

算法设计与分析

主讲人：权丽君

Email: *ljquan@suda.edu.cn*

苏州大学 计算机学院



第8讲 动态规划

内容提要：

- 动态规划
- 贪心算法
- 摊还分析

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很多个。

——这种找最优解的问题通常称为**最优化问题**。

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， $F(X)$ 取得其最大值或最小值。

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

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

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

最优化问题的分类：

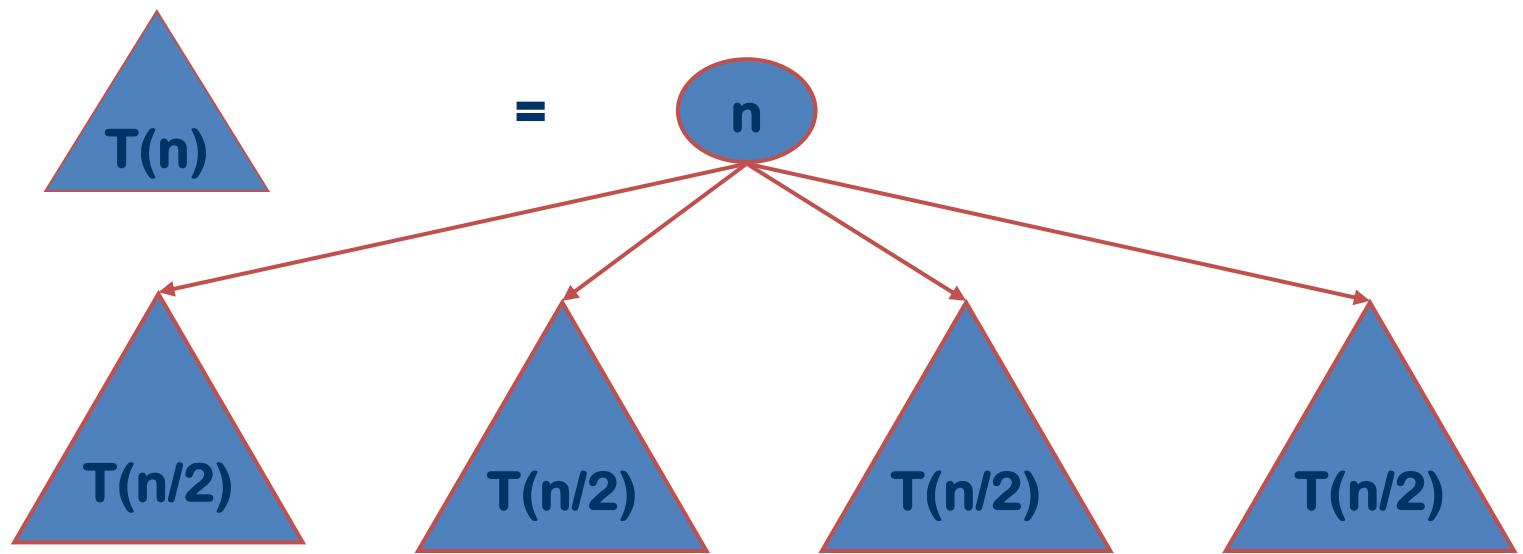
根据描述约束条件和目标函数的数学模型的特性和问题的求解方法的不同，可分为：线性规划、整数规划、非线性规划、动态规划等问题。而研究解决这些问题的科学一般就总称之为**最优化理论和方法**。

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）

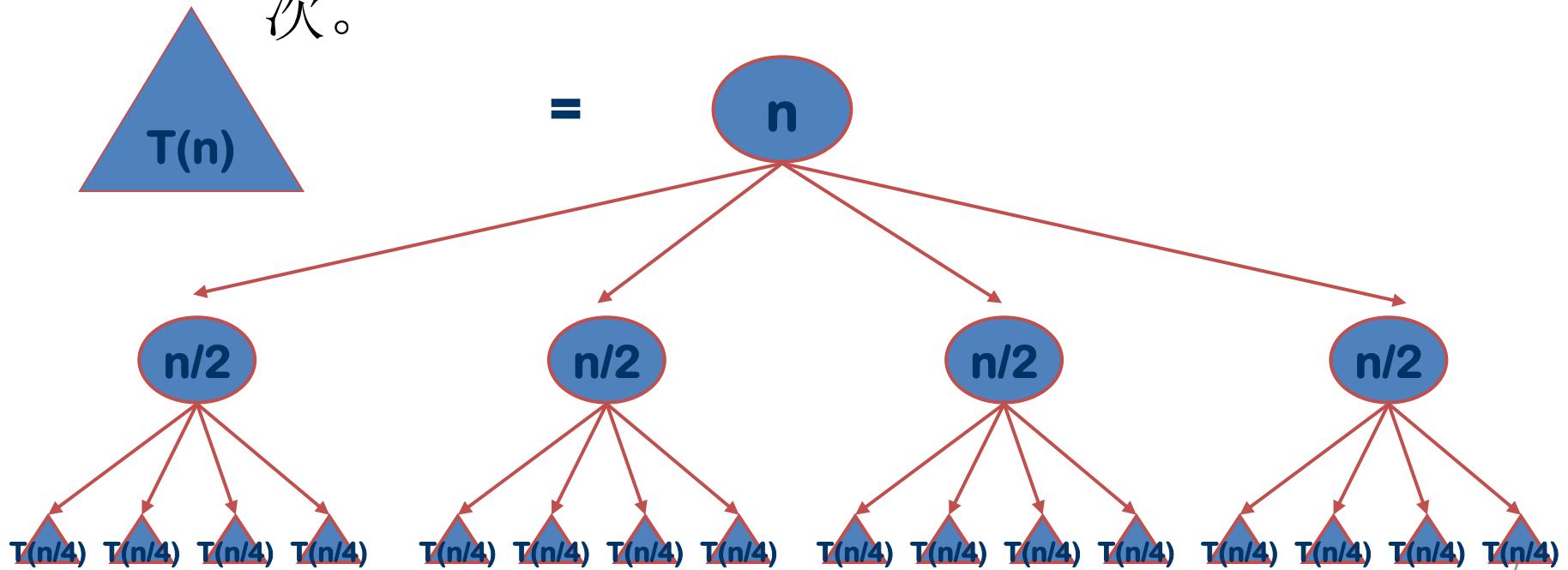
动态规划 总体思想

- 动态规划的思想实质是分治思想和解决冗余
- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



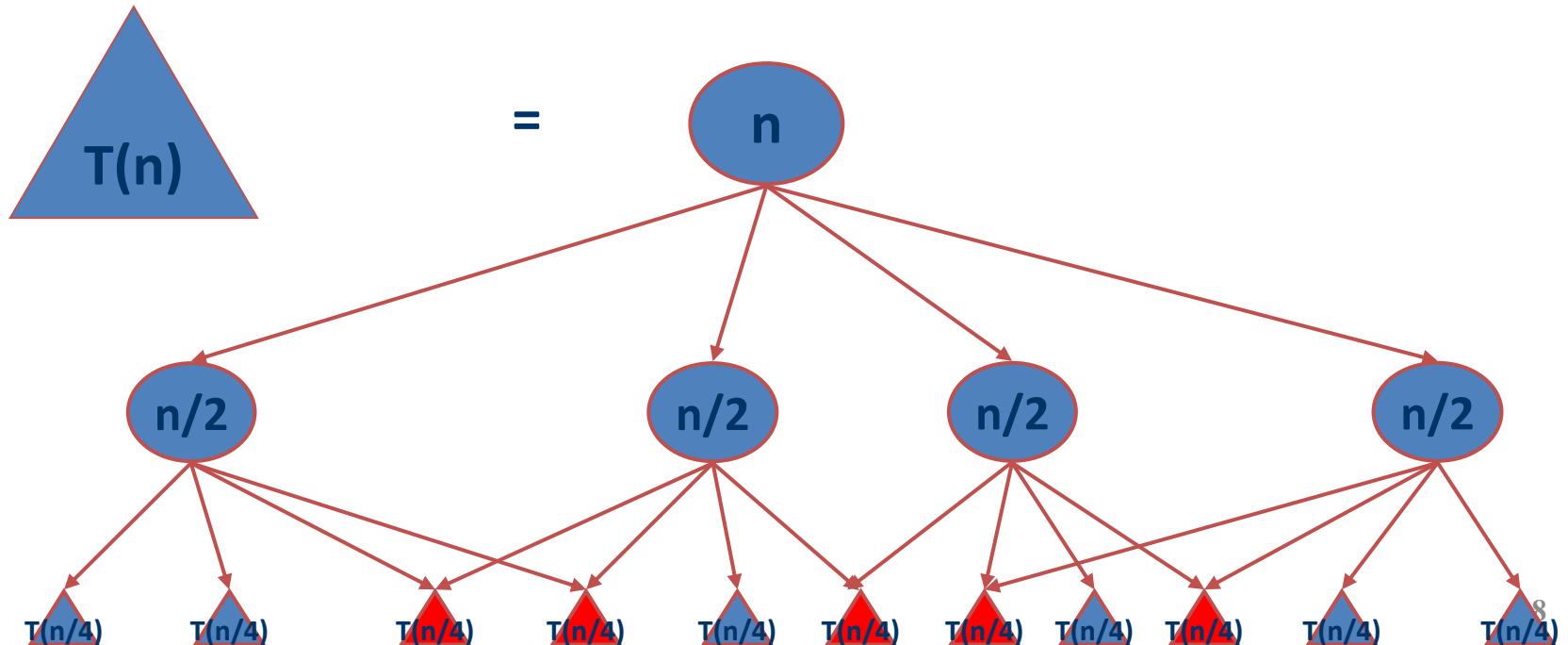
动态规划 总体思想

- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。



动态规划 总体思想

- 如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。动态规划法用一个表来记录所有已解决的子问题的答案。具体的动态规划算法尽管多种多样，但它们具有相同的填表方式。



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，避免了不必要的计算工作。

动态规划 总体思想

基本步骤：

- ① 找出最优解的性质，并刻画其结构特征。 → 划分子问题
- ② 递归地定义最优解的值。 → 给出最优解的递归式
- ③ 按自底向上的方式计算最优解的值。
- ④ 由计算出的结果构造一个最优解。

注：

- 步骤①~③是动态规划算法的基本步骤。如果只需要求出最优值的情形，步骤④可以省略；
- 若需要求出问题的一个最优解，则必须执行步骤④，步骤③中记录的信息是构造最优解的基础；

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。

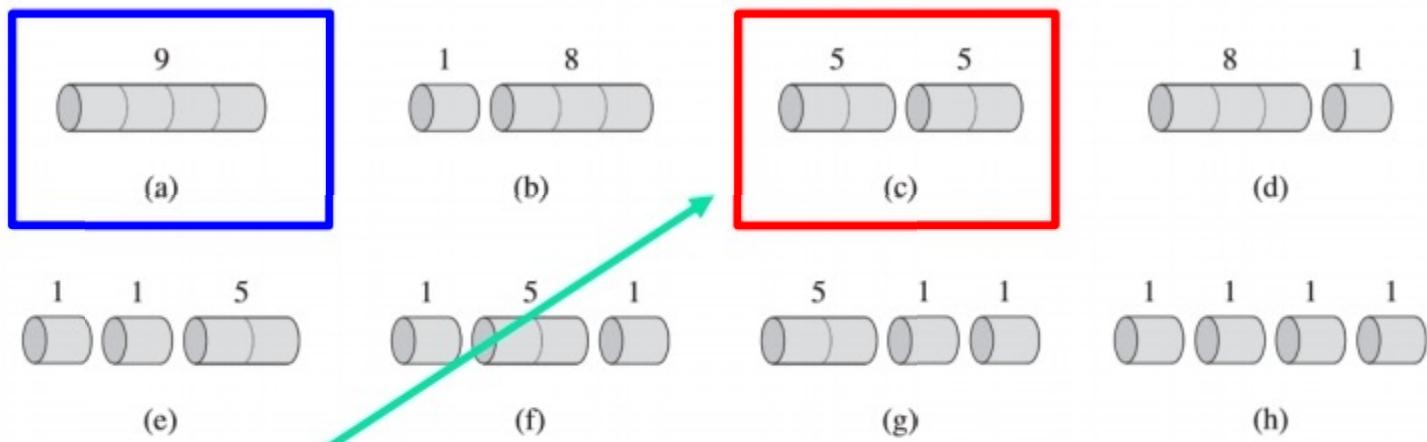
思考：先按 p_i 最大出售，剩余的再按较小 p_j 出售，是否就可以获得最好的收益？（贪心策略）

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：

$$r_n = p_{i1} + p_{i2} + \dots + p_{ik}$$

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）

$r_6=17$ ，切割方案 $6=6$ （无切割）

$r_2=5$ ，切割方案 $2=2$ （无切割）

$r_7=18$ ，切割方案 $7=1+6$ 或 $7=2+2+3$

$r_3=8$ ，切割方案 $3=3$ （无切割）

$r_8=22$ ，切割方案 $8=2+6$

$r_4=10$ ，切割方案 $4=2+2$

$r_9=25$ ，切割方案 $9=3+6$

$r_5=13$ ，切割方案 $5=2+3$

$r_{10}=30$ ，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 :

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

- 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。

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 是钢条长度。

若 $n=0$ ，则收益为0。

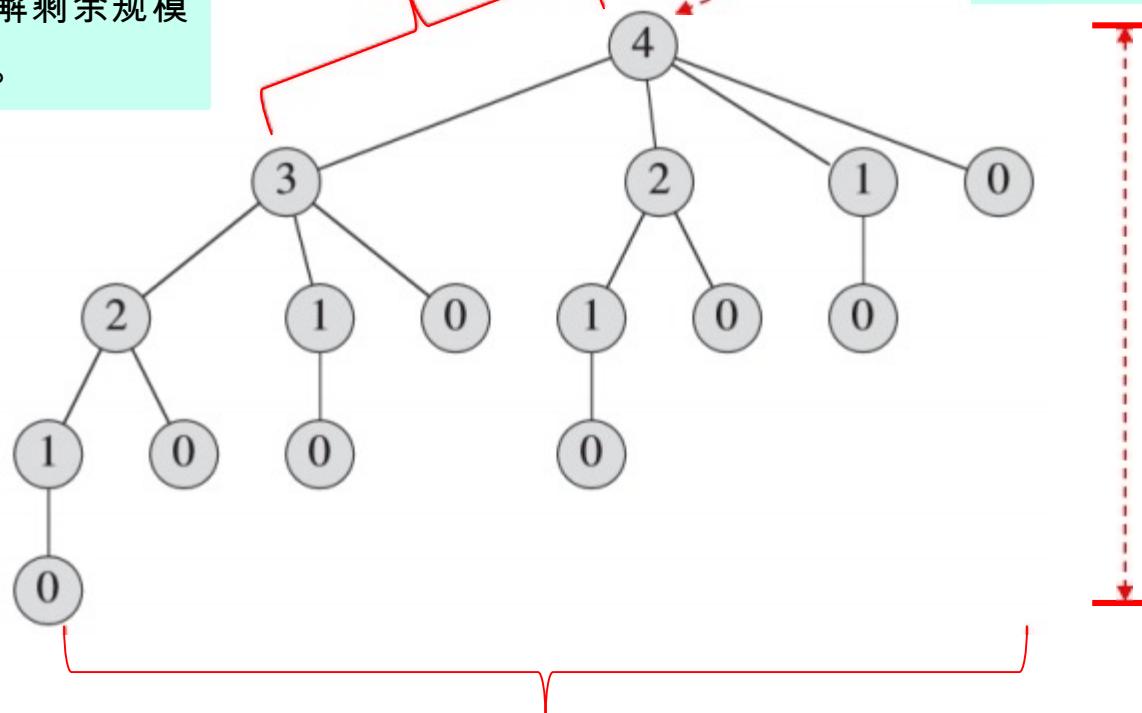
该过程的效率很差：

- 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



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： $T(n) = 2^n$ 练习15.1-1

即CUT-ROD的运行时间为 n 的指数函数

CUT-ROD(p, n)

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

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）。
- 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)

- 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)

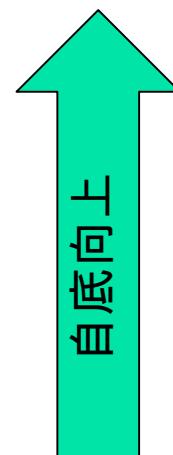
```
1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```



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  $r[n]$ 
```



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。

- 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  $q$ 
```

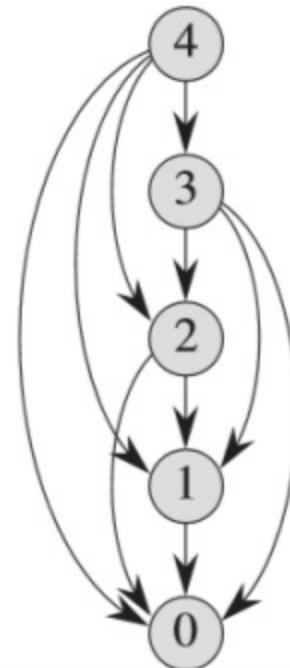
动态规划：递归只是处理的一部分。在求解的过程中记录了重复子问题的解。通过“引用”以前的计算结果，避免重复计算，提高效率。

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：

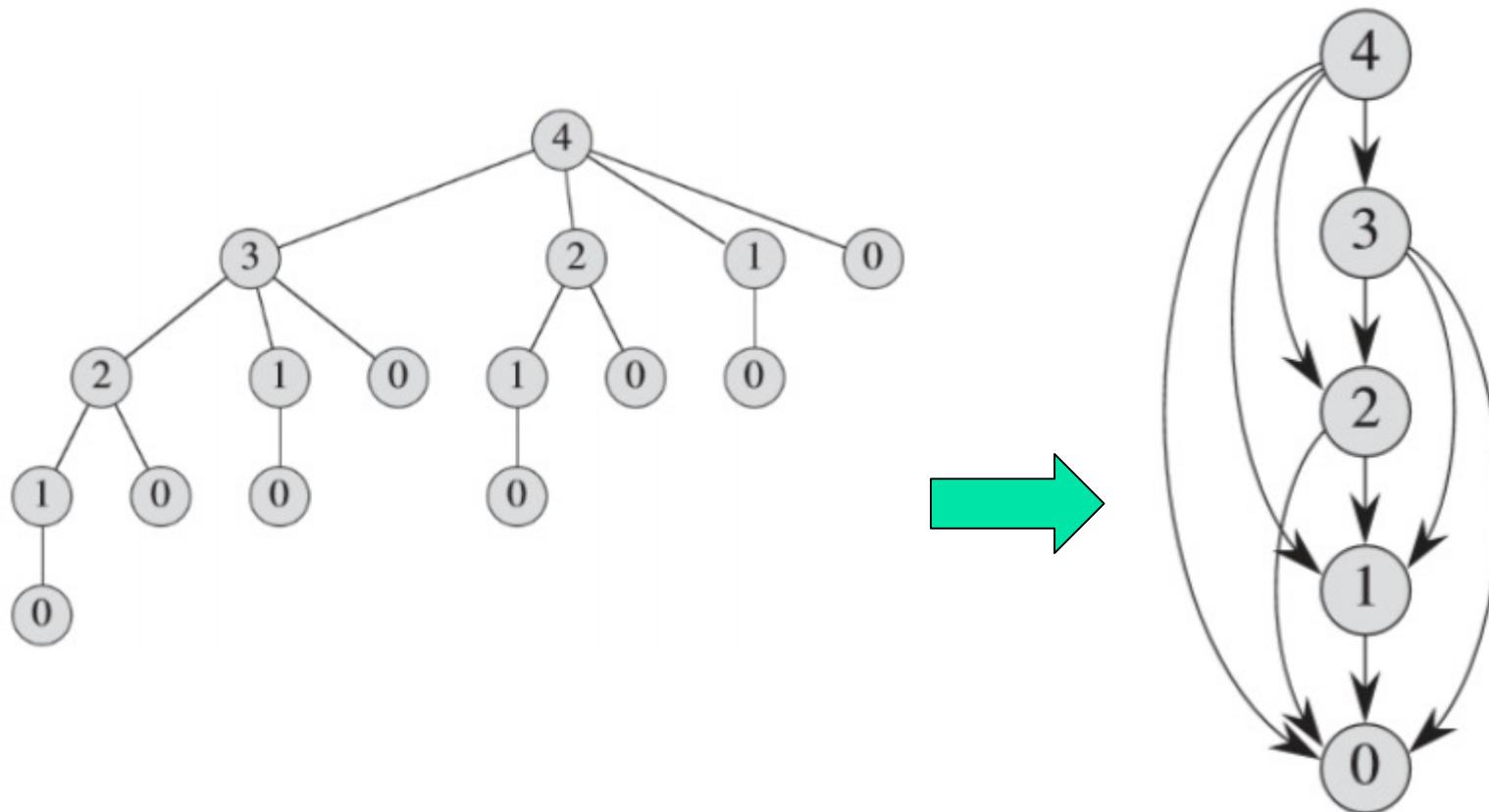
子问题图：用于描述子问题与子问题之间的依赖关系。

- 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。



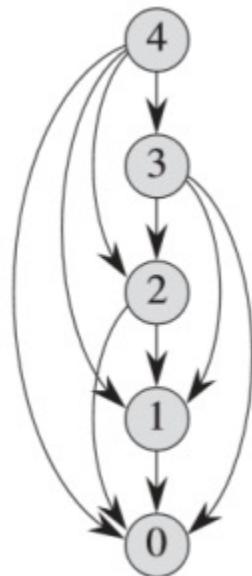
$n=4$ 时，钢条切割问题的子问题图。顶点的标号给出了子问题的规模。有向边 (x,y) 表示当求解子问题x时需要子问题y的解。

- 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。

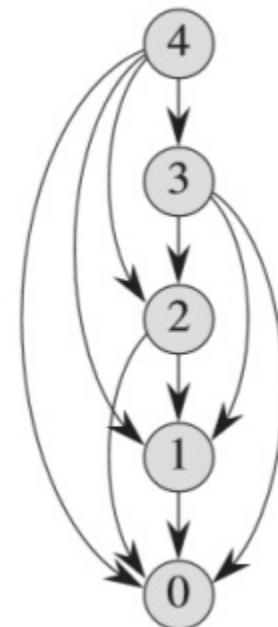
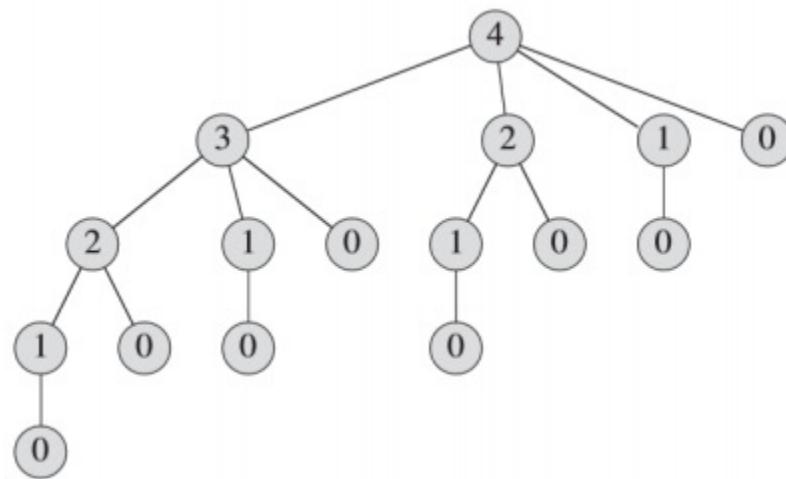
——逆拓扑序，深度优先原则进行处理。



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 $(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$

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：
 - 子问题的定义： $r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$
 - 备忘表的结构： $r[0..n]$

```
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  $q$ 
```

- ◆ **最优化问题**: 问题的可行解有多个, 希望寻找其中的最优解, 记为Min F(X)或Max F(X)
- ◆ **Programming的含义**: 造表
- ◆ **动态规划**: 对重叠子问题只求解一次, 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, 避免重复计算。
- ◆ **动态规划算法的步骤**
 1. 刻画一个最优解的结构特征;
 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。

- ◆ **钢条切割问题**: 零售, 价格表
 - 钢条切割问题具有最优子结构性: 最优方案里的子方案也一定是最优的。
 - 递推求解: $r_n = \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}$
 - 备忘表: $r[0 \dots n]$
 - 重构解: $s[0 \dots n]$
 - 带备忘的自顶向下算法和自底向上算法。
- ◆ **子问题图**: 用于描述子问题与子问题之间依赖关系的图, 是递归调用树的“收缩版”。
 - 动态规划算法的运行时间与子问题图中顶点和边的数量至少呈线性关系。

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：

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  $C$ 
```

注：只有两个矩阵"相容"
(compatible)才能相乘。

} 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 $10 \times 100, 100 \times 5, 5 \times 50$ 。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，得一个 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，得一个 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： $P(n)=\Omega(2^n)$

练习15.2-3

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 $A_i A_{i+1} \dots A_j$ 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：

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

则必须有 $(\overline{A_i A_{i+1} \dots A_k})$ 必是“前缀”子链 $A_i A_{i+1} \dots A_k$ 的一个最优的括号化子方案，记为 $A_{i,k}$ ；同理 $(\overline{A_{k+1} A_{k+2} \dots A_j})$ 也必是“后缀”子链 $A_{k+1} A_{k+2} \dots A_j$ 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： $(A_1 A_{i+1} \dots A_k) (A_{k+1} \dots A_n)$

到哪里找这样的一个k，使得上述计算的标量乘法次数最小呢？

备忘表的结构： $i \sim j$ 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

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

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 $m[i, j]$ 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 $p_{i-1} p_k p_j$ ，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。

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

$m[1, n]$ 是计算 $A_{1,n}$ 的最小代价

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 $p_{i-1} \times p_i$ ， $i = 1, 2, \dots, n$ 。

使 用 辅 助
表 $m[1..n, 1..n]$
保 存 $m[i, j]$ 的 代
价，使用 $s[1..n -$
 $1, 2..n]$ 记录计算
 $m[i, j]$ 时取得最优
代价的 分割点 k 。

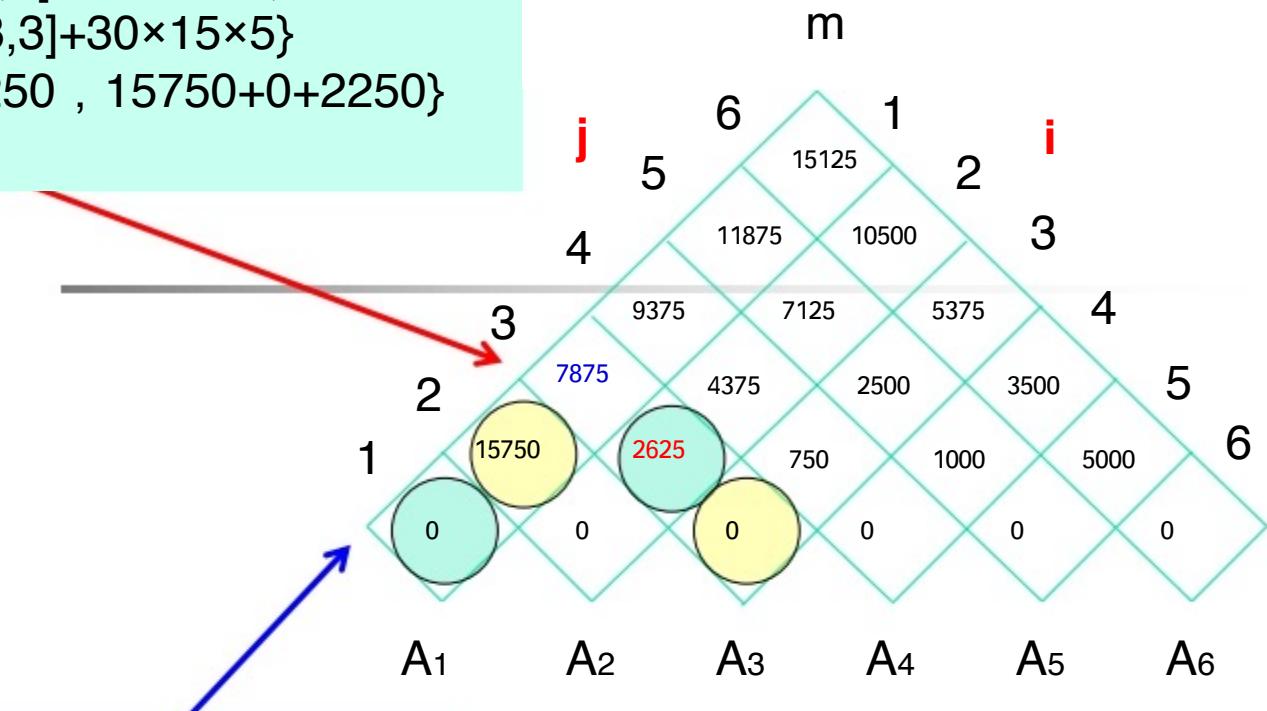
$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

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 $m[1, n]$ 。

$$\begin{aligned}
 m[1,3] &= \min\{m[1,1]+m[2,3]+30\times35\times5, \\
 &\quad m[1,2]+m[3,3]+30\times15\times5\} \\
 &= \min\{0+2625+5250, 15750+0+2250\} \\
 &= 7875
 \end{aligned}$$

例，设

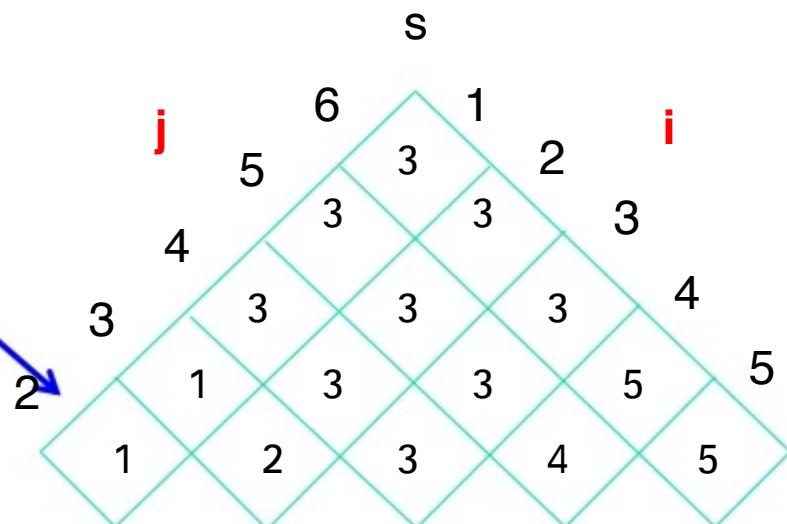
k



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

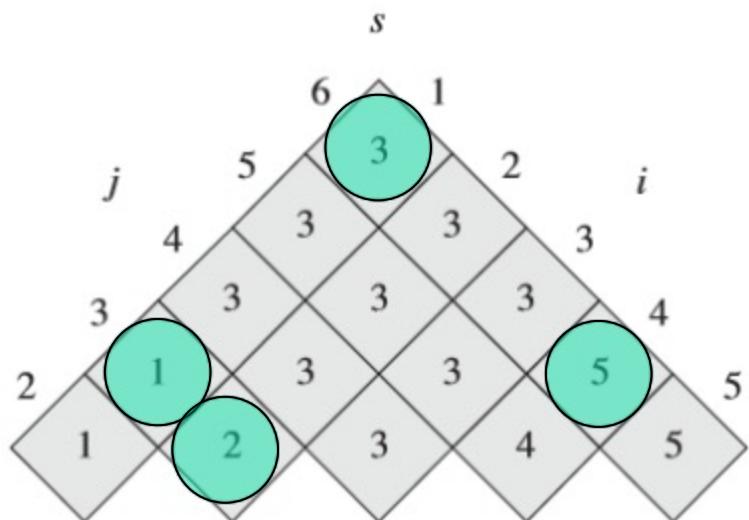
PRINT-OPTIMAL-PARENS(s, i, j)

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



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



回顾：何为Programming？

——造表。

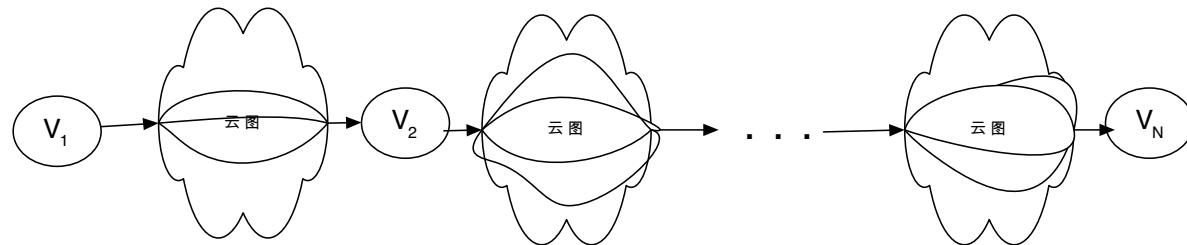
动态规划：与分治不同，适用于**有子问题重叠**的情况。

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、...，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_1, x_2, \dots, x_n)，其中， x_i 表示第*i* 阶段的决策：

$$\begin{array}{ccccccc} & x_1 & & x_2 & & x_3 & \dots & & x_n \\ S_0 & \rightarrow & S_1 & \rightarrow & S_2 & \rightarrow & \dots \dots & \rightarrow & S_n \end{array}$$

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_1 、 p_2 、...、 p_n 种选择，则可能的决策序列将有 $p_1 p_2 \dots p_n$ 个。

2) 动态规划

20世纪50年代初美国数学家RE.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

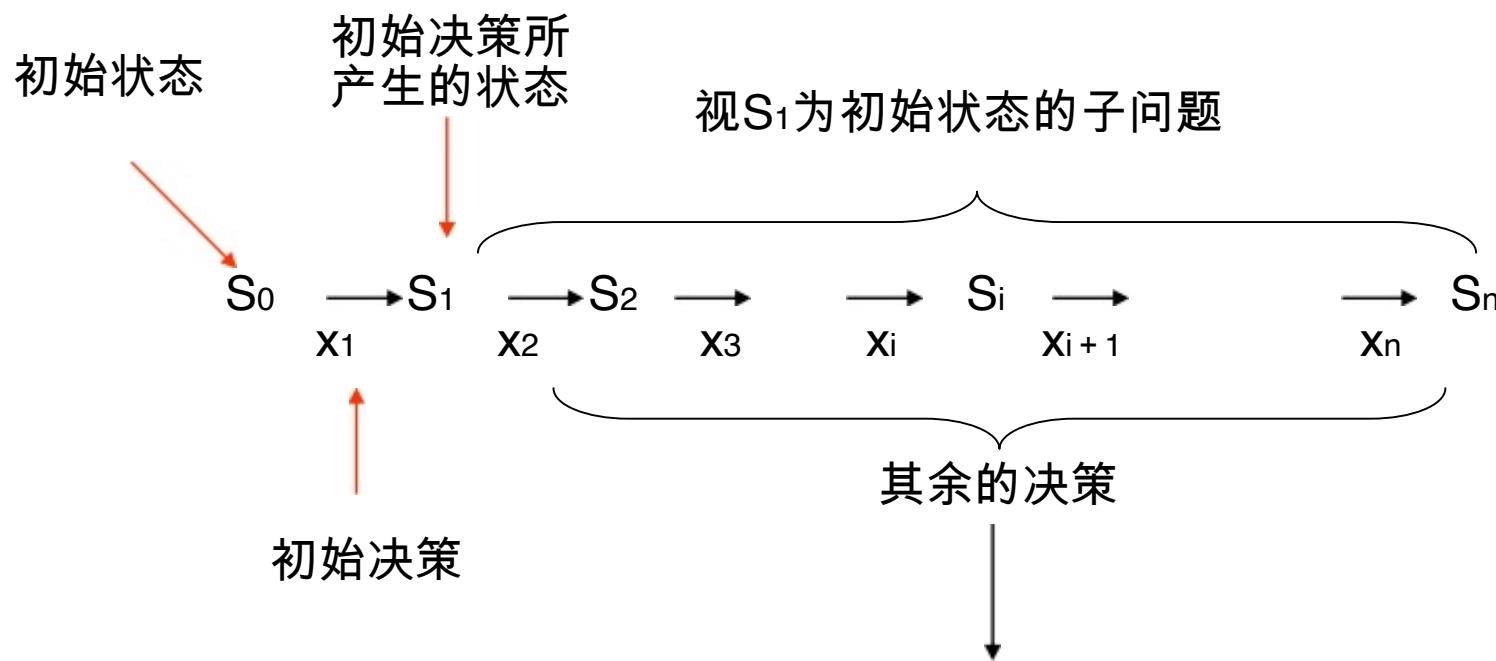
最优化原理（最优子结构性）

无后效性

- 所谓无后效性原则，指的是这样一种性质：某阶段的状态一旦确定，则此后过程的演变不再受此前各状态及决策的影响。也就是说，“未来与过去无关”，当前的状态是此前历史的一个完整的总结，此前的历史只能通过当前的状态去影响过程未来的演变。
- 由此可见，不能使用动态规划的情况
 - 1) 对于不能划分阶段的问题，不能运用动态规划来解；
 - 2) 对于能划分阶段，但不符合最优化原理的，也不能用动态规划来解；
 - 3) 既能划分阶段，又符合最优化原理的，但不具备无后效性原则，还是不能用动态规划来解；
- 误用动态规划程序设计方法求解会导致错误的结果。

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**。

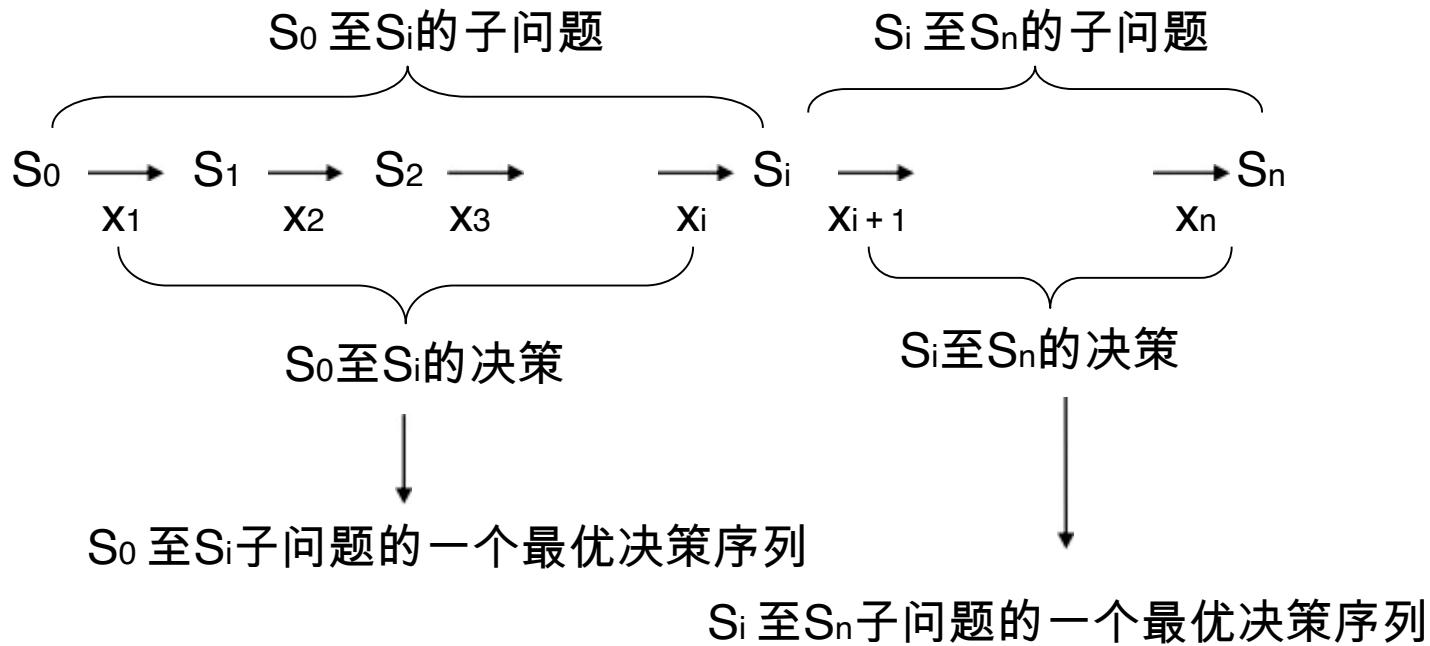


对最优化原理的另外陈述：

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

► $v_2v_3\dots\dots v_n$ 是从 v_2 到 v_n 的最短子路径；

► $v_3\dots\dots v_n$ 是从 v_3 到 v_n 的最短子路径；

.....

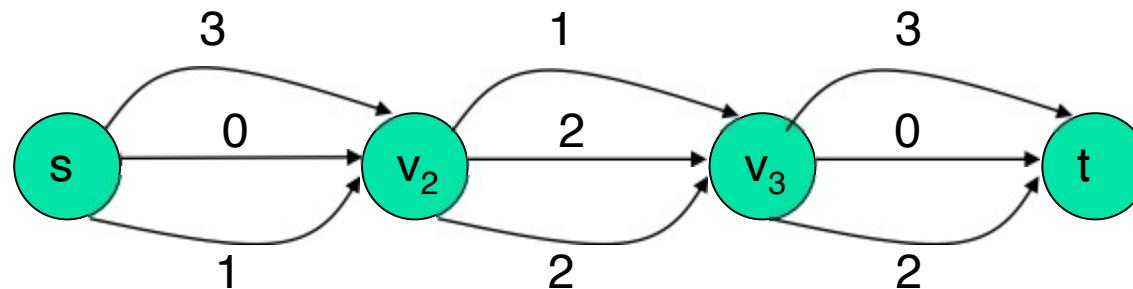
推广：

对 $v_1v_2v_3\dots\dots v_n$ 中的任意一段子路径：

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

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：

$s - \underline{\color{red}3} -> v_2 - \underline{\color{red}2} -> v_3 - \underline{\color{red}3} -> t$

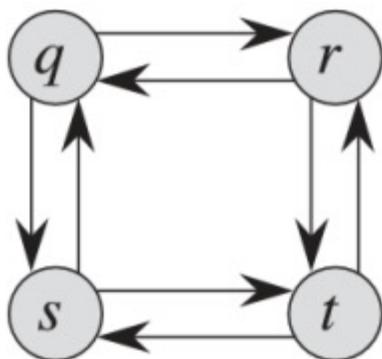
但最优子结构不成立：最优决策序列上的任一子决策序列相对于当前子问题不是最优的。

■ 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

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

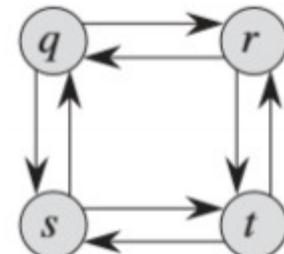
所谓**子问题无关**，是指同一个原问题的一个子问题的解不影响另一个子问题的解，可以各自独立求解。

- **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。



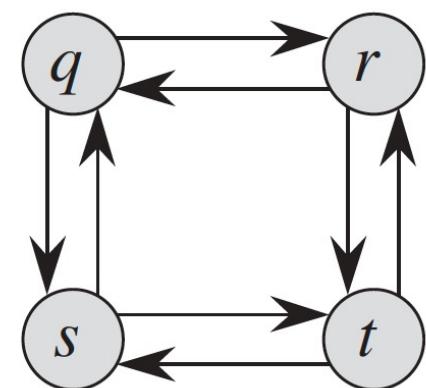
- 详见P217~218.75

最长路径不具有最优子结构

- 设 P 是从 u 到 v 的最长路径， w 是中间某点，则

$$u \xrightarrow{P} v \Rightarrow u \xrightarrow{P_1} w \xrightarrow{P_2} v$$

- 但 P_1 不一定是从 u 到 w 的最长路径， P_2 也不一定是从 w 到 v 的最长路径。例：
- 考虑一最长路径 $q \rightarrow r \rightarrow t$
- 但 q 到 r 的最长路径是： $q \rightarrow s \rightarrow t \rightarrow r$
- r 到 t 的最长路径是： $r \rightarrow q \rightarrow s \rightarrow t$



为什么两问题有差别？

- 最长路径的子问题不是独立的。所谓独立指一个子问题的解不能影响另一个子问题的解。
- 但第一个子问题中使用了s和t，第二个子问题又使用了，使得产生的路径不再是简单路径。

从另一个角度看一个子问题求解时使用的资源(顶点)不能在另一个子问题中再使用。

- 最短路径问题中，两子问题没有共享资源，可用反证法证明之。
- 例：矩阵链乘 $A_{i..j} \Rightarrow A_{i..k} \cdot A_{k+1..j}$

显然两子链不相交，没有资源共享，是相互独立的两子问题

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。

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：即首先求得子问题的最优解，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，求原问题最优解的代价通常就是**求子问题最优解的代价加上由此次选择直接产生的代价**。

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

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

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。

$$f(x_1, x_2, \dots, x_i) \rightarrow x_{i+1}$$

向后递推

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

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化结构性：

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

- 如何描述子问题空间(子问题结构,不同的子问题个数)尽可能使其简单,然后再考虑有没有必要扩展. 例 :

$$A_{1..n} \Rightarrow A_{1..k} \cdot A_{k+1..n} \Rightarrow (A_{1..k_1} \cdot A_{k_1+1..k})(A_{k+1..k_2} \cdot A_{k_2+1..n})$$

由此可见, 最合适的子问题空间描述为 : $A_i A_{i+1} \cdots A_j$

- 最优子结构有关的两方面问题
 - 用在最优解中有多少个子问题
 - 用在最优解中的子问题有多少种选择
 - 例如, 矩阵链乘——两个子问题, $j-i$ 种选择
- 动态规划算法的运行时间
 - 子问题总数
 - 对每个子问题涉及多少种选择
 - 例如:
矩阵链乘共要解 $\Theta(n^2)$ 个子问题: $1 \leq i \leq j \leq n$, 求解每个子问题至多有 $n-1$ 种选择
 \therefore 最终的运行时间为 $\Theta(n^3)$
- 动态规划求解方式
 - 自底向上

重叠子问题

当用递归算法解某问题时，重复访问(计算)同一子问题

- 分治法与动态规划的比较

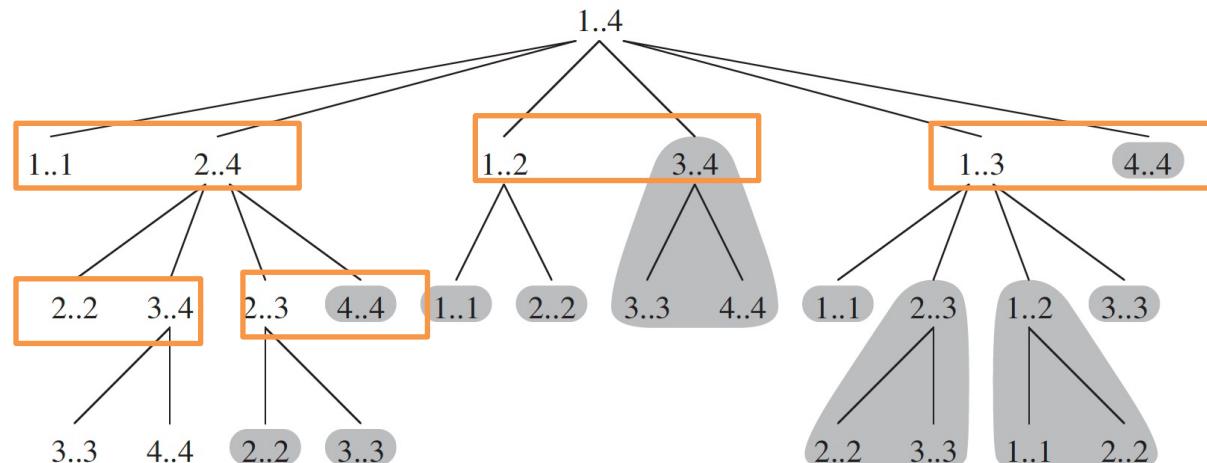
- 当递归每一步产生一个新的子问题时，适合使用分治法
- 当递归中较多出现重叠子问题时，适合使用动态规划，即对重叠子问题只求解一次，然后存储在表中，当需要使用时常数时间内查表。若子问题规模是多项式阶的，动态规划特别有效。

- 用自然递归算法求解

自顶向下的递归算法（没带备忘录）

```
RECURSIVE-MATRIX-CHAIN( $p, i, j$ )
```

```
1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j - 1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
         +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
         +  $p_{i-1} p_k p_j$ 
6      if  $q < m[i, j]$ 
7           $m[i, j] = q$ 
8  return  $m[i, j]$ 
```



1~2行和第6~7行至少各花费单位时间，第5行的加法运算也是如此，因此我们得到如下递归式：

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1), n > 1$$

注意，对 $i=1, 2, \dots, n-1$ ，每一项 $T(i)$ 在公式中以 $T(k)$ 的形式出现了一次，还以 $T(n-k)$ 的形式出现了一次，而求和项中累加了 $n-1$ 个 1，在求和项之前还加了 1，因此公式可改写为：

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n \quad (15.8)$$

下面用代入法证明 $T(n)=\Omega(2^n)$ 。特别地，我们将证明，对所有 $n \geq 1$ ， $T(n) \geq 2^{n-1}$ 都成立。基本情况很简单，因为 $T(1) \geq 1 = 2^0$ ，利用数学归纳法，对 $n \geq 2$ ，我们有

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n = 2 \sum_{i=0}^{n-2} 2^i + n = 2(2^{n-1} - 1) + n \quad (\text{由公式(A.5)}) \\ &= 2^n - 2 + n \geq 2^{n-1} \end{aligned}$$

因此，调用 RECURSIVE-MATRIX-CHAIN($p, 1, n$) 所做的总工作量至少是 n 的指数函数。

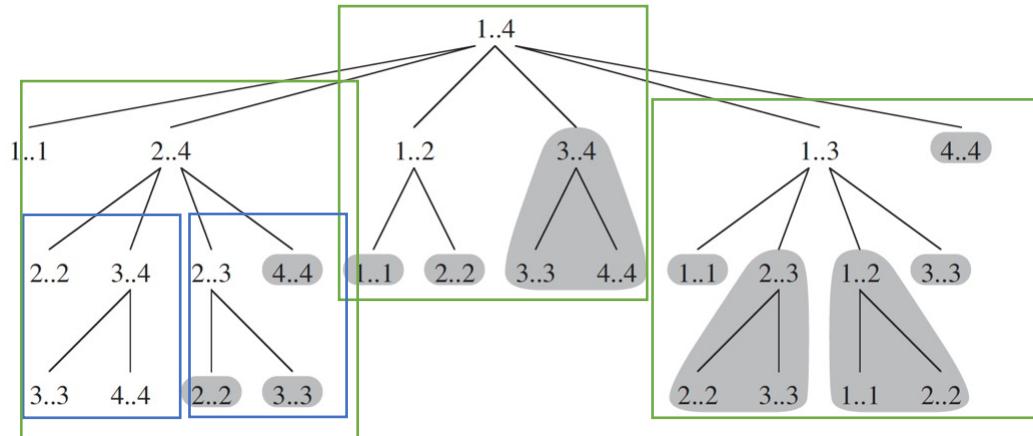
MEMOIZED-MATRIX-CHAIN(p)

```
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  be a new table
3  for  $i = 1$  to  $n$ 
4      for  $j = i$  to  $n$ 
5           $m[i, j] = \infty$ 
6  return LOOKUP-CHAIN( $m, p, 1, n$ )
```

LOOKUP-CHAIN(m, p, i, j)

```
1  if  $m[i, j] < \infty$ 
2      return  $m[i, j]$ 
3  if  $i == j$ 
4       $m[i, j] = 0$ 
5  else for  $k = i$  to  $j - 1$ 
6       $q = \text{LOOKUP-CHAIN}(m, p, i, k)$ 
        +  $\text{LOOKUP-CHAIN}(m, p, k + 1, j) + p_{i-1} p_k p_j$ 
7      if  $q < m[i, j]$ 
8           $m[i, j] = q$ 
9  return  $m[i, j]$ 
```

阴影子树表示查表获得而未通过递归方式计算的子问题



小结

- 1) 若所有子问题须至少解一次，自底向上的动态规划时间常数因子较优(不需要递归开销，维护表的开销较小)
- 2) 若子问题空间有些不需要计算，则备忘型递归具有只需计算需要的子问题的优点。

结论：当自然递归算法是指数阶，但实际不同的子问题数目是多项式阶时，可用动态规划来获得高效算法

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，通过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.

- 矩阵链乘问题属于**突出阶段性的动态规划**实例，每一个阶段都要在全面考虑各种情况下，做出必要的决策
- 下面的例子是另一类动态规划问题，设计角度是从**递推**角度出发，设计过程不太强调阶段性，只需要找出大规模问题与小规模问题（子问题）之间的递推关系，当然每一个子问题是一个比原问题简单的优化问题。

15.4 最长公共子序列

一个应用背景：基因序列比对。

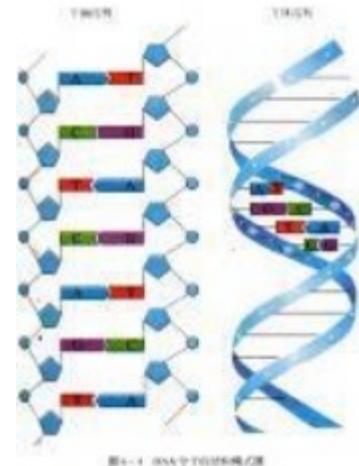
DNA(Deoxyribonucleic Acid , 脱 氧 核 糖 核 酸) 是 染 色 体 的 主 要 组 成 成 分 。 DNA 又 是 由 腺 嘌 呴 (adenine) 、 鸟 嘌 呴 (guanine) 、 胞 嘧 呔 (cytosine) 、 胸 腺 嘧 呔 (thymine) 等 四 种 碱 基 分 子 (canonical bases) 组 成 。

用它们的英文单词的首字母 A 、 C 、 G 、 T 来 代 表 这 四 种 碱 基 ， 这 样 一 个 DNA 被 表 示 为 有 穷 字 符 集 {A,C,G,T} 上 的 一 个 串 。

如：两个有机体的DNA分别为

$S_1 = \text{ACCGGTCGAGTGC}GCGGAAGGCCGGCCGAA$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$



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数量小，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

$S_3=GTGTCGGAAGCCGGCGAA$ 。

$S_1=ACCG\textcolor{red}{GTCGAGTGCGCGAAGCCGGCGAA}$

$S_2=\textcolor{red}{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

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， $LCS=\emptyset$ ；
- 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              $b[i, j] = “↖”$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = “↑”$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = “←”$ 
18 return  $c$  and  $b$ 
```

LCS-LENGTH的时间复杂度为 $O(mn)$

例，下图给出了在 $X=<A,B,C,B,D,A,B>$ 和 $Y=<B,D,C,A,B,A>$ 上运行LCSLENGTH计算出的表。

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	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 ←3
5	D	0	1	2	2	3	3
6	(A)	0	1	2	2	3	④
7	B	0	1	2	2	3	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， $c[7, 6] = 4$ ，
 $LCS(X, Y) = <B, C, B, A>$

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LCSLENGTH计算出的表。

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

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

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

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

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LCSLENGTH计算出的表。

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

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

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

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

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

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

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

```
else if  $c[i - 1, j] \geq c[i, j - 1]$ 
     $c[i, j] = c[i - 1, j]$ 
     $b[i, j] = "\uparrow"$ 
```

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LCSLENGTH计算出的表。

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

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

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

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

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

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

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

```
else if  $c[i - 1, j] \geq c[i, j - 1]$ 
     $c[i, j] = c[i - 1, j]$ 
     $b[i, j] = "\uparrow"$ 
```

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LCSLENGTH计算出的表。

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

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

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

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

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

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

$$b[1, 3] = ' \uparrow '$$

```
else if c[i - 1, j] ≥ c[i, j - 1]
    c[i, j] = c[i - 1, j]
    b[i, j] = "↑"
```

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LCSLENGTH计算出的表。

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

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

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

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

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

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

$$b[1, 4] = '↖'$$

if $x_i == y_j$
 $c[i, j] = c[i - 1, j - 1] + 1$
 $b[i, j] = "↖"$

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LCSLENGTH计算出的表。

j	0	1	2	3	4	5	6	
i	y_j	B	D	C	A	B	A	
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

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

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

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

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

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

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

else $c[i, j] = c[i, j - 1]$
 $b[i, j] = " \leftarrow "$

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LCSLENGTH计算出的表。

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

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

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

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

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

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

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

```

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LCSLENGTH计算出的表。

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

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

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

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

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

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

$$b[2, 1] = '↖'$$

```

if  $x_i == y_j$ 
     $c[i, j] = c[i - 1, j - 1] + 1$ 
     $b[i, j] = "↖"$ 

```

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LCSLENGTH计算出的表。

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

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

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

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

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

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

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

```

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

```

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  $b[i, j] == "\backslash"$ 
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( $b, X, i, j - 1$ )
```

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)
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	$\leftarrow 1$	1
2	(B)	0	①	$\leftarrow 1$	$\leftarrow 1$	1	2	$\leftarrow 2$
3	(C)	0	1	1	②	$\leftarrow 2$	2	2
4	(B)	0	1	1	2	2	③	$\leftarrow 3$
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

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

- **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

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。

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 ← T

while i ≠ 0 do

case

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

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

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

endcase

repeat

end SEARCH

(2) 最优二叉搜索树

给定一个n个关键字的已排序的序列 $K = \langle k_1, k_2, \dots, k_n \rangle$ (不失一般性 , 设 $k_1 < k_2 < \dots < k_n$) , 对每个关键字 k_i , 都有一个概率 p_i 表示其被搜索的频率。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。

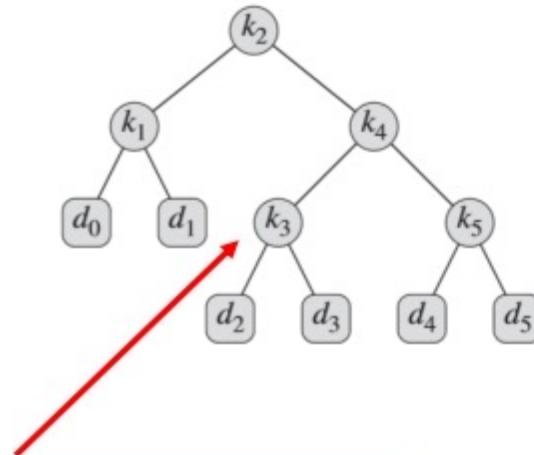
- ◆ d_0 表示所有小于 k_1 的值， d_n 表示所有大于 k_n 的值，对于 $i = 1, \dots, n-1$ ， d_i 表示所有在 k_i 和 k_{i+1} 之间的值。
- 每个 d_i 也有一个概率 q_i ，表示搜索对象x恰好落入区间 d_i 的频率。

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

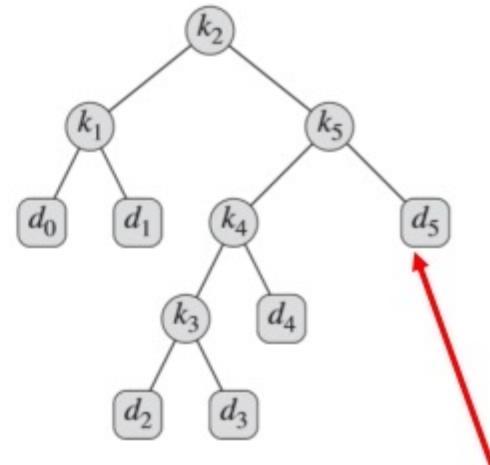
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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

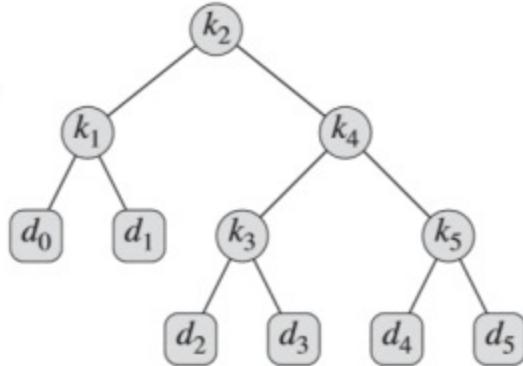
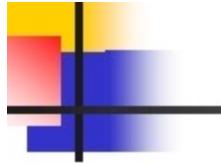
- 一次搜索的代价等于从根结点开始访问结点的数量(包括外部结点)。
 - ◆ 从根结点开始访问结点的数量等于结点在T中的深度+1；
 - ◆ 记 $\text{depth}_T(i)$ 为结点*i* 在T中的深度。
- 二叉搜索树T的期望代价为

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

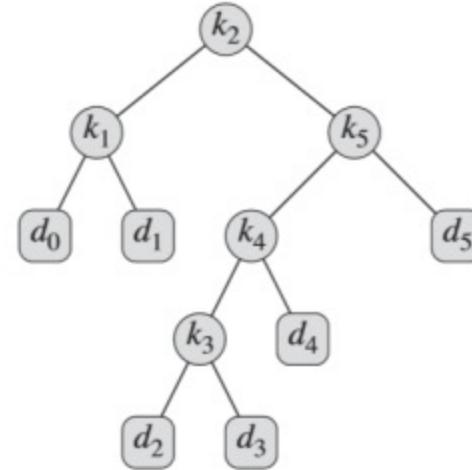
即：加权平均代价，包括所有成功搜索的结点和失败搜索的结点。

例: $n=5$,

i	0	1	2	3	4	5	$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$.
p_i							
q_i	0.05	0.10	0.05	0.05	0.05	0.10	



(a)



(b)

- (a) 的期望搜索代价为 2.80。
- (b) 的期望搜索代价为 2.75。

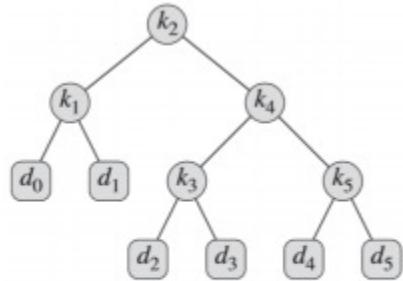
$$\begin{aligned}
 E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\
 &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i,
 \end{aligned}$$

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

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

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

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

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**
 - 最优二叉搜索树根不一定是概率最高的关键字；
 - 最优二叉搜索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_{r+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 $k_i \dots k_j$ 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 ，即包含关键字 k_i, \dots, k_j 的最优二叉搜索（子）树的根。

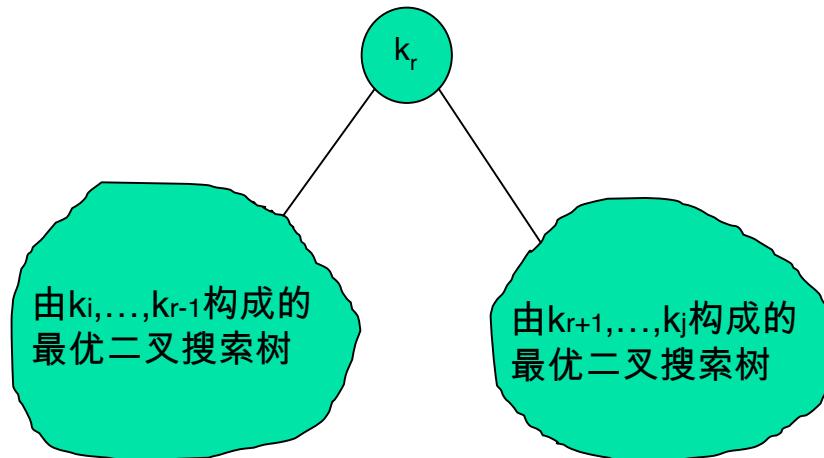
计算过程: 求解包含关键字 k_i, \dots, k_j 的最优二叉搜索树，其中
 $i \geq 1$ ， $j \leq n$ 且 $j \geq i-1$ 。

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

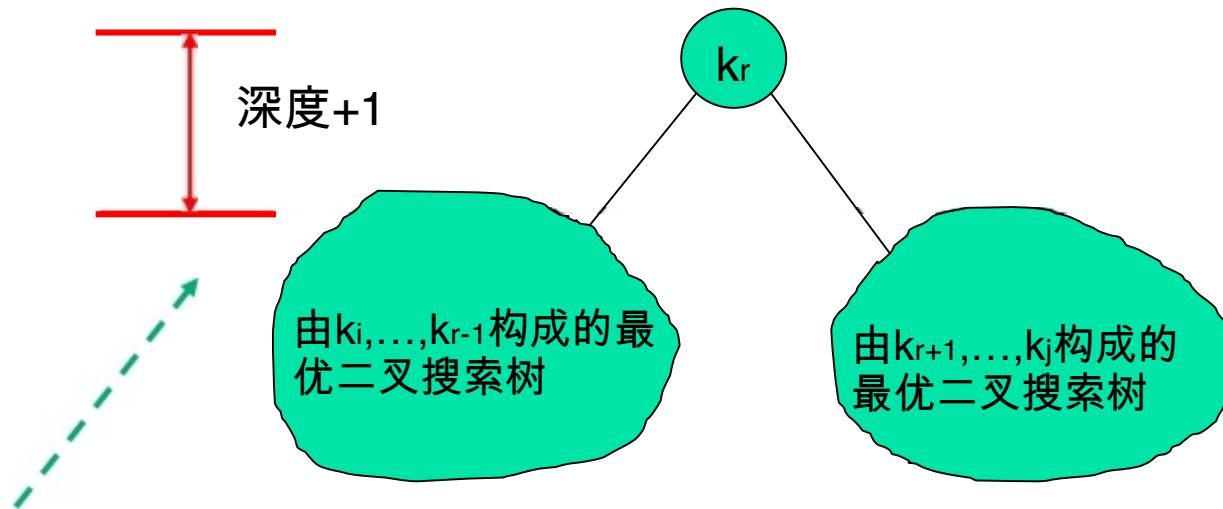
- $e[1, n]$ 为问题的最终解的期望搜索代价。

(1) 当 $j \geq i$ 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

$$(\text{包含外部结点}) : w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l .$$

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搜索代价

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。
- $\text{root}[1..n]$ ：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

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

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  $n$            ← l是区间长度  
6   for  $i = 1$  to  $n - l + 1$     ← i、j是区间下标  
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

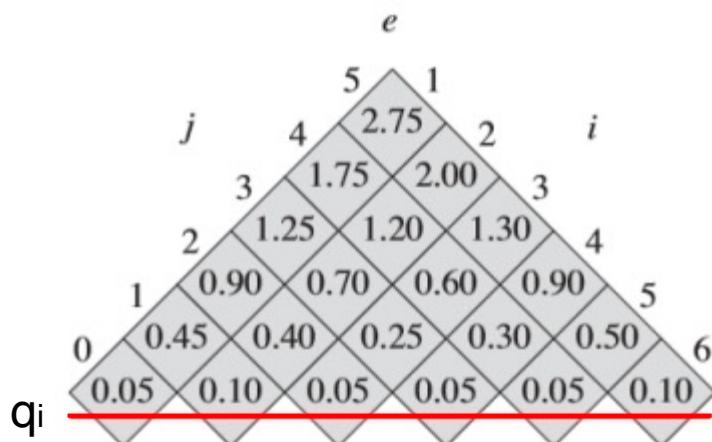
i、j是区间下标

自底向上的迭代计算

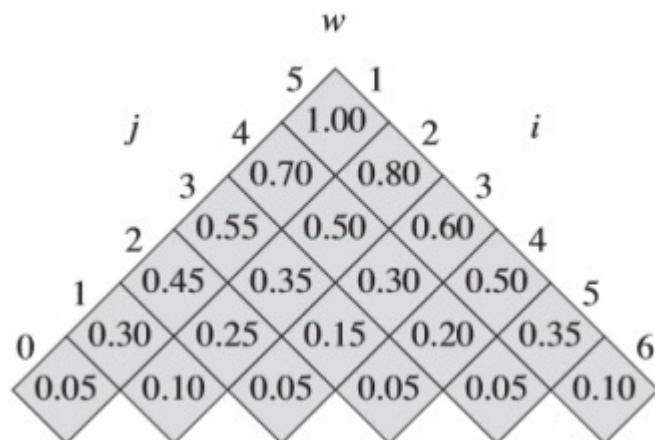
时间复杂度 $\Theta(n^3)$

■ 例: $n=5$,

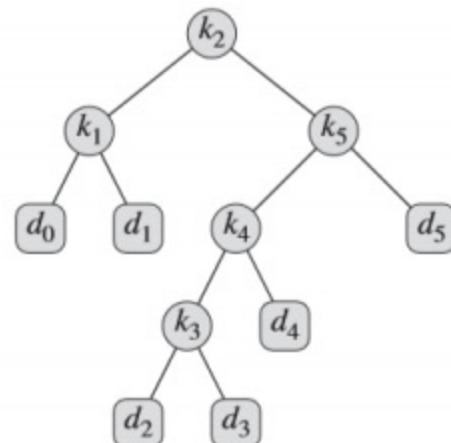
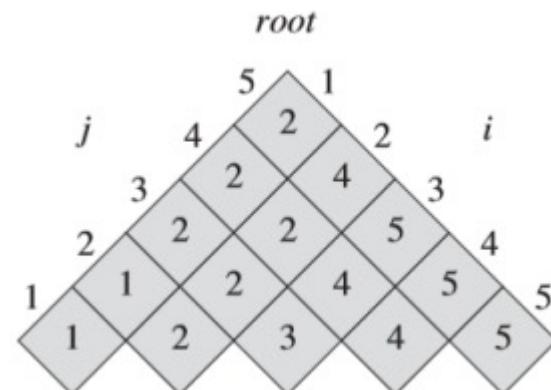
i	0	1	2	3	4	5	$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$
p_i	0.15	0.10	0.05	0.05	0.10	0.20	
q_i	0.05	0.10	0.05	0.05	0.05	0.10	



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

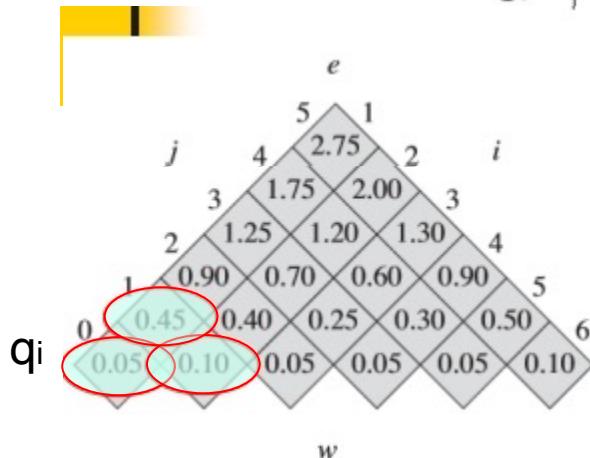


$$w[i, j] = w[i, j - 1] + p_j + q_j.$$



■ 例: $n=5$,

i	0	1	2	3	4	5	$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$
p_i	0.15	0.10	0.05	0.10	0.20		
q_i	0.05	0.10	0.05	0.05	0.05	0.10	



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

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

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

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

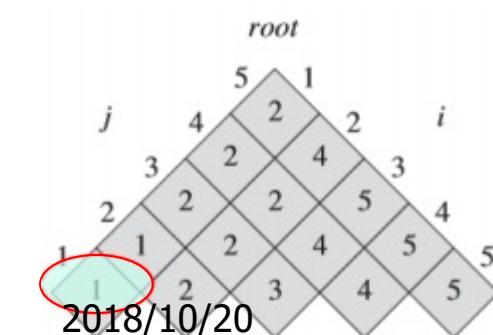
$$w[1, 0] = q_0 = 0.05$$

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

$$e[1, 1] = \min\{e[1, 0] + e[2, 1] + w[1, 1]\}$$

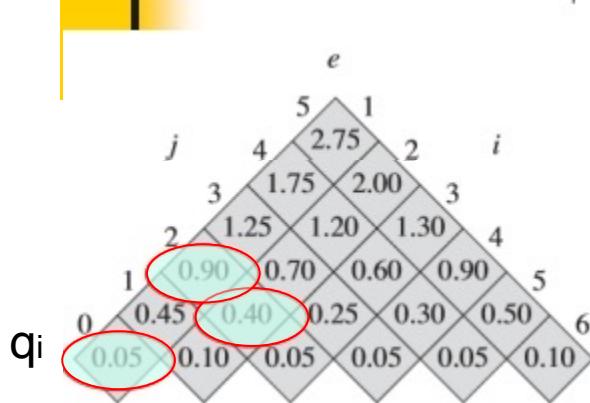
$$= 0.45$$

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



■ 例: $n=5$,

i	0	1	2	3	4	5	$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$
p_i	0.15	0.10	0.05	0.10	0.20		
q_i	0.05	0.10	0.05	0.05	0.05	0.10	



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

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

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

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

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

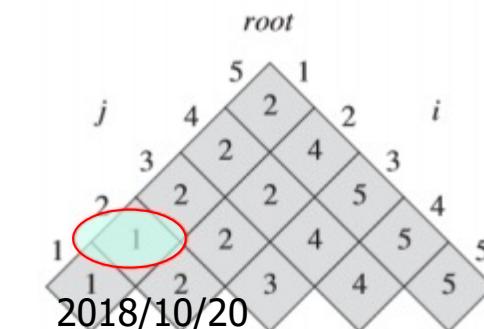
$$e[3, 2] = q_2 = 0.05$$

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

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

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

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

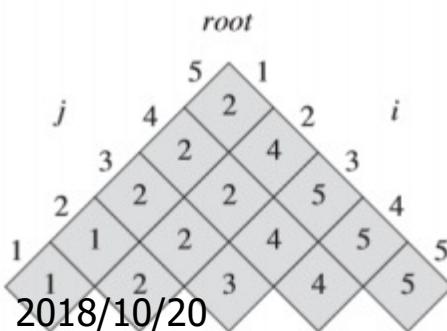
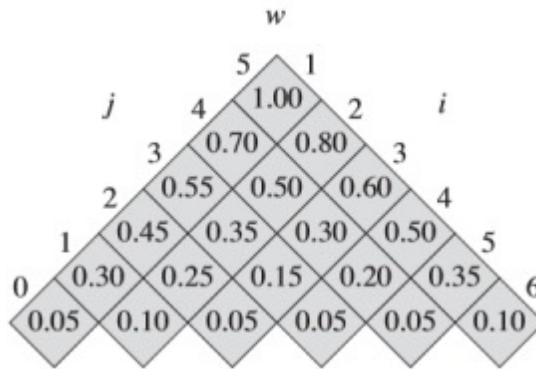
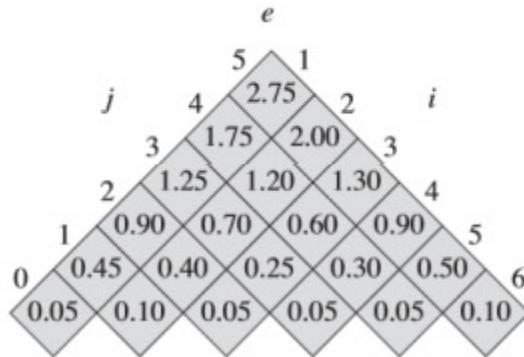


2018/10/20

$n=5$, 

i	0	1	2	3	4	5	$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$
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	

qi



2018/10/20

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

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

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



第8讲 动态规划

内容提要：

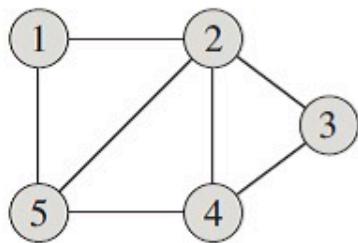
- 动态规划
- 贪心算法
- 摊还分析

第22章 基本的图算法

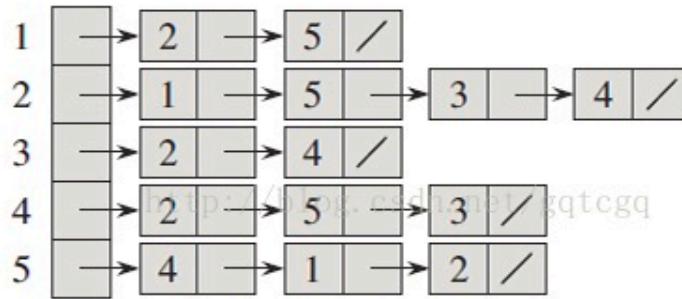
- 图的表示
- 广度优先搜索

图的表示

- 对于图 $G=(V, E)$, 可以有两种表示方法:
 - 邻接链表
 - 邻接链表适合表示稀疏图 (边的条数 $|E|$ 远远小于 $|V|^2$)
 - 本书给出的多数图算法都假定图以邻接链表的方式表示
 - 邻接矩阵。
 - 邻接矩阵适合表示稠密图 ($|E|$ 接近 $|V|^2$) ,
 - 如果需要快速判断两个节点之间是否有边相连, 也可以用邻接矩阵表示法。
- 两种表示方法既可以表示无向图, 也可以表示有向图。



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

邻接链表

- Adj 为一个包含 $|V|$ 条链表的数组，每个节点有一条链表。对于每个结点 $u \in V$ ，邻接链表 $\text{Adj}[u]$ 包含所有与结点 u 之间有边相连的结点 v 。
 - 如果 G 是一个有向图，则对于边 (u, v) 来说，节点 v 将出现在链表 $\text{Adj}[u]$ 里，所以，所有邻接链表的长度之和等于 $|E|$ 。
 - 如果 G 是一个无向图，对于边 (u, v) 来说，结点 v 将出现在链表 $\text{Adj}[u]$ 里，而且结点 u 将出现在链表 $\text{Adj}[v]$ 里，所以，所有邻接链表的长度之和等于 $2|E|$ 。
 - 不管是有向图还是无向图，邻接链表表示法的存储空间均为 $O(V+E)$
- 不足之处：无法快速判断一条边 (u, v) 是否是图中的一条边

邻接矩阵

- 对于邻接矩阵表示法，将G中的结点编为 $1, 2, \dots, |V|$ ，编号可以是任意的。进行编号之后，图G的邻接矩阵表示由一个 $|V|^2$ 的矩阵A表示，矩阵满足下面的条件：

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

- 邻接矩阵的空间需求为 $O(|V|^2)$ 。
- 无向图的邻接矩阵是一个对称矩阵。
- 邻接链表表示和邻接矩阵表示在渐近意义上至少是一样有效的，但由于邻接矩阵简单明了，因而当图较小时，更多地采用邻接矩阵来表示

表示图的属性

- 对图进行操作的多数算法需要维护图中结点或边的某些属性
 - 在伪代码中， $v.d$ 表示结点 v 的属性 d 。 $(u,v).f$ 表示边 (u,v) 的属性。
- 实现节点和边的属性
 - Eg. 邻接链表，额外的数组来表示结点属性， Adj 数组相对应的数组 $d[1..|V|]$

图的搜索

如何遍历一张图？

- 广度优先搜索
 - 许多重要图算法的原型。
 - Prim最小生成树（23.2节）、Dijkstra的单源最短路径算法（24.3节）
- 深度优先搜索

广度优先搜索

- 给定图 $G=(V, E)$ 和一个可以识别的源结点 s , 广度优先搜索系统地探索 G 中的边, 来发现可从 s 到达的所有顶点。
 - 能够计算从源结点 s 到每个可到达的结点的最短距离。
 - 同时还能生成一棵根为 s , 且包括所有 s 的可达顶点的“广度优先搜索树”。
 - 对于每个源结点 s 可以到达的结点 v , 在广度优先搜索树里从节点 s 到结点 v 的简单路径所对应的就是图 G 中从节点 s 到结点 v 的最短路径,
 - 该算法既可以用于有向图, 也可用于无向图。

广度优先搜索

- 始终是将已发现和未发现顶点之间的边界，沿其广度方向向外扩展。
 - 算法首先会发现和 s 距离为 k 的所有顶点，然后才会发现和 s 距离为 $k+1$ 的其他顶点。
- 在概念上将每个结点涂上白色，灰色或黑色。
 - 所有结点一开始均为白色，
 - 在算法推进过程中，这些结点可能会变为灰色或黑色（被“发现”的节点）。
 - 灰色结点代表的是已知和未知两个集合之间的边界。
- 执行广度优先搜索的过程中将构造出一颗广度优先树
 - 一开始仅有根结点 s
 - 在扫描发现结点 u 的邻接链表时，每当发现一个白色结点 v ，就将结点 v 和边 (u,v) 加入树中
 - u 是结点 v 的前驱或父亲结点
 - 祖先和后代关系皆以相对于根结点 s 的位置来进行定义

BFS(breadth-first search)搜索过程

- 假定输入图 $G=(V,E)$ 是以邻接链表所表示的，包含的性质如下：

- $u.\text{color}$ 表示结点 u 的颜色，
- $u.\pi$ 表示在广度优先树中 u 的前驱节点
- $u.d$ 表示从源结点 s 到结点 u 的距离
- 使用先进先出的队列来管理灰色结点集合

BFS(G, s)

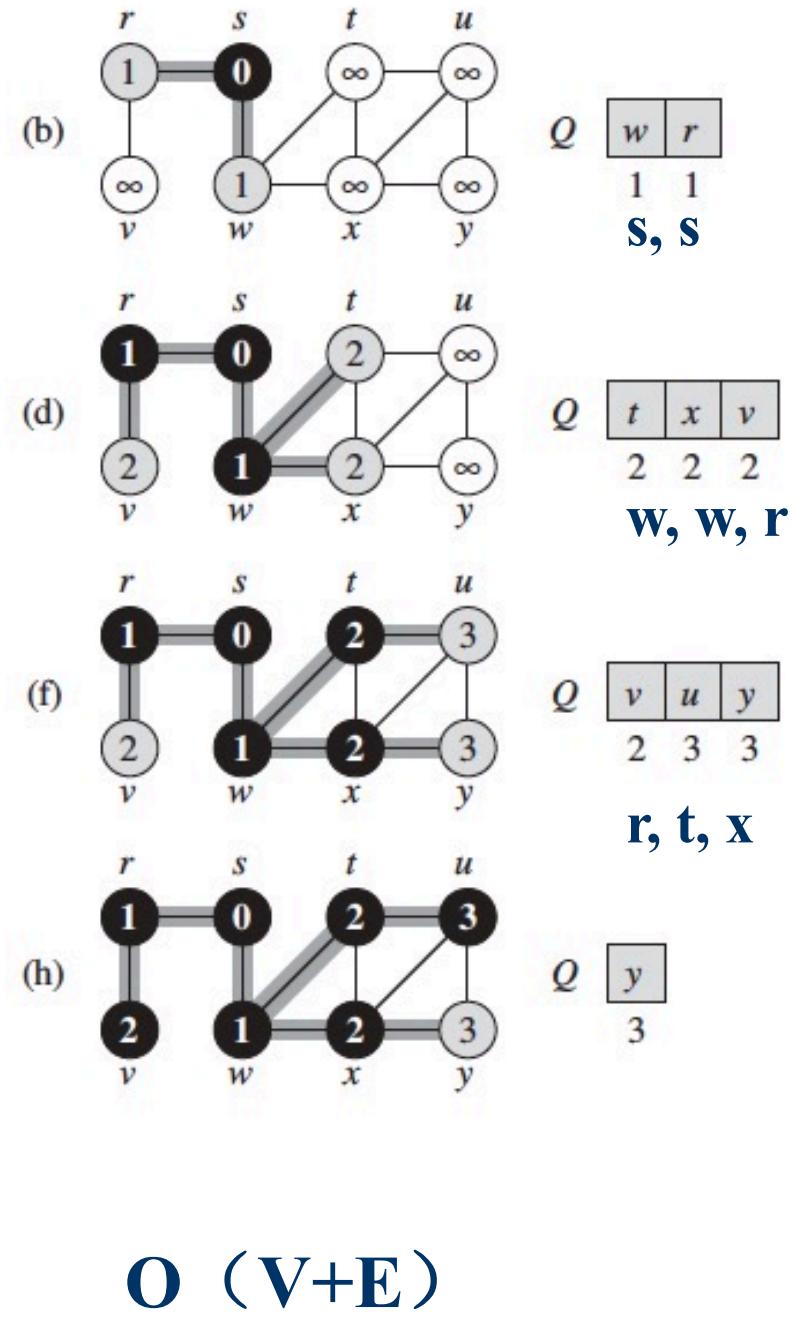
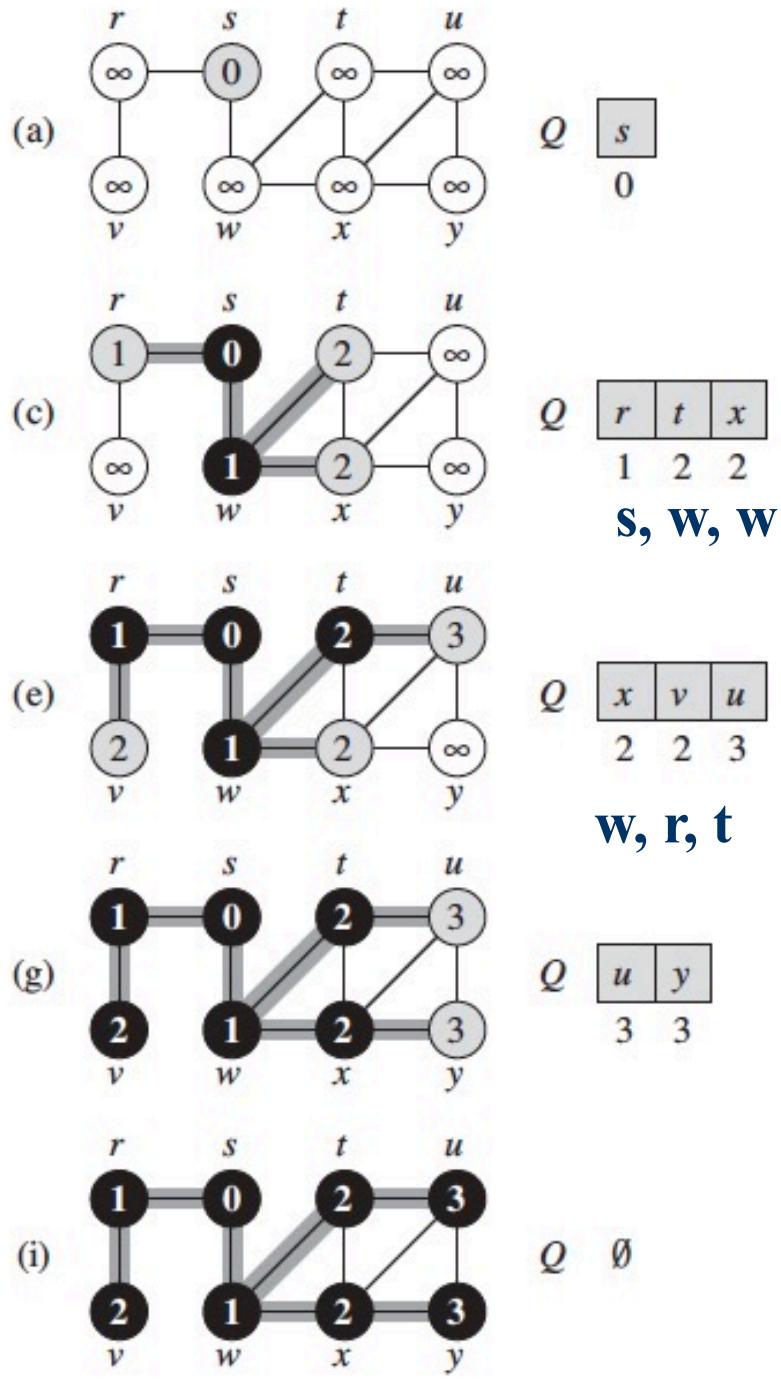
```
1   for each vertex  $u \in G.V - \{s\}$ 
2      $u.\text{color} = \text{WHITE}$ 
3      $u.d = \infty$ 
4      $u.\pi = \text{NIL}$ 
5      $s.\text{color} = \text{GRAY}$ 
6      $s.d = 0$ 
7      $s.\pi = \text{NIL}$ 
8      $Q = \emptyset$ 
9      $Q = \text{ENQUEUE}(Q, s)$ 
10    while  $Q \neq \emptyset$ 
11       $u = \text{DEQUEUE}(Q)$ 
12      for each  $v \in G.\text{Adj}[u]$ 
13        if  $v.\text{color} == \text{WHITE}$ 
14           $v.\text{color} = \text{GRAY}$ 
15           $v.d = u.d + 1$ 
16           $v.\pi = u$ 
17           $\text{ENQUEUE}(Q, v)$ 
18       $u.\text{color} = \text{BLACK}$ 
```

Q包含的是灰色节结点集合

对结点 u 的邻接链表中的每个结点 v 进行考察
，然后将 u 涂黑

结点涂灰色的同时被加入队列Q中

O (V+E)



最短路径

- 广度优先搜索能够找出从源结点 $s \in V$ 到所有可以到达的结点的距离。
 - 定义从源结点 s 到结点 v 的最短路径距离 $\delta(s, v)$ 为从结点 s 到结点 v 之间所有路径里面最少的边数。
 - 如果没有路径 $\delta(s, v) = \infty$ 。
 - 我们称从结点 s 到结点 v 的长度为 $\delta(s, v)$ 的路径为 s 到 v 的最短路径。

广度优先搜索的正确性

- 引理22.5设 $G = (V, E)$ 为一个有向图或无向图，又假设BFS以 s 为结点在图 G 上运行。那么在算法执行过程中，BFS将发现从源结点 s 可以到达的所有结点 $v \in V$ ，并在算法终止时，对于所有的 $v \in V, v.d = \delta(s, v)$ 。而且，对于任意可以从 s 到达的结点 $v \neq s$ ，从源结点 s 到结点 v 的其中一条最短路径为从结点 s 到结点 $v.\pi$ 的最短路径再加上边 $(v.\pi, v)$ 。

广度优先搜索树

- 广度优先搜索算法可以得到从源顶点 s 到所有可达顶点的最短距离 $v.d$ 。同时，该算法能够构造出广度优先树。下面的代码为打印一棵广度优先树中，源节点 s 到任意结点 v 的路径。
- 引理22.6 当运行在一个有向或无向图 $G=(V,E)$ 上时，BFS过程所建造出来的 π 属性使得前驱子图 $G_\pi=(V_\pi, E_\pi)$ 成为一棵广度优先树。

PRINT-PATH(G, s, v)

if $v == s$

print s

else if $v.\pi == \text{NIL}$

print “nopath from” s “to” v “exists”

else PRINT-PATH($G, s, v.\pi$)

print v

第25章 所有节点对的最短路径问题

- 给定的有向加权图 $G=(V, E)$ ，对于所有的节点 $u, v \in V$ ，其中权重函数为 $w: E \rightarrow \mathbb{R}$ ，该函数将边映射到实数值的权重上。希望找到一条从节点 u 到节点 v 的最短路径，其中路径的权重为组成该路径的所有边的权重之和。希望以表格的形式表示输出：第 u 行第 v 列给出的是节点 u 到节点 v 的最短路径权重。

解决方法

- 对于这个问题，如果是运行 $|V|$ 次单源最短路径算法来解决所有节点对的最短路径问题，每一次使用一个不同的节点做为源节点。
- 如果所有边的权值是非负的，可以采用Dijkstra算法。
 - 如果采用数组来实现最小优先队列，算法的运行时间为 $O(V^3 + VE) = O(V^3)$ 。
 - 使用二叉堆实现的最小优先队列将使算法的运行时间降低到 $O(VE \lg V)$ ，这个时间在稀疏图的情况下有很大改进，因为稀疏图 $E < V^2$ 。
 - 如果采用斐波那契堆来实现最小优先队列，其算法运行时间为 $O(V^2 \lg V + VE)$ 。
- 如果图中有权重为负值的边，就必须采用效率更低的Bellman-Ford算法，这样的运行时间将使 $O(V^2 E)$ ，在稠密图的情况下，该运行时间为 $O(V^4)$ 。
- 动态规划算法将能做到更好

邻接矩阵

- 本章的多数算法采用邻接矩阵表示图，假定节点的编号为 $1, 2, \dots, |V|$ ，因此，算法的输入是 $n \times n$ 的矩阵 W ，该矩阵代表一个有 n 个节点的有向图的边的权重：

$$w_{ij} = \begin{cases} 0 & \text{if } i = j , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E . \end{cases}$$

- 算法的输出也是一个 $n \times n$ 的矩阵 $D = (d_{ij})$ 。其中 $d_{ij} = \delta(i, j)$ 。

前驱结点矩阵

- 为了解决所有顶点间最短路径问题，不仅要算出最短路径的权值，而且要计算出一个前驱节点矩阵 $\Pi = (\pi_{ij})$
 - 其中 π_{ij} 在*i=j*或从*i*到*j*没有通路时为NIL，
 - 其他情况下 π_{ij} 表示从*i*到*j*的某条最短路径上*j*的前驱顶点。由 Π 矩阵的第*i*行导出的子图应是根节点为*i*的一棵最短路径树。
- 下面的算法将打印出从节点*i*到节点*j*的一条最短路径，该算法类似于22章的PRINT-PATH过程：

```
PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi$ , i, j)
    if i == j
        print i
    else if  $\pi_{ij}$  == NIL
        print “no path from i to j exists”
    else PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi$ , i,  $\pi_{ij}$ )
        print j
```

最短路径和矩阵乘法

- 动态规划算法的步骤是：
 - 分析最优解的结构，
 - 递归定义最优解的值，
 - 自底向上计算最优解的值。
- 本章第一个是利用传统的动态规划
- 本章第二个也是个动态规划算法，但是基于一种观察结果，他就是warshall算法
- 本章第三个算法是将问题转换为没有负数权重的图，再对每个节点调用Dijkstra算法，他就是Johnson算法

步骤1:最短路径的结构

引理 24.1(最短路径的子路径也是最短路径) 给定带权重的有向图 $G=(V, E)$ 和权重函数 $w: E \rightarrow \mathbf{R}$ 。设 $p = \langle v_0, v_1, \dots, v_k \rangle$ 为从结点 v_0 到结点 v_k 的一条最短路径，并且对于任意的 i 和 j , $0 \leq i \leq j \leq k$, 设 $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ 为路径 p 中从结点 v_i 到结点 v_j 的子路径。那么 p_{ij} 是从结点 v_i 到结点 v_j 的一条最短路径。

证明 如果将路径 p 分解为 $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, 则有 $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$ 。现在, 假设存在一条从 v_i 到 v_j 的路径 p'_{ij} , 且 $w(p'_{ij}) < w(p_{ij})$ 。则 $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ 是一条从结点 v_0 到结点 v_k 的权重为 $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ 的路径, 而该权重小于 $w(p)$ 。这与 p 是从 v_0 到 v_k 的一条最短路径这一假设相矛盾。 ■

步骤1:最短路径的结构

- 引理24.1已经证明一条最短路径的所有子路径都是最短路径。
- 考虑从*i*到*j*的一条最短路径*p*，假定*p*最多包含*m*条边，假定没有权值为负值的环路，且*m*为有限值。
- 如果*i=j*，则*p*的权重为0且不包含任何边。
- 如果*i*和*j*不同，则将*p*分解为 $i \xrightarrow{p'} k \rightarrow j$ ，其中路径*p'*最多包含*m-1*条边。根据引理24.1，*p'*是从结点*i*到结点*k*的一条最短路径。所以 $\delta(i, j) = \delta(i, k) + w_{kj}$ 。

步骤2:所有结点对最短路径的递归解

- 设 $l_{ij}^{(m)}$ 为从i到j的最多包含m条边的任意路径中的最小权重。当m=0时，从i到j之间存在一条没有边的最短路径当且仅当i=j。所以：

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

- 对于m≥1，我们需要计算的 $l_{ij}^{(m)}$ 是
 - $l_{ij}^{(m-1)}$ （从i到j最多由m-1条边组成的最短路径的权重）的最小值和从i到j最多由m条边组成的任意路径的最小权重，我们通过对j的所有可能前驱k进行检查来获得该值。
 - 递归定义为：
$$l_{ij}^{(m)} = \min \left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \right)$$

对于所有的j有 $w_{jj}=0$,所以

$$= \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} .$$

- 如果i到j之间存在一条最短路径，则该路径为简单路径，其中包含的边最多为n-1，所以有： $\delta(i, j) = l_{ij}^{n-1} = l_{ij}^n = l_{ij}^{n+1} = \dots$
- 根据输入矩阵 $W=(w_{ij})$ ，可以计算出矩阵序列 $L^{(1)}, L^{(2)}, \dots, L^{(m)}$ 。
 - 对于 $m=1, 2, \dots, n-1$ ，有 $L^{(m)}=(l_{ij}^{(m)})$ 。
 - 最后的矩阵 $L^{(n-1)}$ 包含的是最短路径的实际权重。
 - $L^{(1)}=(w_{ij})$ ，因此， $L^{(1)}=W$ 。

步骤三 自底向上计算最短路径权重

- 下面算法假定已经知道了 W 和 L^{m-1} 的情况下，给出如何计算 L^m ，其中， L 表示 L^{m-1} ， L' 表示 L^m ，该算法的时间复杂度为 $O(n^3)$

EXTEND-SHORTEST-PATHS(L, W)

```
1   $n = L.\text{rows}$ 
2  let  $L' = (l'_{ij})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4    for  $j = 1$  to  $n$ 
5       $l'_{ij} = \infty$ 
6      for  $k = 1$  to  $n$ 
7         $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$ 
8  return  $L'$ 
```

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.\text{rows}$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4    for  $j = 1$  to  $n$ 
5       $c_{ij} = 0$ 
6      for  $k = 1$  to  $n$ 
7         $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

最短路径和矩阵乘法

- 现在可以看到上面的算法与矩阵乘法的关系了。假定希望计算矩阵乘积 $C = A * B$, 对于 $i, j = 1, 2, \dots, n$, 有 $c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$ 。如果在 $l_{ij}^{(m)} = \min_{1 \leq k \leq n} (l_{ij}^{m-1} + w_{kj})$ 中做出下面的替换:

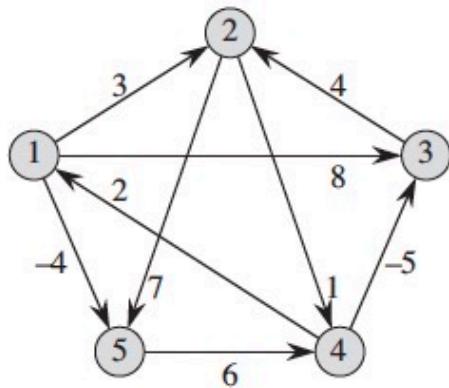
$$\begin{aligned} l^{(m-1)} &\rightarrow a, \\ w &\rightarrow b, \\ l^{(m)} &\rightarrow c \\ \min &\rightarrow +, \\ + &\rightarrow \cdot \end{aligned}$$

- 就可以得到 $c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$ 。因此, 如果对算法 EXTEND-SHORTEST-PATHS 做出上面的替换, 就可以得到标准的矩阵乘法的算法。

- 设 A^*B 表示算法EXTEND-SHORTEST-PATHS(A, B)所返回的矩阵“乘积”，我们可以计算出下面由 $n-1$ 个矩阵所构成的矩阵序列（如左图）。
- 矩阵 $L(n-1)=w_{n-1}$ 包含的是最短路径权重，因此下面伪代码程序在 $\theta(n^4)$ 时间内计算出该矩阵序列（如右图）。

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W = W, \\ L^{(2)} &= L^{(1)} \cdot W = W^2, \\ L^{(3)} &= L^{(2)} \cdot W = W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1}. \end{aligned}$$

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
 $n = W.\text{rows}$
 $L^1 = W$
for $m = 2$ **to** $n-1$
 let L^m **be a new** $n*n$ **matrix**
 $L^m = \text{EXTEND-SHORTEST-PATHS}(L^{m-1}, W)$
return L^{n-1}



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & 7 \\ 3 & \infty & 4 & 0 & 5 & 11 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

一个有向图和由**SLOW-ALL-PAIRS-SHORTEST-PATHS**所计算出的矩阵序列 $L(m)$ 。可以自行验证 $L^{(5)}=L^{(4)}$ ，因此，对于所有的 $m \geq 4$ ，有 $L^{(m)}=L^{(4)}$

改进算法的运行时间

- 我们的目标并不是要计算所有的 L^m 矩阵，我们感兴趣的仅仅是矩阵 L^{n-1} 。因为有：

$$\delta(i, j) = l_{ij}^{n-1} = l_{ij}^n = l_{ij}^{n+1} = \dots$$

- 正如传统的矩阵乘法是相关的，所以由EXTEND-SHORTEST-PATHS过程所定义的算法也是相关的。所以可以采用重复平方的技术计算该矩阵序列，该算法的时间复杂度可以减少为 $O(n^3 \log n)$ ；

$$\begin{aligned} L^{(1)} &= W, \\ L^{(2)} &= W^2 = W \cdot W, \\ L^{(4)} &= W^4 = W^2 \cdot W^2 \\ L^{(8)} &= W^8 = W^4 \cdot W^4, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil}-1} \cdot W^{2^{\lceil \lg(n-1) \rceil}} \end{aligned}$$

由于 $2^{\lceil \lg n \rceil - 1} \geq n - 1$ ，最后的乘积 $L^{(2^{\lceil \lg(n-1) \rceil})}$ 等于 $L^{(n-1)}$

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
  n = W.rows
  L1 = W
  m = 1
  while m < n-1
    let L2m be a new n*n matrix
    L2m = EXTEND-SHORTEST-PATHS(Lm, Lm)
    m = 2m
  return Lm
```

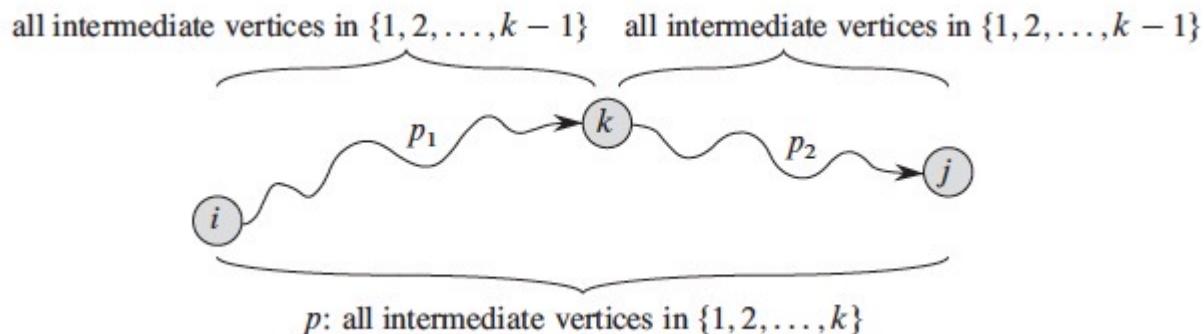
Note: $n - 1 \leq 2m < 2n - 2$

Floyd-Warshall 算法

- Floyd-Warshall算法采用不同的动态规划公式解决所有节点对的最短路径问题，**它的运行时间是 $O(V^3)$** ，该算法同样假设：可以存在负权重的边，但不能存在权重为负值的环路。
- Floyd-Warshall算法考虑一条最短路径上的中间节点，这里简单路径 $p = \langle v_1, v_2, \dots, v_l \rangle$ 上的中间节点指的是路径 p 上除 v_1 和 v_l 之外的任意结点。

步骤一：最短路径结构

- 假定图G的所有节点 $V=\{1, 2, \dots, n\}$, 考虑其中的子集 $\{1, 2, \dots, k\}$ 。对于任意的结点 $i, j \in V$, 考虑从*i*到*j*的所有中间节点均取自集合 $\{1, 2, \dots, k\}$ 的路径。并设 p 为其中的最短路径, 分两种情况讨论:
- a: 如果结点*k*不是 p 上的中间节点, 则路径 p 上的所有中间节点都属于集合 $\{1, 2, \dots, k-1\}$ 。因此, 从*i*到*j*的中间节点均取自集合 $\{1, 2, \dots, k-1\}$ 的一条最短路径, 同样也是从*i*到*j*的中间节点均取自集合 $\{1, 2, \dots, k\}$ 的一条最短路径。
- b: 如果结点*k*是路径 p 上的中间节点, 则 p 可以分解为 $i \xrightarrow{p_1} k \xrightarrow{p_2} j$, 所以, p_1 上的所有中间节点都属于集合 $\{1, 2, \dots, k-1\}$, 同理 p_2 也是。



步骤二：所有结点对最短路径问题 的一个递归解

- 根据上面的观察，设 $d_{ij}^{(k)}$ 为从i到j的所有中间节点全部取自集合{1, 2, ..., k}的一条最短路径的权重。
- 当k=0时，i到j的路径没有中间节点，这样的路径最多只有一条边，所以有 $d_{ij}^{(k)} = w_{ij}$ 。
- 递归式是：
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$
- 因为对于所有的路径，所有的中间节点都属于集合{1, 2, ..., n}，矩阵 $D^{(n)} = (d_{ij}^{(n)})$ 给出的就是最终的矩阵： $d_{ij}^{(n)} = \delta(i, j)$ 。

步骤三：自底向上计算最短路径权重

FLOYD-WARSHALL(W)

```
1   $n = W.\text{rows}$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5      for  $i = 1$  to  $n$ 
6          for  $j = 1$  to  $n$ 
7               $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 
```

运行时间由算法3~7行的3层嵌套的**for**循环所决定。因为第7行的每次执行时间是 $O(1)$ ，因此该算法的运行时间为 $\theta(n^3)$

步骤四：构建一条最短路径

- 计算 D^k 的同时计算前驱矩阵 Π
- 需要计算一个矩阵序列： $\Pi^0, \Pi^1, \dots, \Pi^n$ ，这里 $\Pi = \Pi^n$ ，
- 定义 π_{ij}^k 为从i到j的一条所有中间节点都取自集合{1,2,...,k}的最短路径上，j的前驱结点。
- 所以，可以给出 π_{ij}^k 的一个递归公式，当 $k=0$ 时，从i到j的一条最短路径上没有中间节点，所以：

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ \text{http://baidu.com} & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases}$$

- 如果 $k >= 1$ ，则有

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

在计算 $D^{(n=k)}$ 的同时，计算前驱矩阵 Π^k

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

<http://blog.csdn.net/gqtgcg>

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

0-1背包问题

- 给定n种物品和一背包。物品i的重量是 w_i ，其价值为 v_i ，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？
- ② 0-1背包问题是一个特殊的整数规划问题。

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i \\ s.t \quad & \left\{ \begin{array}{l} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{array} \right. \end{aligned}$$

步骤一 子结构

- 设所给0-1背包问题的子问题

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i \\ s.t \quad & \begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases} \end{aligned}$$

- 设最优值 $m(i,j)$ 为背包容量为 j 、可选择物品为 $1, 2, \dots, i$ 时的最优值(装入包的最大价值)。所以原问题的解为 $m(n, C)$
- 将原问题分解为其子结构来求解。要求原问题的解 $m(n, C)$ ，可从 $m(1, C), m(2, C), m(3, C) \dots$ 来依次求解，即可装包物品分别为(物品1)、(物品1, 2)、(物品1, 2, 3)、.....、(物品1, 物品2, 物品 $n-1$, 物品 n)。最后求出的值即为最优值 $m(n, C)$ 。)

步骤二 递归解

- 若求 $m(i,j)$, 此时已经求出 $m(i-1,j)$, 即第*i*个物品放入和不放入时这二者最大值。对于此时背包剩余容量 $j=0,1,2,3\dots\dots C$, 分两种情况:
 - (1)当 $w[i] > j$, 即第*i*个物品重量大于背包容量j时, $m(i,j)=m(i-1,j)$
 - (2)当 $w[i] \leq j$, 即第*i*个物品重量不大于背包容量j时, 这时要判断物品*i*放入和不放入对*m*的影响。
 - 若不放入物品*i*, 则此时 $m(i,j)=m(i-1,j)$
 - 若放入物品*i*, 此时背包剩余容量为 $j-w[i]$, 在子结构中已求出当容量 $k=0,1,2\dots\dots C$ 时的最优值 $m(i-1,k)$ 。所以此时 $m(i,j)=m(i-1,j-w[i])+v[i]$ 。
 - 取上述二者的最大值, 即 $m(i,j) = \max\{ m(i-1,j), m(i-1,j-w[i])+v[i] \}$
- 总结得出状态转移方程为:

$$m(i,j) = \begin{cases} \max\{m(i-1,j), m(i-1,j-w_i) + v_i\} & j \geq w_i \\ m(i-1,j) & 0 \leq j < w_i \end{cases}$$
$$m(1,j) = \begin{cases} v_1 & j \geq w_1 \\ 0 & 0 \leq j < w_1 \end{cases}$$

15.5 0-1背包问题

设所给0-1背包问题的子问题

$$\begin{aligned} & \max \sum_{k=i}^n v_k x_k \\ & \left\{ \begin{array}{l} \sum_{k=i}^n w_k x_k \leq j \\ x_k \in \{0,1\}, i \leq k \leq n \end{array} \right. \end{aligned}$$

的最优值为 $m(i, j)$, 即 $m(i, j)$ 是背包容量为 j , 可选择物品为 $i, i+1, \dots, n$ 时 0-1 背包问题的最优值。由 0-1 背包问题的最优子结构性质, 可以建立计算 $m(i, j)$ 的递归式如下。

$$m(i, j) = \begin{cases} \max \{m(i+1, j), m(i+1, j - w_i) + v_i\} & j \geq w_i \\ m(i+1, j) & 0 \leq j < w_i \end{cases}$$

66

$$m(n, j) = \begin{cases} v_n & j \geq w_n \\ 0 & 0 \leq j < w_n \end{cases}$$

步骤三 自底向上计算 $m[n, w]$

- 伪代码

```
m[0,j]=0  
m[i,0]=0  
for i=1 to n  
    for j=1 to C  
        if j<wi  
            m[i,j]=m[i-1,j]  
        else  
            m[i,j]=max{m[i-1,j],m[i-1,j-wi]+vi}  
return m[n,w]
```

- 算法复杂度分析：

- 从 $m(i, j)$ 的递归式容易看出，算法需要 $O(nc)$ 计算时间。当背包容量 c 很大时，算法需要的计算时间较多。例如，当 $c > 2^n$ 时，算法需要 $\Omega(n2^n)$ 计算时间。

- 为方便讲解和理解，下面讲述的例子均先用具体的数字代入，即： eg:
number=4, capacity=8

i (物品编号)		1	2	3	4			
w (体积)		2	3	4	5			
v (价值)		3	4	5	6			
i/j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7
3	0	0	3	4	5	7	8	9
4	0	0	3	4	5	7	8	9

如， $i=1, j=1, w(1)=2, v(1)=3$ ， 有 $j < w(1)$ ， 故 $V(1,1)=V(1-1,1)=0$ ；
 $i=1, j=2, w(1)=2, v(1)=3$ ， 有 $j=w(1)$ ， 故 $V(1,2)=\max \{ V(1-1,2), V(1-1,2-w(1))+v(1) \} =\max \{ 0, 0+3 \} =3$ ；
如此下去，填到最后一个，
 $i=4, j=8, w(4)=5, v(4)=6$ ， 有 $j > w(4)$ ， 故 $V(4,8)=\max \{ V(4-1,8), V(4-1,8-w(4))+v(4) \} =\max \{ 9, 4+6 \} =10.....$

步骤四 背包回溯

- 根据填表的原理可以有如下的寻解方式:
- $V(i,j)=V(i-1,j)$ 时，说明没有选择第*i*个商品，则回到 $V(i-1,j)$;
- $V(i,j)=V(i-1,j-w(i))+v(i)$ 时，说明装了第*i*个商品，该商品是最优解组成的一部分，随后我们得回到装该商品之前，即回到 $V(i-1,j-w(i))$;
- 一直遍历到*i=0*结束为止，所有解的组成都会找到。

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

https://blog.csdn.net/uqq_37767455

最优解为 $V(4,8)=10$ ，而 $V(4,8) \neq V(3,8)$ 却有 $V(4,8)=V(3,8-w(4))+v(4)=V(3,3)+6=4+6=10$ ，所以第4件商品被选中，并且回到 $V(3,8-w(4))=V(3,3)$ ；
 有 $V(3,3)=V(2,3)=4$ ，所以第3件商品没被选择，回到 $V(2,3)$ ；
 而 $V(2,3) \neq V(1,3)$ 却有 $V(2,3)=V(1,3-w(2))+v(2)=V(1,0)+4=0+4=4$ ，所以第2件商品被选中，并且回到 $V(1,3-w(2))=V(1,0)$ ；
 有 $V(1,0)=V(0,0)=0$ ，所以第1件商品没被选择。

动态规划策略应用实例-0/1背包问题

一、实验目的

- 1、理解动态规划策略的基本思想
- 2、掌握动态规划策略求解问题的框架，能够运用动态规划策略求解实际应用问题
- 3、掌握动态规划策略求解问题的时间复杂度分析

二、实验内容

1. 问题描述：

给定 n 种物品和一背包。物品 i 的体积是 w_i , 其价值为 v_i , 背包的容量为 C 。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

注：物品只能选择不装或者装入背包，而不分切一小部分装入，即0或者1。因此，0-1背包问题是一个特殊的整数规划问题。

形式化描述：输入： $C > 0, w_i > 0, v_i > 0, 1 \leq i \leq n;$

输出： (x_1, x_2, \dots, x_n) , $x_i \in \{0, 1\}$, 使得 $\sum_{1 \leq i \leq n} v_i \cdot x_i$ 最大。

- (1) 描述0-1背包问题的最优子结构，并利用“剪切-粘贴”技术证明；
- (2) 利用最优子结构，写出原问题最优解的递归表达式；
- (3) 利用自底向上的方法计算最优解的值和最优解，用伪代码进行描述，并分析算法的时间复杂度；
- (4) 用高级编程语言实现(3)中的求解0-1背包问题的动态规划算法，并通过问题实例测试程序，对运行结果截图。

- 贪心策略：图算法
- 第23章：最小生成树
- 第24章 单源最短路径：Dijkstra算法
- 提交：1，报告（按照贪心算法步骤：问题描述；最优子结构，递归式，利用自底向上的方法计算最优解的值和最优解；重构最优解（选做）；案例分析）；用高级编程语言实现，并通过问题实例测试程序，对运行结果截图。
- 2，一份PPT