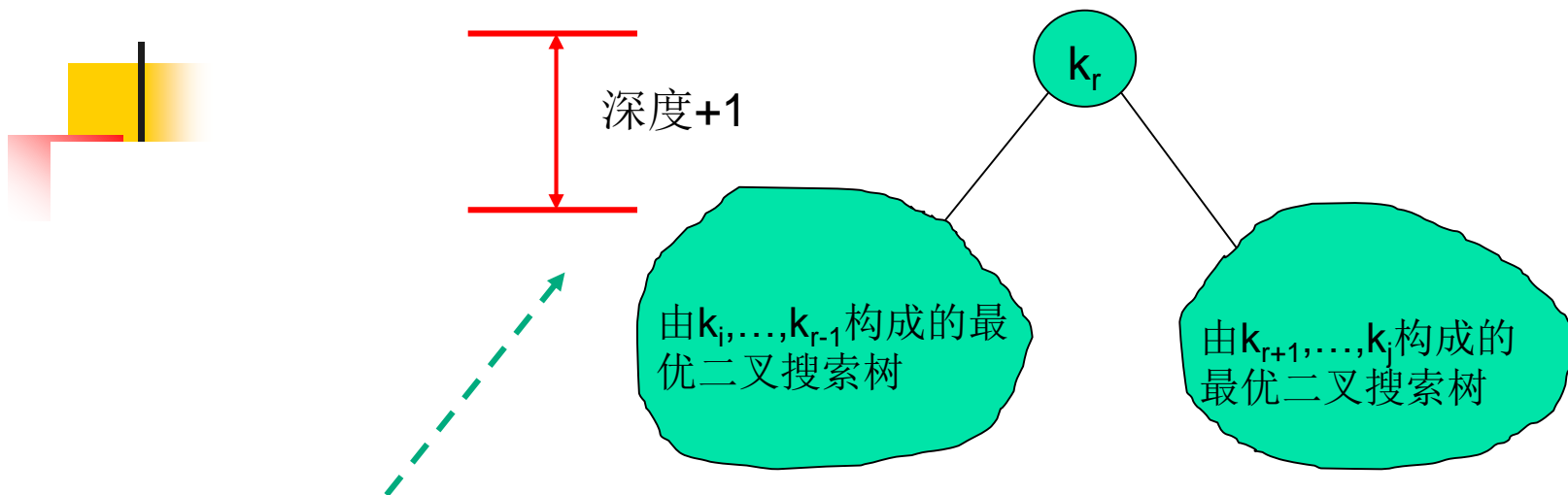
- $e[1,n]$ 为问题的最终解的期望搜索代价。

(1) 当 $i \leq j$ 时，从 k_i, \dots, k_j 中选择出根结点 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 且同样为最优二叉搜索子树。



当一棵树成为另一个结点的子树时，有以下变化：



- 子树的每个结点的深度增加1。
- 根据搜索代价期望值计算公式，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**。
- ◆ 对于包含关键字 k_i, \dots, k_j 的子树，所有结点的概率之和为

（包含外部结点）：

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

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

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

其中， $w(i, r-1)$ 和 $w(r+1, j)$ 是左右子树所有结点的概率之和。且有：

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

故有：

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

因此，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

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

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} ，其期望搜索代价仅为： $e[i, i-1] = q_{i-1}$ 。

(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 ， k_r 即为关键字 k_i, \dots, k_j 的最优二叉搜索（子）树的树根。

在求出 $e[1, n]$ 后，利用 root 即可构造出最终的最优二叉搜索树。

计算最优二叉搜索树的期望值

定义三个表（数组）：

- $e[1..n+1, 0..n]$ ：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。
- $w[1..n+1, 0..n]$ ：用于保存子树的结点概率之和，且有

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

这样，每个 $w[i, j]$ 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  $root$ 
```

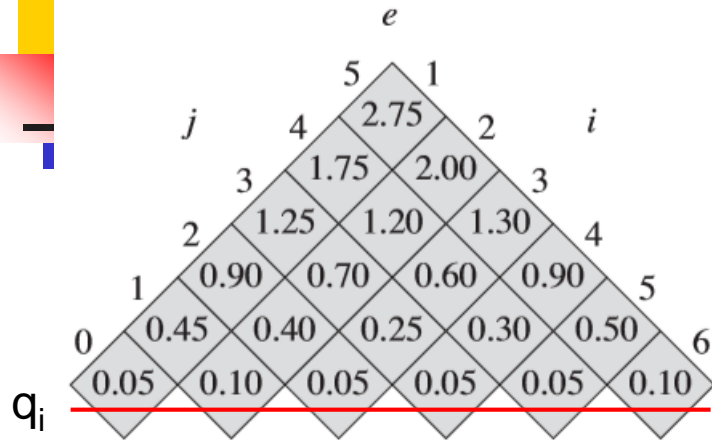
自底向上的迭代计算

时间复杂度 $\Theta(n^3)$

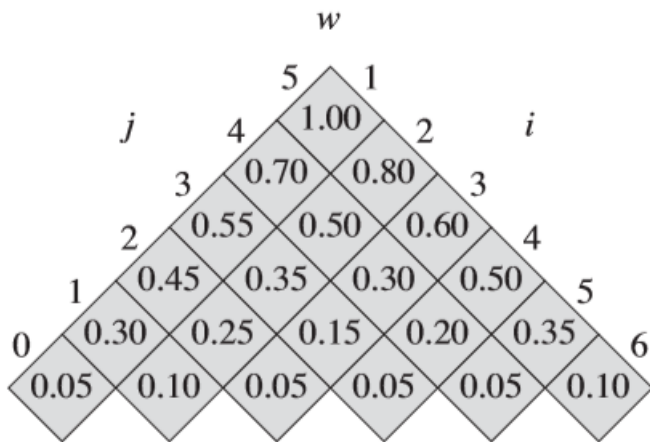
■ 例：n=5，

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

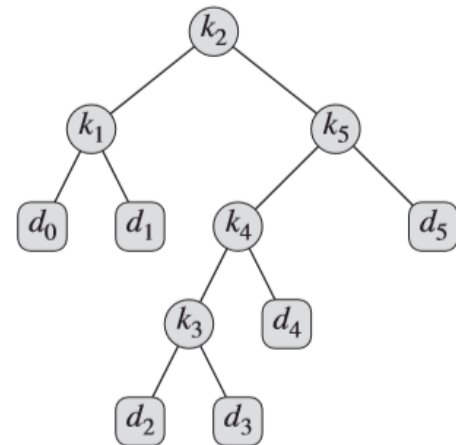
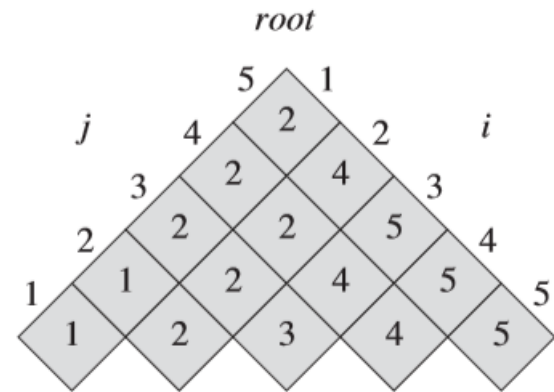
$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1.$$

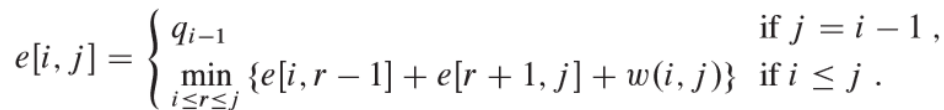


$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j. \end{cases}$$



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



 q_i 

$$e[1, 0] = q_0 = 0.05$$

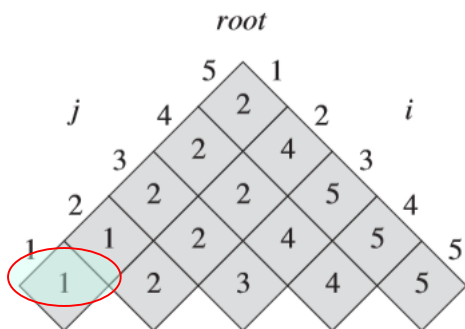
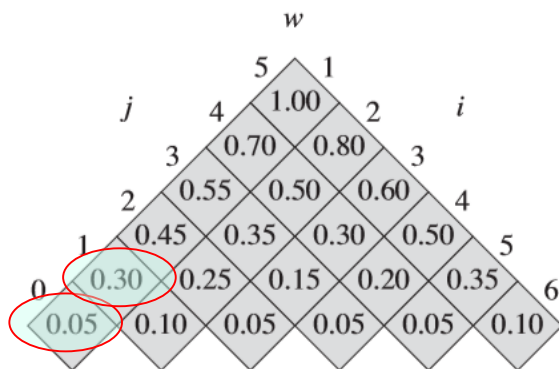
$$e[2, 1] = q_1 = 0.10$$

$$w[1, 0] = q_0 = 0.05$$

$$w[1, 1] = w[1, 0] + p_1 + q_1 = 0.3$$

$$\begin{aligned} e[1,1] &= \min\{e[1,0] + e[2,1] + w[1,1]\} \\ &= 0.45 \end{aligned}$$

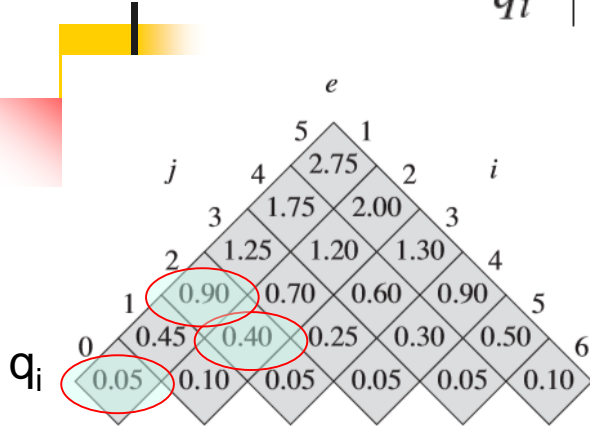
$$r[1, 1] = 1$$



■ 例：n=5，

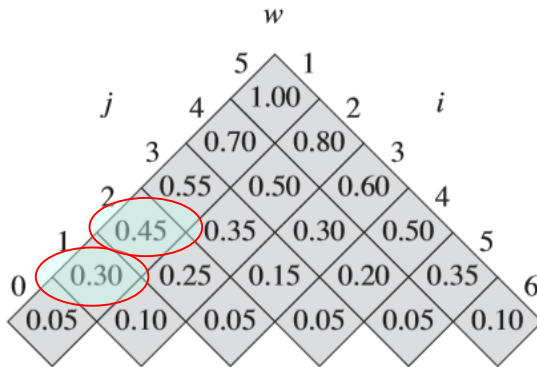
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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



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

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



$$e[1, 0] = q_0 = 0.05$$

$$e[1, 1] = 0.45$$

$$e[2, 2] = 0.4$$

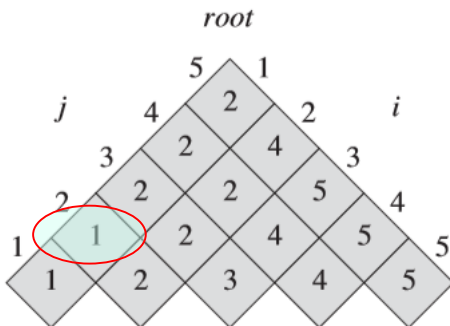
$$e[3, 2] = q_2 = 0.05$$

$$w[1, 1] = 0.3$$

$$w[1, 2] = w[1, 1] + p_2 + q_2 = 0.45$$

$$\begin{aligned} \mathbf{e[1,2]} &= \min\{\mathbf{e[1,0]+e[2,2]+w[1,2]}, \\ &\quad \mathbf{e[1,1]+e[3,2]+w[1,2]}\} \\ &= 0.9 \end{aligned}$$

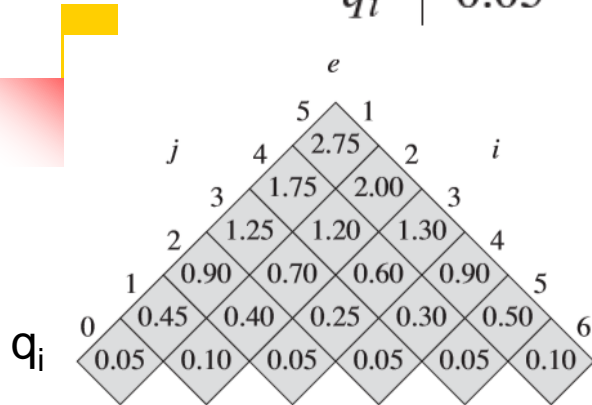
$$r[1, 2] = 1$$



$n=5$,

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

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



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

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

课堂练习：计算

(1) $w[2, 3]$

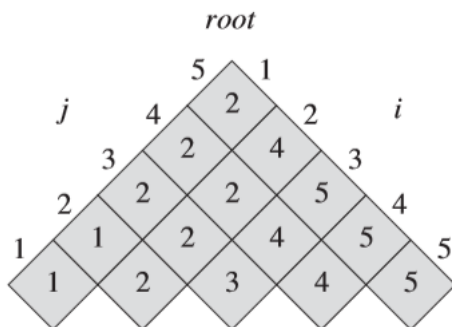
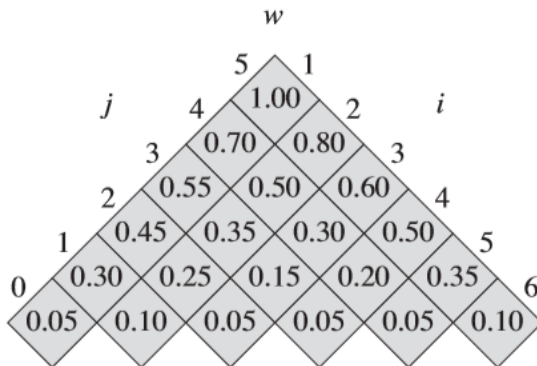
$e[2, 3]$

$r[2, 3]$

(2) $w[1, 5]$

$e[1, 5]$

$r[1, 5]$



动态规划作业：

■ 计算题：

- 15.2-1
- 15.4-1
- 15.5-2

■ 算法设计题：

- 15.1-3
- 15-9
- 15-11

■ 证明题：

- 15.2-5
- 15.3-6：提示 基于以下数据讨论第二问，

		j			
		1	2	3	4
i	1	1	2	5/2	6
	2	1/2	1	3/2	3
	3	2/5	2/3	1	3
	4	1/6	1/3	1/3	1

Let $c_1 = 2$ and $c_2 = c_3 = 3$.