

# 算法设计与分析

Computer Algorithm Design & Analysis  
2019.11

王多强

[dqwang@mail.hust.edu.cn](mailto:dqwang@mail.hust.edu.cn)

群名称: 2019-算法  
群 号: 835135560



群名称: 2019-算法  
群 号: 835135560

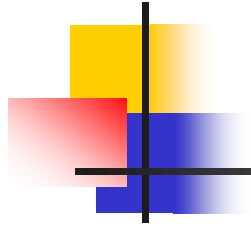


# Chapter 15

## 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。每个解都有一个评价优劣的值，我们希望寻找具有**最优值的解**（最小值或最大值）。

——这种找**最优解**的问题通常称为**最优化问题**。

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。

进一步地，从数学的角度看，所谓最优化问题可以概括为这样一种数学模型：

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：

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

# 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）

# 何为Programming?

—— 造表。

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

分治法：要求将问题划分为互不相交的子问题，递归地求解子问题，然后将子问题的解组合成原问题的解。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：与分治不同，适用于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，避免了对子问题的重复计算。

## 动态规划算法的步骤

1. 刻画一个最优解的结构特征（最优子结构性）；
2. 递归地定义最优解的值（一个递推关系式）；
3. 计算最优解的值（目标函数的最小/最大值）；
4. 利用计算出的信息，构造一个最优解（求解向量）。

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) 如果要求解本身，就要做步骤4，这通常需要在执行步骤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；



# 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

单位：英寸

长度为 $i$ 英寸的钢条可以带来 $p_i$ 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  $r_n$  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。

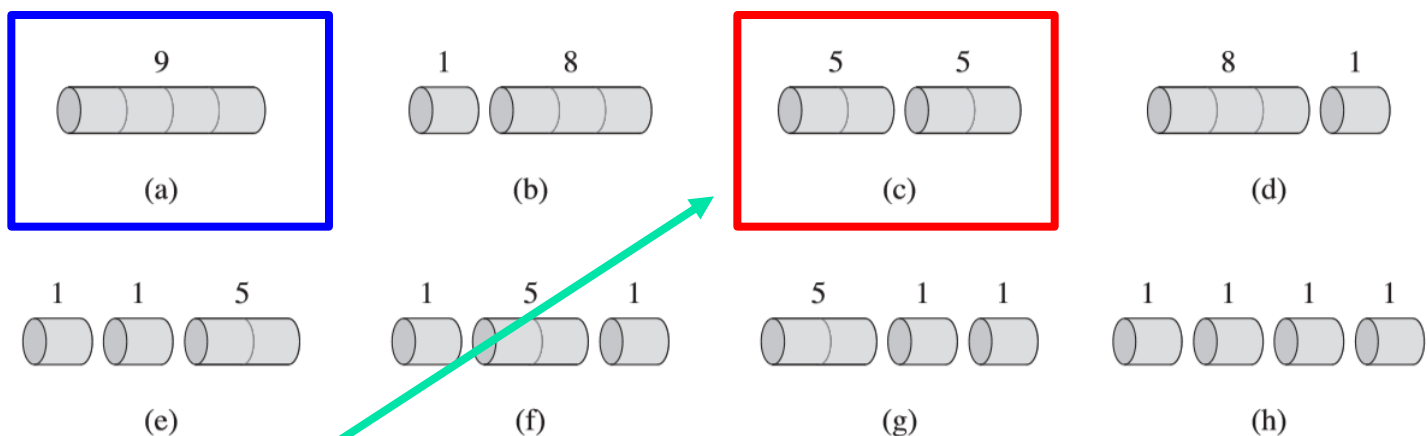
思考：按价格从高到低分段出售，即先卖 $p_i$ 最大的段，然后再卖 $p_j$ 较小的段。是否可以获得最好的收益？（贪心策略）

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有以下几种：



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，每段长2英寸，  
**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 $2^{n-1}$ 种不同的切割方案。
  - 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, 该如何获得最优的切割呢?

- 对**最优切割**, 设**某次切割**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终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，j位置切割后，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**原问题的一个解就可以通过组合这两个相关子问题的最优子解而获得**。

- 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：

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**  $q$

其中， $p$ 是价格数组， $n$ 是钢条总长度。

若 $n=0$ ，则收益为0。

该过程的效率很差：

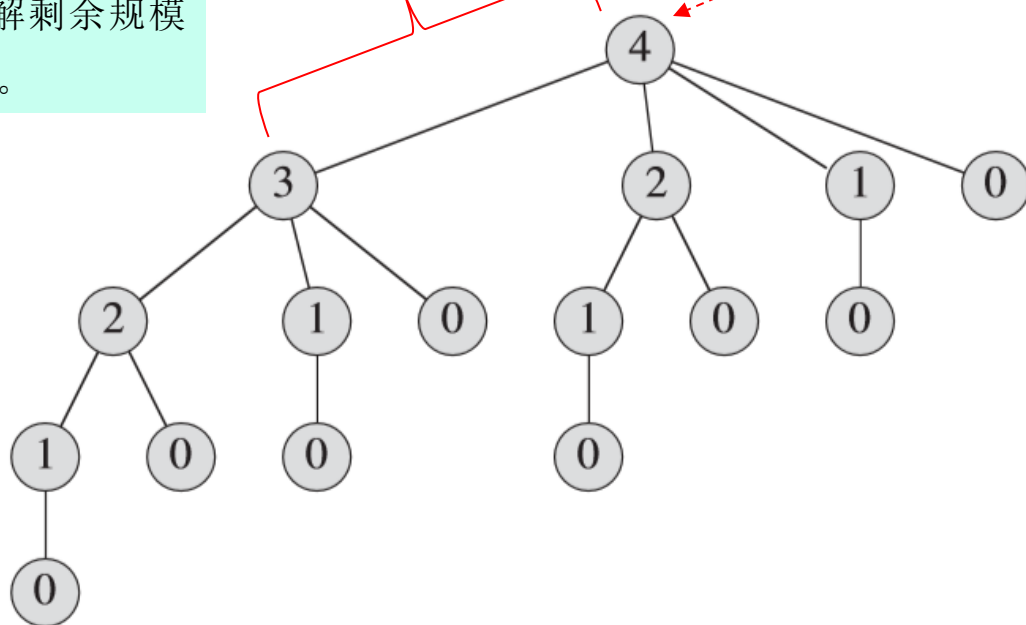
- 存在一些相同长度的子问题，在CUT-ROD的整个执行过程中反复地做一些计算（递归调用）。



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



从根结点到叶结点的一条路径对应长度为 $n$ 的钢条的 $2^{n-1}$ 种切割方案之一。

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。

➤ 因此，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）。

- **可获得的改进**：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。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出每个子问题，则动态规划方法的总运行时间就是**多项式阶**的。

## ■ 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)

- 形式是**递归**的，但处理过程中会**保存每个子问题的解**。
- 过程每次被调用时，首先检查当前子问题是否已经被计算过并保存过子解。
  - ◆ 如果是，则直接返回以前保存的值；
  - ◆ 否则，按照一般方式计算该子问题并保存其解。

**备忘：**一个表，“记住”之前已经计算出来的结果。  
表的形式可以是一个数组或者是散列表

## 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( $p, n, r$ )
```

用辅助数组作为备忘表。

➤ 初始化为 $-\infty$ ;

## 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  $q$ 
```

查表:  $r[n] \geq 0$  表示已经保存过解, 直接引用  $r[n]$ 。

否则, 实施计算,  $r[n]$  保存当前结果;

## (2) 自底向上法 (bottom-up method)

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、原问题的顺序依次求解。一个较大子问题的解可通过组合较小子问题的解而得到，直到原始问题得到答案。

**自底向上求解：**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。

**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  $r[n]$ 
```

从规模为0的“最小”子问题开始依次求解

## ■ 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  $r[n] \geq 0$ 
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

递归设计框架

迭代设计框架

- 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。
  - 而在某些特殊情况下，自顶向下法可能没有递归处理所有可能的子问题（剪枝）而减少工作量。



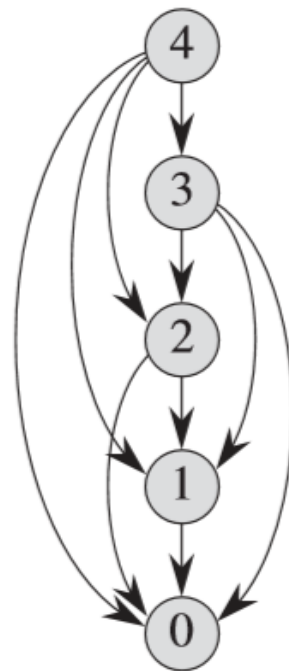


# 子问题图

当思考用动态规划求解一个问题时，应该弄清楚所涉及的子问题以及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。

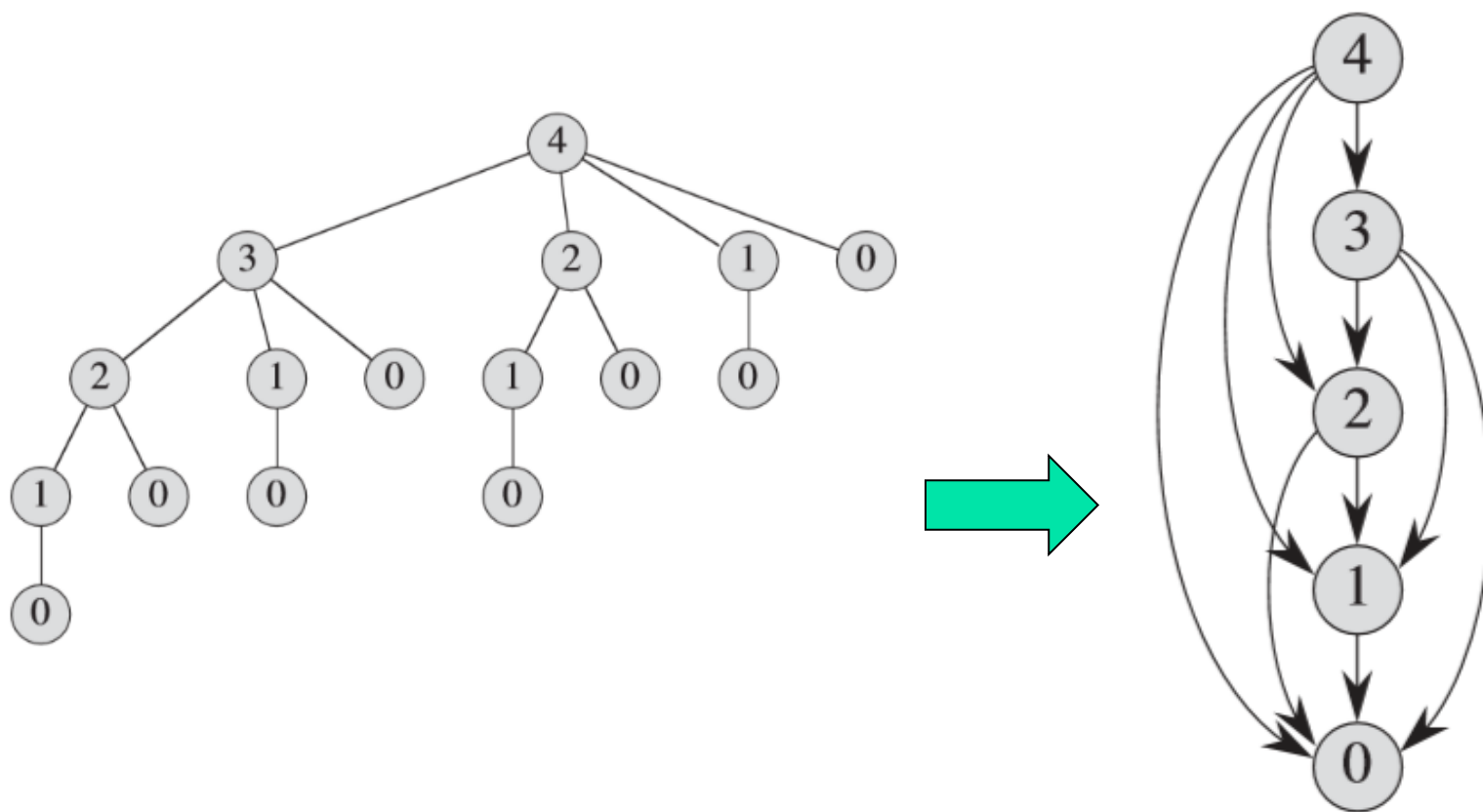
- 子问题图是一个有向图，每个顶点唯一地对应一个子问题。
- 若求子问题 $x$ 的最优解时需要直接用到子问题 $y$ 的最优解（ $x$ 依赖于 $y$ ），则在子问题图中画一条从子问题 $x$ 的顶点到子问题 $y$ 的顶点的有向边。



$n=4$ 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。

- 子问题图是递归调用树的“简化版”或“收缩版”

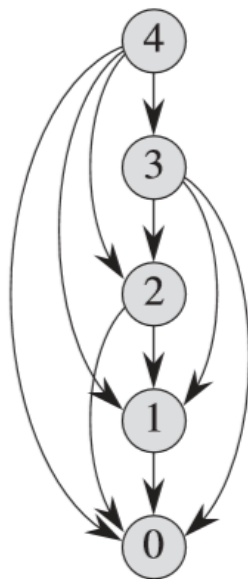
—— 将递归树中对应相同子问题的结点合并为子问题图中的一个顶点，相关的边都从父结点指向子结点。



- 在自底向上的动态规划方法中，

对于一个给定的子问题 $x$ ，在求解它之前应先求解它所依赖的子问题。仅当它依赖的所有子问题都求解完成了，才会求解它。

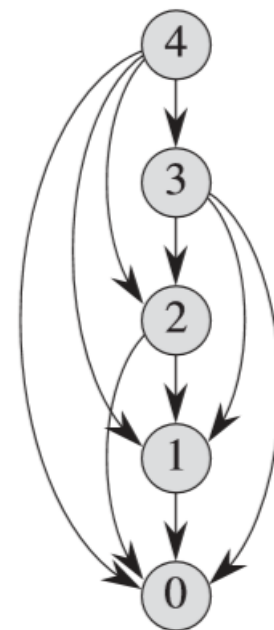
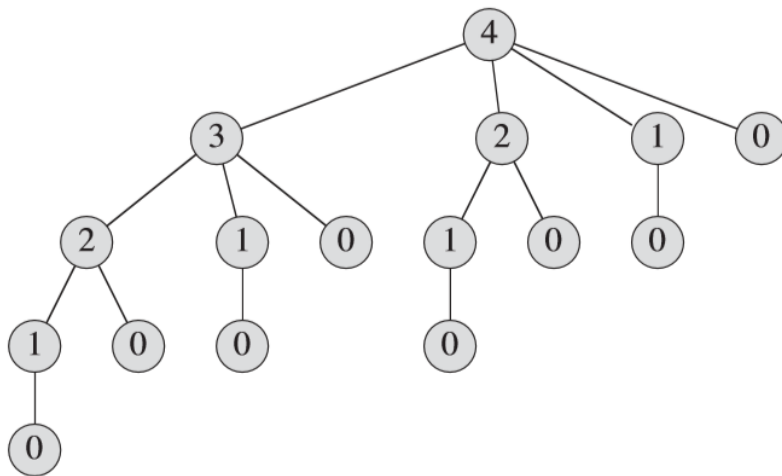
—— 这在子问题图中，对应顶点的一个逆拓扑序，可以按照深度优先原则进行处理。



## 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——目标函数的最大值，但怎么切割的呢？（切割点都在哪里）

扩展上述算法，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用于记录切割点。

◆ 对长度为j的钢条，求出其最优收益，并**记录下获得最优收益时左侧第一个切割点的位置**

## ■ 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

# 建立基于动态规划策略的计算过程容易吗？

- 不是很容易。
- 要根据问题的性质构造递推关系式，形成有效的计算过程。
- 要点：
  - ▣ 首先给出子问题的定义和递推关系式： $r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$
  - ▣ 然后造表：不同的问题表的结构不同： $r[0..n]$

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  $q$ 
```

## 15.2 矩阵链乘法

### 两个矩阵的乘运算：

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），C中共有 $pq$ 个元素，所以计算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, 所以矩阵链乘相当于在矩阵之间加括号。不同的加括号方案导出不同的矩阵链乘计算模式。

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:

$$(A_1(A_2(A_3 A_4))) \quad (A_1((A_2 A_3) A_4))$$

$$((A_1 A_2)(A_3 A_4)) \quad ((A_1(A_2 A_3)) A_4)$$

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

尽管不同的**计算模式**最后得到的结果是一样的，但计算过程中产生的代价是不同的。

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

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 $10 \times 5$ 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 $100 \times 50$ 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，上述两种计算模式的计算量**相差10倍**！

■ **问题：怎么求代价最小的计算模式——最优计算模式呢？**

# 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**。

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： **$P(n) = \Omega(2^n)$**

显然，**穷举**所有可能的括号化方案是不可取的。

# 1) 最优括号化方案的结构特征

## —— 寻找最优子结构

## 子问题的定义

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：

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

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**：若 $A_{i,j}$ 是最优的计算模式，则其中的 $A_{i,k}$ 、 $A_{k+1,j}$ 也都是相应子问题的最优计算模式。

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

到哪里找这个使标量乘法次数最小的k呢？

表的结构：i~j代表区间，但数据结构是二维数组。

## (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（最小值），则

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

含义：

递推方程

- 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。
- 对与某个  $k$ ，要想使得  $m[i, j]$  最小，必然是： $m[i, j]$  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，取能使  $m[i, j]$  最小的  $k$ 。
- $m[1, n]$  是计算  $A_{1, n}$  的最小代价。





构造解：

再设 $s[i, j]$ ，记录使 $m[i, j]$ 取最小值的 $k$ ，后面可以根据 $s$ 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  $s$ 
```

辅助表  $m$  保存所有  $m[i, j]$  的代价， $s[i, j]$  记录使  $m[i, j]$  时取得最优值时的分割点  $k$ 。

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

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 $m[1, n]$ 。

$m[1, n]$ 是计算 $A_{1,n}$ 的最小代价

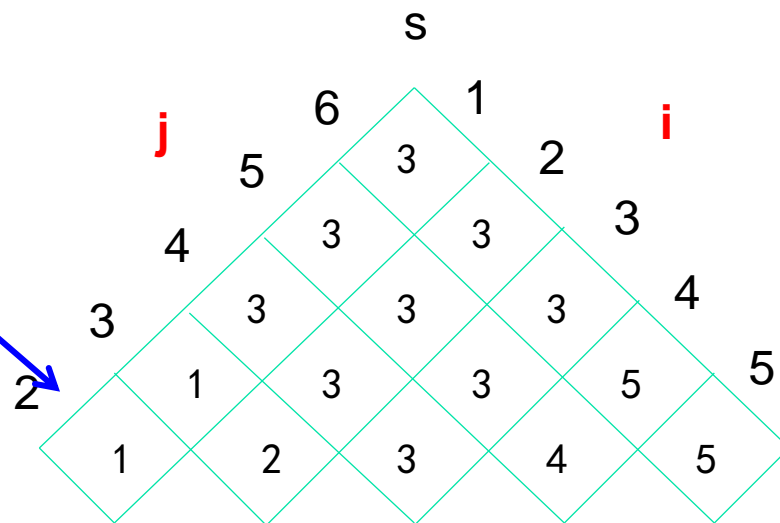
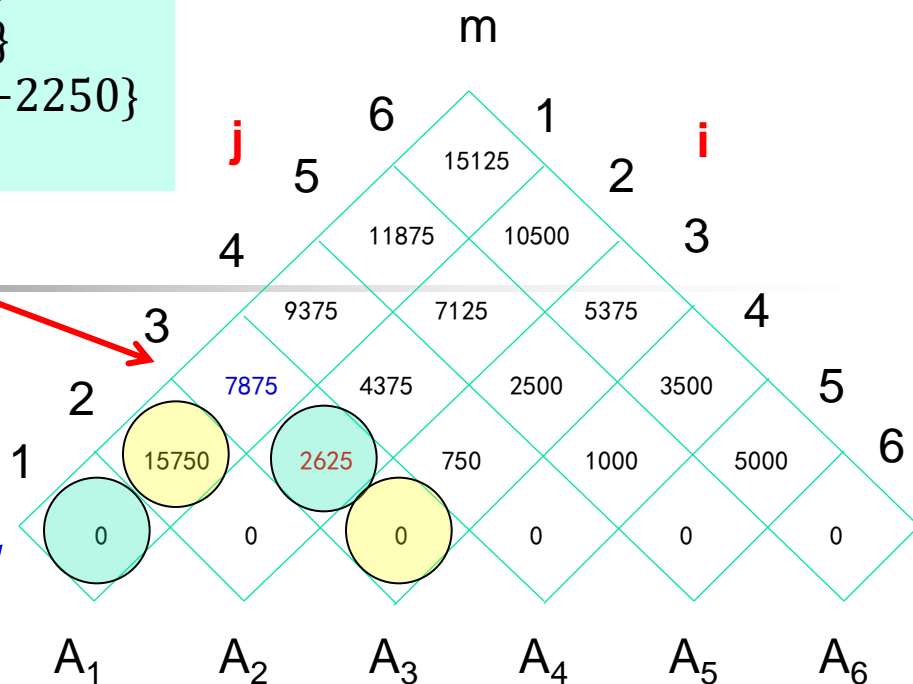
$$\begin{aligned}
 m[1,3] &= \min\{m[1,1]+m[2,3]+30 \times 35 \times 5, \\
 &\quad m[1,2]+m[3,3]+30 \times 15 \times 5\} \\
 &= \min\{0+2625+5250, 15750+0+2250\} \\
 &= 7875
 \end{aligned}$$

例， 设

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

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

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



# 时间复杂度分析

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 $A_i A_{i+1} \dots A_j$ 的最优括号化方案中“首个”加括号的位置点 $k$ 。

- ◆ 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:

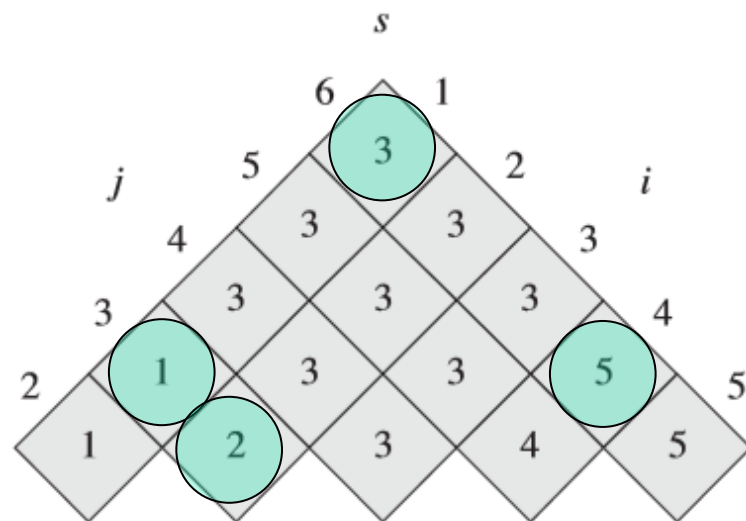
PRINT-OPTIMAL-PARENS( $s, i, j$ )

```
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( $s, 1, 6$ )



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



# 回顾：何为Programming?

——造表。

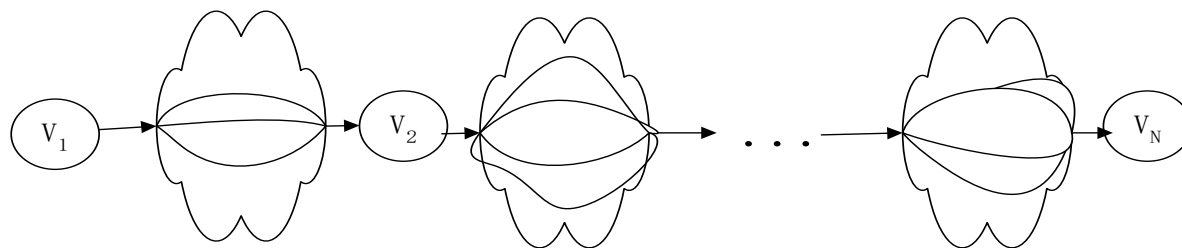
动态规划：与分治不同，适用于有子问题重叠的情况。

动态规划算法对重复的子问题的求解只做一次，并将其最优的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，避免了不必要的计算工作。

# 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



**阶段：**事件的发展过程分为若干个相互联系的阶段。事件的发展总是从初始状态开始，依次经过第一阶段、第二阶段、第三阶段、 $\dots$ ，直至最后一个阶段结束。

**阶段变量：**描述阶段的变量称为**阶段变量**。



**状态：**状态表示每个阶段的自然状况或客观条件，用一组变量的值表示。它既是当前阶段某途径的起点，也是前面阶段某途径的终点。

**状态变量：**过程的状态通常可以用一个或一组数来描述，称为状态变量。

- 如果用一组数表示，状态变量就是多维的，用向量表示。

**状态集合：**当过程按所有可能不同的方式发展时，过程的各个段中，状态变量将在某一确定的范围内取值，状态变量取值的集合称为状态集合。

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，决策变量对应着不同的数值。

**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。

**状态转移方程：**阶段之间状态变量值的变化存在一定的关系。

如果给定*i*阶段的状态变量 $x(i)$ 的值后，第*i*+1阶段的状态变量 $x(i+1)$ 就可以完全确定，即 $x(i+1)$ 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 $x(i+1)$ 的函数关系，表示为：

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

——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:

每一决策都附有一定的“成本”，决策序列的成本是序列中所有决策的成本之和。

设从阶段 $i$ 到阶段 $i+1$ 有 $p_i$ 种不同的选择，则从阶段1至阶段 $n$ 共有 $p_1 p_2 \dots p_n$ 种不同的途径，每个途径对应一个决策序列。

问：这些途径里面，哪一个的成本最小？

——如何求取最优决策序列？



**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。

**目标函数**：用来衡量可行解优劣的标准，通常以函数形式给出。

**最优解**：能够使目标函数取极值的可行解是最优解。

**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

记问题的决策序列为：(x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>)，其中，x<sub>i</sub>表示第i阶段的决策：

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

### 1) 枚举法

**穷举**可能的决策序列，从中选取可以获得最优解的决策序列：

若问题的决策序列由n次决策构成，每一阶段分别有p<sub>1</sub>、p<sub>2</sub>、...、p<sub>n</sub>种选择，则可能的决策序列将有p<sub>1</sub>p<sub>2</sub>...p<sub>n</sub>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，提出了著名的**最优化原理**(principle of optimality)，从而把多阶段过程转化为一系列单阶段问题，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。

动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便。





---

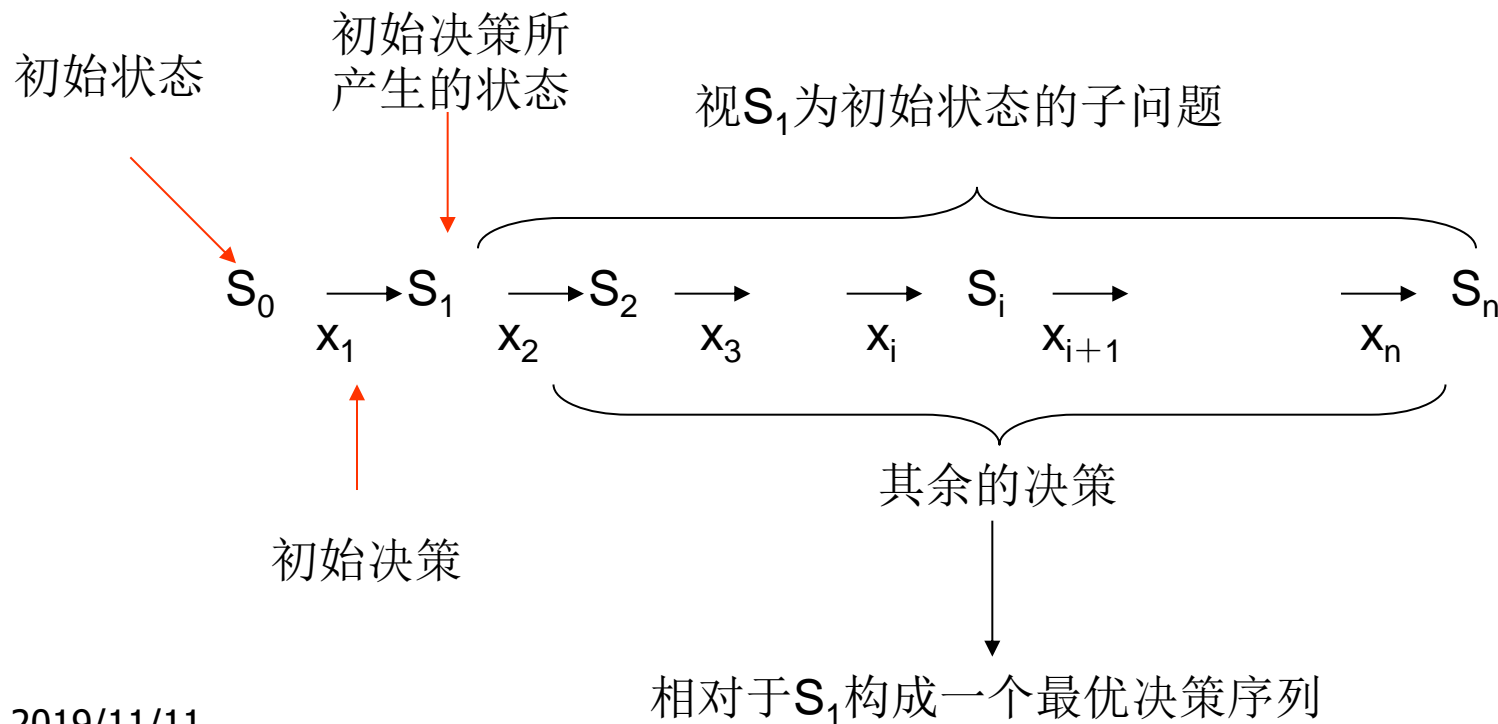
用动态规划策略求解，需要问题满足两个基本性质：

无后效性

最优化原理（最优子结构性）

### 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。

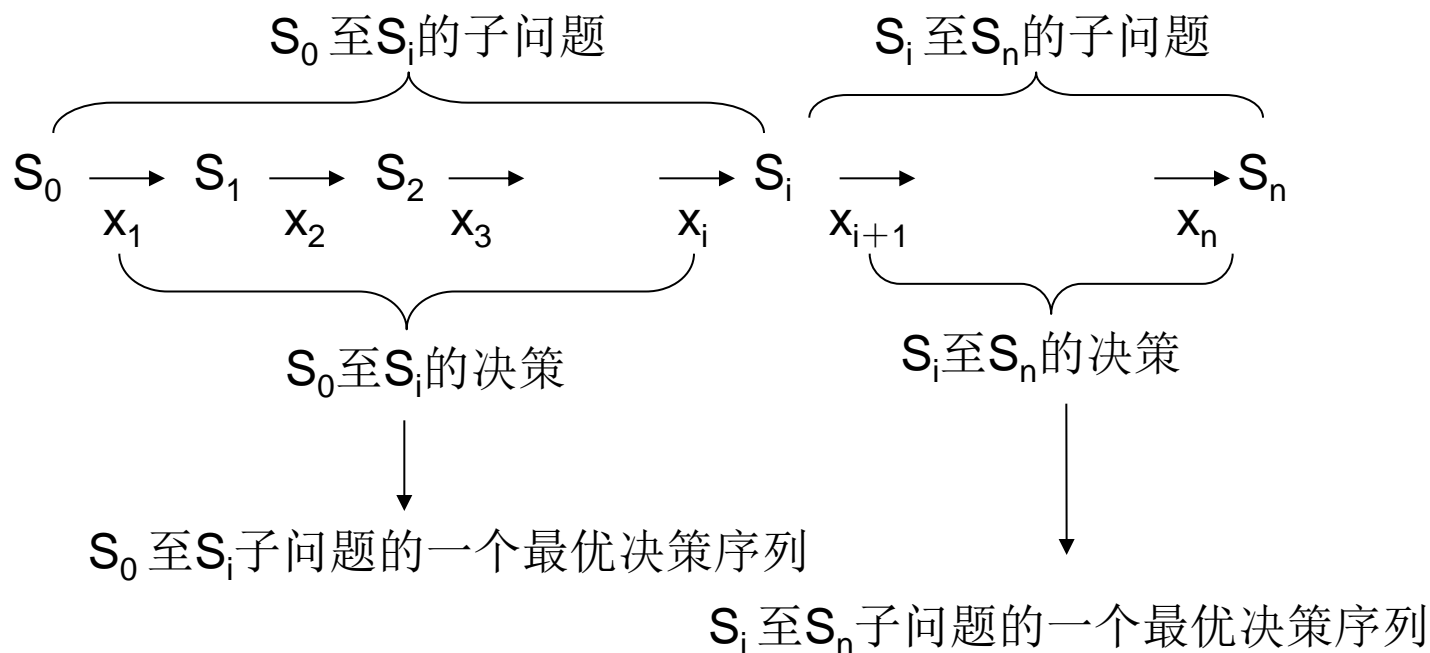


## 对最优性原理的另外陈述：

一个最优策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的决策必须构成最优（子）策略。

简而言之，一个最优策略的子策略应是最优的，故又称“**最优子结构性性质**”。

延伸：若全局是最优的，则局部亦是最优的



特征：如果整个序列是最优决策序列，则该序列中的任何一段子序列将是相对于该子序列所对应子问题的最优决策子序列（**最优子结构**）。

## 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：

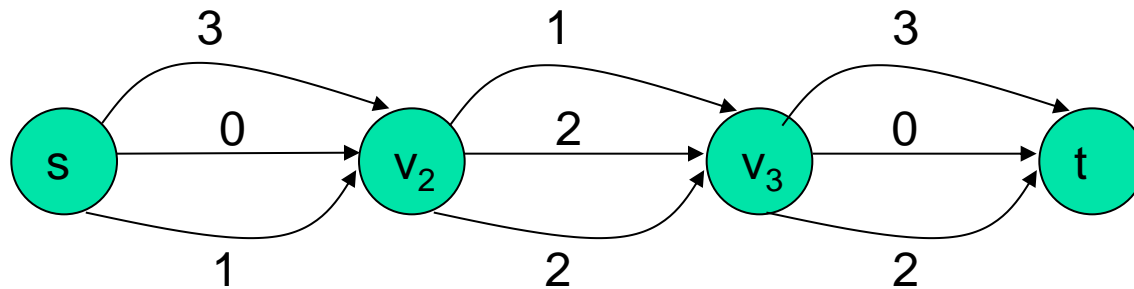
对  $v_1v_2v_3\cdots v_n$  中的任意一段子路径：

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

将代表从  $v_p$  至  $v_q$  的最短子路径。

**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（即 $\text{Length}(s, t) \bmod 4$ ）最小。



此时，问题的一个最优决策序列是：

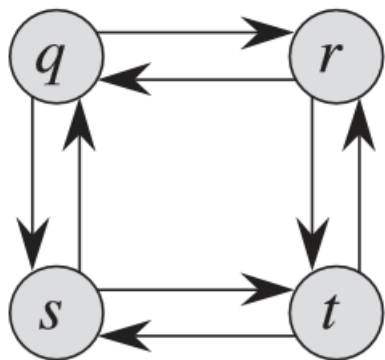
$s \xrightarrow{3} v_2 \xrightarrow{2} v_3 \xrightarrow{3} t$

但最优子结构不成立：最优决策序列上的任一子决策序列相对于当前子问题不是最优的。

## ■ 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

**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。



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

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。

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

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



## 回顾:

1) 不管是钢管切割问题还是矩阵链乘问题, 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: 
$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

# 关于动态规划求解策略的进一步说明

1) 动态规划算法通常用于求解具有某种最优性质的问题。

- 在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。

2) 动态规划是一种策略，不是一个具体的算法。

- 因此，动态规划求解不象搜索或数值计算，具有一个标准的数学表达式和明确清晰的解题方法。
- 动态规划针对的最优化问题，由于问题性质的不同，确定最优解的条件的不同，动态规划的设计对不同的问题，有各具特色的解题方法。

### 3) 对动态规划带来的改进的理解

**改进一：**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，因此可能有**多项式**的计算复杂度。

**改进二：重叠子问题性：**动态规划与分治法也不同，分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则有些子问题被重复计算了很多次。动态规划**保存了已解决的子问题的答案**，在需要时找出已求得的答案，**避免了大量的重复计算**，节省了时间。

# 动态规划的技术要点：

- 用一个表（备忘）来记录所有已解的子问题的答案。
- 不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。
- 动态规划算法都**具有类似的填表模式**。

利用查表，避免对重复子问题的重复求解，动态规划可以将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法，这是动态规划算法的根本目的。

■ 详见P218~220.

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.

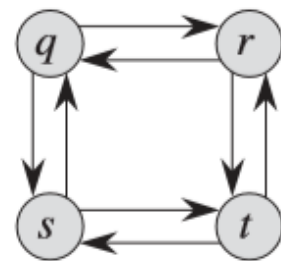
# 子问题无关性

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于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。

## 关于动态规划求解代价的说明

在动态规划方法中，我们通常**自底向上**地使用最优子结构，即首先求子问题的最优解，然后求原问题的最优解。

在求解原问题的过程中，需要在涉及的子问题中做出选择，选出能得到原问题最优解的子问题。这样，**求解原问题的代价通常就是求子问题最优解的代价加上此次选择直接产生的代价。**

如矩阵链乘算法，计算矩阵链 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案，若选择划分位置为 $A_k$ ，则计算矩阵链 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案的代价就是子问题 $A_i \cdots A_k$ 和 $A_{k+1} \cdots A_j$ 的最优括号化方案的代价加上此次选择本身产生的代价 $p_{i-1} p_k p_j$ 。

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



# 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

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

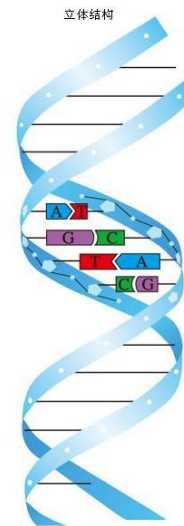

图 15-4 DNA 分子的结构和碱基

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。



子问题

## 2) 递推关系式

表结构，二维数组

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

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

**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



$X_m$ 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	
1	A		0	↑	↑	↑	↖1	↖1	
2	(B)		0	↖①	←-1	←-1	↑1	↖2	
3	(C)		0	↑1	↑1	↖②	←-2	↑2	
4	(B)		0	↖1	↑1	↑2	↑2	↖③	
5	D		0	↑1	↖2	↑2	↑2	↑3	
6	(A)		0	↑1	↑2	↑2	↖3	↖④	
7	B		0	↖1	↑2	↑2	↑3	↖4	

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 <sub>j</sub> (B) D (C) A (B) (A)						
i	x <sub>i</sub>								
0			0	0	0	0	0	0	0
1	A		0	↑	↑	↑ ↖	1	←1	1
2	(B)		0	↖ ①	←1	←1	↑	↖ 2	←2
3	(C)		0	↑	↑	↖ ②	←2	↑	↑
4	(B)		0	↑	↑	↑	↑	↖ ③	←3
5	D		0	↑	↖	↑	↑	↑	↑
6	(A)		0	↑	↑	↑	↖	↑	↖ ④
7	B		0	↑	↑	↑	↑	↖	↑

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	$x_i$								
0			0	0	0	0	0	0	
1	A		0	0	0	1	1	1	
2	(B)		0	①	1	1	2	2	
3	(C)		0	1	1	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

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

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

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

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	↑	↑	②	←2	2	↑
4	(B)		0	↑	↑	↑	↑	③	←3
5	D		0	↑	2	2	2	↑	3
6	(A)		0	↑	2	2	3	↑	④
7	B		0	↑	2	2	3	4	↑

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

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

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

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

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

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

**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 <sub>j</sub> (B) D (C) A (B) (A)						
i	x <sub>i</sub>								
0	x <sub>i</sub>	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	←2
3	(C)	0	1	1	②	←2	2	2	2
4	(B)	0	1	1	2	2	③	←3	
5	D	0	1	2	2	2	3	3	3
6	(A)	0	1	2	2	3	3	④	
7	B	0	1	2	2	3	4	4	4

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

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

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

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

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

$$b[1, 2] = \text{'}\uparrow\text{'}$$

**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 <sub>j</sub> (B) D (C) A (B) (A)						
i									
0	x <sub>i</sub>		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 <sub>j</sub> (B) D (C) A (B) (A)						
i	x <sub>i</sub>								
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 <sub>j</sub> (B) D (C) A (B) (A)						
i	x <sub>i</sub>								
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
			$y_j$ <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span> D <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">C</span> A <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span> <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">A</span>						
i	$x_i$								
0	$x_i$	0	0	0	0	0	0	0	0
1	A	0	↑	↑	↑	↖	1	←1	↖1
2	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span>	0	↖①	←1	←1	↑1	↑2	←2	
3	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">C</span>	0	↑1	↑1	↖②	←2	↑2	↑2	
4	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span>	0	↖1	↑1	↑2	↑2	↖③	←3	
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3	
6	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">A</span>	0	↑1	↑2	↑2	↖3	↑3	↖④	
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4	

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

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

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

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

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

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

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 <sub>j</sub> (B) D (C) A (B) (A)						
i	x <sub>i</sub>								
0	x <sub>i</sub>		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

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

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

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

$$i = 2, j = 1 \quad x_2 = y_1$$

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

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

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
			$y_j$ <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span> D <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">C</span> A <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span> <span style="border: 1px solid green; border-radius: 50%; padding: 2px;">A</span>						
i	$x_i$								
0	$x_i$		0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	↗	1
2	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span>		0	↖	←1	←1	↗	↖	←2
3	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">C</span>		0	↑	↑	↖	←2	↑	↑
4	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">B</span>		0	↖	↑	↑	↑	↖	↖
5	D		0	↑	↑	↑	↑	↑	↑
6	<span style="border: 1px solid green; border-radius: 50%; padding: 2px;">A</span>		0	↑	↑	↑	↖	↑	↖
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追踪。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  $b[i, j] == \text{“}\nwarrow\text{”}$ 
```

```
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
```

```
5      print  $x_i$ 
```

```
6  elseif  $b[i, j] == \text{“}\uparrow\text{”}$ 
```

```
7      PRINT-LCS( $b, X, i - 1, j$ )
```

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

} 注意递归调用  
和print  $x_i$ 的先  
后顺序

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

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

## 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。

不是一般性，这里假设所有元素互异

# 二叉搜索树的检索算法

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

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

:  $X = \text{IDENT}(i) : \text{return}$  //X等于IDENT(i)，则返回//

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

endcase

repeat

end SEARCH

# 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？
- **引入新的因素**：使用频率

# 最优二叉搜索树的定义：

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。

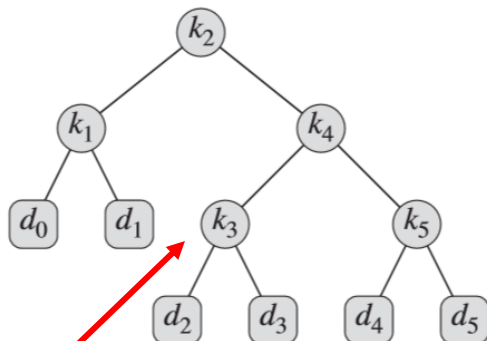


例：设有 $n=5$ 个关键字的集合，每个 $k_i$ 的概率 $p_i$ 和相应 $d_i$ 的概率 $q_i$ 如表所示：

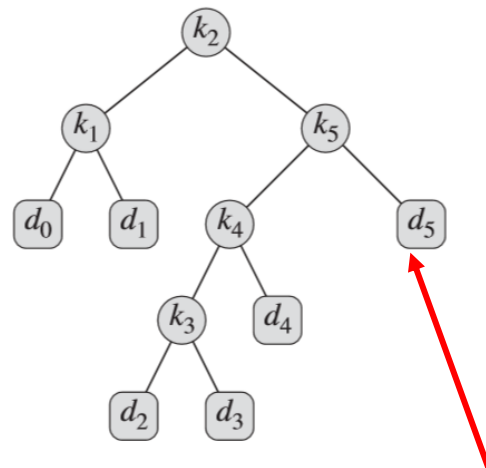
$i$	0	1	2	3	4	5
$p_i$		0.15	0.10	0.05	0.10	0.20
$q_i$	0.05	0.10	0.05	0.05	0.05	0.10

这里有： $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$ 。成功检索和不成功检索总共有 $2n+1$ 种情况

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。

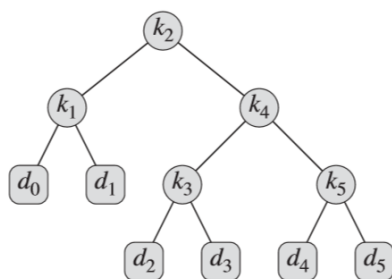
# 二叉搜索树的期望搜索代价

对于特定搜索对象，搜索过程是从根开始到某个结点的检索过程。成功搜索结束与内结点，不成功搜索结束与外部结点。

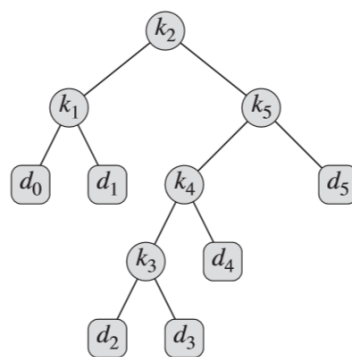
记一次搜索的代价等于从根结点开始访问结点的数量(含内部结点和外部结点)。

记 $\text{depth}_T(i)$ 为结点 $i$ 在 $T$ 中的深度（根到 $i$ 的路径上的边数）。

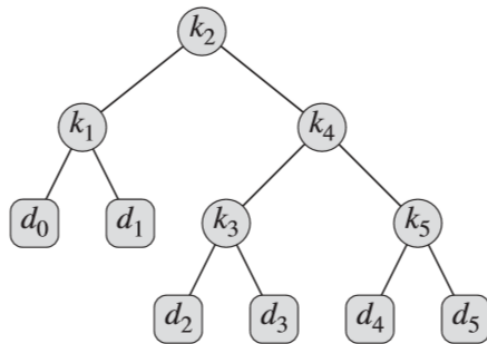
则从根结点开始访问结点 $i$ 的数量等于 $\text{depth}_T(i) + 1$ ；



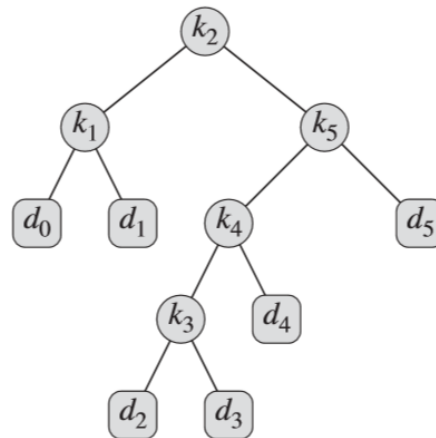
(a)



(b)



(a)



(b)

则二叉搜索树T的期望代价为

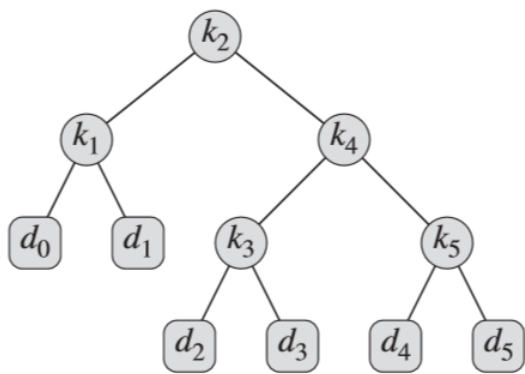
$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i \end{aligned}$$

即：加权平均代价，包括所有成功搜索的结点和失败搜索的结点。

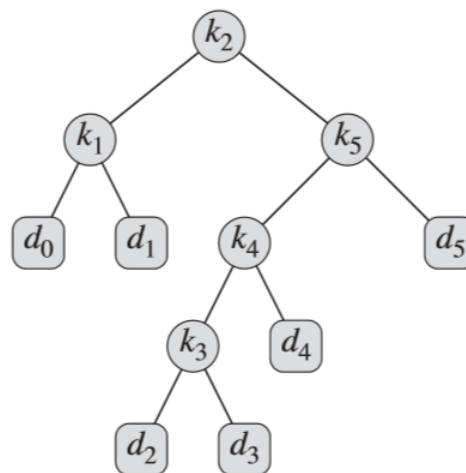
例:  $n=5$ ,

$i$	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)

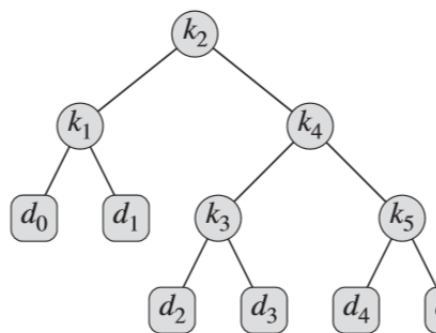
$$\begin{aligned}
 E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\
 &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i,
 \end{aligned}$$

$n=5,$

$i$	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

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

逐结点计算树(a)的期望搜索代价，如表所示：



(a)

(a) 的期望搜索代价为2.80

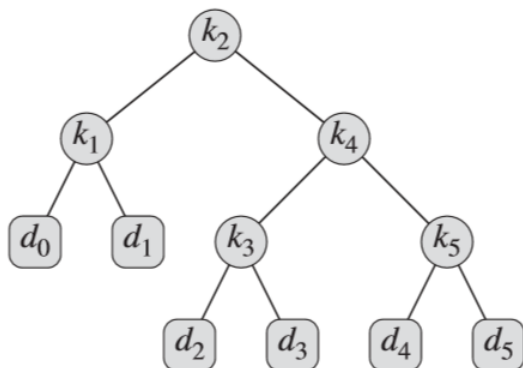
$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i, \end{aligned}$$

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

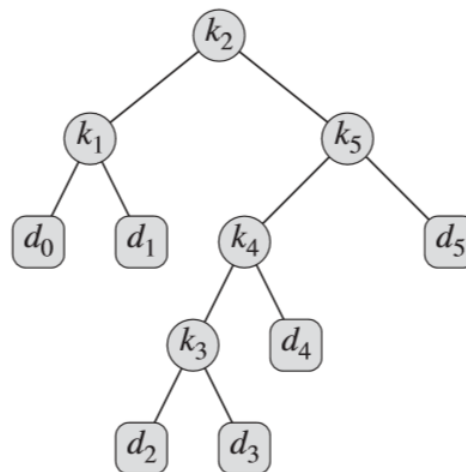
例:  $n=5$ ,

$i$	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。

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

## 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。

枚举？ No！

# 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？

## 谁可能是这个根呢?

$k_r$  是  $k_i \cdots k_j$  中的一个。

**策略：** 在  $i \leq l \leq j$  的范围内检查所有可能的结点  $k_l$ 。

对每一个  $l$ ，事先分别求出包含关键字  $k_i, \dots, k_{l-1}$  和关键字  $k_{l+1}, \dots, k_j$  的最优二叉搜索子树，通过组合左、右子树找到具有最小期望成本的  $k_r$ ，它就是包含关键字  $k_i, \dots, k_j$  的最优二叉搜索（子）树的根。

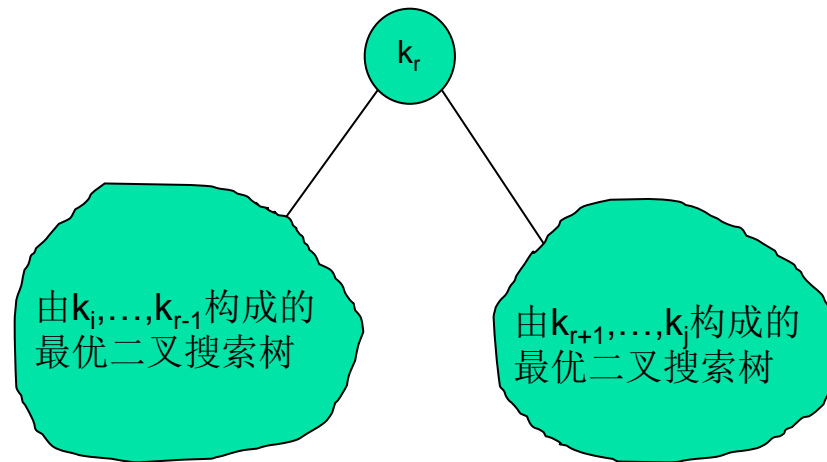
**计算过程：** 求解包含关键字 $k_i, \dots, k_j$ 的最优二叉搜索树，其中  
 $1 \leq i, j \leq n$  且  $i \leq j$ 。

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

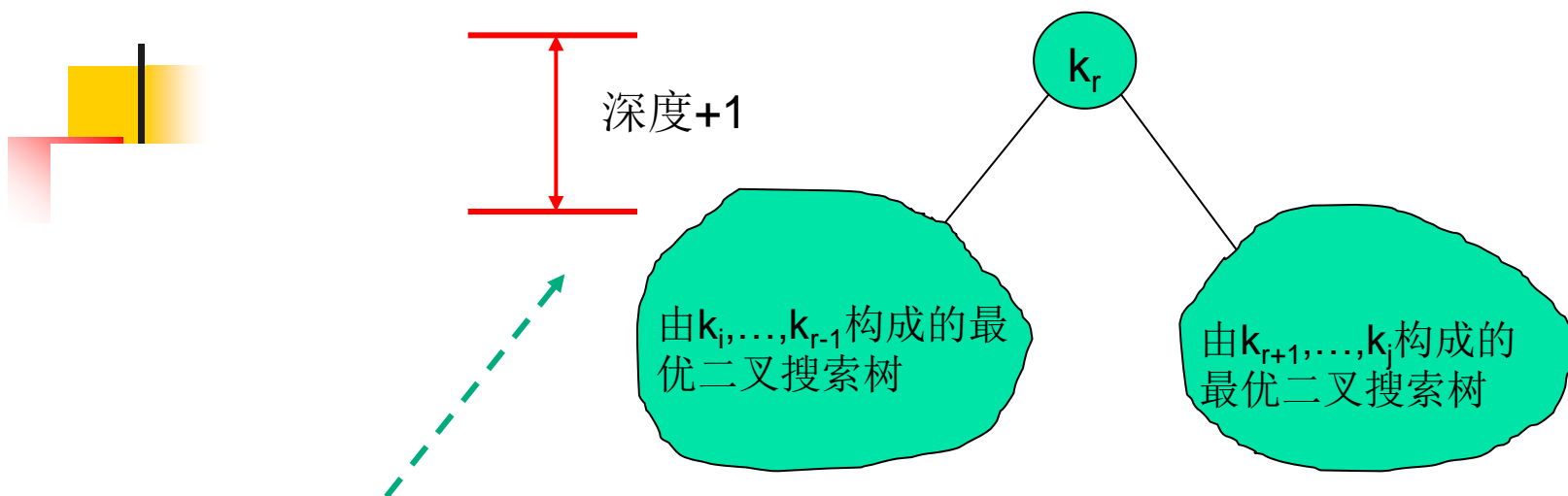
- $e[1, n]$ 为问题的最终解的期望搜索代价。

(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 $r$ ， $k_r$ 即为相对于关键字 $k_i, \dots, k_j$ 的最优二叉搜索（子）树的树根。

在求出 $e[1, n]$ 后，利用 $\text{root}$ 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

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

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

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  $root$ 
```

l是区间长度

i、j是区间下标

自底向上的迭代计算

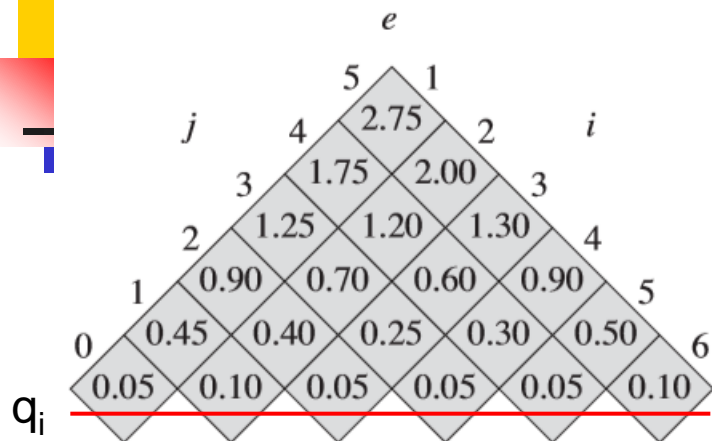
时间复杂度 $\Theta(n^3)$



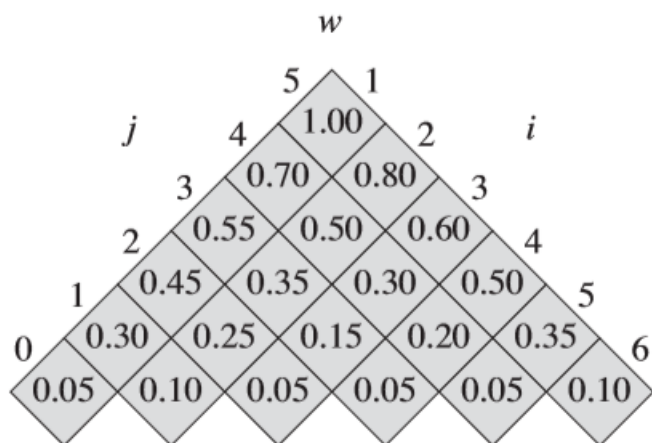
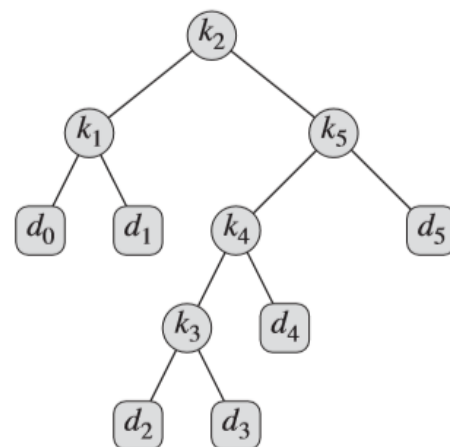
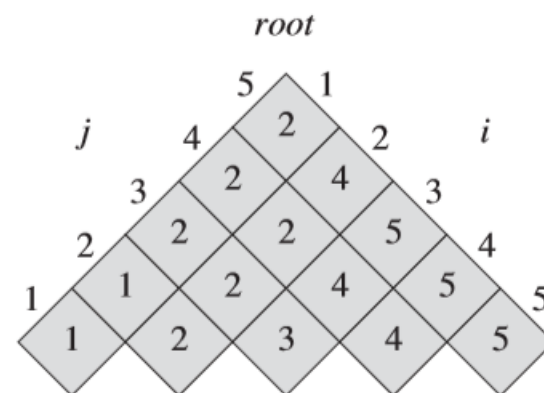
■ 例:  $n=5,$

$i$	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

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



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

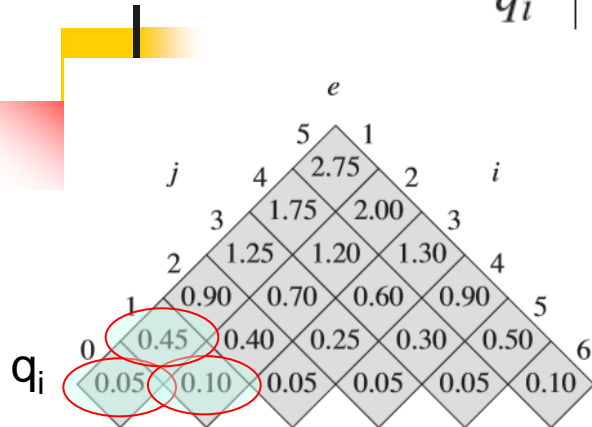


$$w[i, j] = w[i, j-1] + p_j + q_j.$$

■ 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

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



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

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

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

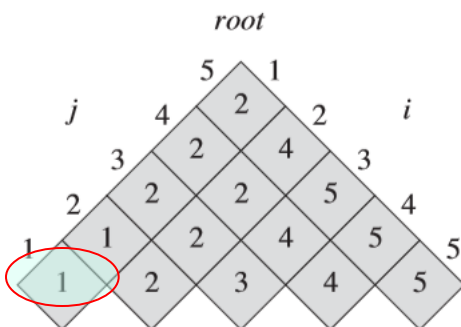
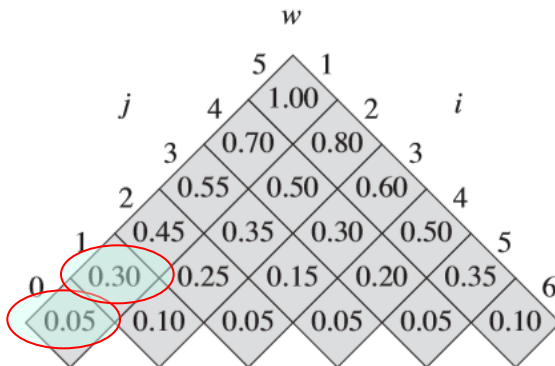
$$e[2, 1] = q_1 = 0.10$$

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

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

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

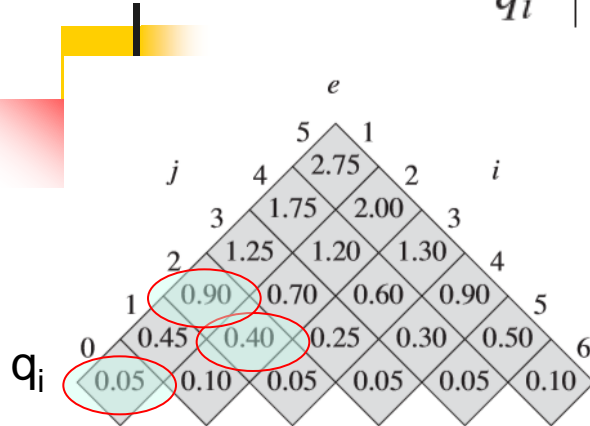
$$r[1, 1] = 1$$



■ 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

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



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

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

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

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

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

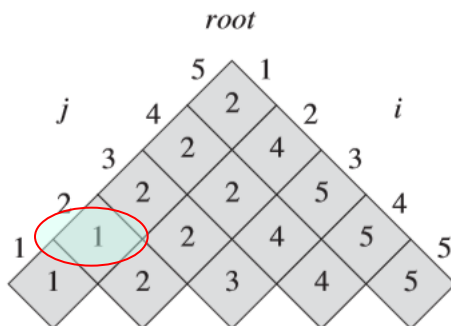
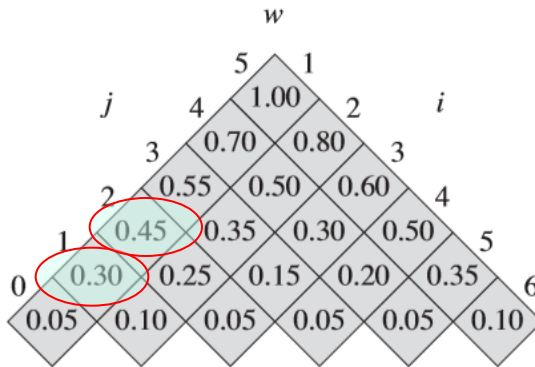
$$e[3, 2] = q_2 = 0.05$$

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

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

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

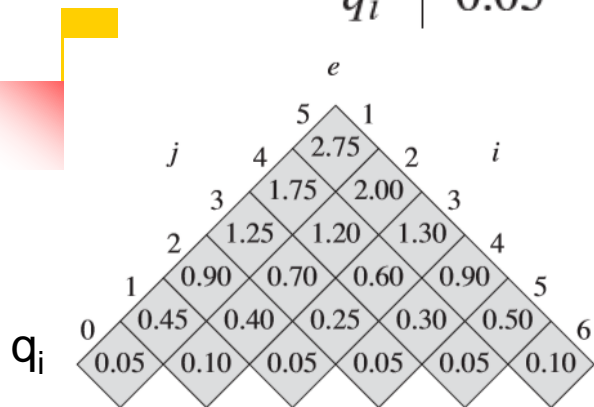
$$r[1, 2] = 1$$



$n=5,$

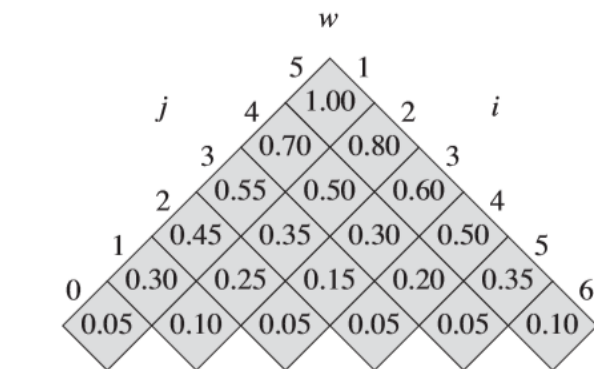
$i$	0	1	2	3	4	5
$p_i$		0.15	0.10	0.05	0.10	0.20
$q_i$	0.05	0.10	0.05	0.05	0.05	0.10

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



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

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



课堂练习：计算

(1)  $w[2, 3]$

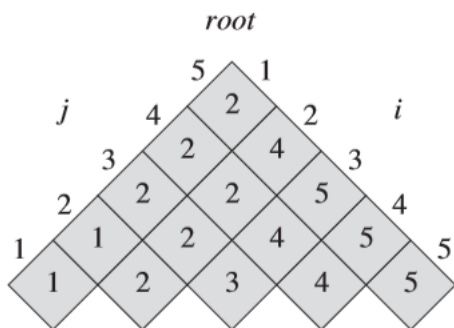
$e[2, 3]$

$r[2, 3]$

(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，

		$j$			
$i$	$r_{ij}$	1	2	3	4
	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  $c_2 = c_3 = 3$ .