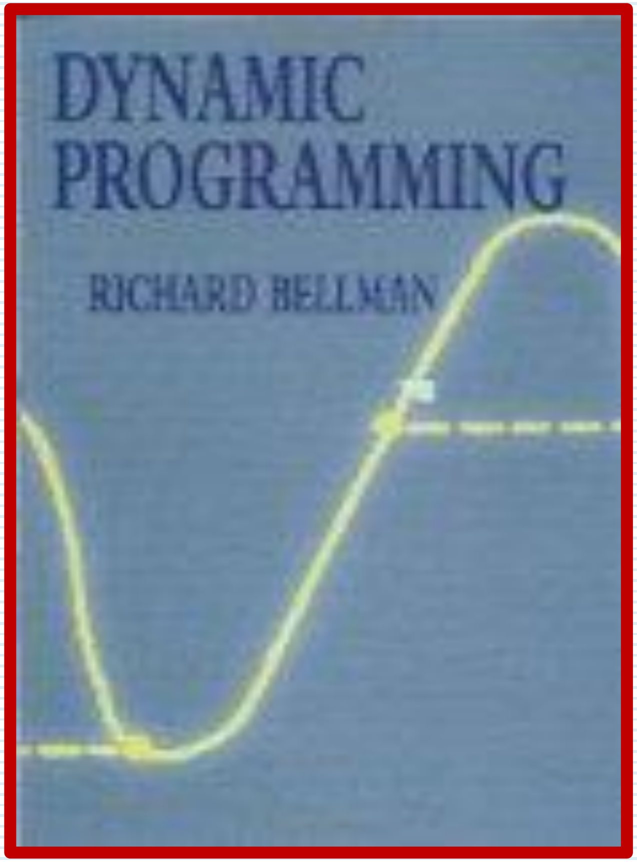
算法分析与设计

主讲教师： 刘峤

第3章：动态规划
Dynamic Programming



《 Dynamic Programming 》

Richard Bellman (1957)

用于解决分段决策过程的最优化问题

知识要点

- 理解动态规划算法的概念和
  - 两大基本要素：最优子结构性质和重叠子问题性质
- 掌握动态规划算法的设计方法
  - 最优解的结构特征及最优值的定义
  - 自底向上计算最优值并构造最优解
- 通过应用范例学习动态规划算法设计策略
  - 矩阵连乘问题、最长公共子序列问题、最大子段和问题
  - 凸多边形最优三角剖分问题、图像压缩问题、0-1背包问题

动态规划算法

- 算法基本思想
  - 将待求解问题分解成若干个子问题
- 与分治法的区别在于
  - 适用动态规划算法求解的问题，子问题往往不是互相独立的
  - 若用分治法求解，则分解得到的子问题数目太多，由于子问题被重复计算而导致最终解决原问题需指数时间
  - 思路：如果可以保存已解决的子问题的答案，就可以避免大量重复计算，从而得到多项式时间的算法
- 动态规划法的基本思路是：构造一张表来记录所有已解决的子问题的答案（无论算法形式如何，采用填表的方式是相同的）

动态规划算法的基本步骤

1. 找出最优解的性质（分析其结构特征）
2. 递归地定义最优值（优化目标函数）
3. 以自底向上的方式计算出最优值
4. 根据计算最优值时得到的信息，构造最优解

1. 矩阵连乘问题

（Matrix-Chain Multiplication）

矩阵相乘：标准解法

```
void Multi ( int **A, int **B, int **C, int p, int q, int r){
    for (int i=0; i<p; i++){
        for (int j=0; j<r; j++) {
            int sum = 0;
            for (int k=0; k<q; k++){
                sum += A[i][k]*B[k][j];
            }
            C[i][j]=sum;
        }
    }
}
```

A是p x q的矩阵

B是q x r的矩阵

时间复杂度：O(n^3)

数乘次数为：p x q x r

矩阵连乘问题

- 问题描述
  - 给定n个矩阵：{$\mathbf{A_1}$, $\mathbf{A_2}$, ……, $\mathbf{A_n}$}，其中$\mathbf{A_i}$与$\mathbf{A_{i+1}}$可乘
  - 求解这n个矩阵的连乘积： $\mathbf{M=A_1A_2 \ldots\ldots A_n}$
  - 问题：如何确定计算矩阵连乘积**M**的计算次序
    - 使得依此次序计算**M**需要的**数乘次数最少**？
- 问题分析
  - 矩阵乘法满足结合率，因此矩阵连乘有多种计算次序
  - 通过加括号的方式可以确定矩阵连乘问题的计算次序
  - 概念定义：完全加括号
    - 若矩阵连乘积**M**的计算次序完全确定，则称**M**已**完全加括号**
    - 可以按计算次序反复调用两个矩阵相乘的标准算法求解

完全加括号的矩阵连乘积

- 完全加括号的矩阵连乘积可递归地定义如下
  - 单个矩阵是完全加括号的
  - 矩阵连乘积**M**是完全加括号的，则**M**可以表示为两个完全加括号的矩阵连乘积A和B的乘积并加括号，即：M = (AB)
  - 例如：设有四个矩阵：A_{50x10}，B_{10x40}，C_{40x30}，D_{30x5}
  - 则：连乘积**M=ABCD**总共有五种完全加括号的方式
    - **（A（（BC）D））** **数乘次数：16000**
    - **（A（B（CD）））** **数乘次数：10500**
    - **（（AB）（CD））** **数乘次数：36000**
    - **（（（AB）C）D）** **数乘次数：87500**
    - **（（A（BC））D）** **数乘次数：34500**

矩阵连乘问题

- 问题：确定矩阵连乘积M的计算次序，使所需数乘次数最少
- 解决方案1：穷举法
  - 列举出所有可能的计算次序，从中找出数乘次数最少的次序
  - 算法复杂度分析：
    - 设n个矩阵连乘积所有可能的计算次序总数为P(n)
    - 对矩阵加括号：相当于分割序列 $(A_1 \dots A_k)(A_{k+1} \dots A_n)$
    - 每种加括号方式都可以分解为两个子矩阵的加括号问题
    - 由此可以得到关于P(n)的递推式如下：

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases} \quad \Rightarrow \quad P(n) = \Omega(4^n / n^{3/2})$$

矩阵连乘问题

- 问题：确定矩阵连乘积M的计算次序，使所需数乘次数最少
- 解决方案2：动态规划法
  - 首先分析问题的**最优解结构特征**
  - 符号约定：将矩阵连乘积 $(A_i\ A_{i+1}\ \ldots\ A_j)$ 简记为**A[i:j]**
  - 考察计算A[1:n]的最优计算次序：
    - 设最优计算次序在A_k和A_{k+1}之间断开矩阵链（$1 \le k < n$）
    - 则相应的完全加括号方式为： $(A_1\ \ldots\ A_k)(A_{k+1}\ \ldots\ A_n)$
  - 总计算量为如下三部分计算量之和：
    - 分别求解**A[1:k]**和**A[k+1:n]**的计算量
    - 矩阵**A[1:k]**与**A[k+1:n]**相乘的计算量

矩阵连乘问题

- 动态规划第一步小结：分析最优解的结构
  - 上述最优划分的关键结构特征在于
  - A[1:n]的最优计算次序所包含的矩阵子链也是最优的
  - 即：A[1:k]和A[k+1:n]自身的计算次序也是最优的
- 最优子结构性质
  - 矩阵连乘计算次序问题的最优解包含着其子问题的最优解
  - 这种性质称为最优子结构性质
  - 该性质是问题是否可用动态规划算法求解的显著特征之一！

动态规划法求解矩阵连乘问题

- 第二步：建立递归关系（递归地定义最优值）
  - 思考：矩阵连乘问题的最优值是什么？
  - 设：计算A[i:j]所需要的最少数乘次数为m[i][j]
    - 其中：1≤i≤j≤n
  - 问题：原问题的最优值是什么？
    - 原问题的最优值为m[1][n]
  - 当 i=j 时，有：A[i:j]=A_i
    - 因此：m[i][i]=0

动态规划法求解矩阵连乘问题

- 第二步：建立递归关系（递归地定义最优值）
  - 设：计算A[i:j]所需要的最少数乘次数为m[i][j]
  - 当 i<j 时，可利用最优子结构性质来计算m[i][j]
    - 设：A_i 的维度为 P_{i-1} x P_i
    - 设：A[i:j]的最优划分位置为k
    - 则：$m[i][j]=m[i][k]+m[k+1][j]+P_{i-1}\ P_k\ P_j$
    - k的取值只有j-i个可能，即：$k\in\{i,\ i+1,\ \ldots,\ j-1\}$
  - 综上可以得到矩阵连乘问题的最优值m[i][j]的定义：

$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\} & i<j \end{cases}$$

动态规划法求解矩阵连乘问题

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

- 第三步：计算最优值
  - 问题：直接递归计算m[1][n]将耗费指数时间
    - 许多子问题被重复计算多次
  - 分析：考虑 1≤i≤j≤n 的所有可能情况
    - 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：

$$\binom{n}{2}=\frac{n(n-1)}{2}=\frac{n^2-n}{2}$$

动态规划法求解矩阵连乘问题

- 第三步：计算最优值
  - 矩阵连乘问题中不同子问题的个数为问题规模n的多项式
    - 不同子问题个数：（n^2-n）/2
    - 这是该问题可用动态规划算法求解的又一显著特征
  - 解法：依据其递归式以**自底向上**的方式进行计算
    - 在计算过程中，保存已解决的子问题答案
    - 每个子问题只计算一次，从而避免大量的重复计算
    - 最终得到多项式时间的算法

动态规划法求解矩阵连乘问题

第三步：计算最优值——案例分析

- 例如：有矩阵链如下

A1	A2	A3	A4	A5	A6
30x35	35x15	15x5	5x10	10x20	20x25

- 矩阵维数序列如下（数组）：

P0	P1	P2	P3	P4	P5	P6
30	35	15	5	10	20	25

- 求最优完全加括号方式：使得矩阵元素相乘次数最少

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0					
A2		0				
A3			0			
A4				0		
A5					0	
A6						0

- **计算方法：根据递归式自底向上计算**　**思考：自底向上的含义？**

$$m[i,j]=\begin{cases}0 & \textbf{m[i][i]=0} \quad i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0	15750				
A2		0	2625			
A3			0	750		
A4				0	1000	
A5					0	5000
A6						0

$$m[1,2] = m[1,1] + m[2,2] + P_0P_1P_2\ (k=1) = 0+0+30\times35\times15 = 15750$$

$$m[2,3] = m[2,2] + m[3,3] + P_1P_2P_3\ (k=2) = 0+0+30\times15\times5 = 2625$$

$$m[3,4] = m[3,3] + m[4,4] + P_2P_3P_4\ (k=3) = 0+0+15\times5\times10 = 750$$

$$m[4,5] = m[4,4] + m[5,5] + P_3P_4P_5\ (k=4) = 0+0+5\times10\times20 = 1000$$

$$m[5,6] = m[5,5] + m[6,6] + P_4P_5P_6\ (k=5) = 0+0+10\times20\times25 = 5000$$

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0	15750	7875			
A2		0	2625	4375		
A3			0	750	2500	
A4				0	1000	3500
A5					0	5000
A6						0

$$\mathrm{m}[1,3] = \min \begin{cases} m[1,1] + m[2,3] + P_0 P_1 P_3 \ (k=1) \\ m[1,2] + m[3,3] + P_0 P_2 P_3 \ (k=2) \end{cases} = 7875$$

计算最优值

$$\mathrm{m}[1,3] = \min\begin{cases} m[1,1] + m[2,3] + P_0P_1P_3 \ (k=1) \\ m[1,2] + m[3,3] + P_0P_2P_3 \ (k=2) \end{cases} = 7875$$

$$\mathrm{m}[2,4] = \min\begin{cases} m[2,2] + m[3,4] + P_1P_2P_4 \ (k=2) \\ m[2,3] + m[4,4] + P_1P_3P_4 \ (k=3) \end{cases} = 4375$$

$$\mathrm{m}[3,5] = \min\begin{cases} m[3,3] + m[4,5] + P_2P_3P_5 \ (k=3) \\ m[3,4] + m[5,5] + P_2P_4P_5 \ (k=4) \end{cases} = 2500$$

$$\mathrm{m}[4,6] = \min\begin{cases} m[4,4] + m[5,6] + P_3P_4P_6 \ (k=4) \\ m[4,5] + m[6,6] + P_3P_5P_6 \ (k=5) \end{cases} = 3500$$

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0	15750	7875	9375		
A2		0	2625	4375	7125	
A3			0	750	2500	5375
A4				0	1000	3500
A5					0	5000
A6						0

$$\mathrm{m}[1,4] = \min\left\{\begin{array}{l} \underline{m[1,1]+m[2,4]}+P_0P_1P_4\ (k=1) \\ \underline{m[1,2]+m[3,4]}+P_0P_2P_4(k=2) \\ \underline{m[1,3]+m[4,4]}+P_0P_3P_4(k=3) \end{array}\right\}$$

$$= \min\left\{\begin{array}{l} 0+4375+30\times35\times5 \\ 15750+750+30\times15\times5 \\ 7850+0+30\times5\times10 \end{array}\right\} = 9375$$

计算最优值

$$m[1,4] = \min \begin{Bmatrix} m[1,1] + m[2,4] + P_0 P_1 P_4 \ (k=1) \\ m[1,2] + m[3,4] + P_0 P_2 P_4 (k=2) \\ m[1,3] + m[4,4] + P_0 P_3 P_4 (k=3) \end{Bmatrix} = 9375$$

$$m[2,5] = \min \begin{Bmatrix} m[2,2] + m[3,5] + P_1 P_2 P_5 (k=2) \\ m[2,3] + m[4,5] + P_1 P_3 P_5 (k=3) \\ m[2,4] + m[5,5] + P_1 P_4 P_5 (k=4) \end{Bmatrix} = 7125$$

$$m[3,6] = \min \begin{Bmatrix} m[3,3] + m[3,6] + P_2 P_3 P_6 (k=3) \\ m[3,4] + m[5,6] + P_2 P_4 P_6 (k=4) \\ m[3,5] + m[6,6] + P_2 P_5 P_6 (k=5) \end{Bmatrix} = 5375$$

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0	15750	7875	9375	11875	
A2		0	2625	4375	7125	10500
A3			0	750	2500	5375
A4				0	1000	3500
A5					0	5000
A6						0

$$m[1,5] = \min \begin{cases} m[1,1] + m[2,5] + P_0P_1P_5 (k=1) \\ m[1,2] + m[3,5] + P_0P_2P_5 (k=2) \\ m[1,3] + m[4,5] + P_0P_3P_5 (k=3) \\ m[1,4] + m[5,5] + P_0P_4P_5 (k=4) \end{cases} = 11875$$

$$\mathrm{m}[2,6] = \min \begin{cases} m[2,2] + m[3,6] + P_1P_1P_5 (k=2) \\ m[2,3] + m[4,6] + P_1P_2P_5 (k=3) \\ m[2,4] + m[5,6] + P_1P_3P_5 (k=4) \\ m[2,5] + m[6,6] + P_1P_4P_5 (k=5) \end{cases} = 10500$$

计算最优值

m	A1	A2	A3	A4	A5	A6
A1	0	15750	7875	9375	11875	15125
A2		0	2625	4375	7125	10500
A3			0	750	2500	5375
A4				0	1000	3500
A5					0	5000
A6						0

$$\mathrm{m}[1,6]=\min\begin{cases} m[1,1]+m[2,6]+P_0P_1P_6\,(k=1) \\ m[1,2]+m[3,6]+P_0P_2P_6\,(k=2) \\ m[1,3]+m[4,6]+P_0P_3P_6\,(k=3) \\ m[1,4]+m[5,6]+P_0P_4P_6\,(k=4) \\ m[1,5]+m[6,6]+P_0P_5P_6\,(k=5) \end{cases}=15125$$

思考：为什么要填这张表？ **提示：m[1][n]是什么？**

动态规划法求解矩阵连乘问题

- 第四步：构造最优解对应的问题解
  - 方法：另外设置一张表 S
    - 在填充表 m 的过程中记录各子链取最优值时的分割位置k
    - S[i][j]=k表示：A[i:j] 的最优划分是 (A[i:k])(A[k+1:j])
  - 构造原问题的最优计算次序
    - 从S[1,n]记录的信息可知A[1:n]的最佳划分方式
      - S[1, n]=**k** 表示最佳划分为：(A[1 : **k**])(A[**k+1** : n])
    - 其中A[1 : k]和A[k+1:n]的最佳划分方式可以递归地得到
      - S[1, k]=**x** → (A[1 : **x**])(A[**x+1**: k])
      - S[k+1, n]=**y** → (A[k+1 : **y**])(A[**y+1**: n])
  - 由此递归下去可以构造出原问题的一个最优解

动态规划法求解矩阵连乘问题

```
void MatrixChain(int *p, int n, int *m, int *s){
  for (int i = 1; i < n; i++) m[i][i] = 0;
  for (int r = 2; r <= n; r++){  // r表示子链长度，取值2～n
    for (int i = 1; i <= (n-r+1); i++) { // i为每“行”元素个数
      int j = i+r-1;      // j为列序号，“逐行”填充右上三角矩阵
      s[i][j] = i;        // i为初始断开位置
      m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
      for (int k = i+1; k < j; k++) {  // 尝试各种切分位置
        int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (t < m[i][j]){            思考：可否K从i开始取值？
          m[i][j] = t;  s[i][j] = k;
}}}}}
```

最优解？

构造最优解

最优值？

m	A1	A2	A3	A4	A5	A6
A1	0	15750	7875	9375	11875	15125
A2		0	2625	4375	7125	10500
A3			0	750	2500	5375
A4				0	1000	3500
A5					0	5000
A6						0

s	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

该矩阵连乘问题的最优解为: (A1(A2A3))((A4A5)A6)

从最优值得到最优解

```
void traceback(int s[], int i, int j) {
    if(i==j) return;
    traceback(s, i, s[i][j]);
    traceback(s, s[i][j]+1, j);
    output(); // A[i,s(i,j)] and A[s(i,j)+1, j]
}
```

初始调用参数：traceback(s, 1, n-1);

动态规划法求解矩阵连乘问题

- 算法复杂度分析
  - 思考：算法的主要计算量取决于什么？
    - 算法中的3重循环（分别对r、i 和 k）
    - 循环体内的计算量为：**O(1)**
    - 而3重循环的总次数为：**$O(n^3)$**
  - 因此算法的计算复杂度上界为：**$O(n^3)$**
  - 算法的空间复杂度？
    - 提示：填充两张表（m 和 S）
    - **$O(n^2)$**

小结：动态规划算法的基本要素

动态规划算法的基本要素

- 要素1：问题具有最优子结构性质
  - 最优子结构性质：可以分解为若干个规模较小的相同问题
    - 例如：矩阵连乘的计算次序问题？
    - 该问题的最优解包含着其子问题的最优解
  - 如何分析问题的最优子结构性质？
    - 首先假设由问题的最优解导出的子问题的解不是最优的
    - 然后证明在该假设下可构造出比原最优解更好的解
    - 通过矛盾法证明由最优解导出的子问题的解也是最优的

动态规划算法的基本要素

- 要素1：问题具有最优子结构性质
  - 解题方法：利用问题的最优子结构性质
    - 从子问题的最优解出发
    - 自底向上递归地构造出整个问题的最优解
  - 动态规划是解决多阶段决策最优化问题的思路而非算法
    - 动态规划程序设计往往是针对特定的最优化问题
    - 同一个问题可以有多种方式刻划它的最优子结构
    - 解题时需发挥想像力和创造性
    - 最优子结构是问题能用动态规划算法求解的前提！

动态规划算法的基本要素

- 要素2：问题具有重叠子问题性质
  - 递归求解问题时产生的子问题并不总是独立的
    - 子问题的重叠性导致有些子问题被反复计算多次
    - 通常独立的子问题个数随问题的规模呈多项式增长
  - 动态规划的特点
    - 对每个子问题只求解一次，并将结果保存在一个表格中
    - 当再次需要求解该子问题时可以用常数时间查表得到
    - 因此采用动态规划求解此类问题只需要多项式时间

2. 凸多边形最优三角剖分问题

Optimal Triangulation of a Convex Polygon

凸多边形最优三角剖分

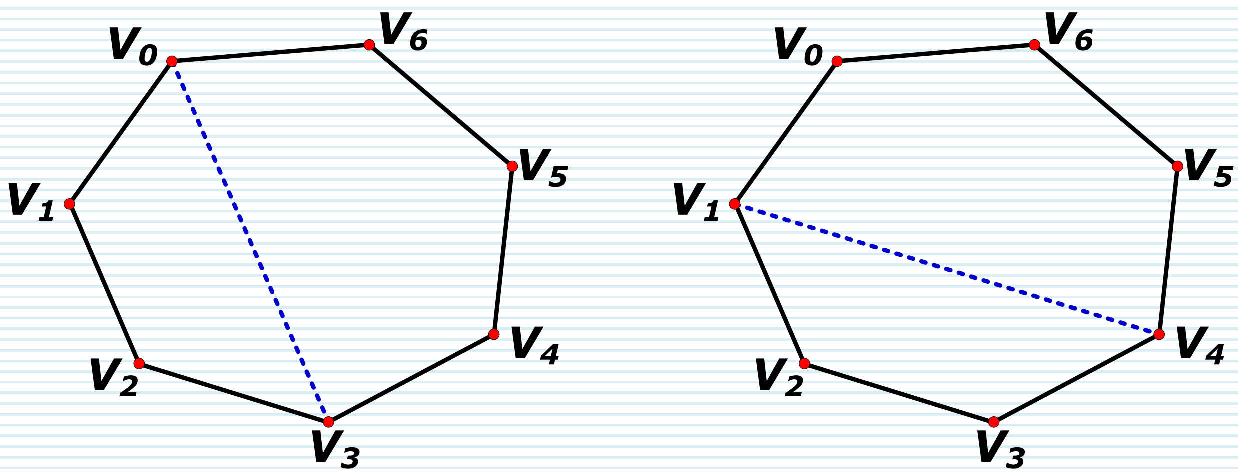
- 定义1：多边形
  - 平面上由一系列首尾相接的直线段组成的分段线性闭曲线
- 定义2：简单多边形
  - 多边形的边除了顶点外没有别的交点
- 定义3：凸多边形
  - 当一个简单多边形及其内部构成一个闭凸集时称为凸多边形
  - 凸集的含义：凸多边形边界或内部的任意两点所连成的直线段上的所有点均在凸多边形的内部或边界上
- 通常用顶点的逆时针序列表示凸多边形（约定：$v_0 = v_n$）
  - 即：$V=\{v_0, v_1, \ldots\ldots, v_{n-1}\}$ 表示具有n条边（v_0, v_1），（v_1, v_2），……，（v_{n-1}, v_n）的一个凸多边形

凸多边形最优三角剖分

凸多边形的分割

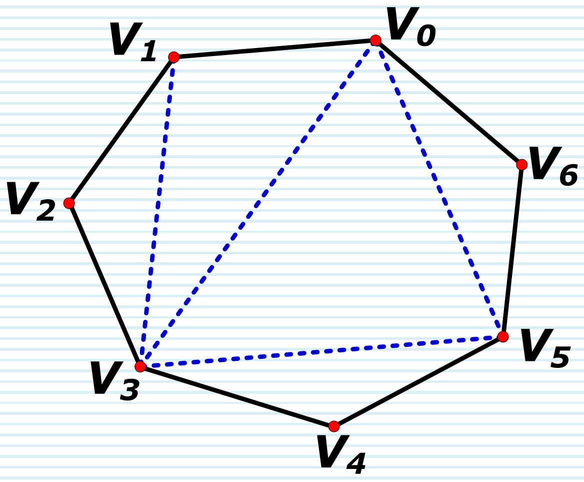
- 若 v_i 和 v_j 是多边形中两个不相邻的顶点
  - 则线段（v_i, v_j）称为多边形的一条**弦**
- 一条弦将多边形分割成两个多边形：
  - { v_i, v_{i+1}, ……, v_j } 和 { v_j, v_{j+1}, ……, v_i }

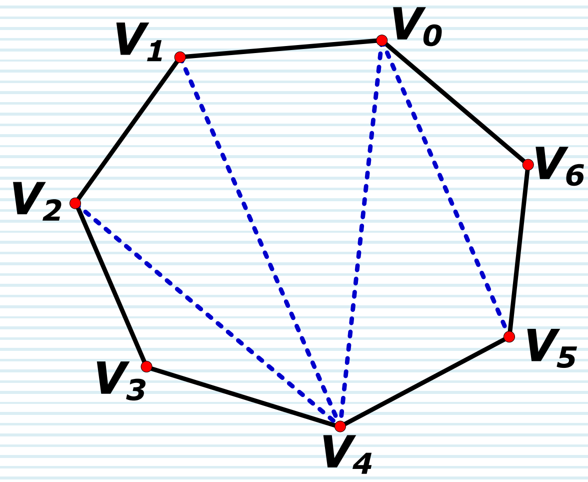
- 例1：{ v_0, v_1, v_2, v_3 } 和 { v_3, v_4, v_5, v_6, v_0 }



- 例2：{ v_1, v_2, v_3, v_4 } 和 { v_4, v_5, v_6, v_0, v_1 }

凸多边形最优三角剖分





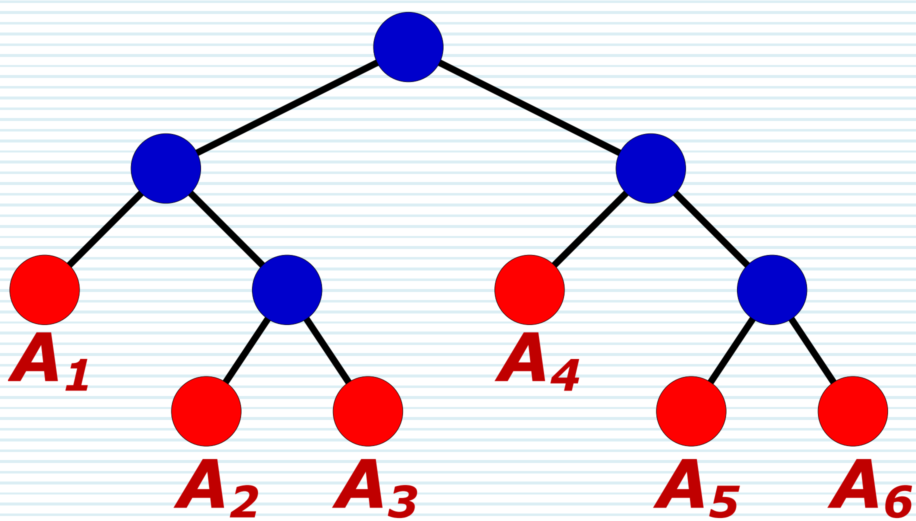
- **凸多边形的三角剖分**
  - 将多边形P分割成互不相交的三角形的弦的集合T
  - 在该剖分中各弦互不相交，且集合T已达到最大
- 在有n个顶点的凸多边形的三角剖分中
  - 恰有n-3条弦和n-2个三角形

凸多边形最优三角剖分

- 问题描述：对于给定的凸多边形P
  - 以及定义在由多边形的边和弦组成的三角形上的权函数W
  - 要求确定该凸多边形的三角剖分
    - 使得该三角剖分中诸三角形上权值之和为最小
- 三角形的权函数W可以有多种定义方式
  - 例如：$W(v_iv_jv_k)=|v_iv_j|+|v_jv_k|+|v_kv_i|$
    - 其中：$|v_iv_j|$ 表示顶点 v_i 到 v_j 的欧式距离
  - 则：对应于该权函数的最优三角剖分称为最小弦长三角剖分
  - 本节介绍的算法可以适用于任意权函数情况

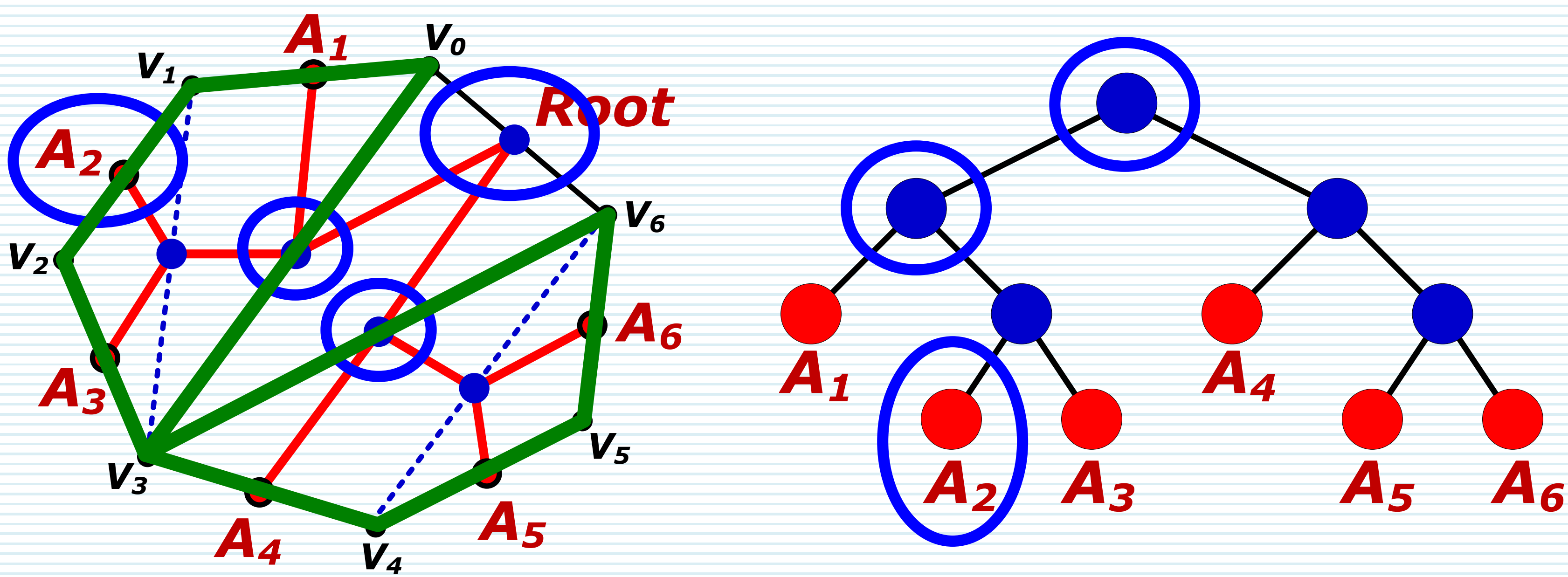
完全加括号表达式的语法树

- 矩阵连乘的最优计算次序等价于矩阵链的最优完全加括号方式
  - 一个表达式的完全加括号方式相当于一棵平衡二叉树
  - 例如：完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$
    - 可以表示为如下的平衡二叉树，其中叶节点为表达式中的元素，树根表示左右子树相结合
    - 这样的二叉树称为该表达式的语法树



凸多边形三角剖分与表达式完全加括号问题的语法同构性

- 凸多边形三角剖分也可以用语法树来表示（如图）
  - 该语法树的根节点为边（v_0, v_6）
  - 三角剖分中的弦组成其余的内节点（子树的根节点）
  - 多边形中除（v_0, v_6）外的各条边都是语法树的一个叶节点
  - 例如：以弦（v_0, v_3）和（v_3, v_6）为根的子树表示？
    - 凸多边形{ v_0, v_1, v_2, v_3} 和 { v_3, v_4, v_5, v_6}的三角剖分



凸多边形三角剖分问题与矩阵连乘问题的同构关系

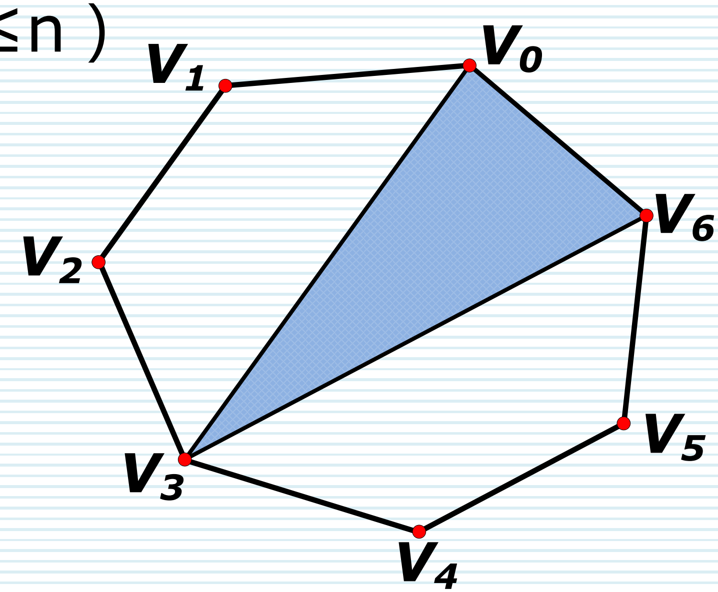
- n个矩阵乘积的完全加括号可以表示为一棵语法树
  - 思考：叶节点个数？
  - 该问题和有n个叶节点的语法树存在1:1对应关系
- 凸n边形的三角剖分可以表示为n-1个叶节点的语法树
  - 思考：叶节点个数？
  - 该问题和有n-1个叶节点的语法树存在1:1对应关系
- 推论： 这两个问题的可行解之间也存在1:1对应关系
  - 矩阵 A_i 对应于凸多边形中的一条边（v_{i-1}, v_i）
  - 每条弦（v_i, v_j）对应于一组矩阵的连乘积A[i+1, j]
- 思考：讨论这种同构关系的意义何在？

三角剖分的结构及其相关问题

- 矩阵连乘的最优计算次序问题是凸多边形最优三角剖分的特例
  - 对于给定的矩阵链：$(A_1\ A_2\ \ldots\ A_n)$
  - 定义一个与之相应的凸多边形：$P=\{v_0, v_1, \ldots\ldots, v_n\}$
  - 使得矩阵 A_i 与凸多边形的边（v_{i-1}, v_i）形成1:1对应
  - 若矩阵 A_i 的维数为：$p_{i-1} \times p_i$
  - 定义三角形（$v_iv_jv_k$）上的权值：$w(v_iv_jv_k)=p_i \times p_j \times p_k$
  - 则：凸多边形P的最优三角剖分所对应的语法树
  - 同时也给出了矩阵链$A_1A_2\ \ldots\ A_n$ 的最优完全加括号方式

凸多边形最优三角剖分的最优子结构性质

- 设：T为凸多边形P={v_0, v_1, ……, v_n}的一个最优三角剖分
- 不妨设：T包含三角形$v_0v_kv_n$（1≤k≤n）
- 则：T的权为三部分权之和（如图）
  - 三角形$v_0v_kv_n$的权
  - 子多边形{v_0,v_1,...v_k}的权
  - 子多边形{v_k,v_{k+1},...v_n}的权



- 断言：由T所确定的这两个子多边形的三角剖分也是最优的
  - 反证：若子多边形有更小权的三角剖分
  - 则：三部分的权值之和将小于T的权值
  - 则：T不是最优三角剖分，与前提假设矛盾

凸多边形最优三角剖分的递归结构

- 思考：问题的最优值如何定义？
  - 提示1：求解问题要用到最优子结构性质
  - 提示2：矩阵链的完全加括号问题与该问题同构
- 定义最优值：**t[i][j]**（$1 \le i \le j \le n$）
  - 为凸子多边形$P=\{v_{i-1},v_i,\ldots\ldots.v_j\}$三角剖分的最优值
    - 即：P的最优三角剖分所对应的权函数值
  - 则：原问题的最优权值为？ **t[1][n]**
    - 请问：这是几边形？ **凸(n+1)边形**
  - 思考：有没有什么情况没有考虑在内？
    - 设：退化的两顶点多边形{vi-1vi} 权值为0（t[i][i]=0）

凸多边形最优三角剖分的递归结构

- 设：t[i][j]为凸子多边形 **P**={v_{i-1},v_i,…v_j}三角剖分的最优值
- 则：当（j-i）≥1时：凸子多边形**P**至少有三个顶点
- 设：**k**为其中一个中间点（i≤ k < j）
  - 由最优子结构性质：t[i][j]的值应为三部分权值之和
    - 三角形 $v_{i-1}v_kv_j$ 的权值
    - 两个凸子多边形的最优权值：t[i][k]和t[k+1][j]
  - 思考：k 的可能位置有几个？ **j-i**
    - 问题转化为：在其中选择使得t[i][j]达到最小的位置
- 相应地得到t[i][j]的递归定义如下：

$$t[i][j]=\begin{cases} 0 & i=j \\ \min\limits_{i<=k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j \end{cases}$$

计算凸多边形最优三角剖分的最优值

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\left\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\right\} & i<j\end{cases}$$

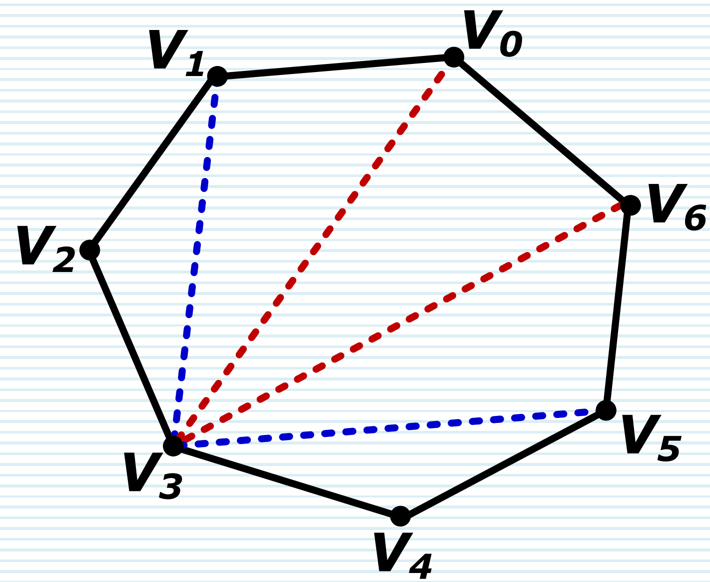
- 与矩阵连乘问题相比
  - t[i][j]与m[i][j]的递归式几乎形式上完全相同
  - 唯一的区别在于权函数的定义
- 因此：只需对MatrixChain算法做少量修改即可

```
void Triangulation(int *p, int num, int *T, int *s) {
    n = num +1;
    for (int i = 1; i <= num ; i++) T[i][i] = 0;
    for (int r = 2; r <= num; r++){
        for (int i = 1; i <= (num-r+1); i++) {
            int j = i+r-1;
            T[i][j] = T[(i+1)][j]+W(i-1, i, j);
            s[i][j] = i;
            for (int k = i+1; k < j; k++) {
                int u = T[i][k] + T[(k+1)][j]+W(i-1, k, j);
                if (u < T[i][j]) {
                    T[i][j] = u;  s[i][j] = k;
} } } } }
```

算法复杂度？ $O(n^3)$

构造凸多边形最优三角剖分（最优解）

S	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0



- 凸多边形最优三角剖分的最优解
  - 计算最优值t[1][n]时，设置数组S记录三角剖分信息
  - S[i][j]记录与（v_{i-1}, v_j）组成三角形的第三个顶点的位置
  - 可以在O(n)时间内构造出最优三角剖分当中的所有三角形

3. 最长公共子序列问题

Longest Common Subsequence

最长公共子序列

- 子序列的概念
  - 给定序列 $\mathbf{X=\{x_1, x_2, \ldots\ldots, x_n\}}$
  - X的子序列是从该序列中删去若干元素后得到的序列
- 注意：子序列中的元素顺序与给定序列保持一致
  - 称：$\mathbf{Z=\{z_1, z_2, \ldots, z_k\}}$ 是 X 的子序列
  - 是指：存在一个严格递增的下标序列：$\mathbf{\{i_1, i_2, \ldots, i_k\}}$
  - 使得：对于所有 $j=1,2,\ldots,k$ 有：$z_j = x_{i_j}$
- 子序列示例：给定序列X={A,**B**,**C**,B,**D**,A,**B**}
  - 则：Z={B,C,D,B} 是的子序列
  - 相应的递增下标序列为{2,3,5,7}

最长公共子序列

- 公共子序列（Common Subsequence）
  - 给定两个序列：**X** 和 **Y**
  - 若存在另一序列 **Z**：既是**X**的子序列，又是**Y**的子序列
  - 则称：Z是序列X和Y的公共子序列
- 最长公共子序列（LCS）问题
  - 给定两个序列：**X** 和 **Y**
    - $\mathbf{X=\{x_1, x_2, \ldots\ldots, x_m\}}$
    - $\mathbf{Y=\{y_1, y_2, \ldots\ldots, y_n\}}$
  - 找出X和Y的一个最长公共子序列

最长公共子序列问题

- 问题分析
  - 要求找出“一个” 而不是“唯一的”最长公共子序列
  - 公共子序列在原序列当中不一定是连续的
    - X：{ A B C B D A B }
    - Y：{ C B D C A B A }

$$\left.\begin{array}{l}X:\{A\ B\ C\ B\ D\ A\ B\}\\ Y:\{C\ B\ D\ C\ A\ B\ A\}\end{array}\right\}\mathrm{LCS}(X,Y)=\{B\ C\ B\ A\}$$

- 解决思路1：穷举搜索
  - 枚举**X**的所有子序列：分别检查它是否是**Y**的子序列
    - 如果是，则记录当前最长的公共序列
  - 算法复杂度分析　**$O(n2^m)$**
    - 思考：X有多少个可能的子序列？　**2^m个**　**$O(n)$**
    - 思考：对每条子序列检查是否是Y的子序列需时多少？

最长公共子序列问题

- 考察：最长公共子序列问题是否具有最优子结构性质
  - 给定：$X=\{x_1, x_2, \ldots\ldots, x_m\}$ 和 $Y=\{y_1, y_2, \ldots\ldots, y_n\}$
  - 设：它们的一个LCS为 $Z=\{z_1, z_2,\ldots, z_k\}$ ，则：
    - 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LCS
    - 若：$x_m \neq y_n$ 且 $z_k \neq x_m$
      - 则： Z 是 X_{m-1} 和 Y 的LCS
    - 若：$x_m \neq y_n$ 且 $z_k \neq y_m$
      - 则： Z 是 X 和 Y_{n-1} 的LCS
  - 可见：LCS(X,Y) 包含了这两个序列的前缀子序列的LCS
  - 因此：最长公共子序列问题具有最优子结构性质

动态规划求解LCS问题

- 定义递归解（分析子问题的递归结构）
  - 由LCS问题的最优子结构性质可知：为求解X和Y的一个LCS
    - 当 $x_m = y_n$ 时：须找出 **LCS(X_{m-1}，Y_{n-1})**
      - 将 x_m（或 y_n ）添加到这个LCS上得到**LCS(**X,Y**)**
    - 当 $x_m \neq y_n$ 时，须解决两个子问题：
      - 找出一个**LCS(X_{m-1} ,Y)**和一个**LCS(X, Y_{n-1})**
      - 这两个LCS中较长的一个就是**LCS(**X,Y**)**
  - 由此递归结构可以看出LCS问题具有重叠子问题性质
    - **LCS(X_{m-1},Y)**和**LCS(X, Y_{n-1})**都包含一个公共子问题
    - 即都需要求解： **LCS(X_{m-1}, Y_{n-1})**

动态规划求解LCS问题

- 建立递归关系（递归地定义最优值）
  - 用**c[i][j]**表示序列 X_i 和 Y_j 的最长公共子序列的**长度**
  - 其中： $X_i=\{x_1, x_2, \ldots\ldots, x_i\}$；$Y_j=\{y_1, y_2, \ldots\ldots, y_j\}$
    - 若：其中一个序列长度为0（ i=0或j=0 ）
    - 则：LCS（X,Y）的长度也是0
  - 根据最优子结构性质建立递归关系如下：

$$c[i][j]=\begin{cases} 0 & i=0, j=0 \\ c[i-1][j-1]+1 & i,j>0; x_i=y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i,j>0; x_i\neq y_j \end{cases}$$

  - 如直接用递归算法求解，则时间随输入规模呈指数增长

动态规划求解LCS问题的最优值

- 分析子问题空间
  - **思考**：总共包含多少个不同的子问题？
    - 总共Θ(mn)个不同的子问题……子问题空间不大
  - 因此考虑采用动态规划法：自底向上计算最优值
- 设置两个数组作为输出
  - 用**c[i][j]**表示序列 $\mathbf{X}_i$ 和 $\mathbf{Y}_j$ 的最长公共子序列的长度
    - 问题的最优值记为**c[m][n]**，即LCS(X,Y)的长度
  - 用**b[i][j]**记录**c[i][j]**是从哪一个子问题的解得到的
    - **思考**：怎样利用数组b构造最长公共子序列（最优解）？

根据LCS问题的最优值构造最优解

- 思考：对最优解的推导应该从哪里开始？
  - 提示：c[m][n]表示LCS(X,Y)的长度
  - 答案：推导应首先从b[m][n]开始
- 思考： b[m][n]中的值是什么？
  - b[i][j] = 1：表示c[i][j]从左上方**c[i-1][j-1]**得到
  - b[i][j] = 2：表示c[i][j]从正上方**c[i-1][j]**得到
  - b[i][j] = 3：表示c[i][j]从正左方**c[i][j-1]**得到
- 思考：怎样构造出LCS序列？
  - 按照b[i][j]的值代表的方向往回搜索
  - 当b[i][j] = 1，以i 和 j 作为序列下标可以构造出LCS

示例：动态规划求解LCS问题

C	x_i	B	D	C	A	B	A
y_j	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

x	1	2	3	4	5	6	7
	A	B	C	B	D	A	B

y	1	2	3	4	5	6
	B	D	C	A	B	A

求得：|LCS(x,y)|=4

示例：动态规划求解LCS问题

b	x_i	B	D	C	A	B	A
y_j	0	0	0	0	0	0	0
A	0	2	2	2	1	3	1
B	0	1	3	3	2	1	3
C	0	2	2	1	3	2	2
B	0	1	2	2	2	1	3
D	0	2	1	2	2	2	2
A	0	2	2	2	1	2	1
B	0	1	2	2	2	1	2

x	1	2	3	4	5	6	7
	A	B	C	B	D	A	B

y	1	2	3	4	5	6
	B	D	C	A	B	A

思考：怎样改进？ **LCS(x,y) =**

B	C	B	A

动态规划法求解LCS问题

```
int lcs (char x[], char y[], int b[], int c[]){
    // 将辅助表c[i][j]的第一行和第一列初始化为0
    for(int i=1; i <= m; i++){
        for(int j=1; j <= n; j++){
            if(x[i]==y[j]){
                c[i][j] = c[i-1][j-1]+1;  b[i][j] = 1; }
            else if(c[i-1][j] >= c[i][j-1]){
                c[i][j] = c[i-1][j];  b[i][j] = 2; }
            else {
                c[i][j] = c[i][j-1];  b[i][j] = 3; }
        }
    }
    return c[m][n];
}
```

算法复杂度？

O(mn)

从最优值得到最优解

```
void traceback(int i, int j, char x[], int b[]){
    if(i==0||j==0)  return;
    if(pb[i][j]==1){
        traceback(i-1, j-1, x, b);
        output(x[i]);
    }
    else if (pb[i][j]==2){
        traceback(i-1, j, x, b);
    }
    else {
        traceback(i, j-1, x, b);
    }
}
```

初始调用参数：traceback(m, n, x, b);

4. 最大子段和问题

Maximum Sub-Sequence Sum

最大子段和问题

问题描述

- 给定n个整数（可能为负数）组成的序列 **$a_1,a_2,\ldots,a_n$**
- 求该序列形如下式的子段和的最大值：$S=\max\sum_{k=i}^{j}a_k$
- 当所有整数均为负整数时定义其最大子段和为0
- 据此定义该问题的最优值为：

$$S=\max\left\{0,\ \max_{1\le i\le j\le n}\sum_{k=i}^{j}a_k\right\}$$

- 例如：(a1,a2,a3,a4,a5,a6)=(-2, 11, -4, 13, -5, -2)
- 该序列的最大子段和为：$S=\sum_{k=2}^{4}a_k=20$

最大子段和问题：简单算法

```
int MaxSum(int n, int *a, int *besti, int *bestj){
    int sum = 0;
    for(int i=1; i <= n; i++){
        for(int j=i; j <= n; j++){
            int tmp = 0;
            for(int k=i; k<=j; k++){
                tmp += a[k];
            }
            if(tmp > sum) {
                sum = tmp;  *besti=i;  *bestj=j;
            }
        }
    }
    return sum;
}
```

思考：怎样改进？

$$\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$$

时间复杂度：O（n^3）

最大子段和问题：简单算法（改进版）

```
int MaxSum(int n, int *pa, int *besti, int *bestj){
    int sum = 0;
    for(int i=1; i <= n; i++){
        int tmp = 0;
        for(int j=i; j <= n; j++){
            tmp +=pa[j];
            if(tmp > sum) {
                sum = tmp;  *besti=i;  *bestj=j;
            }
        }
    }
    return sum;
}
```

时间复杂度？ O（n^2）

思考：能否改进？

最大子段和问题：分治算法

- 问题分析
  - 如果将序列a[1:n]分为等长的两段？
    - 设：m = n/2　得到：a[1:m] 和 a[m+1:n]
  - 思考：分段之后如何求解原问题？
    - 首先分别求出a[1:m]和a[m+1:n]的最大子段和
  - 分析：原问题的实际情况只可能是如下三种情形之一
    - a[1:n]的最大子段和与a[1:m]的最大子段和S1相同
    - a[1:n]的最大子段和与a[m+1:n]的最大子段和S2相同
    - a[1:n]的最大子段和产生于跨越两段分界点的子序列

最大子段和问题：分治算法

- 考虑第三种情况　　　时间复杂度？**O（nlogn）**
  - a[1:n]的最大子段和产生于跨越两段分界点的子序列
  - 则：位于边界的a[m]和a[m+1]在最优子序列中
  - 思考：这意味着什么？　　**思考：能否改进？**
    - 提示：这个最优子序列在a[1:m]中的部分有什么特点？
  - 思考：如何实现问题求解？（算法设计）
    - 分别求得包含a[m]和a[m+1]的极大子段和S_{m1}和S_{m2}
    - 则：同时包含这两个点的极大子段和 Sm = $S_{m1}+S_{m2}$
    - a[1:n]的最大子段和 S = max{S1, S2, Sm}

最大子段和问题：动态规划

- 思考：如果采用动态规划思想，怎样看待该问题？
  - 提示：首先考虑如何（递归）定义问题的最优值
  - 提示：序列a[1:n]的最大子段和可以表示为

$$S=\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\left(\max_{1\le i\le j}\sum_{k=i}^{j}a[k]\right)$$

  - 设：序列a[1:j]中包含a[j]的最大子段和为b[j]

$$b[j]=\max_{1\le i\le j}\left\{\sum_{k=i}^{j}a[k]\right\}\quad (1\le j\le n)$$

  - 有：
    - 若b[j-1]>0：b[j] = b[j-1]+a[j]
    - 若b[j-1]≤0：b[j] = a[j]

最大子段和问题：动态规划

- 设：序列a[1:j]中包含a[j]的最大子段和为b[j]

$$b[j]=\max_{1\le i\le j}\left\{\sum_{k=i}^{j}a[k]\right\}\quad (1\le j\le \mathrm{n})$$

- 由上式得到b[j]的动态规划递归式：

b[j]=max{ b[j-1]+a[j], a[j] } （1≤j≤n）

- 思考： b[j]与求解目标最优值S有什么关系？

• 提示：序列a[1:n]的最大子段和可以表示为

$$S=\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\left(\max_{1\le i\le j}\sum_{k=i}^{j}a[k]\right)$$

最大子段和问题：动态规划算法

```
int MaxSum (int *a, int n) {
    int sum = 0, b = 0;
    for(int i = 1;  i <= n;  i++){
        if(b > 0)
            b += a[i];
        else
            b = a[i];
        if(b > sum)
            sum = b;
    }
    return sum;
}
```

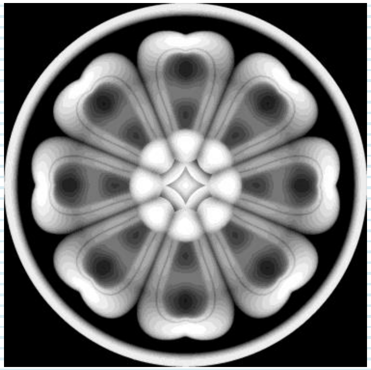
时间复杂度？ O (n)

5. 图像压缩问题

（Image Compression Problem）

图像压缩问题

- 灰度图
  - 灰度是在白色和黑色之间分的若干个等级
    - 其中最常用的是256级灰度图
  - 灰度就是没有色彩，RGB色彩分量全部相等
    - 例如：RGB(100,100,100) 代表灰度为100
    - 例如：RGB(50,50,50) 代表灰度为50
  - 在计算机中常用像素点的灰度值序列来表示图像
    - $P = \{p_1, p_2, \ldots p_n\}$
  - 灰度图在医学、航天等领域有着广泛的应用



256×256

图像压缩问题

- 可以将彩色图像（RGB三色图）转换为灰度图
  - 常用方法1：比例法
    - 根据人眼对红绿蓝的敏感程度
    - 使用以下比例式进行转换
    - Gray = R×0.3+G × 0.59+B × 0.11
    - 这也是最常用的一种转换
  - 常用方法2：平均值法
    - Gray =（R+G+B）/ 3
    - 即：取红绿蓝三色的平均值为灰度

图像压缩问题

- 灰度图的压缩
  - 例如：图像A的像素点灰度值序列为：$\{p_1,p_2,\ldots p_n\}$
  - 其中：像素点灰度值取值范围[0-255]
  - 每个像素点的灰度值表示为8位二进制数
  - 减少表示像素点的位数，可以降低图像的空间占用需求

图像压缩问题

- 图像的变位压缩存储
  - 对于给定像素点序列：$\{p_1, p_2, \ldots, p_n\}$
  - 将其分割成**m**个连续分段：$S_1, S_2, \ldots, S_m$
    - 设第 i 个像素段 S_i 中：有**N[i]**个像素（1≤i≤m）
    - 设第 i 个像素段 S_i 中：每个像素都用**b[i]**位表示
  - 前（i-1）个分段的像素总数：

$$t[i] = \sum_{k=1}^{i-1} N[k], \ (1 \le i \le m)$$

  - 第 i 个像素段序列 S_i 可以表示为：

$$S_i = \left\{ p_{t[i]+1}, \ldots, p_{t[i]+N[i]} \right\} \quad (1 \le i \le m)$$

图像压缩问题

- 图像的变位压缩存储格式
  - 设： h_i 为分段 S_i 中最大灰度值所对应的二进制数的位数

$$h_i = \left\lceil \log\left(\max_{t[i]+1 \le k \le t[i]+N[i]} p_k + 1 \right) \right\rceil$$

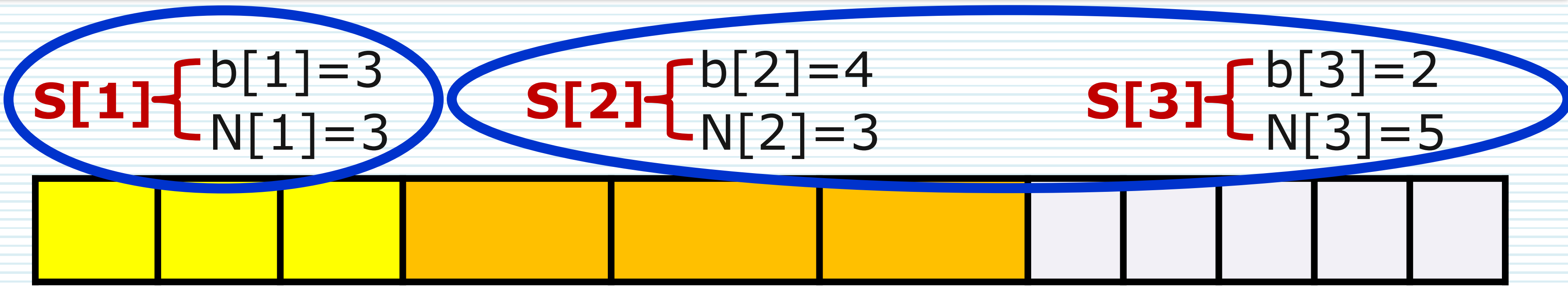
  - 由b[i] 的定义可知：$h_i \le b[i] \le 8$
    - 因此：表示b[i]需要用3位（bit）
  - 如果进一步限制：分段序列中的像素个数不超过255个
    - 即：$1 \le N[i] \le 255$，则需要用8位来表示N[i]
  - 像素段 S_i 所需存储空间为： $N[i] * b[i] + 11$
  - 压缩后像素序列$\{p_1, p_2, \ldots, p_n\}$所需存储空间为：

$$\sum_{i=1}^{m} N[i] \times b[i] + 11m$$

图像压缩问题

- 问题提出：对于给定像素序列 $P = \{p_1, p_2, \ldots, p_n\}$
  - 要求确定其最优分段，使得依此分段所需的存储空间最少
  - 并且要求每个分段的长度不超过256位
- 问题示例
  - 设：P = {10，12，15，255，1，2，1，1，2，2，1，1}
    ① S1 = {10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1}
    ② 分成12个组，每组仅包含一个像素
    ③ S1 = {10, 12, 15} S2 = {255} S3 = {1, 2, 1, 1, 2, 2, 1, 1}
  - 所需存储空间
    - 分法1：8×12 + 11×1 = **107**
    - 分法2：4×3 + 8×1 + 1×5 + 2×3 + 11×12 = **163**
    - 分法3：4×3 + 8×1 + 2×8 + 11×3 = **69**

图像压缩问题

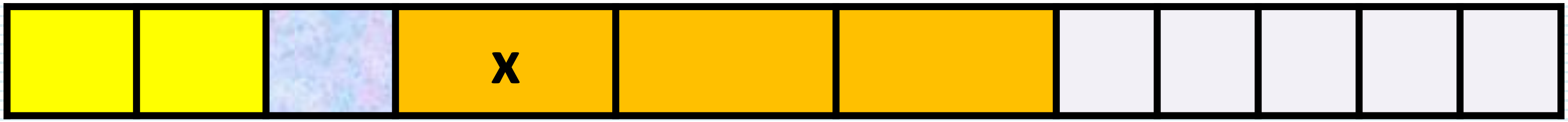


- 图像压缩问题的最优子结构性质
  - 设：N[i]，b[i] 是 $\{p_1, p_2, \ldots, p_n\}$ 的最优分段($1 \le i \le m$)
  - 则：N[1]，b[1] 是 $\{p_1, p_2, \ldots, p_{N[1]}\}$的最优分段
  - 且：N[i]，b[i] 是 $\{p_{N[1]+1}, \ldots, p_n\}$ 的最优分段（$2 \le i \le m$）
  - 即：图象压缩问题满足最优子结构性质

思考：这个最优子结构性质对于解决问题有何意义？

图像压缩问题的最优子结构性质

S[1] b[1]=3 N[1]=3　　S[2] b[2]=4 N[2]=3　　S[3] b[3]=2 N[3]=5



- 猜测：最优划分结果是否是（唯一）确定的？
- 考虑S[1]：若最优分段不是确定的，则存在如下两种可能性
  - S[1] 可以被进一步划分
    - 若分段后长度之和小于S[1]，则与最优划分假设矛盾
  - S[1] 可以被延长
    - 不妨设 x 并入 S[1] 后，同样得到最优分段结果
    - 注意由最优划分假设可知： b[1] ≠ b[2]

图像压缩问题的最优子结构性质

- 思考：讨论S[1]的边界，意义何在？
  - 意义（1）：再次确认该问题的最优子结构性质
  - 意义（2）：提供解题思路：实现顺序递推求解

- 若： b[1]>b[2]
  - 若N[2]>1：则x加入S[1]会导致存储总长度增加，矛盾
  - 若N[2]=1：0<b[1]-b[2]<7<11，总长度减少，矛盾
- 若： b[1]<b[2]
  - 若N[2]=1：**需增大b[1]**：S[1]的增量==b[2]+11？
  - 若N[2]>1，且b[1]可容纳x：存储总长度减少，矛盾
  - 若N[2]>1，且b[1]无法容纳x：**需增大b[1]**：矛盾？

图像压缩问题的最优子结构性质

示例：P = {10，12，15，255，1，2，1，1，2，2，1，1}

P	10	12	15	255	1	2	1	1	2	2	1	1
b	4	4	4	8	1	2						
S	15	**4 + 11 = 15**										
S	15	19	**15 + 4 + 11 = 30**									
S	15	19	23		**4 × 8 + 11 = 43**							
S	15	19	23	42	**23 + 8 + 11 = 42**							

$$b_i = \lceil \log_2 (p_i + 1) \rceil$$

图像压缩问题的最优子结构性质

示例：P = {10，12，15，255，1，2，1，1，2，2，1，1}

P	10	12	15	255	1	2	1	1	2	2	1	1
b	4	4	4	8	1	2	1	1	2	2	1	1
S	15	19	23	42	50							
S	15	19	23	42	50	57						
S	15	19	23	42	50	57	59					
S	15	19	23	42	50	57	59	61	63	65	67	69

图像压缩问题的最优子结构性质

P	10	12	15	255	1	2	1	1	2	2	1	1
b	4	4	4	8	1	2	1	1	2	2	1	1
S	15	19	23	42	50	57	59	61	63	65	67	69

P	10	12	15	255	1	2	63	1	2	2	1	1
b	4	4	4	8	1	2	6	1	2	2	1	1
S	15	19	23	42	50	57	66	74	81	83	85	87

图像压缩问题

递归定义最优值

- 设：数组元素 S[i] 表示最优分段的存储长度
- 设： $bm(i,j)=\left\lceil \log\left(\max_{i\le k\le j}\{p_k\}+1\right)\right\rceil \quad (1\le i\le n)$
  - 从像素 i 到 j 中选择像素灰度值 pi 最大的值
  - 求出：表示该像素点最少需要的二进制位数
- 由最优子结构性质易知

$$S[i]=\min_{1\le k\le \min\{i,256\}}\{S[i-k]+k\times bm(i-k+1,i)\}+11$$

图像压缩问题

S	15	19	23	42	50	57	66	74	81	83	85	87
A	1	2	3	1	2	2	4	5	2	3	4	5

- 构造最优解
  - 用 A[i] 记录最优分段的信息 …… 怎样查看？
    - 提示：最优分段的最后一段的信息记录在哪里？
    - 答案：A[n]存储最后一段中的像素个数
  - 思考：倒数第二个最优分段的信息记录在哪里？
    - 段长度存储于： A[n-A[n]]
  - 以此类推，可以在O(n)时间内构造出最优解

```
void compress(int *b, int *s, int *pn, int n){
    s[0] = 0;    // s保存最优分段的存储长度（bit位数）
    for(int i = 1; i < n+1; i++){
        int bm = b[i];              // b[i]存放每个像素实际所需长度
        A[i] = 1;                   // 将当前像素点作为独立分段
        s[i] = s[i-1] + bm;         // 将新分段长度累加到最优分段
        for(int k = 2; k <= i && k < 256; k++){
            if(bm < b[i-k+1]){      // 从i-1往前扫描b
                bm = b[i-k+1]; }
            if(s[i] > s[i-k] + k*bm){
                s[i] = s[i-k] + k*bm;
                A[i] = k;}
        }
        s[i] += 11;
} }
```

图像压缩问题

算法复杂度分析

- Compress算法的基本思想是逐一确定像素点的分段归属
- 算法的时间复杂度？
  - 求解最优分段的算法中对k的循环次数不超过256
  - 故对每个像素点 i，可在O(1)时间完成对S[i]的计算
  - 因此：整个算法的时间复杂度为：O(n)
- 算法的空间复杂度？
  - 需要三个辅助数组（b，S，A）：O(n)

6. 0/1背包问题

（0/1 Knapsack Problem）

0-1背包问题

- 问题描述
  - 给定：n种物品和一个背包
    - 物品 i 的重量是 w_i，其价值为 v_i
    - 背包的容量为：Capacity
  - 约束条件：
    - 对于每种物品，旅行者只有两种选择：放入或舍弃
    - 每种物品只能放入背包一次
  - 问题：如何选择物品，使背包中物品的总价值最大?

0-1背包问题

0-1背包问题的形式化描述

- 优化目标函数： $\max\left(\sum_{i=1}^{n} v_i x_i\right)$
- 其中： x = (x$_1$, x$_2$, …, x$_n$) 为 n 元 0-1 向量
- 约束条件： $\begin{cases} \sum_{i=1}^{n} w_i x_i \le Capacity \\ x_i \in \{0,1\}, \quad 1 \le i \le n \end{cases}$

0-1背包问题

递归定义最优值

- 设所给0-1背包问题的子问题的最优值为：m(i，c)
  - 即m(i，c)是如下0-1背包问题的最优值：
    - 背包容量为 c，可选择物品为{ i，i+1，…，n}
  - 显然： 0-1背包问题具有最优子结构性质
- 根据最优子结构性质可以建立如下递归式：

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

0-1背包问题示例

设：Cap=10, w={2,2,6,5,4}, v={6,3,5,4,6}

	weight	value	0	1	2	3	4	5	6	7	8	9	10
1	2	6	0	0	6	6	9	9	12	12	15	15	15
2	2	3	0	0	3	3	6	6	9	9	9	10	11
3	6	5	0	0	0	0	6	6	6	6	6	10	11
4	5	4	0	0	0	0	6	6	6	6	6	10	10
5	4	6	0	0	0	0	6	6	6	6	6	6	6

x={1, 1, 0, 0, 1} 所选择物品为：1 2 5

```
void knapsack(int*v, int *w, int *m, int cap, int len){
    int cmax = min(w[n]-1, cap);    // 将最后一个物品放入包中
    for(int i = 0; i <= cmax; i++){
        m[n][i] = 0;                // 最后一行的前 cmax 个元素
    }
    for(int i = w[n]; i <= cap; i++){
        m[n][i] = v[n];             // 思考：if(w[n]==cap)？
    }
    for(int i = n-1; i >= 1; i--){      // 从后向前依次将剩余物品放入包中
        cmax = min(w[i]-1, cap);
        for(int c = 0; c <= cmax; c++){     // 剩余容量不足
            m[i][c] = m[(i+1)][c];
        }
        for(int c = w[i]; c <= cap; c++){   // 剩余容量允许
            m[i][c] = max( m[(i+1)][c], m[(i+1)][(c-w[i])]+v[i]);
        }
    } return;}
```

0-1背包问题

- 算法复杂度分析

  - 根据m(i，c)的递归式

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

  - 容易看出算法的计算复杂度为：O(n×c)
  - 显然：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Ω(n×2^n)计算时间
  - 算法的空间复杂度： O(n×c)

7．最优二叉查找树

（Optimal Binary Search Tree）

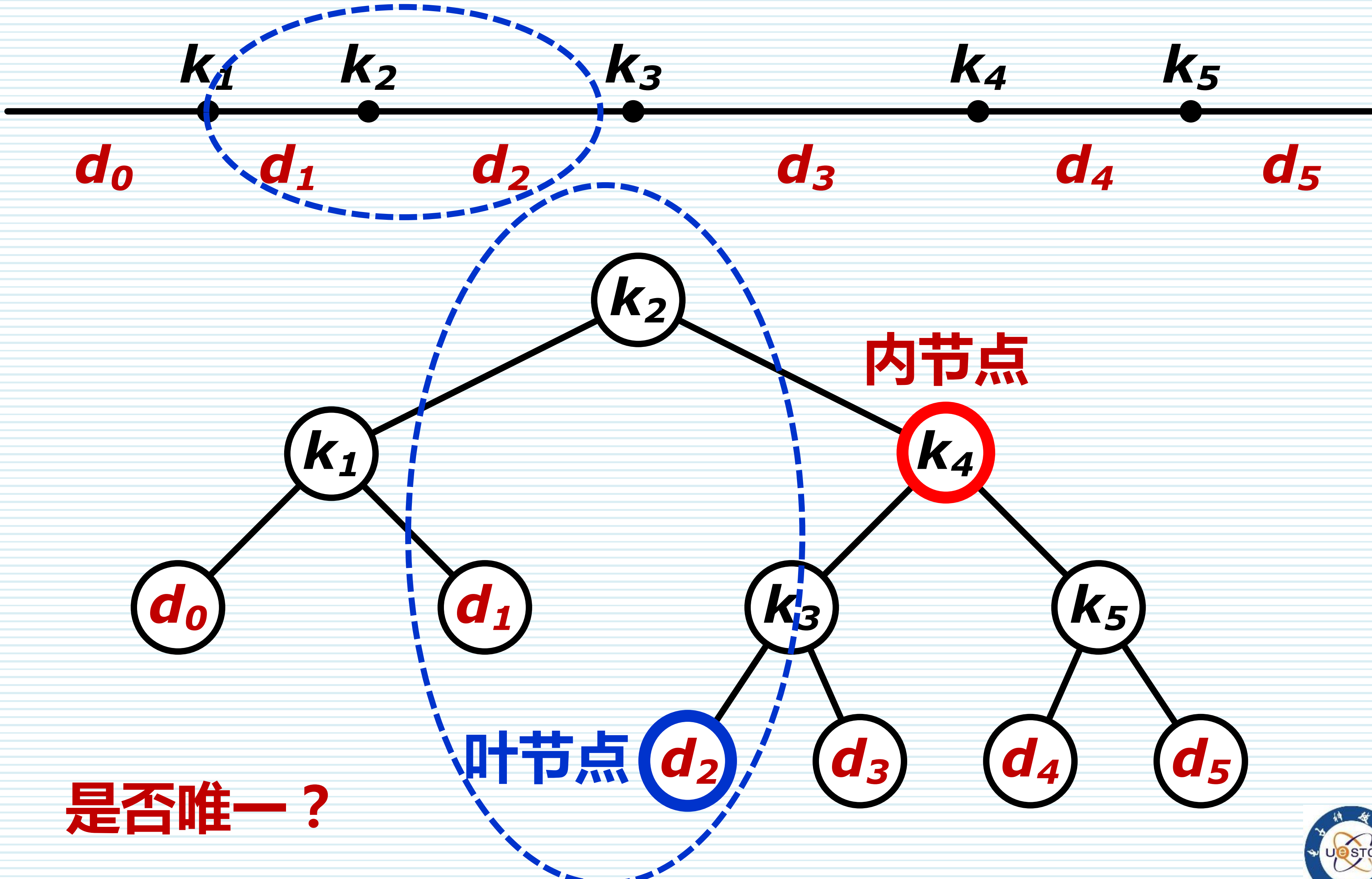
最优二叉查找树

- 二叉查找树（也称为二叉排序树，或二叉搜索树）
  - 设：S = $\{k_1, k_2, \dots k_n\}$是n个互异的关键字组成的有序集
    - 且：$k_1 < k_2 < \dots < k_n$
  - 二叉查找树：利用二叉树的节点存储有序集S中的元素
- 二叉查找树的性质
  - 二叉查找树BST中任意节点
    - 大于其左子树中任意节点；小于等于右子树中任意节点
  - 某些搜索的值可能不在BST内
    - 因此BST中有n+1个“虚拟键”（叶结点）
    - 所有关键字k_i为内部结点，每个虚拟键d_i构成一个叶结点
    - 例如：d_0代表所有小于k_1 的值，d_n代表所有大于k_n的值

最优二叉查找树

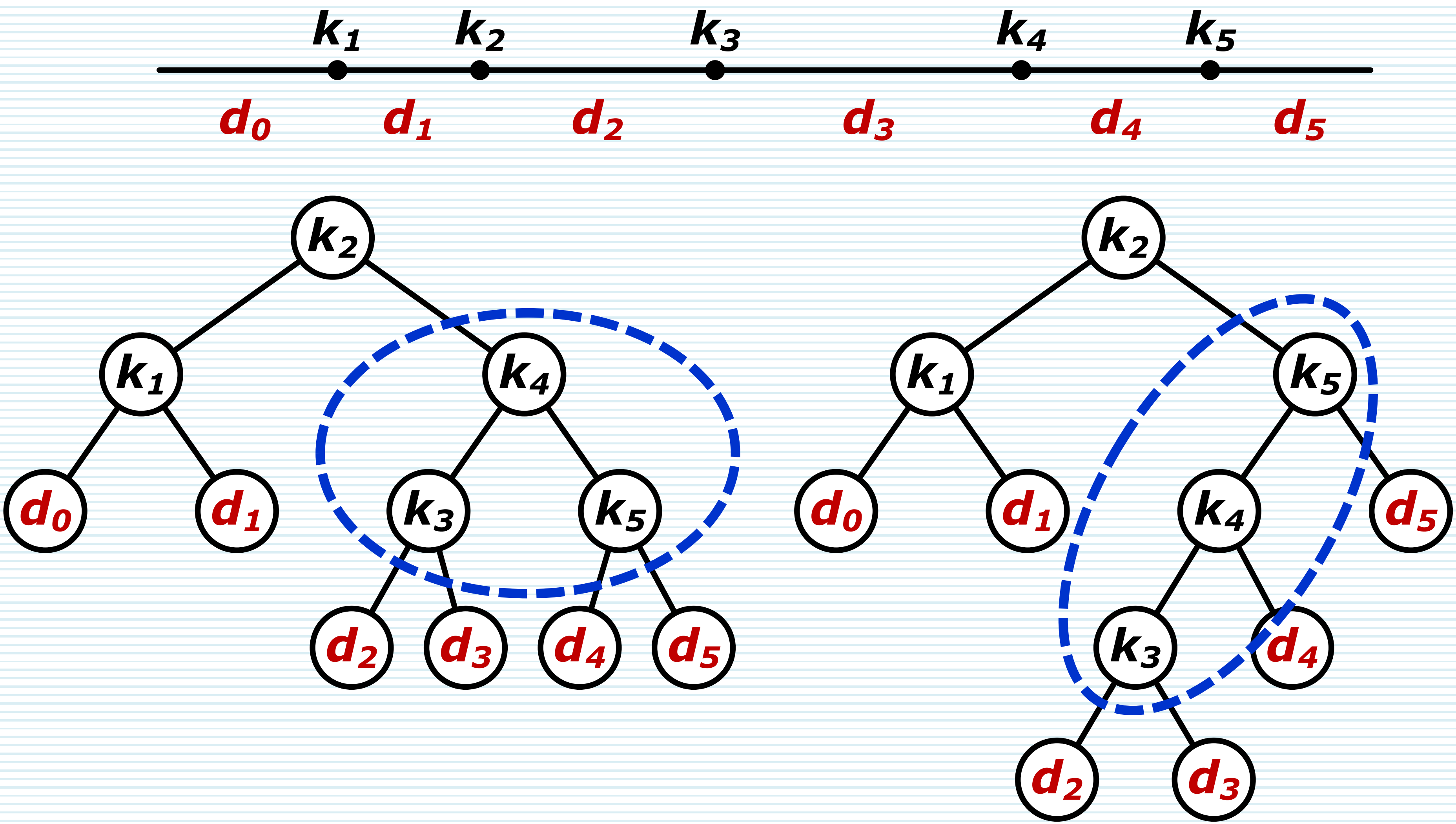
- 二叉查找树的性质
  - 所有关键字k_i为内部结点，每个虚拟键d_i构成一个叶结点
  - 规定：二叉查找树的叶节点是形如(d_i, d_{i+1})的开区间
    - 以符号d_i表示虚拟的叶节点
    - 约定：$d_0 = -\infty$; $d_{n+1} = \infty$
  - 在二叉查找树中搜索一个元素 k，返回的结果有两种情况
    - 在二叉查找树的内节点中找到：$k_i == k$
    - 在二叉查找树的叶节点中确定：$k \in (d_i, k_{i+1})$

二叉查找树



是否唯一？

二叉查找树



在两棵树上对某些元素进行查找，比较的次数是不同的

二叉查找树的期望搜索代价

- 对于有序集S = {k1, k2, … kn}构成的二叉查找树 T
  - 对于任意给定的关键字 k
    - 设：在其中找到元素k_i == k的概率为p_i
    - 设：返回k ∈ (k_i, k_{i+1}) 的概率为q_i
    - 则：$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$ 查找成功和失败的概率之和为1
  - 设：内节点k_i 的深度为H(k,i)；叶节点d_i 的深度为H(d,i)
    - 则：$E(T) = \sum_{i=1}^{n} (H(k,i)+1) \cdot p_i + \sum_{i=0}^{n} (H(d,i)+1) \cdot q_i$
    - E(T) 称为二叉查找树T的期望搜索代价
    - 表示：在T中做一次查找所需的平均比较次数

最优二叉查找树

- 最优二叉查找树问题定义
  - 对于有序集S = $\{k_1, k_2, \ldots k_n\}$构成的二叉查找树集合{T}
  - 若：S中元素的存取概率分布为（$p_0, q_0, \ldots, p_n, q_n$）
  - 目标：在集合{T}中找出一棵具有最小期望代价的BST
- 最优二叉查找树问题具有最优子结构性质
  - 若：最优二叉查找树T有一棵包含关键字$\{k_i,\ldots,k_j\}$ 的子树T’
    - 相应的最优二叉树叶节点为：$\{ k_{i-1}, k_i \ldots, k_j\}$
  - 则：T’ 对于该关键字集合构成的子问题也必定是最优的
  - 证明：如果存在T”比T’更优，用T”替换T中的T’
  - 从而产生比T更优的树，这与T的最优性质相矛盾

最优二叉查找树

- 最优子结构分析
  - 对于给定关键字序列{$k_i,\ldots,k_j$ }（1≤i≤j≤n）
  - 假设：kr 是包含该序列的一棵最优子树的根（i≤r≤j）
    - 根 kr 的左子树包含：k_i, …, k_{r-1} （以及$d_{i-1},\ldots,d_{r-1}$）
    - 根 kr 的右子树包含：$k_{r+1},\ldots,$ k_j （以及d_r, …, d_j）
  - 问题的解法
    - 依次检查所有的侯选根 （设为 kr ）
    - 确定包含关键字$k_i,\ldots,k_{r-1}$和$k_{r+1},\ldots,k_j$ 的最优BST
    - 就可以保证找到最优二叉查找树
    - 问题的关键：确认“权重”函数（子树的期望代价）

最优二叉查找树

- 最优值的递归表达式
  - 设T′为包含关键字{k_i,…,k_j } 的最优BST（1≤i≤j≤n）
    - 定义E[i, j]为搜索最优二叉查找树T的期望代价
  - 考虑到特殊情况：j = i-1 （此时只有虚拟节点d_{i-1} ）
    - 期望的搜索代价为： $E[i,i-1]=q_{i-1}$
  - 因此选取子问题域为：寻找一棵包含关键字{k_i,…,k_j} 的最优二叉查找树，其中： 1≤i，j≤n，i-1 ≤ j
  - 当 i ≤ j 时：需要从{ki,…,kj }中选择一个根kr
    - 用k_i, …, k_{r-1}构造最优二叉查找树作为 kr 的左子树
    - 用k_{r+1},…, k_j构造最优二叉查找树作为 kr 的右子树

最优二叉查找树

- 最优值的递归表达式
  - 当树T′成为根节点kr 的子树时，其期望搜索代价怎么变化？
    - T′中每个节点的深度增加1
  - 根据二叉查找树的期望搜索代价定义式：

$$E(T) = 1 + \sum_{i=1}^{n} H(k,i) \cdot p_i + \sum_{i=0}^{n} H(d,i) \cdot q_i$$

  - 因此子树T′的期望搜索代价增量为子树中所有节点概率之和
  - 以符号 w（i,j）表示期望搜索代价增量：

$$w[i,j] = \sum_{t=i}^{j} p_t + \sum_{t=i-1}^{j} q_t$$

最优二叉查找树

- 最优值的递归表达式
  - 如果kr是一棵包含关键字{ki,…,kj }的最优子树的根
  - 则该子树的期望搜索代价为：

$$E[i,j] = p_r + E[i,r-1] + w[i,r-1] + E[r+1,j] + w[r+1,j]$$

  - 注意到如下关系式成立：

$$w[i,j] = w[i,r-1] + p_r + w[r+1,j]$$

  - 代入上式可得（当j≥i时）：

$$E[i,j] = E[i,r-1] + E[r+1,j] + w[i,j]$$

  - 已知：当j = i-1 时（此时只有虚拟节点di-1）

$$E[i,j] = q_{i-1}$$

最优二叉查找树

- 经过整理得到最优值的递归表达式如下

$$
\begin{cases}
E[i,j] = w[i,j] + \min\limits_{i \le r \le j}\{E[i,r-1] + E[r+1,j]\} & (1 \le i \le j \le n) \\
E[i,i-1] = q_{i-1} & (j = i-1)
\end{cases}
$$

- 所求的最优二叉查找树的期望搜索代价最优值为：E(1,n)
- 问题：如何求解w[i,j]？

$$
\begin{cases}
w[i,i-1] = q_{i-1} & (1 \le i \le n+1) \\
w[i,j] = w[i,j-1] + p_j + q_j & (j \ge i, 1 \le i \le n+1)
\end{cases}
$$

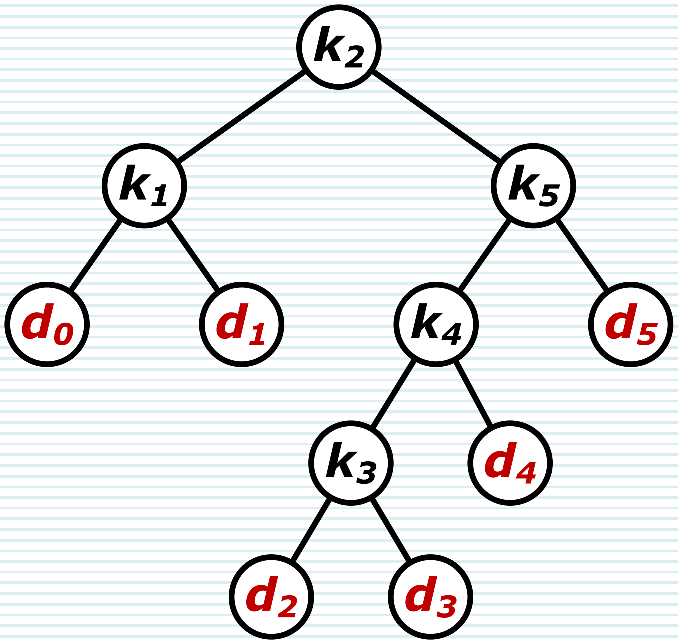
最优二叉查找树

- 构造最优解
  - 采用R[i][j]保存以Kr为根的最优子树T[i,j]的根节点
  - 若：R[1][n]=r，则：kr 为所求二叉查找树的根节点
    - 其左子树为 T[1,r-1]，右子树为 T[r+1,n]
  - 若：R[1][r-1]=s，则 ks 是子树 T[1,r-1] 的根节点
  - 依此类推，可以用O(n)时间构造出最优二叉查找树

```
void BST(float* e, float* w, int* kr){
    for(int i = 1; i <= ncol; i++){
        w[i][(i-1)] = q[i-1];
        e[i][(i-1)] = q[i-1];
    }
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n-k+1; i++){
            int j = i + k - 1;
            e[i][j] = INT_MAX;
            w[i][j] = w[i][(j-1)]+p[j]+q[j];
            for(int r = i; r <= j; r++) {
                float tmp = e[i][(r-1)] + e[(r+1)][j] + w[i][j];
                if(tmp < e[i][j]){
                    e[i][j] = tmp; kr[i][j] = r;
                }
}}}}
```

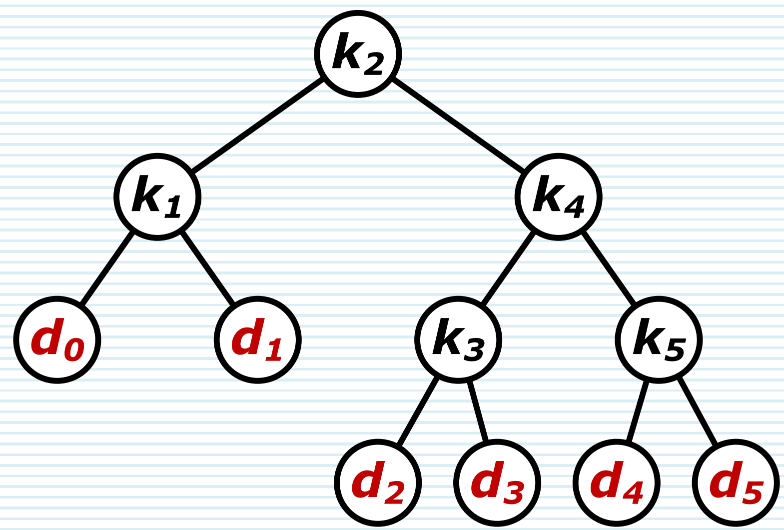
最优二叉查找树示例

p	0.00	0.15	0.10	0.05	0.10	0.20
q	0.05	0.10	0.05	0.05	0.05	0.10



0.05	0.45	0.90	1.25	1.75	2.75
0.00	0.10	0.40	0.70	1.20	2.00
0.00	0.00	0.05	0.25	0.60	1.30
0.00	0.00	0.00	0.05	0.30	0.90
0.00	0.00	0.00	0.00	0.05	0.50
0.00	0.00	0.00	0.00	0.00	0.10

E



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

R

最优二叉查找树

- 算法的时间复杂度： O(n3)
- 算法的空间复杂度
  - 算法中需要用到3个二维数组
  - 用于分别存储不同关键字范围(1≤i≤j≤n)下的：
    - 最优值：E[i][j]
    - 子树权重：W[i][j]
    - 根节点标识：R[i][j]
  - 因此算法的空间复杂度为：O(n2)

本章小结

- 动态规划算法的基本要素
  - 最优子结构性质
    - 问题具备最优子结构性质，说明：
    - 可以根据子问题的最优解，来构造原问题的最优解
  - 重叠子问题性质
    - 子问题的类型有限，且易于被重复利用
- 动态规划算法的设计步骤
  - 找出最优解的性质，并刻画其结构特征
  - 递归地定义最优值
  - 以自底向上的方式计算最优解的值
  - 根据计算最优值时得到的信息，构造最优解

作业二

1. 采用动态规划法求解字符串的编辑距离（edit distance）
   - 问题描述：对于给定的字符串A和B，它们之间的编辑距离定义为：允许对B进行如下操作：(1) 删去一个字符；(2) 插入一个字符；(3) 替换一个字符。每操作一次，则计数加1。将B转化为A所需的最小操作次数即为A和B之间的编辑距离。
   - 要求：给出完整的C语言代码，含测试代码。
2. 采用动态规划法求解最长公共子序列问题（完整源码，含测试代码）

- 作业提交方式：请发至邮箱 **cuestc@163.com**
  - 源代码以.c后缀保存为文本文件（第一题示例：1.c）
  - 将三道题的.c文件打包成一个rar文件（不要工程文件）
  - Rar文件命名方式：**学号-姓名-HW2.rar**
  - 截止日期：**10月20日 24:00** （过时不再收取）

电子科技大学

UESTC

1956