

算法设计与分析

主讲人：吴庭芳

Email: *tfwu@suda.edu.cn*

苏州大学 计算机学院



第十一讲 动态规划

内容提要：

- 动态规划策略概念
- 钢条切割
- 矩阵链乘法
- 动态规划原理
- 最长公共子序列



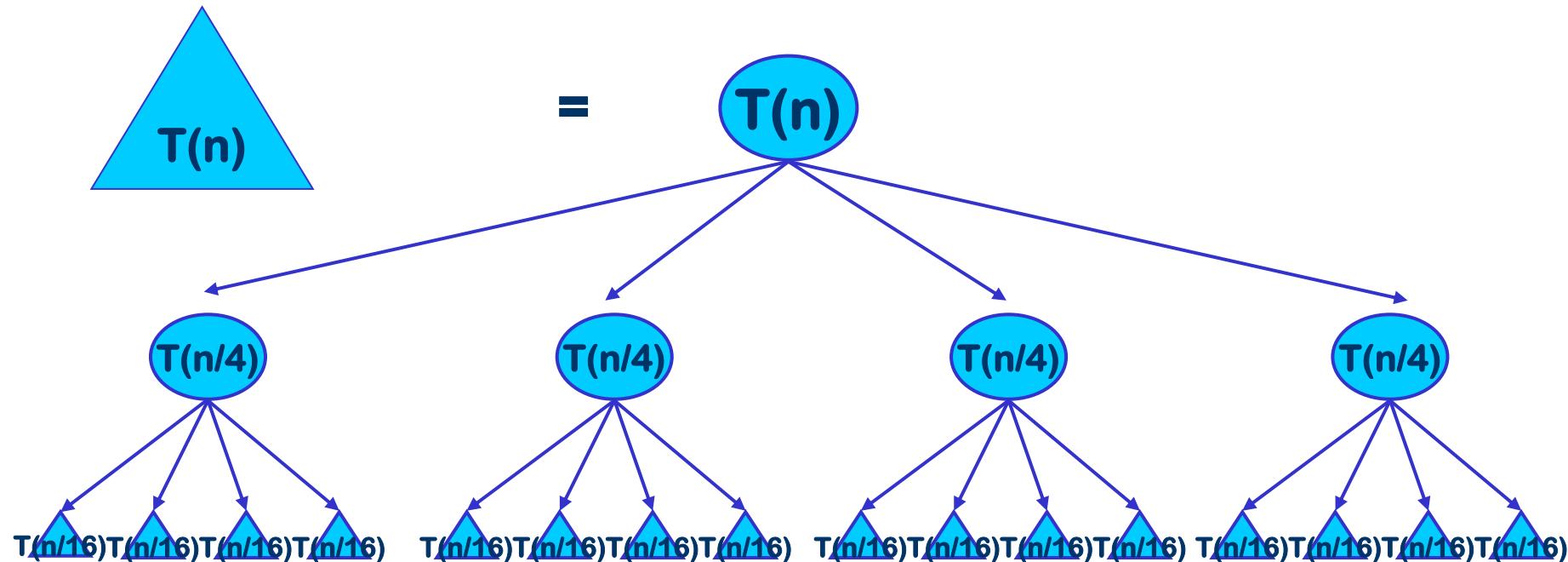
动态规划策略概念

- 动态规划 (Dynamic Programming) 是运筹学的一个分支，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 (Multistep Decision Process) 的优化问题时，提出了著名的**最优化原理**：把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划
- 动态规划是解决问题的一种策略 (方法)
- 动态规划问世以来，在最短路线、库存管理、资源分配、装载等问题得到了广泛的应用，用动态规划方法比其它优化方法求解更为优越



动态规划策略概念

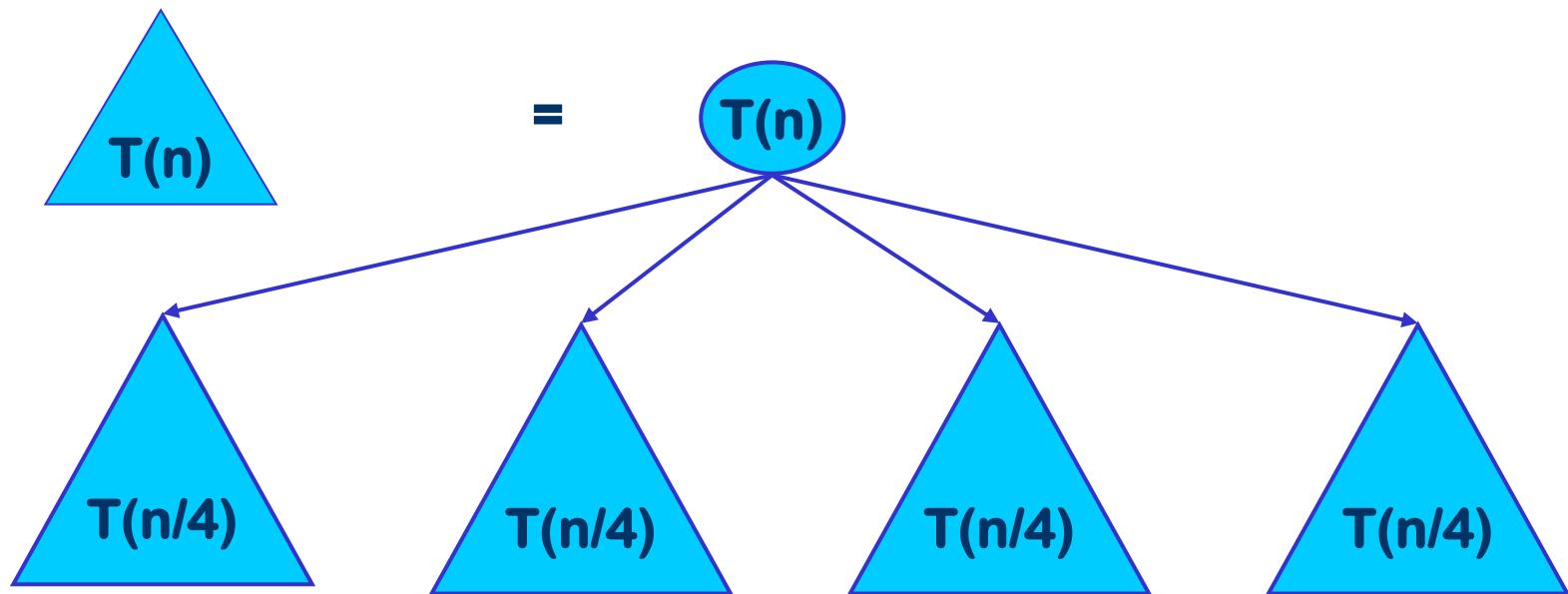
- 分治策略：首先将原问题分解为互不相交的子问题，然后递归求解这些子问题，最后将子问题的解合并成原问题的解





动态规划策略概念

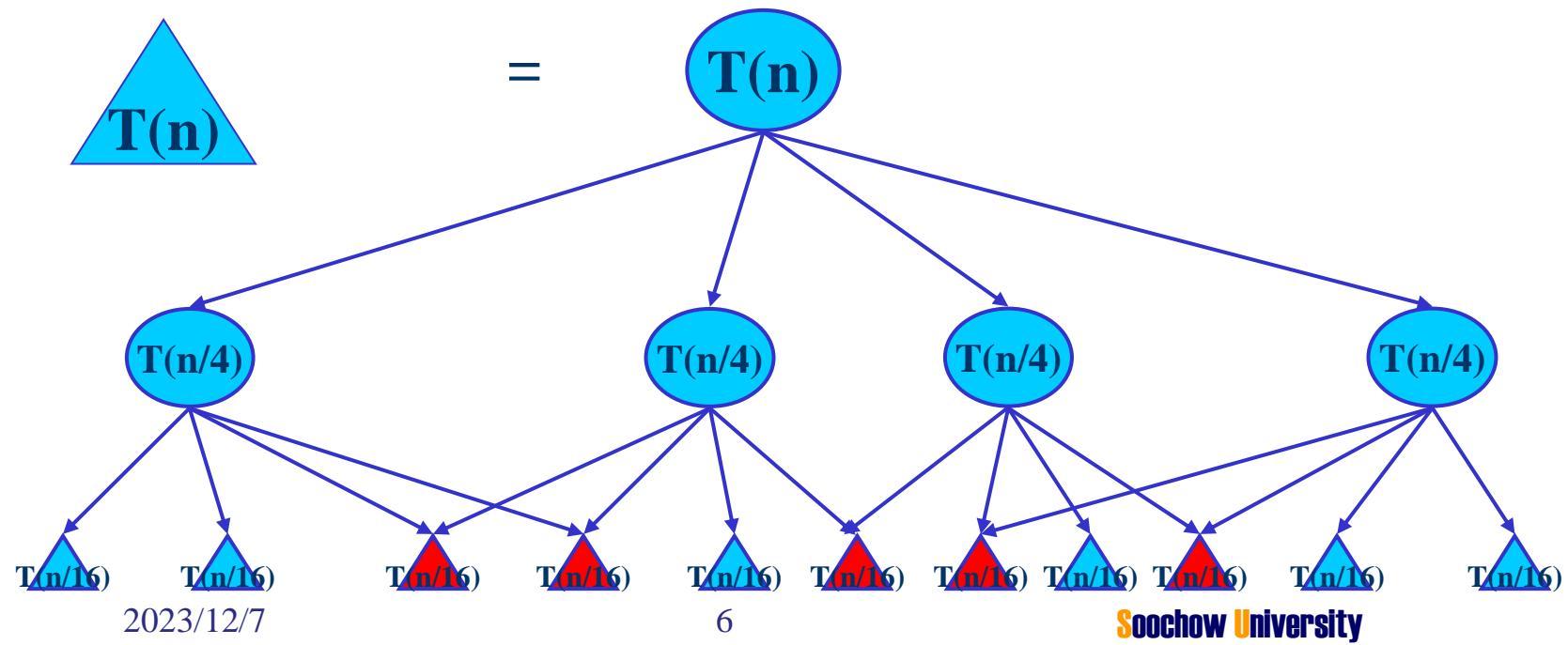
- 动态规划的基本思想：**分治思想和解决冗余**
- 动态规划策略与分治策略类似，也是将待求解的原问题分解成若干个子问题





动态规划策略概念

- 动态规划：但与分治策略不同的是，动态规划适用于**子问题重叠**的情况，即不同子问题**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，只需从表格中找到上次计算的结果加以调用即可，避免了不必要的计算工作





动态规划策略概念

- 从数学的角度看，最优化问题可以概括为这样一种数学模型：给定一个“函数” $F(X)$ ，其中“自变量” X 满足一定条件，求 X 为怎样的值时， $F(X)$ 取得其最优值（最大值或最小值）
 - $F(X)$ 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： $X \in D$
 - 使目标函数 $F(X)$ 取最小值或最大值的可行解 X 称为**最优解**。相应地，目标函数的最小值或最大值 $\text{Min}F(X)$ 或 $\text{Max}F(X)$ 称为**最优值**（或者**最优解的值**）



动态规划策略概念

□ 动态规划策略的步骤（框架）：

- 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 ---> 给出最优解的值的递归式
 - ③ 按自底向上的方式计算出最优解的值
 - ④ 由计算出的最优值构造一个最优解
- 步骤 ①~③ 是动态规划算法的基本步骤。如果仅需要求出最优解的值，而非最优解本身，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



第十一讲 动态规划

内容提要：

- 动态规划策略概念
- 钢条切割
- 矩阵链乘法
- 动态规划原理
- 最长公共子序列



钢条切割

□ 钢条切割问题：

- Serling 公司购买长钢条，将其切割为短钢条出售。不同的切割方案，收益是不同的，那么如何切割才能获得最大的收益？
 - 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：长度为 i 英寸的钢条可以带来 p_i 美元的收益

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



钢条切割

□ 钢条切割问题：

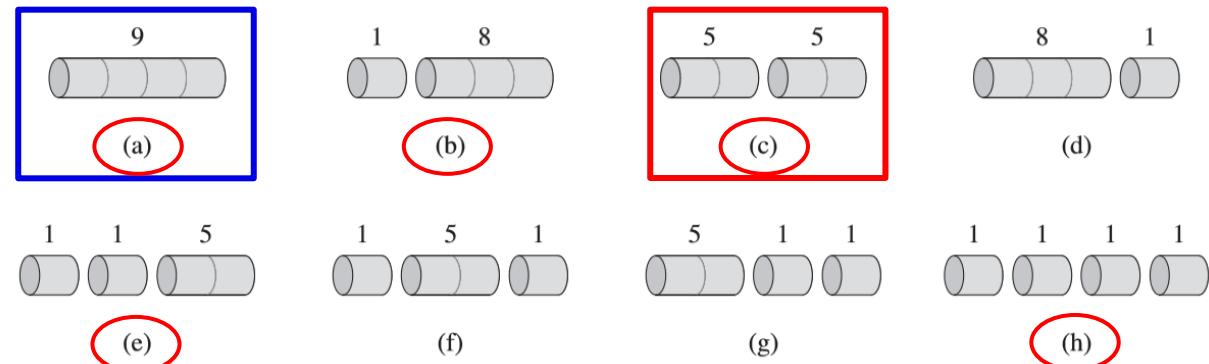
- › 给定一段长度为 n 英寸钢条和一个价格表 P , 求钢条切割方案, 使得销售收益 r_n 最大, 即 $\text{Max}r_n$
- › 分析: 如果长度为 n 英寸的钢条的价格 p_n 足够大, 则可能完全不需要切割, 出售整条钢条是最好的收益; 但由于每个 p_i 不同, 可能切割后的销售收益会更好一些

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



钢条切割

□ 考虑 $n = 4$ 的情况



□ 长度为 n 英寸的钢条共有 2^{n-1} 种不同的切割方案：

- 在距离钢条左端 i ($i = 1, 2, 3, \dots, n-1$) 英寸处可选择切割或不切割，共有 $n-1$ 个切割点
- 如果按长度非递减的顺序切割小段钢条，切割方案会少很多，但切割方案的数量还是有划分函数给出，仍然远远大于任何 n 的多项式

□ **最优方案：** 方案 c，将 4 英寸的钢条切割为两段长度各长为 2 英寸的钢条，此时可产生的最大收益为 10 美元
2023/12/7 12



钢条切割

□ 如果一个**最优解** (即切割方案) 将长度为 n 的钢条切割为 k 段, 每段的长度为 $i_j (1 \leq j \leq k)$, 则有 $n = i_1 + i_2 + \cdots + i_k$, 得到**最优解的值** (即最大收益) 为:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

如, 从价格表可得以下基本方案:

$r_1=1$, 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

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

$r_6=17$, 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)



钢条切割

□ 对于长度为 n ($n \geq 1$) 的钢条，用更短的钢条的最优切割收益来描述：

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

- 对每个切割位点 $i = 1, 2, \dots, n-1, n$ ，首先钢条切割为长度为 i 和 $n-i$ 两段；接着，求解 i 和 $n-i$ 两段的最优切割收益 r_i 和 r_{n-i} ，其中每种切割方案的最优收益为两段的最优收益之和
- 因此，长度为 n 的钢条的最优切割收益 r_n 表示为两段更短的钢条的最优切割收益： $r_n = r_i + r_{n-i}$
- 实际这样的 i 有 n 种选择，对应 n 种切割方案（包括不切割）。由于无法预知哪种切割方案会获得最优收益，所以必须考察所有可能的 i ，选取其中收益最大者



钢条切割

- 对于长度为 n ($n \geq 1$) 的钢条，用更短的钢条的最优切割收益来描述：

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

- 注意到，为了求解规模为 n 的原问题，先求解形式完全一样，但规模更小的子问题： i 和 $n-i$ 。即当完成首次切割后，**将两段钢条看成两个独立的钢条切割问题实例，即子问题**
- 通过组合两个相关子问题的最优解，并在所有可能的两段切割方案中选取组合收益最大者，构成原问题的最优解
- **这里体现了钢条切割问题的最优子结构性质：问题的最优解由相关子问题的最优解构成，而这些子问题可以独立求解**



钢条切割

□ 钢条切割问题的朴素递归求解过程：

- 对上述切割过程做如下简化，得到更为简单的递归求解方法：将钢条从左边切割下长度为 i 的一段，只对右边剩下长度为 $n-i$ 的一段继续切割（递归求解），对左边的一段不再进行切割
- 此时，问题的分解方式为：将长度为 n 的钢条分解为左边开始一段，以及剩余部分继续分解的结果，则得到最优切割收益 r_n 的简化版本：

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

- 原问题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



钢条切割

- 一个自顶向下的递归求解过程 $\text{CUT-ROD}(p, n)$: 其中 p 是价格数组, n 是钢条长度

$\text{CUT-ROD}(p, n)$

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

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

从左边切割下长度为 i 的一段,
对右边剩余长度为 $n-i$ 的一段递归求解

- 该求解过程的效率很差: 存在大量相同的子问题, CUT-ROD 反复地用一些相同的参数做重复的递归调用



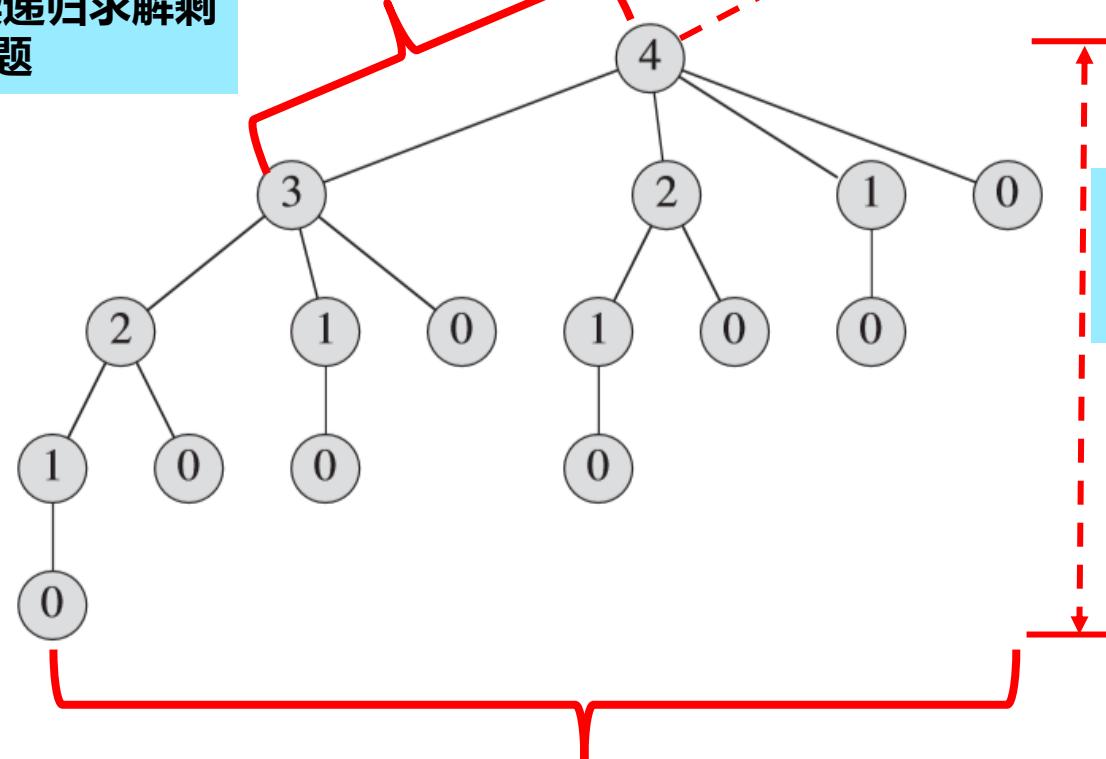
钢条切割

□ 如 $n = 4$, CUT-ROD 的递归执行过程可以用递归调用树表示如下:

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一种切割方案



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, 表示 2^{n-1} 种切割方案



钢条切割

- 令 $T(n)$ 表示第二个参数值为 n 时 CUT-ROD 的调用次数，这个值等于递归调用树中根为 n 的子树中的结点总数：

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

- 第一项 “1” 表示函数的第一次调用（递归调用树的根节点对应的最初的一次调用）
 - $T(j)$ ($j = n-i$) 为调用 $CUT-ROD(p, n-i)$ 所产生的所有调用的次数
 - 证明得到： $T(n) = 2^n$ ，即 CUT-ROD 的运行时间为 n 的指数函数
- 对于长度为 n 的钢条，CUT-ROD 显然考察了所有 2^{n-1} 种可能的切割方案。递归调用树中共有 2^{n-1} 个叶节点，每个叶节点对应一种可能的钢条切割方案



钢条切割

□ 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，则只需要查找保存的结果，而不必重新计算
 - 动态规划方法需要付出**额外的空间**来保存子问题的解，是一种典型的**时空权衡** (time-memory trade off)
-
- **可获得的时间效率改进：** 动态规划方法将一个**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每个子问题，因为动态规划方法每个子问题只需求解一次，因此动态规划方法的总运行时间就是**多项式阶的**



钢条切割

□ 动态规划求解的两种方法：

- ① 带备忘的自顶向下法 (top-down with memorization)
 - 依旧按照**递归调用**的形式编写求解过程，但在处理过程中会**保存每个子问题的解**
 - 当需要时，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



钢条切割

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 $-\infty$, 常见的表示“未知值”的方法
- 检查所需值是否已知, 如果 $r[n] \geq 0$ 时, 表示该子问题已经求解, 直接返回已保存的值
- 第 3~7 行按照 CUT-ROD 过程计算所需值 q
- 当有新的结果时, $r[n]$ 保存结果 q



钢条切割

□ 动态规划求解的两种方法：

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

- 按由小到大的顺序依次求解：

当求解某个子问题时，它所依赖的更小的子问题都已求解完毕，结果已经保存，因此可以直接引用并组合出它自身的解



钢条切割

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  $r[n]$ 
```

- 对于 $j = 1, 2, \dots, n$ 按升序求解每个规模为 j 的子问题
- 求解规模为 j 的子问题的方法与 CUT-ROD 所采用的方法相同，只是直接访问数组元素 $r[j-i]$ 来获得规模为 $j-i$ 的子问题的解，不必进行递归调用
- 规模为 j 的子问题的解存入 $r[j]$



钢条切割

□ 对比：朴素递归 VS 带备忘的自顶向下法

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  $q$ 
```

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  $q$ 
```

朴素递归：“硬”（暴力）求解。整个过程对重叠的子问题进行重复计算，造成效率低下

带备忘的自顶向下动态规划：只需递归求解一部分子问题。在求解的过程中记录了重叠子问题的解。通过“引用”之前的计算结果，避免重复计算，提高效率



钢条切割

□ 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  $q$ 
```

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  $r[n]$ 
```

自顶向下

递归调用设计框架

自底向上

迭代设计框架



钢条切割

□ 通常，自顶向下法和自底向上法具有相同的渐近运行时间：

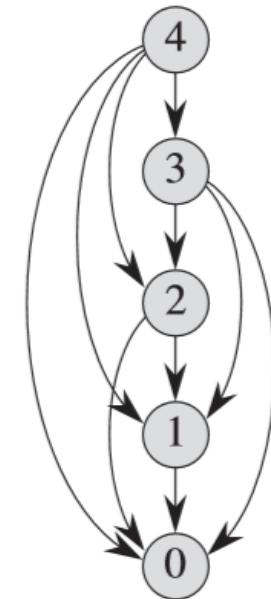
- 如果子问题空间中某些子问题没有必要求解，带备忘的方法有着只需要求解那些必须要求解的子问题的优点（剪枝）
- 由于自底向上法没有频繁的递归函数调用的开销，所以自底向上法的时间复杂性函数通常具有更小的系数



钢条切割

□ 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**，可用子问题图来描述：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

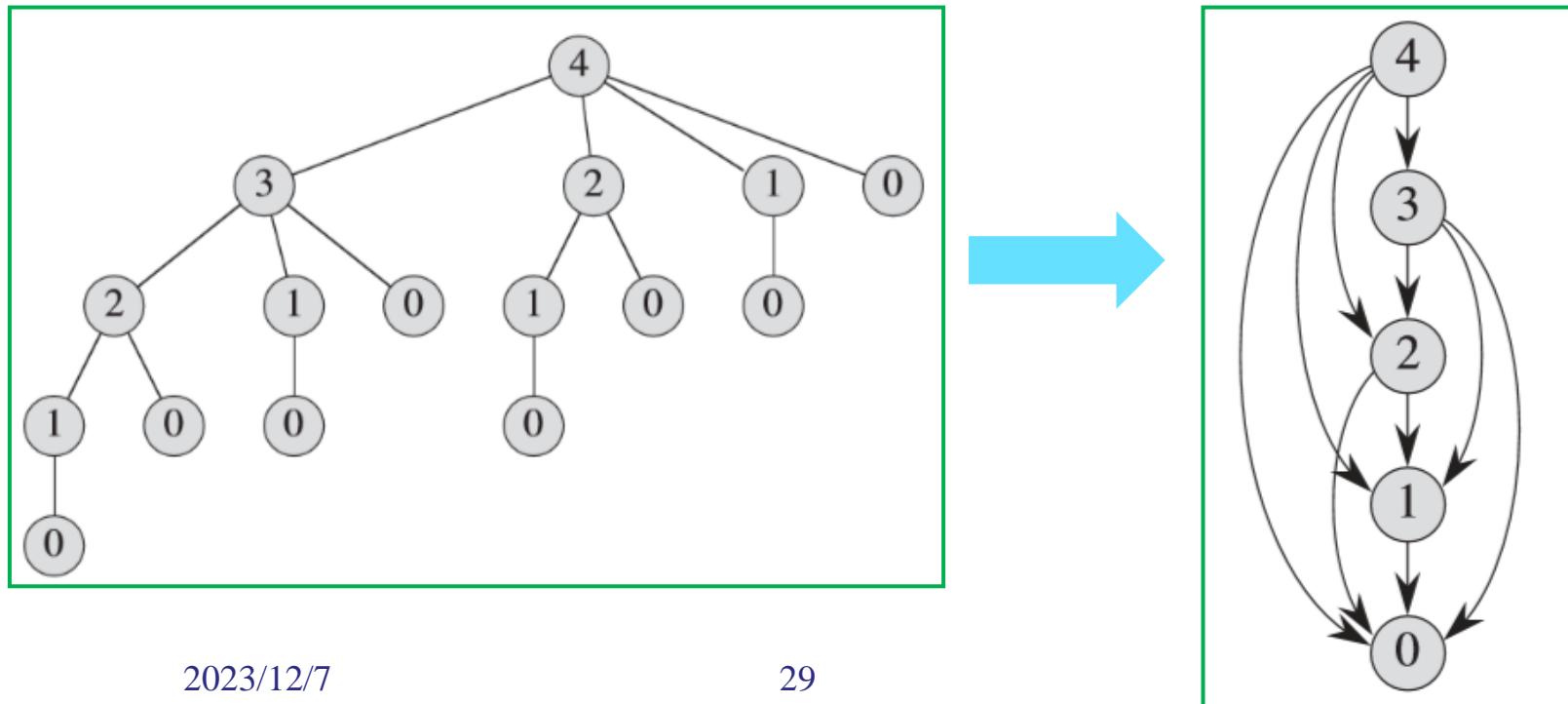


$n = 4$ 时，钢条切割问题的子问题图。顶点的标号给出了子问题的规模。有向边 (x, y) 表示当求解子问题 x 时需要子问题 y 的解



钢条切割

- 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





钢条切割

□ 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 y
- 自底向上动态规划算法是按 “**逆拓扑序**” 来处理子问题图中的顶点。即对于任何子问题，仅当它依赖的所有子问题都求解完成，才会求解它



钢条切割

口 基于子问题图 “**估算**” 算法的运行时间： $G = (V, E)$ 的规模可以帮助确定动态规划算法的运行时间

- 由于每个子问题只会求解一次，因此算法运行时间等于每个子问题求解时间之和
- 子问题图中，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（对于 for 循环中迭代的次数）
- 因此，一般情况下，动态规划算法的运行时间与**顶点和边的数量**呈线性关系 $O(V+E)$



钢条切割

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

- 过程 BOTTOM-UP-CUT-ROD 的主体是嵌套的双重循环，内层 for 循环（第 5~6 行）的迭代次数构成一个等差数列
- 自顶向下的 MEMOIZED-CUT-ROD 过程：MEMOIZED-CUT-ROD 过程对每个子问题（规模为 $0, 1, \dots, n$ ）也只求解一次，当求解一个之前已经计算出结果的子问题时，递归调用会立即返回；为求解规模为 n 的子问题，第 6~7 行的循环会迭代 n 次。因此，MEMOIZED-CUT-ROD 进行的所有递归调用执行此 for 循环的迭代次数也是一个等差数列



钢条切割

□ 带备忘的自顶向下法 VS 自底向上法

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

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

MEMOIZED-CUT-ROD-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  $q$ 
```



钢条切割

□ 重构解：

- BOTTOM-UP-CUT-ROD 算法给出了**最优切割收益**，但是**最优切割方案**（最优解）：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 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**最优解对应的第一段钢条的切割长度 s_j**



钢条切割

□ 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实例

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

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

2023/12/7

35

➤ PRINT-CUT-ROD-SOLUTION($p, 10$): 10

Soochow University



第十一讲 动态规划

内容提要：

- 动态规划算法概念
- 钢条切割
- 矩阵链乘法
- 动态规划原理
- 最长公共子序列



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

- 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），则计算出矩阵 C 共需要 pqr 次标量乘法运算



矩阵链乘法

□ 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  $C$ 
```

只有两个矩阵“相容”才能相乘

三重循环结构



矩阵链乘法

□ 给定一个包含 n 个矩阵的序列 (矩阵链) $\langle A_1, A_2, \dots, A_n \rangle$, 要计算这 n 个矩阵的乘积: $A_1 A_2 \cdots A_n$

- 可以先用**括号**明确计算次序, 然后用标准的矩阵相乘算法计算
- 矩阵乘法满足**结合律**, 不满足交换律

□ 以矩阵链 $\langle A_1, A_2, A_3 \rangle$ 相乘为例, **不同的加括号方式会导致不同的计算代价**

- 设有三个矩阵 A_1, A_2, A_3 的维数分别为 $10 \times 100, 100 \times 5, 5 \times 50$
- 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 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



矩阵链乘法

- 如果按 $(A_1(A_2A_3))$ 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计算代价最小呢？



矩阵链乘法

口 矩阵链乘法问题 (Matrix-Chain Multiplication Problem) :

- 给定 n 个矩阵的链 $\langle A_1, A_2, \dots, A_n \rangle$, 其中 $i = 1, 2, \dots, n$,
矩阵 A_i 的维数为 $p_{i-1} \times p_i$ 。求一个完全括号化方案, 使得计算
乘积 $A_1 A_2 \cdots A_n$ 所需的标量乘法次数最少
- **N:** 求解矩阵链乘法问题并不是要真正进行矩阵相乘运算, 只是
确定计算代价最低的计算顺序; 确定最优计算顺序所花费的时
间通常要比随后真正进行矩阵相乘所节省的时间要少



矩阵链乘法

口 矩阵乘积链的**完全括号化**: 它是单一矩阵, 或者是两个完全括号化的矩阵乘积链的积, 且已外加括号

- 如已知四个矩阵 A_1, A_2, A_3, A_4 , 乘积 $A_1A_2A_3A_4$ 有五种不同的完全括号化方案:

$$(A_1(A_2(A_3A_4))) \quad (A_1((A_2A_3)A_4)) \quad ((A_1A_2)(A_3A_4))$$

$$((A_1(A_2A_3))A_4) \quad (((A_1A_2)A_3)A_4)$$



矩阵链乘法

口 穷举法：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的标量乘法次数，从中找出一种次数最少的计算次序

- 复杂性分析：用 $p(n)$ 表示 n 个矩阵链乘的可供选择的括号化方案的数量，如果将 n 个矩阵从第 k 和第 $k+1$ 处隔开，对两个子序列再分别加扩号，则可以得到下面递归式：

$$p(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & n > 1 \end{cases}$$

$\Rightarrow p(n) = C(n-1)$ 为 Catalan 数

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right) \quad \text{呈指数增长}$$

- 因此，穷举法不是一个有效算法



矩阵链乘法

□ 动态规划方法求解：

① 寻找最优括号化方案的**最优子结构**：原问题的最优解由相关子问题的最优解构成，然后可以利用**最优子结构从子问题的最优解构造出原问题的最优解**

- 用记号 $\overline{(A_i A_{i+1} \cdots A_j)}$ 表示矩阵链 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案。假设 $A_i A_{i+1} \cdots A_j$ 的**最优括号化方案的分割点**在 A_k 和 A_{k+1} 之间 ($i \leq k < j$)，得到两个子矩阵链 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$
- 那么，继续对两个子矩阵链 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 进行括号化时，应该直接采用**独立求解这两个子矩阵链时所得的最优括号化方案**，Why？

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



矩阵链乘法

□ 剪切-粘贴证明：反证法

- 如果不采用独立求解子矩阵链 $A_i A_{i+1} \cdots A_k$ 所得的最优括号化方案 $\overline{(A_i A_{i+1} \cdots A_k)}$ ，那么可以假设 $\overline{(A_i A_{i+1} \cdots A_k)}'$ 是一个代价更小的最优括号化方案，将此最优解带入 $A_i A_{i+1} \cdots A_j$ 最优解中，代替原来对子矩阵链 $A_i A_{i+1} \cdots A_k$ 进行括号化的方案
- 这样得到的最优解 $\overline{(A_i A_{i+1} \cdots A_j)}'$ 的计算代价比原来的“最优解” $\overline{(A_i A_{i+1} \cdots A_j)}$ 更小，这与 $\overline{(A_i A_{i+1} \cdots A_j)}$ 是最优括号化方案相矛盾。对 $A_{k+1,j}$ 亦然
- 因此，在原问题 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案中，对划分得到的两个子矩阵链 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 进行括号化的方法，就是独立求解该子矩阵链自身的最优括号化方案



矩阵链乘法

- 上述体现了矩阵链乘法问题的最优子结构：一个非平凡的矩阵链乘法问题实例的任何解都需要划分链，而任何最优解都是由子问题实例的最优解构成的
- 利用最优子结构性，为构造一个矩阵链乘法问题实例的最优解，可以将问题划分为两个子问题（即 $A_i A_{i+1} \cdots A_k$ 和 $A_{k+1} A_{k+2} \cdots A_j$ 最优括号化问题），求出子问题实例的最优解，然后将子问题的最优解组合起来，从而构造出原问题的最优解
- 必须保证在确定分割点 k 时，已经考察了所有可能的划分点，从而可以保证不会遗漏最优解



矩阵链乘法

② 建立递归关系：用子问题的最优解来递归地定义原问题最优解的代价

- 用 $A_{i,j}$ 表示 $A_i A_{i+1} \cdots A_j$ 乘积的结果矩阵，令 $m[i, j]$ 表示计算矩阵 $A_{i,j}$ 所需最少标量乘法次数（对应 $A_i A_{i+1} \cdots A_j$ 的最优括号化问题），通过①中找到的最优子结构来计算 $m[i, j]$ ，有：

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

- 原问题的最优解 $m[1, n]$ 就是计算结果矩阵 $A_{1,n}$ 所需的最小代价
- $m[i, j]$ 备忘表的结构：数据结构是二维数组



矩阵链乘法

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

- 对于 $i = j$ 时的平凡问题，矩阵链只包含唯一的矩阵 A_i ，不需要做任何标量乘法运算，因此，对所有 $i = 1, 2, \dots, n$, $m[i, i] = 0$
- 对于 $i < j$ 时的非平凡问题，假设 $A_i A_{i+1} \cdots A_j$ 最优括号化方案的分割点在矩阵 A_k 和 A_{k+1} 之间， $i \leq k < j$, $m[i, j]$ 就等于计算 $A_{i, k}$ 的最小代价 $m[i, k]$ + 计算 $A_{k+1, j}$ 的最小代价 $m[k+1, j]$ + 两个结果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
- 再设 $s[i, j]$ ，保存 $A_i A_{i+1} \cdots A_j$ 最优括号化方案的分割点位置 k ，则可以依靠 s 求出最优链乘模式（即最优解）



矩阵链乘法

③ 计算最优代价

- 对于 $1 \leq i \leq j \leq n$, 不同的有序对 (i, j) 对应于不同的子问题。因此, 不同子问题的个数最多只有

$$\binom{n}{2} + n = \Theta(n^2)$$

- 用动态规划算法求解, 依据其递归式以**自底向上的方式**进行计算:
 - $j-i+1$ 个矩阵链相乘的最优计算代价 $m[i, j]$ 只依赖于那些少于 $j-i+1$ 个矩阵链相乘的最优计算代价
 - 按矩阵链长度递增的顺序求解矩阵链括号化问题

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



矩阵链乘法

□ MATRIX-CHAIN-ORDER 过程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  $s$ 
```

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

对 $l = 2, \dots, n$ 按升序依次计算链长为 l 的最小计算代价 $m[i, j] = m[i, i+l-1]$

计算代价 $m[i, j]$ 仅依赖于已经计算出的表项 $m[i, k]$ 和 $m[k+1, j]$



矩阵链乘法

- MATRIX-CHAIN-ORDER 过程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  $s$ 
```

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 $m[1, n]$



矩阵链乘法

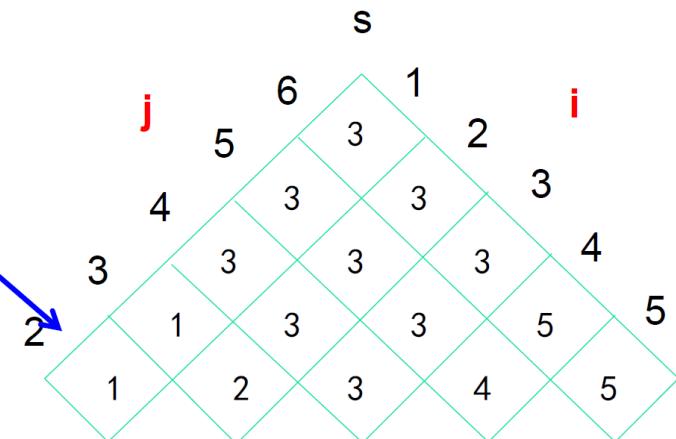
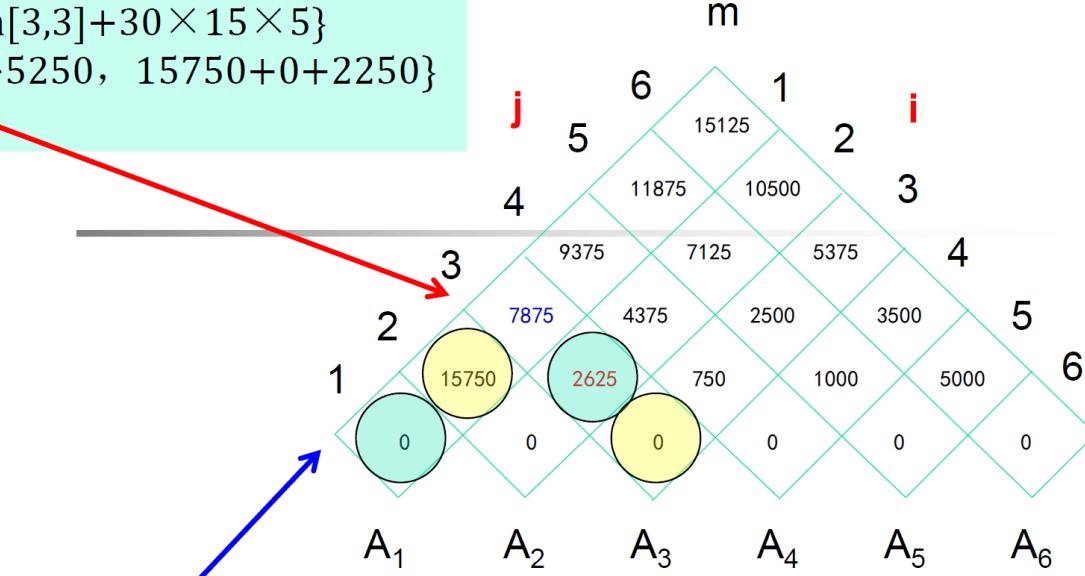
$$\begin{aligned}
 m[1,3] &= \min\{m[1,1]+m[2,3]+30 \times 35 \times 5, \\
 &\quad m[1,2]+m[3,3]+30 \times 15 \times 5\} \\
 &= \min\{0+2625+5250, 15750+0+2250\} \\
 &= 7875
 \end{aligned}$$

例，设

矩阵	维数
A ₁	30×35
A ₂	35×15
A ₃	15×5
A ₄	5×10
A ₅	10×20
A ₆	20×25

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

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





矩阵链乘法

□ MATRIX-CHAIN-ORDER 过程时间复杂度分析：

- 算法的主体由一个三层循环构成：外循环执行 $n-1$ 次，内层循环至多执行 $n-1$ 次，所以 MATRIX-CHAIN-ORDER 的算法复杂度是 $O(n^3)$
- 算法需要 $\Theta(n^2)$ 的空间保存 m 和 s



矩阵链乘法

④ 构造最优解

- $s[i, j]$ 记录了 $A_iA_{i+1}\cdots A_j$ 的最优括号化方案的“首个”分割点 k
 - 基于 $s[i, j]$, 对 $A_iA_{i+1}\cdots A_j$ 的括号化方案是:
$$(A_iA_{i+1}\cdots A_{s[i, j]})(A_{s[i, j]+1}\cdots A_j)$$
- 用递归的方法求出两个子问题 $A_iA_{i+1}\cdots A_{s[i, j]}$ 、 $A_{s[i, j]+1}\cdots A_j$ 的最优括号化方案



矩阵链乘法

□ 根据 s 求出矩阵链乘的最优计算模式

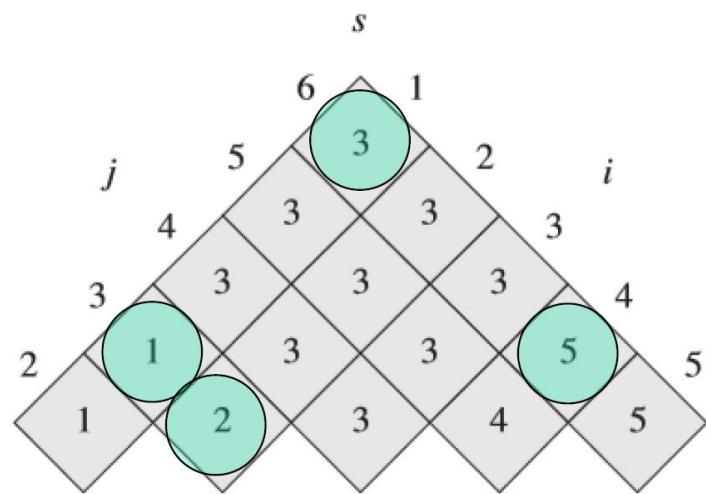
PRINT-OPTIMAL-PARENS(s, i, j)

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



$((A_1(A_2A_3))((A_4A_5)A_6))$





第十一讲 动态规划

内容提要：

- 动态规划算法概念
- 钢条切割
- 矩阵链乘法
- 动态规划原理
- 最长公共子序列



动态规划原理

适合采用动态规划方法的最优化问题中的两要素：

- **最优子结构**
- **重叠子问题**



动态规划原理

口 利用动态规划求解问题的方法：

➤ 第一步：刻画最优解的结构

- 如果一个问题的最优解包含其子问题的最优解，称此问题具有**最优子结构性质**
- 判断某个问题是否适用动态规划算法，判断其是否具有最优子结构性质（也可能意味着适合贪心策略）
- 使用动态规划方法时，通常是**利用子问题的最优解来构造原问题的最优解**；因此，必须确保考察了最优解中用到的所有子问题



动态规划原理

□ 寻找最优子结构的通用模式：

- 证明问题最优解的第一个组成部分是做出一个选择，即划分子问题，例如：选择钢条第一次切割位置，选择矩阵链的首次划分位置等。做出这次选择会产生一个或多个待求解的子问题
- 对一个给定的问题，在其可能的第一步选择中，假定已经知道哪种选择才会得到最优解（不必关心如何得到这个最优选择，只是假定已经知道了这种选择）
- 在已知该选择后，要确定哪些子问题会随之产生，以及如何最好地描述所得到的子问题空间
- 利用“剪切-粘贴”技术证明：作为原问题最优解的组成部分，**每个子问题的解就是它本身的最优解**



动态规划原理

口 采用“剪切-粘贴”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解不是其自身的最优解，那么我们可以从原问题的解中“剪切”掉这些非最优解，而将最优解“粘贴”进去，从而得到一个原问题更优的解，这与最初的解是原问题的最优解的前提假设相矛盾
- 所以作为原问题最优解构成的子问题的解必须是其本身最优解，最优子结构性成立



动态规划原理

□ 子问题空间的描述：

- 遵循如下规则：尽量保持子问题空间简单，需要时再扩充它
- 对于不同问题领域，最优子结构的不同体现在两个方面：
 - 原问题的最优解中涉及多少个子问题
 - 在确定最优解使用哪些子问题时，需要考察多少种选择



动态规划原理

□ 分析动态规划算法的运行时间：

- 一个动态规划算法的运行时间依赖于两个因素的乘积：**子问题的总个数**×**每个子问题中有多少种选择**（也就是每个子问题的求解时间）
- 对于钢条切割问题，共有 $\Theta(n)$ 个子问题，每个子问题最多需要考察 n 种选择，因此运行时间为 $O(n^2)$ ；对于矩阵链乘法问题，共有 $\Theta(n^2)$ 个子问题，每个子问题中又至多有 $n-1$ 个选择，因此执行时间为 $O(n^3)$



动态规划原理

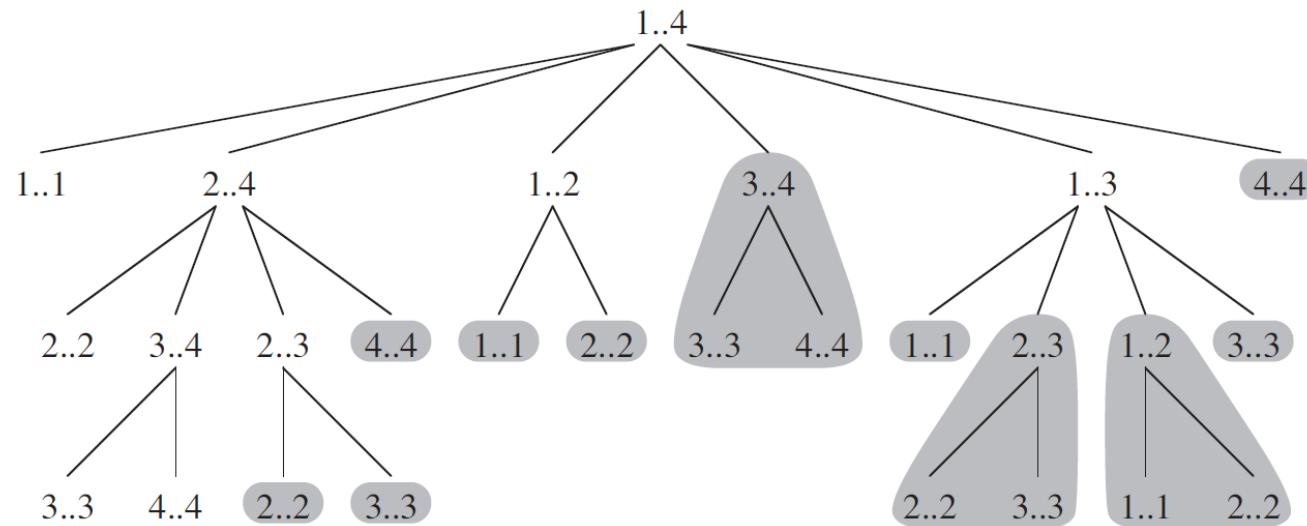
口 重叠子问题

- 适用于动态规划求解的最优化问题的第二个要素是子问题的空间必须足够“小”，即问题的递归算法会反复求解相同的子问题，而不是一直生成新的子问题
- 不同子问题的总数是输入规模的多项式函数
- 如果递归算法反复求解相同的子问题，称最优化问题具有重叠子问题性质
- 而适合用分治策略求解的问题通常在递归的每一步都生成全新的子问题



动态规划原理

- 动态规划算法，充分利用重叠子问题，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果





动态规划算法概念

□ 对动态规划带来的改进的理解：

- 改进一：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，分解得到的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



动态规划算法概念

- 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



第十一讲 动态规划

内容提要：

- 动态规划算法概念
- 钢条切割
- 矩阵链乘法
- 动态规划原理
- 最长公共子序列



最长公共子序列

一个应用背景：基因序列比对

- DNA（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) 四种碱基分子构成
- 用它们的英文单词的首字母 A、G、C、T 来代表这四种碱基，这样一个 DNA 可以表示为**有穷字符集** {A, C, G, T} 上的一个串
- 例如：两个有机体的 DNA 分别为

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$



最长公共子序列

- 通过比较两个有机体的 DNA 来确定两个有机体有多“相似”，这在生物学上叫做“**基因序列比对**”
- 比较两个 DNA 相似性转换为计算机语言，可以看作是**对两个由 A、C、G、T 组成的字符串的比较**
- 那么如何度量 DNA 相似性?
 - 如果一个 DNA 序列是另一个 DNA 序列的**子序列**，那么称这两个 DNA 序列相似
 - 当两个 DNA 序列互不为对方子序列的时候，如何度量?
 - 方法一：统计将其中一个转换成另一个所需改变的数量，如果需要改变的数量小，则可称两个 DNA 序列相似



最长公共子序列

□ 那么如何度量 DNA 相似性?

- 当两个 DNA 序列互不为对方子序列的时候，如何度量？
 - 方法二：在 S_1 和 S_2 中寻找第三个存在的 S_3 ，使得 S_3 中的碱基以同样的先后顺序出现在 S_1 和 S_2 中，但不一定连续；然后根据 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
 - 比如上面的两个 DNA 序列中，最长的公共存在是：

$S_3 = \text{GTCGTCGGAAGGCCGGCCGAA}$ 。

$S_1 = \text{ACCGGTCGAGTGC} \color{red}{\text{CGC}} \text{GG} \color{red}{\text{AAGGCCGGCCGAA}}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTG} \color{red}{\text{TAAA}}$



最长公共子序列

□ 子序列定义

- 给定序列 $X = \langle x_1, x_2, \dots, x_m \rangle$, 如果序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ 是 X 的一个子序列, 则在 X 中存在一个对应的严格递增下标序列 $\langle i_1, i_2, \dots, i_k \rangle$, 使得对于所有 $j = 1, 2, \dots, k$, 有 $z_j = x_{i_j}$
- 例如序列 $Z = \langle B, C, D, B \rangle$ 是序列 $X = \langle A, B, C, B, D, A, B \rangle$ 的一个子序列, X 中对应的递增下标序列为 $\langle 2, 3, 5, 7 \rangle$

□ 两个序列的公共子序列定义

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



最长公共子序列

□ 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 $\langle B, C, B, A \rangle$

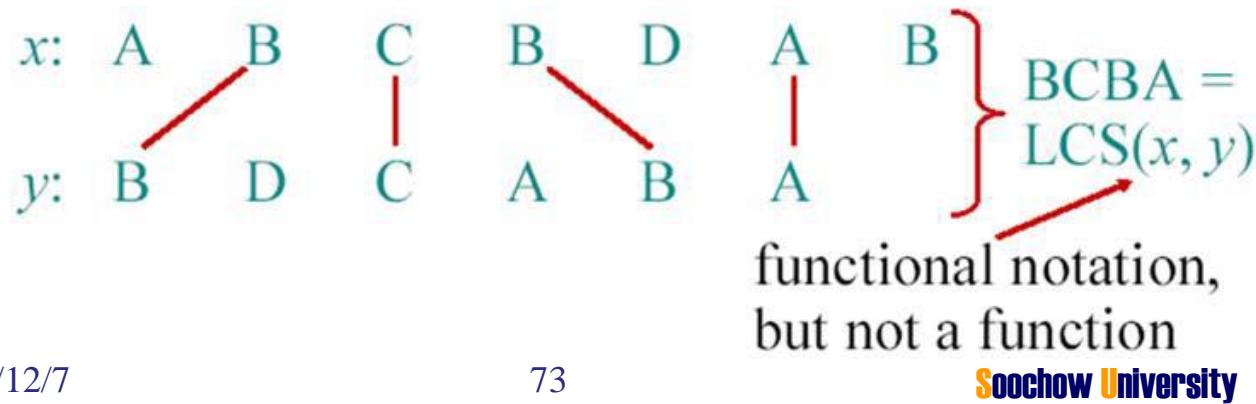


最长公共子序列

- 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
- 暴力检测法

- 检查 X 的每一个子序列是否是 Y 的子序列
- 判断一个序列是否为另一个序列的子序列时间复杂度为 $O(n)$
- 最坏情况的运行时间:

$$T(m, n)=O(\min(m2^n, n2^m))$$





最长公共子序列

□ 高效算法思路：采用动态规划策略

① 步骤一：LCS 问题的最优子结构性

➤ 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 $X_4 = \langle A, B, C, B \rangle$,

$$X_0 = \Phi$$



最长公共子序列

- 定理 15.1 (LCS 的最优子结构) : 令 $X_m = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y_n = \langle y_1, y_2, \dots, y_n \rangle$ 为两个序列, $Z = \langle z_1, z_2, \dots, z_k \rangle$ 为 X_m 和 Y_n 的任意 LCS。
 1. 如果 $x_m = y_n$, 则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS
 2. 如果 $x_m \neq y_n$, 那么 $z_k \neq x_m$ 意味着 Z 是 X_{m-1} 和 Y 的一个 LCS
 3. 如果 $x_m \neq y_n$, 那么 $z_k \neq y_n$ 意味着 Z 是 X 和 Y_{n-1} 的一个 LCS

“ X_m 和 Y_n 的 LCS” 划分为三个子问题 “ 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”

上述定理说明: 两个序列的 LCS 包含两个序列的前缀的 LCS,
因此, LCS 问题具有最优子结构性质



最长公共子序列

② 步骤二：一个递归解

— 定理 15.1 意味着求 X_m 和 Y_n 的一个 LCS 时需要求解一个或两个子问题

(1) 如果 $x_m = y_n$, 那么应该求解 X_{m-1} 和 Y_{n-1} 的一个 LCS, 将 $x_m = y_n$ 追加到这个 LCS 的末尾, 就得到 X 和 Y 的一个 LCS

(2) 如果 $x_m \neq y_n$, 那么求解两个子问题: 求 X_{m-1} 和 Y 的一个 LCS 与 X 和 Y_{n-1} 的一个 LCS。两个 LCS 较长者即为 X 和 Y 的一个 LCS

- LCS 问题的最优子结构性质: 由于这些情况覆盖了所有可能性, 因此知道必然有一个子问题的最优解出现在 X 与 Y 的 LCS 中
- LCS 问题的重叠子问题性质: 为了求 X 与 Y 的一个 LCS, 可能需要求 X_{m-1} 和 Y 的一个 LCS 及 X 和 Y_{n-1} 的一个 LCS。但这些子问题都包含求解 X_{m-1} 和 Y_{n-1} 的 LCS 的子子问题



最长公共子序列

- 设计 LCS 问题的递归算法首先建立最优解的递归式，记 $c[i, j]$ 表示前缀序列 X_i 和 Y_j 的一个 LCS 的长度，则有：

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

- 若 $i = 0$ 或 $j = 0$ ，即其中一个序列的长度为零，则二者的 LCS 长度为 0，即 $\text{LCS} = \Phi$
- 通过限制条件限定需要求解哪些子问题。若 $x_i = y_j$ ，只需要求解一个子问题： X_{i-1} 和 Y_{j-1} 的一个 LCS，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 $c[i, j] = c[i-1, j-1] + 1$



最长公共子序列

- 设计 LCS 问题的递归算法首先建立最优解的递归式，记 $c[i, j]$ 表示前缀序列 X_i 和 Y_j 的一个 LCS 的长度，则有：

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

- 若 $x_i \neq y_j$, 此时需要求解两个子问题： X_{i-1} 和 Y_j 的一个 LCS, 与 X_i 和 Y_{j-1} 的一个 LCS, 两个 LCS 较长者即为 X_i 和 Y_j 的一个 LCS, 所以 $c[i, j] = \max(c[i-1, j], c[i, j-1])$
- 根据上述公式，可以很容易地设计一个指数时间的递归算法来计算两个序列的 LCS 的长度。但是，由于 LCS 问题只有 $\Theta(mn)$ 个不同的子问题，可以用动态规划方法自底向上地计算



最长公共子序列

③ 步骤三：计算 LCS 的长度

— 数据结构设计

- $c[0..m, 0..n]$, 存放最优解的值 $c[i, j]$, 计算时按行主次序
- $b[1..m, 1..n]$, 帮助构造最优解, $b[i, j]$ 指向的表项对应计算 $c[i, j]$ 时所选择的子问题最优解

$$b[i, j] = \begin{cases} \nwarrow & \text{如果 } c[i, j] \text{ 由 } c[i-1, j-1] \text{ 确定} \\ \uparrow & \text{如果 } c[i, j] \text{ 由 } c[i-1, j] \text{ 确定} \\ \leftarrow & \text{如果 } c[i, j] \text{ 由 } c[i, j-1] \text{ 确定} \end{cases}$$

当构造解时，从 $b[m,n]$ 出发，上溯至 $i=0$ 或 $j=0$ 止
上溯过程中，当 $b[i,j]$ 包含“ \nwarrow ”时打印出 $x_i(y_j)$



最长公共子序列

- 下述过程 $\text{LCS-LENGTH}(X, Y)$ 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

$\text{LCS-LENGTH}(X, Y)$

```
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] ≥ 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
```

- 过程 LCS-LENGTH 以两个序列 X 和 Y 为输入，它把 $c[i, j]$ 的值填入一个按行计算表项的表 $c[0..m, 0..n]$ 中
- $c[m, n]$ 包含 X 和 Y 的一个 LCS 的长度



最长公共子序列

- 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
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
5	D	0	1	2	2	3	3
6	(A)	0	1	2	2	3	④
7	B	0	1	2	2	3	4

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

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

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



最长公共子序列

- 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
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	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

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

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

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

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

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

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

```

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



最长公共子序列

- 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
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
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	④
7	B	0	1	2	2	3	4

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

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

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

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

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

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

```

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



最长公共子序列

- 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
i	y_j	(B)	D	(C)	A	(B)	(A)	
	x_i	0	0	0	0	0	0	0
0	A	0	0	0	0	1	$\leftarrow 1$	1
1	(B)	0	①	$\leftarrow 1$	$\leftarrow 1$	1	\uparrow	$\leftarrow 2$
2	(C)	0	1	1	②	$\leftarrow 2$	2	\uparrow
3	(B)	0	1	1	2	2	③	$\leftarrow 3$
4	D	0	1	2	2	3	3	\uparrow
5	(A)	0	1	2	2	3	3	④
6	B	0	1	2	2	3	4	\uparrow

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

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

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

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

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

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

```

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



最长公共子序列

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

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

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

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

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

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

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

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



最长公共子序列

- 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
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
5	D	0	1	2	2	2	3
6	(A)	0	1	2	2	3	④
7	B	0	1	2	2	3	4

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

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

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

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

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

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

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



最长公共子序列

- 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
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

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

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

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

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

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

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

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



最长公共子序列

- 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
i	y_j	(B)	D	(C)	A	(B)	(A)	
0	x_i	0	0	0	0	0	0	0
1	A	0	0	↑	0	↑	1	←1
2	(B)	0	0	0	0	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

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

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

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

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

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

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

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



最长公共子序列

口 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
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
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

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

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

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

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

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

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

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



最长公共子序列

④ 步骤四：构造一个 LCS

- 表 b 用来构造序列 $X_m = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y_n = \langle y_1, y_2, \dots, y_n \rangle$ 的一个 LCS
- 构造思路：**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

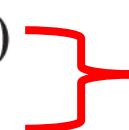


最长公共子序列

□ 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  $b[i, j] == “↖”$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  elseif  $b[i, j] == “↑”$ 
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 $m = 0$ or $n = 0$ ，算法结束，所以 PRINT-LCS 的时间复杂度为 $O(m+n)$



最长公共子序列

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	-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

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



谢谢!

Q & A

作业：计算题：15.2-1 15.4-1 15.5-2

算法设计题：15.1-3 15.5-1 15-11