课程编号：20006026

算法分析与设计

主讲教师：刘 瑶

电子科技大学信息与软件工程学院

第3章：动态规划

Dynamic Programming

知识要点

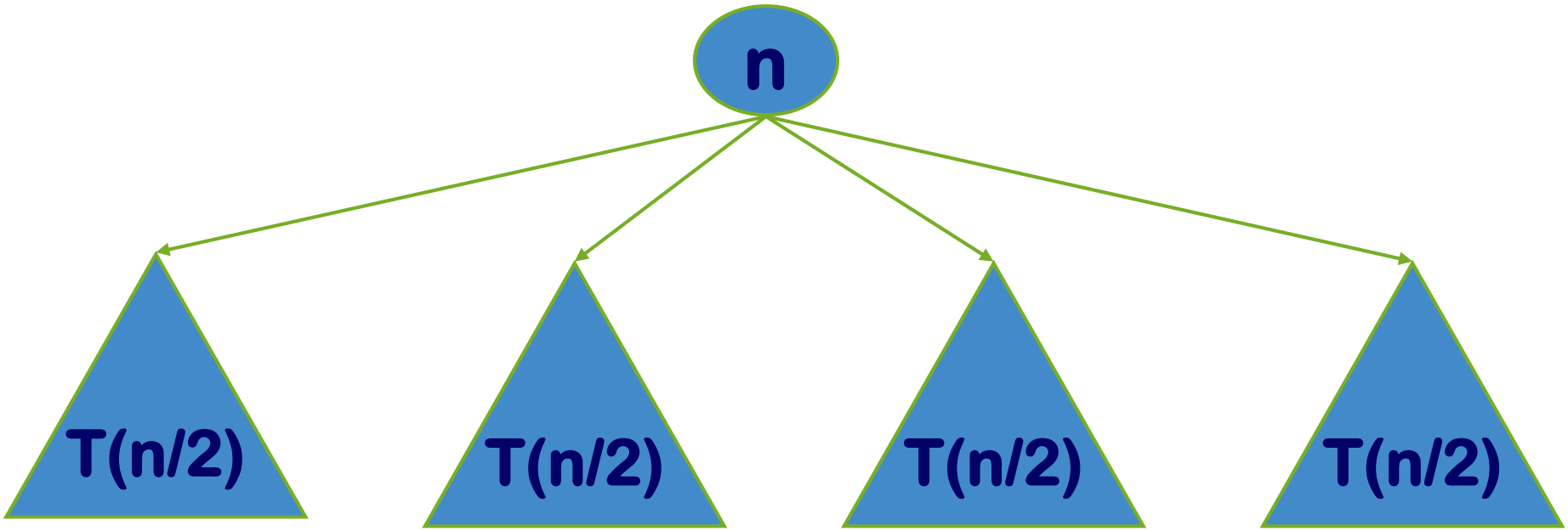
- **理解动态规划算法的概念**
- **掌握动态规划算法的基本要素**
  - 最优子结构性质
  - 重叠子问题性质
- **掌握动态规划算法的设计方法**
  - 找出最优解的性质，并刻划其结构特征
  - 递归地定义最优值
  - 以自底向上的方式计算出最优值
  - 根据计算最优值时得到的信息，构造最优解

知识要点

- **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

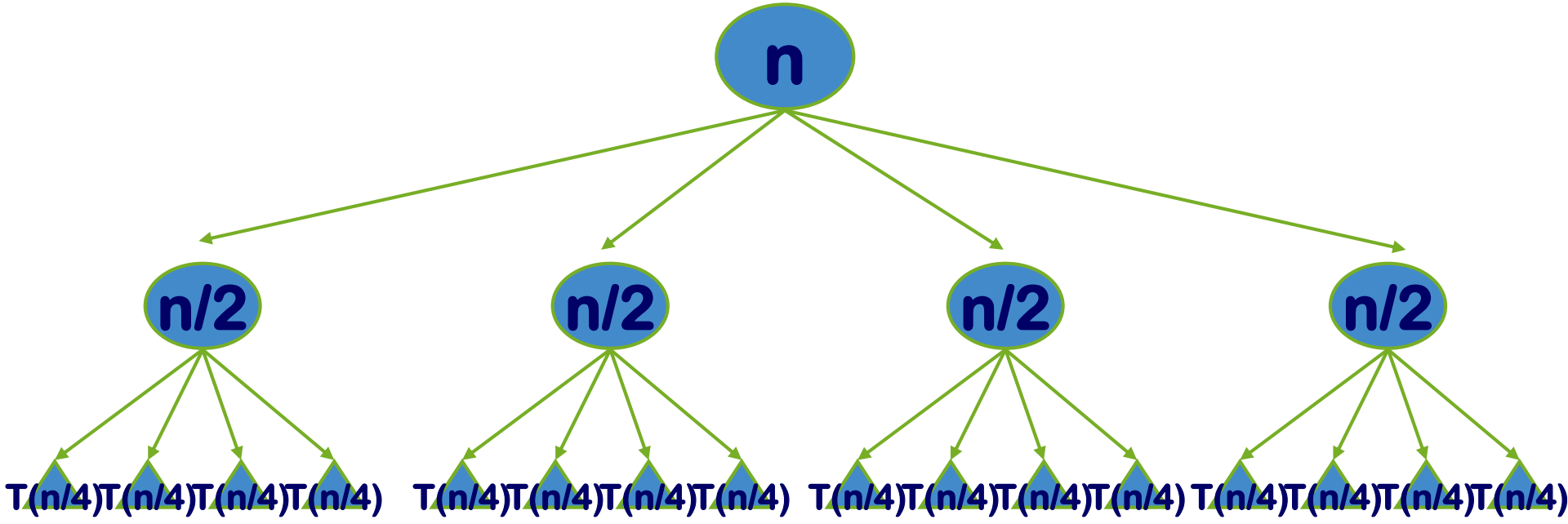
算法总体思想

- 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



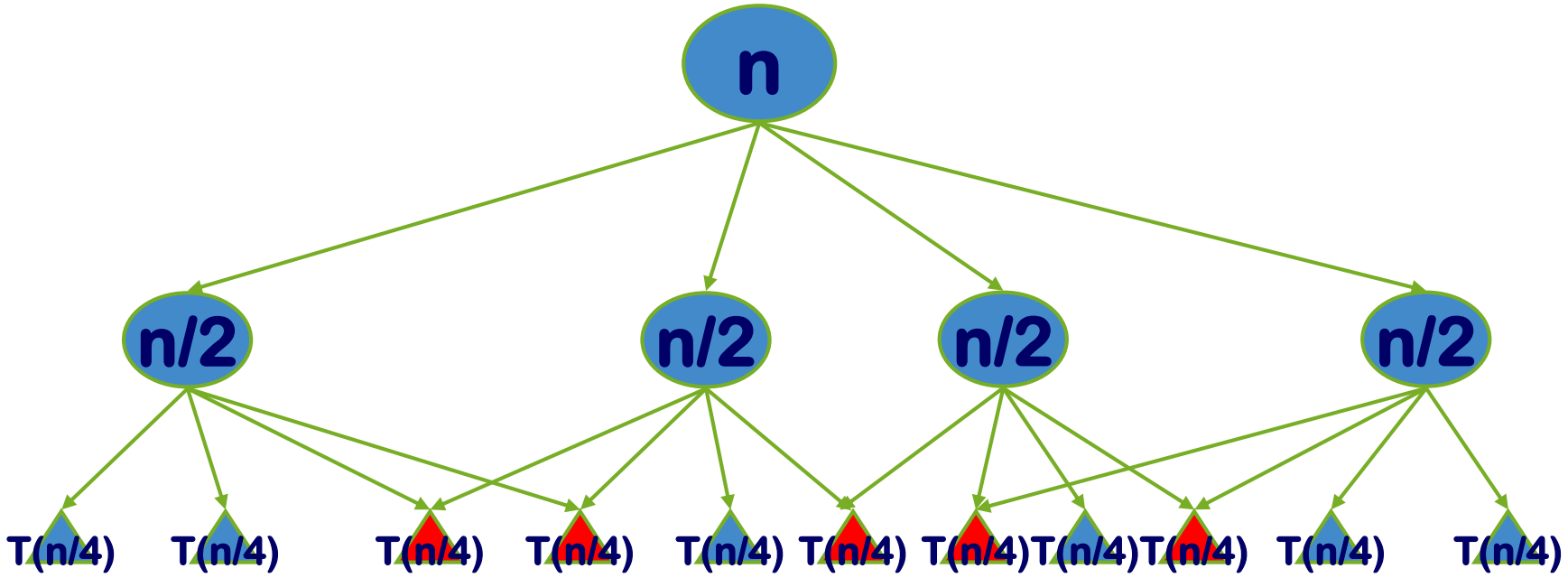
算法总体思想

- 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。



算法总体思想

- 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。



动态规划算法

算法总体思想

- 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
- 与分治法的区别在于
  - 适用于动态规划算法求解的问题，经分解得到的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，导致最终解决原问题需指数时间，
  - 原因在于：虽然子问题的数目常常只有多项式量级，但在用分治法求解时，有些子问题被重复计算了许多次
- 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，其填表格式是相同的）

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

3.1 矩阵连乘问题

(Matrix-Chain Multiplication)

两个矩阵相乘：标准解法

```
void matrixMultiply( int **Ma, int **Mb, int **Mc,
                     int ra, int ca, int rb, int cb ) {
    if (ca != rb) error("矩阵不可乘");
    for (int i=0; i<ra; i++){
        for (int j=0; j<cb; j++) {
            int sum=a[i][0]*b[0][j];
            for (int k=1; k<ca; k++){
                sum += a[i][k]*b[k][j];
            }
            c[i][j]=sum;
        }
    }
}
```

- 设A是p x q的矩阵，B是q x r的矩阵，数乘次数为p x q x r
- 算法的时间复杂度为：$O(n^3)$

矩阵连乘问题

- **问题描述**
  - 给定n个矩阵：{A_1, A_2, ……, A_n}，其中A_i与A_{i+1}可乘
  - 求解这n个矩阵的连乘积： A_1A_2…… A_n
  - 问题：矩阵乘法满足结合率，因此矩阵连乘有多种计算次序
- **问题分析**
  - 通过加括号的方式可以确定矩阵连乘问题的计算次序
  - 若矩阵连乘的计算次序完全确定，则称该连乘积已**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可递归定义如下：
    - 单个矩阵是完全加括号的
    - 矩阵连乘积A是完全加括号的，则A可以表示为两个完全加括号的矩阵连乘积B和C的乘积并加括号，即：A = (BC)

完全加括号的矩阵连乘积

- 设有四个矩阵：A_{50x10}，B_{10x40}，C_{40x30}，D_{30x5}
- 连乘积ABCD总共有五种完全加括号的方式
  - ① (A ((BC) D)) 计算次数：16000
  - ② (A (B (CD))) 计算次数：10500
  - ③ ((AB) (CD)) 计算次数：36000
  - ④ (((AB) C) D) 计算次数：87500
  - ⑤ ((A (BC)) D) 计算次数：34500
- 计算次数（以 ① 为例）
  - $N_{(BC)}= 10x40x30=12000$;
  - $N_{((BC)D)}= 10x30x5=1500$;
  - $N_{(A((BC)D))}= 50x10x5=2500$
  - $N_{(1)} = 12000 + 1500 + 2500 = 16000$ ……

矩阵连乘问题

- **问题描述**
  - 给定n个矩阵：{A_1, A_2, ……, A_n}，其中A_i与A_{i+1}可乘
  - 求解这n个矩阵的连乘积： A_1A_2…… A_n
  - 问题：如何确定计算矩阵连乘积的计算次序
  - 使得依此次序计算矩阵连乘积需要的**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个矩阵连乘积可能的计算次序总数为P(n)
  - 由于每种加括号方式都可以分解为两个子矩阵的加括号问题：$(A_1…A_k)(A_{k+1}…A_n)$，可以得到关于P(n)的递推式如下

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

卡特兰数

- **卡特兰数（Catalan number），是组合数学中一个常出现在各种计数问题中的数列。**
- **其前几项为：**

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, ...

- **卡特兰数 h(n) 满足以下递推关系：**

$$h(n) = h(0)h(n-1) + h(1)h(n-2) + \cdots + h(n-1)h(0)$$

$$(n \geq 2,\ h_0 = 1,\ h_1 = 1)$$

通项： $h(n) = C(2n,n)/(n+1) = C(2n,n) - C(2n,n-1)$

$$(\mathrm{n}=0,1,2,\ldots)$$

矩阵连乘问题

- **解决方案2：动态规划法**
  - 应用动态规划，首先应分析问题的最优解结构特征
  - 将矩阵连乘积 $(A_i A_{i+1} \cdots A_j)$ 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}之间将矩阵链断开（$1 \le k < n$）
    - 则相应的完全加括号方式为：$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$
    - 总计算量为如下三部分计算量之和：
      - 求解**A[1:k]**的计算量
      - 求解**A[k+1:n]**的计算量
      - 求解**A[1:k]**和**A[k+1:n]**相乘的计算量

矩阵连乘问题

- **第一步：分析最优解的结构**
  - 上述划分的关键特征在于：
  - **A[1:n]**的一个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]**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**
  - 该性质是该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）**
  - 设：计算**A[i:j]**所需要的最少数乘次数为**m[i][j]** **(1≤i≤j≤n)**
  - 则：原问题的最优值为**m[1][n]**
  - 当 **i=j** 时，**A[i:j]=A_i**，因此：**m[i][i]=0**
  - 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 ，假设A[i:j]的最优划分位置为k
    - 则：**m[i][j]=m[i][k]+m[k+1][j]+** $P_{i-1}P_kP_j$
    - k的取值只有j-i个可能，即：k∈{i, i+1, ..., j-1}
    - k是其中使计算量达到最小的位置，因此m[i][j]可定义为

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

动态规划法求解矩阵连乘问题

- **第三步：计算最优值**
  - 简单地递归计算**m[1][n]**将耗费指数时间
    - 在递归计算时，许多子问题被重复计算多次
  - 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：

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

  - 这也是该问题可用动态规划算法求解的又一显著特征
  - 采用动态规划法，可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，而在后面需要时只要简单查一下，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（续）
  - 示例：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

动态规划法求解矩阵连乘问题

- **第三步：计算最优值（续）**
  - **方法：根据递归式自底向上计算** **思考：自底向上的含义?**

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

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

m[i][i]=0

根据递归式自底向上计算

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

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

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

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

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

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

根据递归式自底向上计算

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

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

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

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

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

根据递归式自底向上计算

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

$$m[1,4]=\min\begin{cases} m[1,1]+m[2,4]+P_0P_1P_4 \ (k=1) \\ m[1,2]+m[3,4]+P_0P_2P_4\,(k=2) \\ m[1,3]+m[4,4]+P_0P_3P_4\,(k=3) \end{cases}=\min\begin{cases} 0+4375+30\times35\times5 \\ 15750+750+30\times15\times5 \\ 7875+0+30\times5\times10 \end{cases}=9375$$

$$m[2,5]=\min\begin{cases} m[2,2]+m[3,5]+P_1P_2P_5\,(k=2) \\ m[2,3]+m[4,5]+P_1P_3P_5\,(k=3) \\ m[2,4]+m[5,5]+P_1P_4P_5\,(k=4) \end{cases}=7125$$

$$m[3,6]=\min\begin{cases} m[3,3]+m[3,6]+P_2P_3P_6\,(k=3) \\ m[3,4]+m[5,6]+P_2P_4P_6\,(k=4) \\ m[3,5]+m[6,6]+P_2P_5P_6\,(k=5) \end{cases}=5375$$

根据递归式自底向上计算

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

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

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

根据递归式自底向上计算

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

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

思考：这张表的意义何在？是否可以利用它得到最优解？

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值

- 需设置另一张表 S ： 在填充表 m 的过程中
  - 记录各个子矩阵链取最优值时的分割位置k
  - S[i][j]=k表示：A[i:j] 的最优划分方式是 (A[**i:k**])(A[**k+1:j**])
- 可以据此构造矩阵链的最优计算次序
  - 从s[1,n]记录的信息可以知道A[1:n]的最佳划分方式
  - 即：**(**A[1 : **k**]**)(**A[**k+1** : n]**)** 其中：**k = S[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
  - 即：**(**A[1 : x]**)(**A[x+1: k]**)** 其中：**x = S[1, k]**
  - 和：**(**A[k+1 : y]**)(**A[y+1: n]**)** 其中：**y = S[k+1, n]**
- 由此递归下去，可以得到最优完全加括号方式，即构造出一个最优解

构造最优解

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)

构造最优解

- 上述算法中，s[1..n,1..n]记录了各个子矩阵链取最优值时分割位置k。
- s[i,j]表示A[i,j]的最佳方式是(A[i,k])(A[k+1,j])。
- 从s[1,n]记录的信息可以知道A[1,n]的最佳方式，它是(A[1,s[1,n]])(A[s[1,n]+1,n])。
- 其中，A[1,s[1,n]]最佳方式可以递归地得到，它是(A[1,s[1,s[1,n]]])(A[s[1,s[1,n]]+1,s[1,s[1,n]]])
- 由此递归下去，可以得到最优完全加括号方式，即构造出一个最优解。

构造最优解

- 构造最优解的过程是一个递归过程
- 算法：

```
Pubic 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);
    System.out.println("Multiply A"+i+","+s[i,j]+
     "and A"+(s[i,j]+1)+","+j);
}
```

- 算法的调用：

```
void main(){
  int[] P={30,35,15,5,10,20,25};
  int n=P.length;
  int[n][n] m,s;
  MatrixChain(P,n,m,s); //动态规划算法
  TraceBack(s,1,n);
}
```

动态规划方法

- MatrixChain(形参表)
- {
- 初始化；
- 自底向上地计算每一个m[i][j]并将结果填入表中。
- }

初始化是将m[i][i]，即对角线元素，赋值为0。

底是m[i][i]，即对角线元素。最顶层是m[1][n]。

动态规划方法的数据结构

- 形参表中应有n和P[n+1]。
- 算法需要两个二维数组：
- 二维矩阵m[n][n]。其每个元素m[i][j]，$1 \le i \le j \le n$ 为A[i, j] 的最少数乘次数。
- 二维矩阵s[n][n]，其元素s[i][j]，$1 \le i \le j \le n$ 为计算A[i, j] 的断点位置。

动态规划方法

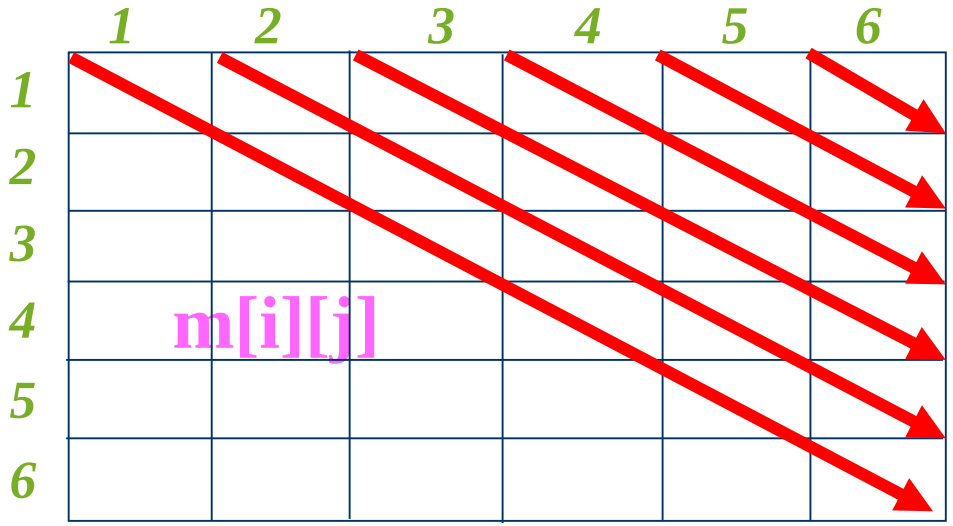
当r > 2，每对(i, j)中的断点k有r – 1个，越往高层断点数目越多。这样自底向上完成整个m[i][j]的计算。

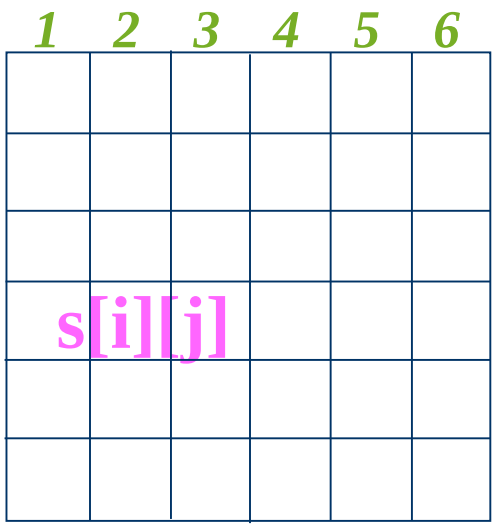
```
vo
{
    for (int r = 2; r <= n; r++)      //r表示链长，取值2~n
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;                  //j依次取值i+1,i+2,.....,n
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
          //即m[i][j] = m[i][i]+m[i+1][j]+ p[i-1]*p[i]*p[j]
        s[i][j] = i;                  //i为初始断开位置
          for (int k = i+1; k < j; k++) {//依次设断开位置为i+1,i+2,.....
            int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (t < m[i][j]) { m[i][j] = t; s[i][j] = k;}
      }
    }
}
```

算法复杂度分析：
算法matrixChain的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。因此算法的计算时间上界为$O(n^3)$。
算法所占用的空间显然为$O(n^2)$。

MatrixChain的运行举例

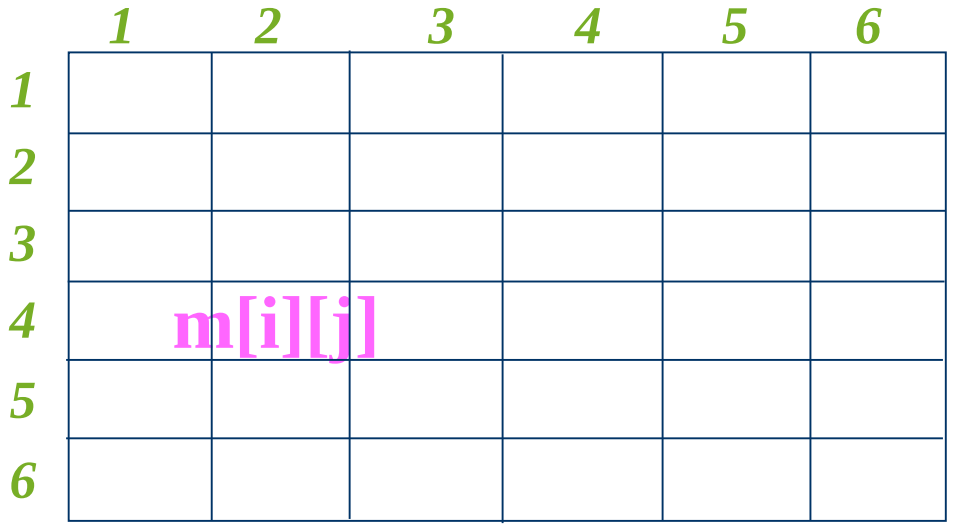
☞MatrixChain将如下面红色箭头所示的过程逐个计算子问题A[i: j] 。

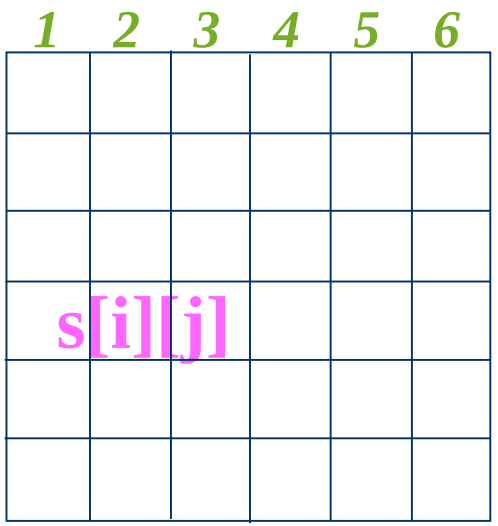




MatrixChain的运行举例

- 设要计算矩阵连乘积$A_1A_2A_3A_4A_5A_6$，其维数分别为$30\times35, 35\times15, 15\times5, 5\times10, 10\times20, 20\times25$，即$p_0=30, p_1=35, p_2=15, p_3=5, p_4=10, p_5=20, p_6=25$。





MatrixChain的运行举例

- 执行for (int i = 1; i <= n; i++) m[i][i] = 0后将对角线元素全部置零，即子问题A[i][i] = 0。

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4		m[i][j]		0		
5					0	
6						0

	1	2	3	4	5	6
1						
2						
3						
4		s[i][j]				
5						
6						

MatrixChain的运行举例

- 当r=2，对每个i，完成相邻矩阵数乘次数计算，即m[i][i+1]=p[i–1]*p[i]*p[j]，并在s[i][j]中添入了相应的断点。

m[i][j]

	1	2	3	4	5	6
1	0	15750				
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当链长r = 3，i = 1时，断点k有两个；
- 对断点k = 1，计算A[1:1]A[2:3]有m[1][3] = m[2][3] + p[0]*p[1]*p[3] = 2625 +30*35*5 = 7875

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2			
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r=3，i=1时，断点k有两个：
- 对断点k = 2，计算A[1:2]A[3:3]有m[1][3] = m[1][2] + m[3][3] + p[0]*p[2]*p[3] = 15750 >7875。m[1][3]仍为7875。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2			
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 2时，断点k有两个：
- 对断点k = 2，计算A[2:4]有m[2][4] = m[3][4] + p[1]*p[2]*p[4] = 750 +35*15*10 = 6000。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	6000		
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	2		
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 2时，断点k有两个：
- 对断点k = 3，计算A[2:2]A[3:4]有m[2][4] = m[2][3] + m[4][4] + p[1]*p[3]*p[4] = 2625 + 0 + 35*5*10 = 4375 ＜ 6000。m[2][4]改为4375，断点改为3。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750		
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3		
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 3时，断点k有两个：
- 对断点k = 3，计算A[3:3]A[4:5]有m[3][5] = m[4][5] + m[4][5] + p[2]*p[3]*p[5] = 1000 +15*5*20 = 2500。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 3时，断点k有两个：
- 对断点k = 4，计算A[3:4]A[5:5]有m[3][5] = m[3][4] + m[5][5] + p[2]*p[4]*p[5] = 750 + 0 + 15*10*20 = 3750 ＞ 2500。m[3][5]仍为2500，断点仍为3。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 4时，断点k有两个：
- 对断点k = 4，计算A[4:4]A[5:6]有m[4][6] = m[4][4] + m[5][6] + p[3]*p[4]*p[6] = 5000 +5*10*25 = 6250

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	6250
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	4
5						5
6						

MatrixChain的运行举例

- 当r = 3，i = 4时，断点k有两个：
- 对断点k = 5，计算A[4:5]A[6:6]有m[4][6] = m[4][5] +m[6][6]+ p[3]*p[5]*p[6] = 1000+0+5*20*25 = 3500 ＜6250。m[4][6]改为3500，断点改为5。

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	3500 6250
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	5 4
5						5
6						

MatrixChain的运行举例

- 类似的，当r=4、5、6时，可计算出相应的m[i][j]及其相应的断点s[i][j]，如下图中所示：

m[i][j]

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

MatrixChain的运行举例

☙由m[1][6]知此矩阵连乘的最小数乘量为15125。

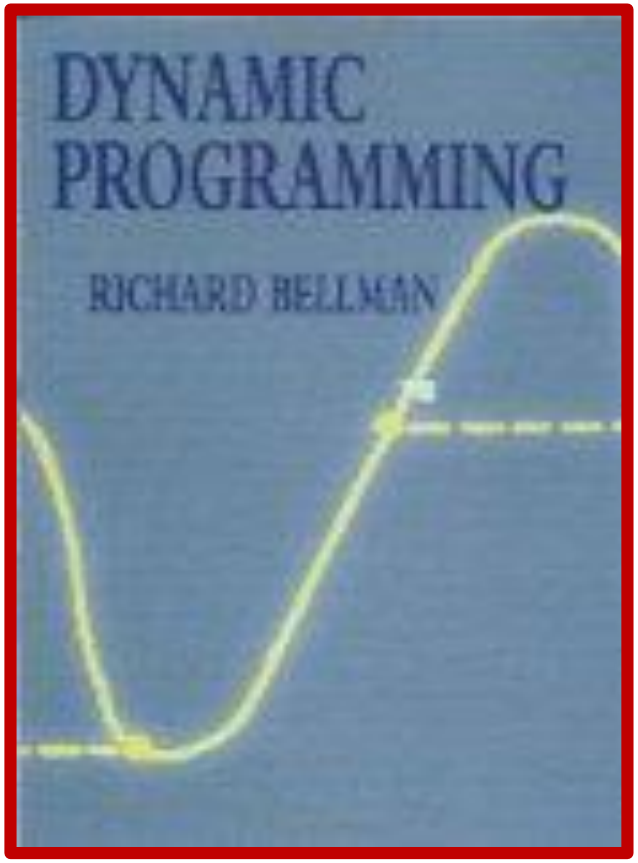
m[i][j]

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s[i][j]

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

3.2 动态规划算法的基本要素



Richard Bellman (1957)，分段最优决策问题

动态规划算法的基本要素

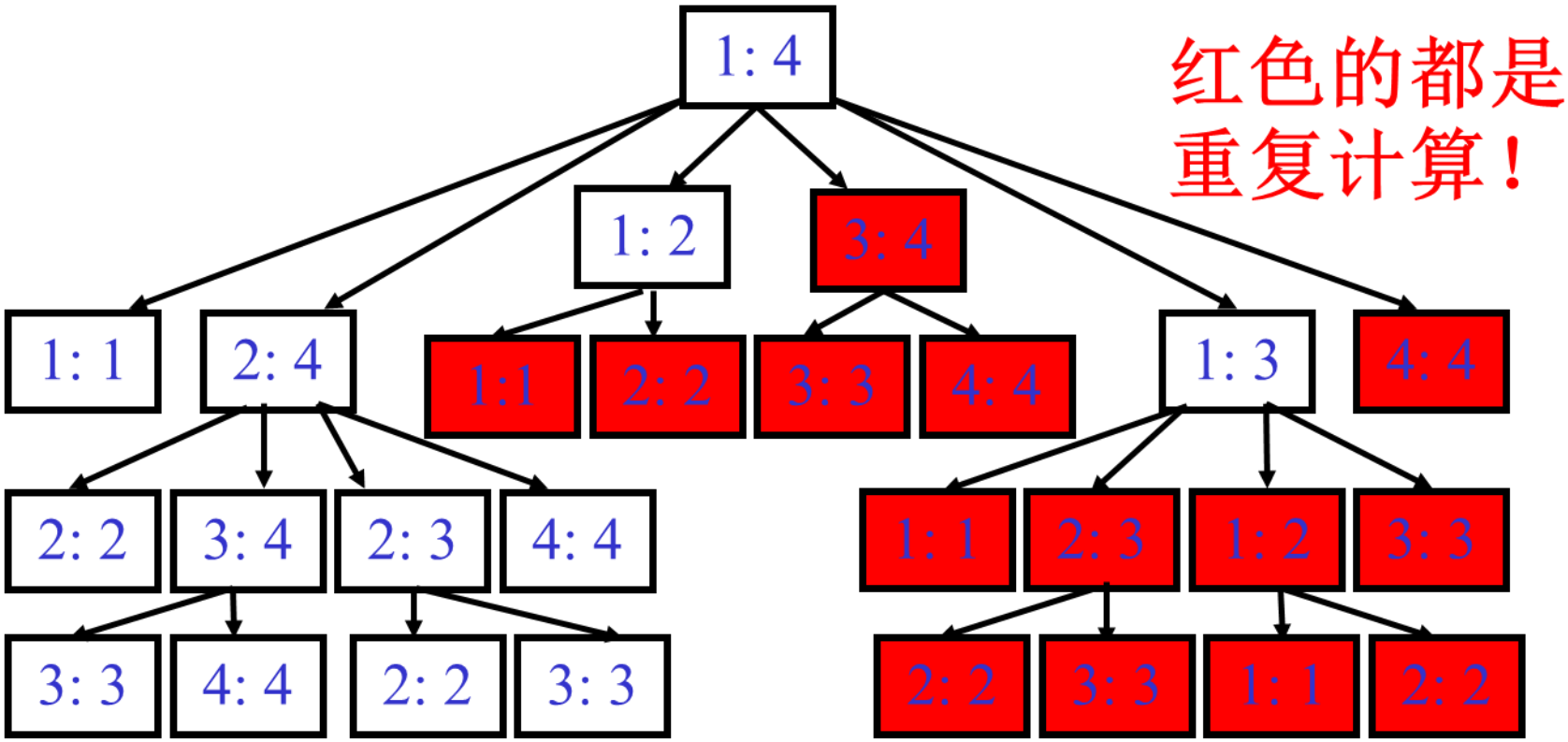
1. 最优子结构

- 矩阵连乘的计算次序问题具有最优子结构性质
  - 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
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：
  - 首先假设由问题的最优解导出的子问题的解不是最优的
  - 然后设法证明在该假设下可构造出比原问题最优解更好的解
  - 通过矛盾法证明由最优解导出的子问题的解也是最优的
- 解题方法：利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解
- 最优子结构是问题能用动态规划算法求解的前提
  - 同一个问题可以有多种方式刻划它的最优子结构
  - 有些表示方法的求解速度更快（空间占用小，问题的维度低）

动态规划算法的基本要素

2. 重叠子问题

- 子问题的重叠性
  - 采用递归算法求解问题时，产生的子问题并不总是独立的
  - 有些子问题被反复计算多次，称为子问题的重叠性质



动态规划算法的基本要素

3. 备忘录方法

- 备忘录方法是动态规划算法的一种变形
  - 它也用表格来保存已解决的子问题答案，以避免重复计算
- 与动态规划的区别在于
  - 备忘录方法的递归方式是自顶向下的
- 备忘录方法的控制结构与直接递归方法的控制结构相同
  - 区别在于备忘录方法为每个解过的子问题建立了备忘录
  - 以备需要时查看，从而避免了相同子问题的重复求解
- 算法复杂度分析
  - 算法的时间复杂度为：$O(n^3)$
  - 算法的空间复杂度为：$O(n^2)$

备忘录方法

```
int MemoizedMaxtrixChain(int n, int **m; int **s)
{  for(int i=1; i<=n; i++)
      for(int j=i; j<=n; j++) m[i][j]=0;
    return LookupChain(1,n);
}
```

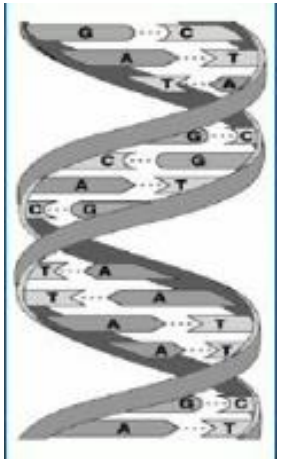
```
int LookupChain(int i，int j)
{
      if (m[i][j] > 0) return m[i][j];
      if (i == j) return 0;
      int u = LookupChain(i，i) + LookupChain(i+1，j) + p[i-1]*p[i]*p[j];
      s[i][j] = i;
      for (int k = i+1; k < j; k++) {
        int t = LookupChain(i，k) + LookupChain(k+1，j) + p[i-1]*p[k]*p[j];
        if (t < u) { u = t; s[i][j] = k;}
        }
      m[i][j] = u; //备忘录m_nxn
      return u;
}
```

3.3 最长公共子序列问题

(Longest Common Subsequence)

序列比对
(Sequence Alignment)

- **在生物学的应用中，经常要比较两个不同有机体的DNA序列。生物有机体的DNA可以表示为四种碱基{A,C,T,G}的字符序列。可以通过序列之间相似性比较来推断不同物种之间的进化关系。**

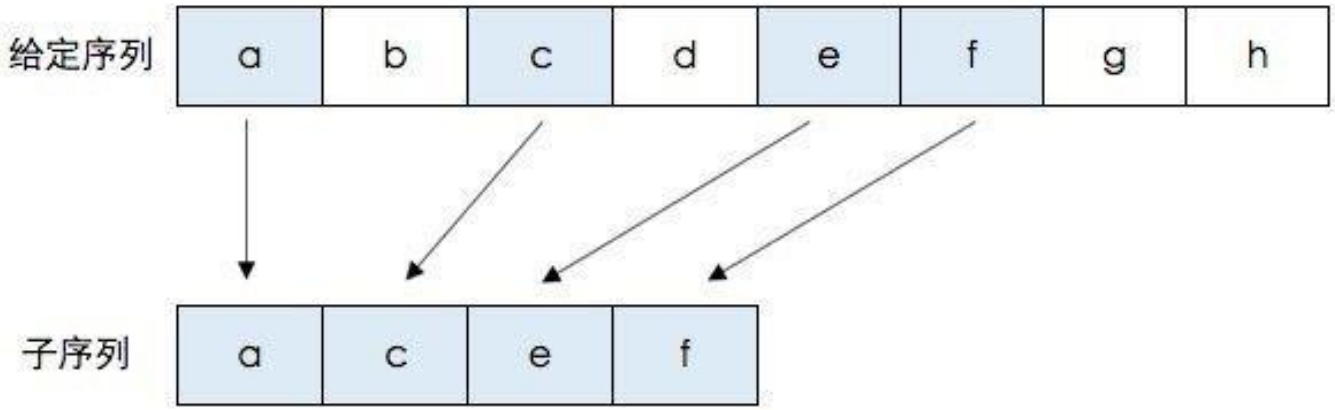


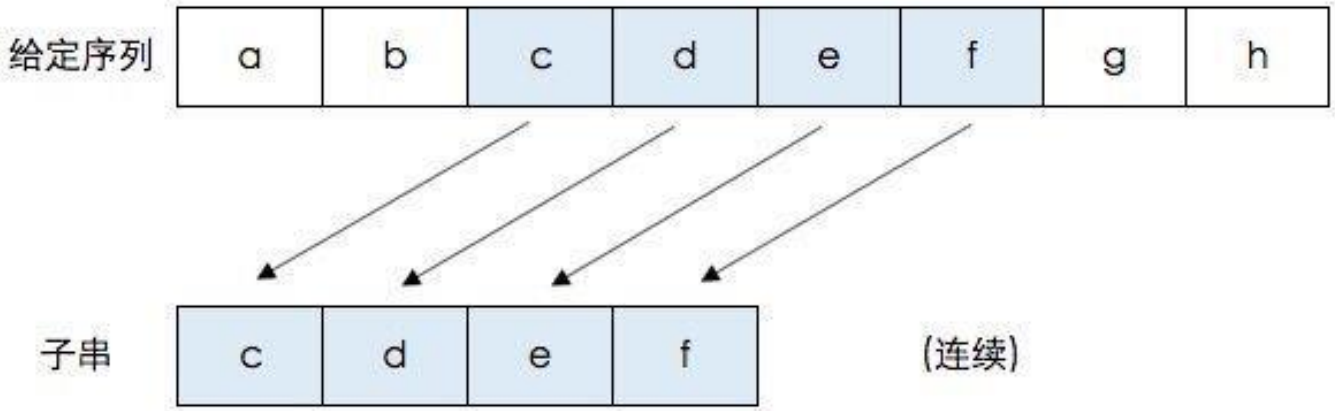
  - 我们把这种相似度概念形式化为最长公共子序列问题。公共子序列越长，可以认为相似度越高。

S_1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA

GTCGTCGGAAGCCGGCCGAA

子序列和子串





最长公共子序列

- **子序列**
  - 给定序列的子序列是在该序列中删去若干元素后得到的序列
  - 若：给定序列 $X=\{x_1, x_2, \ldots\ldots, x_n\}$
  - 称：另一序列 $Z=\{z_1, z_2, \ldots, z_k\}$ 是 X 的子序列
  - 是指存在一个严格递增的下标序列：$\{i_1, i_2, \ldots, i_k\}$
  - 使得：对于所有 $j=1,2,\ldots,k$ 有：$z_j = x_{i_j}$
  - 例如：**Z={B,C,D,B}是X={A,B,C,B,D,A,B}的子序列**
    - 相应的递增下标序列为**{2,3,5,7}**
- **公共子序列**
  - 给定：序列**X**和**Y**
  - 若：另一序列**Z**：既是**X**的子序列，又是**Y**的子序列
  - 称：**Z**是序列**X**和**Y**的公共子序列（**LCS**）

最长公共子序列问题

- **最长公共子序列（LCS）问题**
  - 给定2个序列$X=\{x_1, x_2, \ldots\ldots, x_m\}$和$Y=\{y_1, y_2, \ldots\ldots, y_n\}$
  - 找出X和Y的一个最长公共子序列
- **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}

    } LCS(X,Y)= {B C B A}
  - 解决思路1：穷举搜索

穷举搜索法（Brute-force）

- **用穷举搜索法求解**
  - 穷举法：对**X**的所有子序列，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
    - 设给定序列为 $X=\{x_1, x_2, \ldots\ldots, x_m\}$和$Y=\{y_1, y_2, \ldots\ldots, y_n\}$
    - 思考：**X**有多少个可能的子序列？ **2^m个**
    - 对每条子序列，检查是否是**Y**的子序列，需要**O(n)**时间
    - 从而需要**$O(n2^m)$**，即指数时间来完成搜索
  - 解决思路2：动态规划

最长公共子序列问题

1. 最长公共子序列问题具有最优子结构性质

- 给定序列 $X=\{x_1, x_2, \ldots\ldots, x_m\}$ 和 $Y=\{y_1, y_2, \ldots\ldots, y_n\}$
- 设它们的一个最长公共子序列为 $Z=\{z_1, z_2, \ldots, z_k\}$，则：

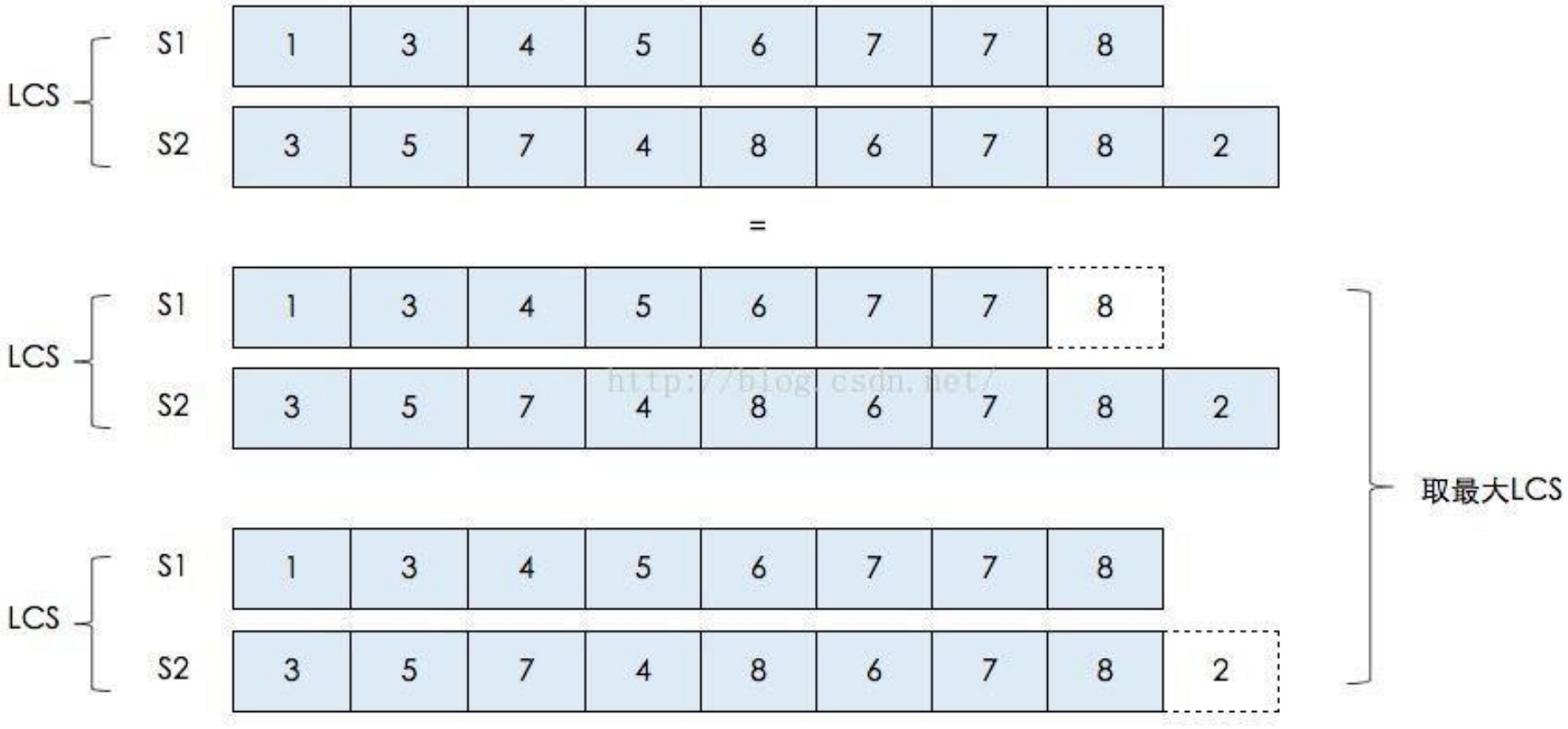
① 若$x_m = y_n$；则：$z_k = x_m = y_n$，且Z_{k-1}是X_{m-1} 和 Y_{n-1} 的LCS

② 若$x_m \neq y_n$且 $z_k \neq x_m$；则：Z 是X_{m-1} 和 Y 的LCS

③ 若$x_m \neq y_n$且 $z_k \neq y_n$；则：Z 是 X 和 Y_{n-1} 的LCS

- 可见：**LCS(X,Y)** 包含了这2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

最长公共子序列问题



假如S1的最后一个元素与S2的最后一个元素不等，那么S1和S2的LCS就等于 ： {S1减去最后一个元素} 与 S2 的LCS， {S2减去最后一个元素} 与 S1 的LCS 中的最大的那个序列。

动态规划求解LCS问题

- **2. 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})
    - 然后将 x_m（或 y_n）添加到这个**LCS**上得到**LCS(X,Y)**
  - 当 $x_m \neq y_n$ 时，须解决如下两个子问题：
    - 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
    - 因为**LCS**(X_{m-1},**Y**)和**LCS**(**X**, Y_{n-1})都包含一个公共子问题
    - 即：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递归式，则时间随输入规模成指数增长

动态规划求解LCS问题

3. 计算LCS的长度（最优值）

- 子问题空间分析
  - **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，所以子问题空间不大
  - 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
    - 数组**b**用于构造最长公共子序列（最优解）

计算最优值

由于在所考虑的子问题空间中，总共有θ(mn)个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

```
void LCSLength(int m，int n，char *x，char *y，int **c，int **b)
{     int i，j;
      for (i = 1; i <= m; i++) c[i][0] = 0;
      for (i = 1; i <= n; i++) c[0][i] = 0;
      for (i = 1; i <= m; i++)
        for (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; }
          }
}
```

动态规划求解LCS问题

- **算法复杂度分析**
  - 数组c的初始化
    - 将第1列（c[i][0]）和第1行（c[0][j]）初始化为0
    - 耗费时间O(m)和O(n)
  - 每个表项的计算时间为O(1)
    - 共有mn个表项需要计算
  - 因此运算时间的复杂度为**O(mn)**

示例：动态规划求解LCS问题

已知：

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

C	y_j	B	D	C	A	B	A
x_i	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

求得：|LCS(x,y)|=4

示例：动态规划求解LCS问题

已知：

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

b	y_j	B	D	C	A	B	A
x_i	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

动态规划求解LCS问题

- **4. 构造最优解（最长公共子序列）**
  - 辅助表**b[i][j]** 记录了**c[i][j]**值是从哪个子问题的解获得的
  - 可以根据它快速地构造出**LCS**：
    - 首先从**b[m][n]**开始
      - c[m][n]表示序列 X 和 Y 的最长公共子序列的长度
    - 按照**b[i][j]**的值表示的方向往回搜索
      - b[i][j] = 1：表示从左上方c[i-1][j-1]得到
      - b[i][j] = 2：表示从上方c[i-1][j]得到
      - b[i][j] = 3：表示从左方c[i][j-1]得到
    - 根据 **i 和 j** 分别对应序列 **x** 和 **y** 的下标可以构造出LCS
      - **思考：怎样构造出LCS序列?**

构造最长公共子序列

```
void LCS(int i，int j，char *x，int **b)
{
    if (i ==0 || j==0) return;
    if (b[i][j]== 1)
        { LCS(i-1,j-1,x,b);
            cout<<x[i]; }
    else if (b[i][j]== 2) LCS(i-1,j,x,b);
    else LCS(i,j-1,x,b);
   }
```

示例：动态规划求解LCS问题

已知：

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

b	y_j	B	D	C	A	B	A
x_i	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

LCS(x,y) = | B | C | B | A |

思考：怎样改进?

示例：动态规划求解LCS问题

已知：

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

LCS(x,y) =

B	C	B	A

C	y_j	B	D	C	A	B	A
x_i	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

3.4 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：$\max\sum_{k=i}^{j}a_k$
- 当所有整数均为负整数时定义其最大子段和为0
- 依次定义，所求的最优值为：

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

- 例如：**$(a_1,a_2,a_3,a_4,a_5,a_6)=(-2,11,-4,13,-5,-2)$**
- 该序列的最大子段和为：$\sum_{k=2}^{4}a_k=20$

最大子段和问题：简单算法

```
int MaxSum(int n, int *pa, 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 +=pa[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）

最大子段和问题：分治算法

- **算法设计** **时间复杂度：O（nlogn）**
  - 如果将序列a[1:n]分为等长的两段：a[1:n/2]和a[n/2+1:n]
  - 可以分别求出a[1:n/2]和a[n/2+1:n]的最大子段和
  - 原问题（a[1:n]的最大子段和）有三种情形：
    - a[1:n]的最大子段和与a[1:n/2]的最大子段相同
    - a[1:n]的最大子段和与a[n/2+1:n]的最大子段相同
    - a[1:n]的最大子段和产生于跨越两段分界点的子序列
      - 显然此时有：a[n/2]和a[n/2+1]在最优子序列中
      - 可分别求得包含a[n/2]和a[n/2+1]的极大子段和S_1和S_2
      - 则：a[1:n]的最大子段和=S_1+S_2

```
int MaxSubSum(int *a, int left, int right)
{  int sum=0;
   if(left=right) sum=a[left]>0 ? A[left] : 0;
   else {  int center=(left+right)/2;
           int leftsum=MaxSubSum(a,left,center);
           int rightsum=MaxSubSum(a,center+1,right);
           int s1=0;
           int lefts=0;
           for(int i=center;i>=left;i--) {
                 lefts +=a[i];
                 if(lefts>s1) s1=lefts;
           }
           int s2=0;
           int rights=0;
           for(int i=center+1;i<=right;i++) {
                 rights +=a[i];
                 if(rights>s2) s2=rights;
           }
           sum=s1+s2;
           if(sum<leftsum) sum=leftsum;
           if(sum<rightsum) sum=rightsum;
   }
   return sum;
}
```

```
int MaxSum(int n, int *a)
{  return( MaxSubSum(a,1,n);
}
```

时间复杂度 **T(n)=O(n log n)**

最大子段和问题：动态规划算法

算法设计

思考：时间复杂度：？

- 通过对分治算法的分析可知，若记：

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

- 则所求的最大子段和为：

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

- 由b[j]的定义可知：
  - 当b[j-1]>0时：b[j]=b[j-1]+a[j]；否则：b[j]=a[j]
- 由此可得b[j]的动态规划递归式：
  - b[j]=max{ b[j-1]+a[j], a[j] } (1≤j≤n)

最大子段和问题：动态规划算法

```
int MaxSum (int n, int *a) {
    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）

3.5 凸多边形最优三角剖分问题

Optimal Triangulation of a Convex Polygon

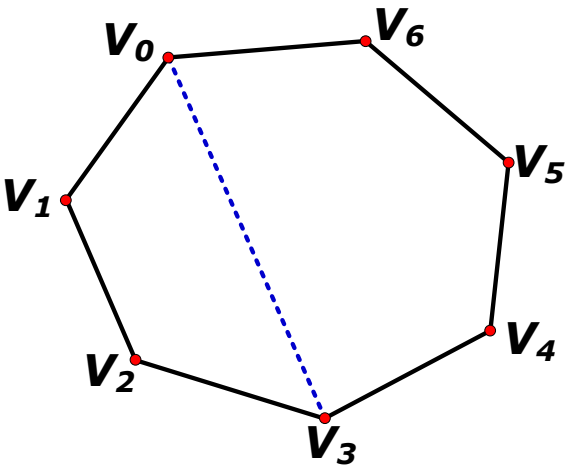
凸多边形最优三角剖分

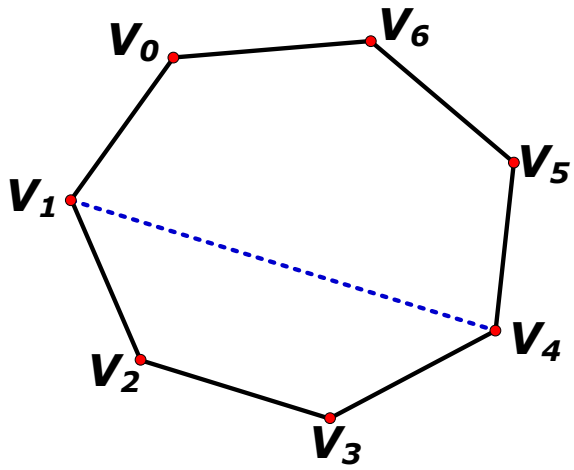
- **多边形**
  - 平面上由一系列首尾相接的直线段组成的分段线性闭曲线
- **简单多边形**
  - 若多边形的边除了连接顶点外没有别的交点，称为简单多边形
- **凸多边形**
  - 当一个简单多边形及其内部构成一个闭凸集时，称为凸多边形
  - 凸集的含义：凸多边形边界或内部的任意两点所连成的直线段上的所有点均在凸多边形的内部或边界上
  - 通常用多边形顶点的逆时针序列表示凸多边形
  - 即：$V=\{v_0, v_1, \ldots\ldots, v_{n-1}\}$ 表示具有n条边（v_0, v_1），（v_1, v_2），……，（v_{n-1}, v_n）的一个凸多边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

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：
  - $\{ v_i, v_{i+1}, \cdots\cdots, v_j \}$ 和 $\{ v_j, v_{j+1}, \cdots\cdots, v_i \}$
- 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 $\{ v_4, v_5, v_6, v_0 , v_1 \}$

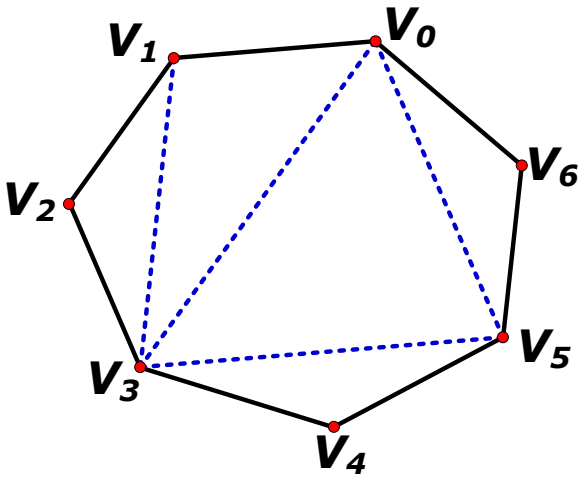


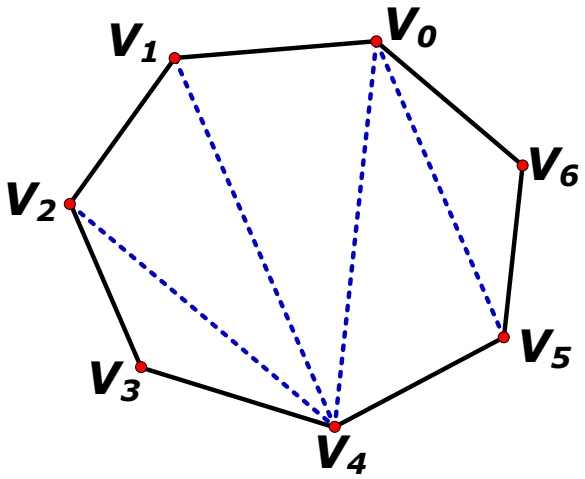


凸多边形最优三角剖分

凸多边形的三角剖分

- 凸多边形的三角剖分是
  - 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





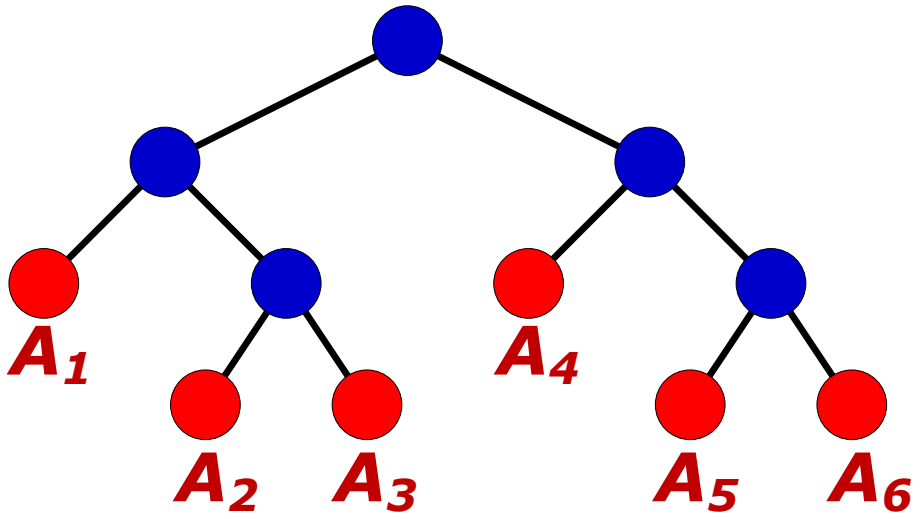
凸多边形最优三角剖分

凸多边形的三角剖分问题

- 给定凸多边形**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
  - 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

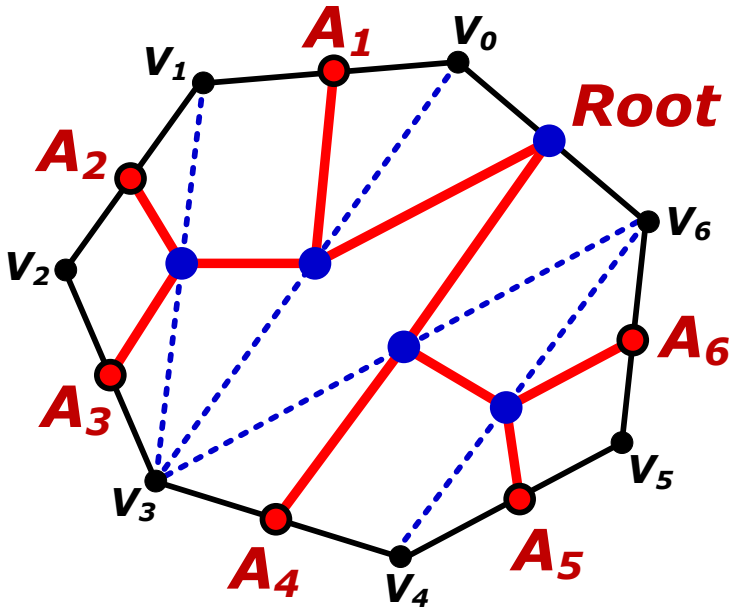
完全加括号表达式的语法树

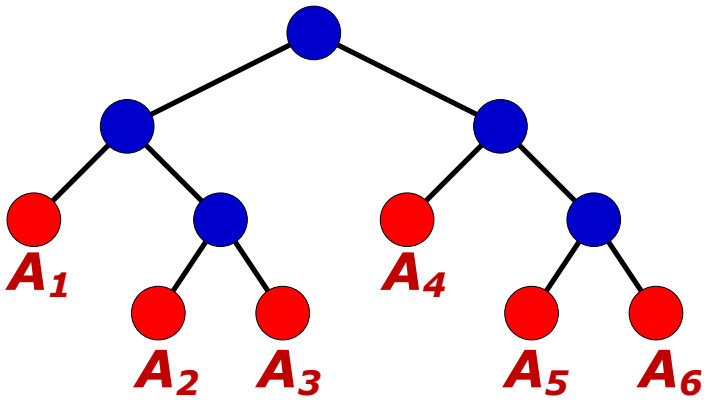
- **矩阵连乘的最优计算次序等价于矩阵链的最优完全加括号方式**
- **一个表达式的完全加括号方式相当于一棵平衡二叉树**
- **例如：完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$**
  - 可以用如下的平衡二叉树进行表示
  - 其中：叶节点为表达式中的原子；树根表示左右子树相结合
  - 这样的二叉树称为该表达式的语法树



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





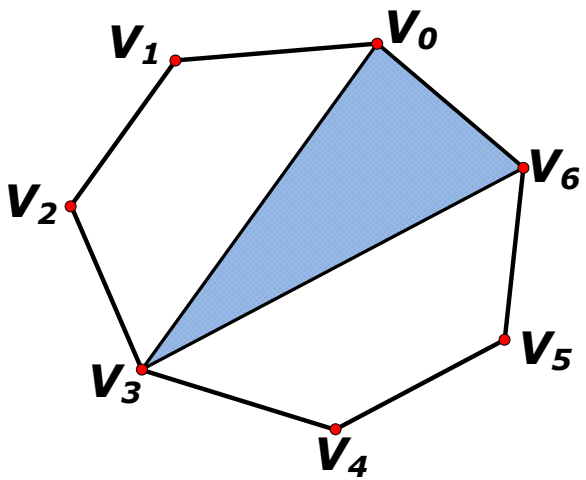
三角剖分的结构及其相关问题

- **凸多边形三角剖分与矩阵连乘问题的同构关系**
  - 凸n边形的三角剖分和有n-1个叶节点的语法树存在一一对应关系。
  - n个矩阵的完全加括号乘积和有n个叶节点的语法树存在一一对应关系。
  - 推论：n个矩阵连乘的完全加括号和凸n+1边形的三角剖分也存在一一对应关系。其中，矩阵A_i对应于凸多边形中的一条边（v_{i-1}, v_i），三角剖分中的每条弦（v_i, v_j）对应于一组矩阵的连乘积$A[i+1, j]$
- **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）一一对应
  - 若矩阵 A_i 的维数为：$p_{i-1} \times p_i$
  - 定义三角形（$v_i v_j v_k$）上的权函数值：$w(v_i v_j v_k)=p_i \times p_j \times p_k$
  - 则：凸多边形**P**的最优三角剖分所对应的语法树 同时 也给出了该矩阵链$A_1 A_2 \ldots A_n$ 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

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$P=\{v_0, v_1, \ldots\ldots, v_n\}$的一个最优三角剖分
- 并设：T包含三角形$v_0v_kv_n$ （$1\le k\le n$）
- 则：T的权为三部分权之和：三角形$v_0v_kv_n$的权，以及两个子多边形$\{v_0,v_1,\ldots v_k\}$和$\{v_k,v_{k+1},\ldots v_n\}$的权之和（如图）
- 可以断言：由T所确定的这两个子多边形的三角剖分也是最优的
- 这是因为：若子多边形有更小权的三角剖分，则三部分之和将小于T的值，这将导致T不是最优三角剖分的矛盾



凸多边形最优三角剖分的递归结构

- 设：**t[i][j]**（$1 \le i \le j \le n$）为凸子多边形p{$v_{i-1}, v_i, \ldots\ldots v_j$}的最优三角剖分所对应的权函数值，即三角剖分的最优值
- 设：退化的两顶点多边形{$v_{i-1}v_i$}的具有权值0（t[i][i]=0）
- 则：原问题（凸(n+1)边形）的最优权值为：**t[1][n]**
- 当（j-i）≥1时：凸子多边形{$v_{i-1}, v_i, \ldots v_j$}至少有三个顶点
  - 设k为其中一个中间点（$i \le k < j$），由最优子结构性质
  - t[i][j]的值应为三部分权值之和：两个凸子多边形的最优权值t[i][k]和t[k+1][j]，加上三角形$v_{i-1}v_kv_j$的权值
  - 由于k的可能位置有**j-i**个，因此问题转化为：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

- 与矩阵连乘问题相比，除了权函数的定义外，**t[i][j]**与**m[i][j]**

的递归式完全相同，因此只需对MatrixChain算法做少量修改即可

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

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

计算最优值

```
    template<class type>
void  minweighttriangulation(int n, type ** t,  int **s)
  {  for (int i=1; i<=n; i++)  t[i][i]=0;
     for (int r=2; r<=n; r++)
        for (int i=1; i<=n-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<=i+r+1; 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;  }
                  }
          }
 }
```

计算凸多边形最优三角剖分的最优值

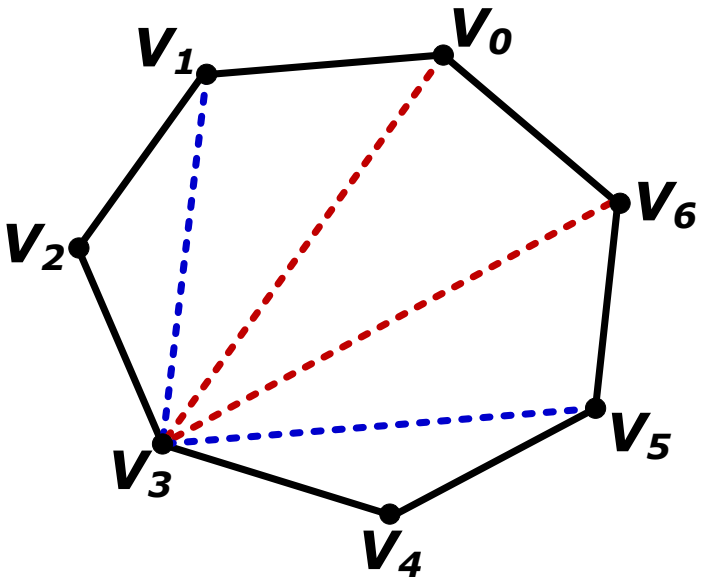
- 复杂度分析：
  - 与矩阵连乘算法的复杂度是一样的
  - 算法有三重循环，元运算的总次数为$O(n^3)$
  - 因此算法的计算时间上界为$O(n^3)$
  - 算法所占用的空间为$O(n^2)$

构造凸多边形最优三角剖分（最优解）

凸多边形最优三角剖分的最优解

- 计算最优值**t[1][n]**时，可以用数组**S**记录三角剖分信息
- S[i][j]记录与（v_{i-1}, v_j）共同组成三角形的第三个顶点的位置
- 据此在O(n)时间内可以构造出最优三角剖分当中的所有三角形

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

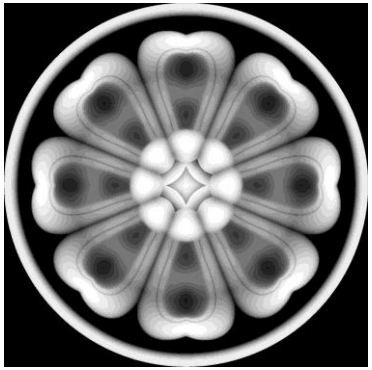


3.7 图像压缩问题

(Image Compression Problem)

图像压缩问题

灰度图



256×256

- 灰度图是指用灰度表示的图像
- 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，也就是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
  - $P = \{p_1, p_2, \cdots p_n\}$
- 灰度图在医学、航天等领域有着广泛的应用

图像压缩问题

- **可以将彩色图像（RGB三色图）转换为灰度图**

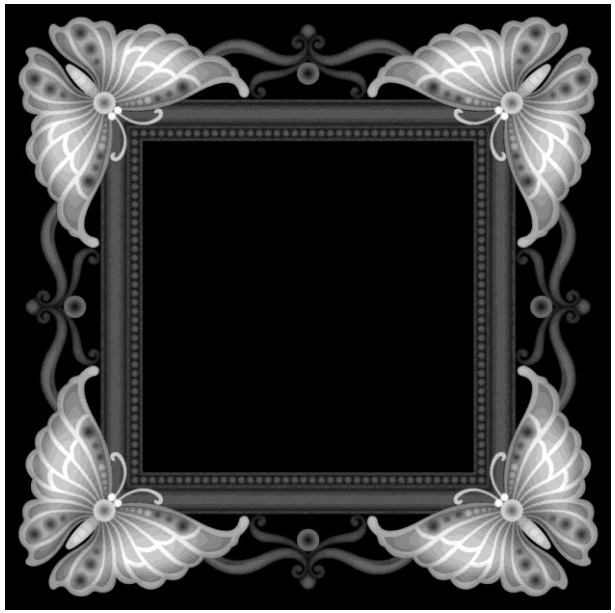
  - 常用方法1：比例法
    - 根据人眼对红绿蓝的敏感程度
    - 使用以下比例式进行转换
    - **Gray = R×0.3+G × 0.59+B × 0.11**
    - 这也是最常用的一种转换

  - 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：$\{p_1,p_2,...p_n\}$
- 其中：整数值p_i $(0 \le p_i \le n)$ 表示像素点 i 的灰度值
- 像素点灰度值取值范围[0-255]，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 $\mathbf{S_i}$ 中：有**N[i]**个像素 **(1≤i≤m)**
  - 设第 **i** 个像素段 $\mathbf{S_i}$ 中：每个像素都用**b[i]**位表示
- 前 **(i-1)** 个分段的像素个数：

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

- 则第 **i** 个像素段序列 $\mathbf{S_i}$ 可以表示为：

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

图像压缩问题

图像的变位压缩存储格式

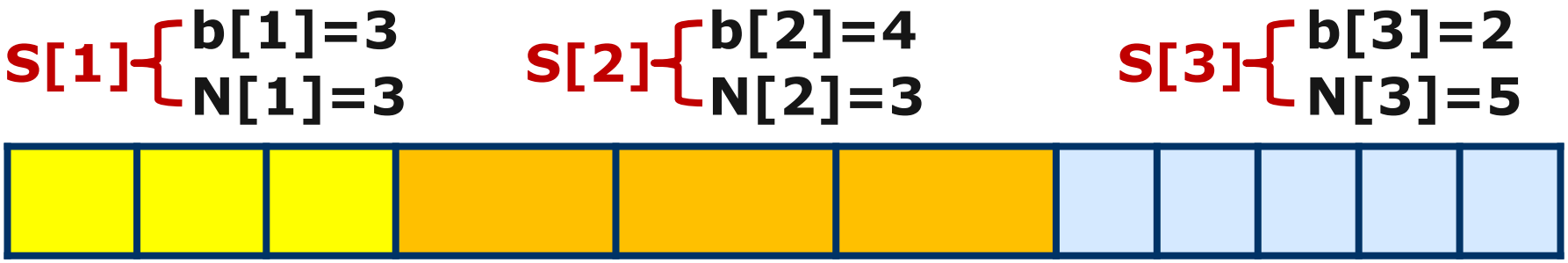
- 设： $\mathbf{h_i}$ 为分段$\mathbf{S_i}$中最大灰度值所对应的二进制数的位数
- 即： $h_i = \left\lceil \log\left(\max_{t[i]+1 \le k \le t[i]+N[i]} p_k + 1 \right) \right\rceil$ 且有： $h_i \le b[i] \le 8$
- 因此：需要用**3位**来表示**b[i]**（分段中每个像素用 b[i] 位表示）
- 如果进一步限制： $1 \le$ **N[i]** ≤ 255 **(分段序列中的像素个数)**
- 则：需要用**8位**来表示**N[i]**
- 像素段 $\mathbf{S_i}$ 所需存储空间为： N[i] * b[i] + 11
- 像素序列$\{p_1, p_2, \cdots p_n\}$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 $P = \{p_1, p_2, \dots p_n\}$**
  - 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** √

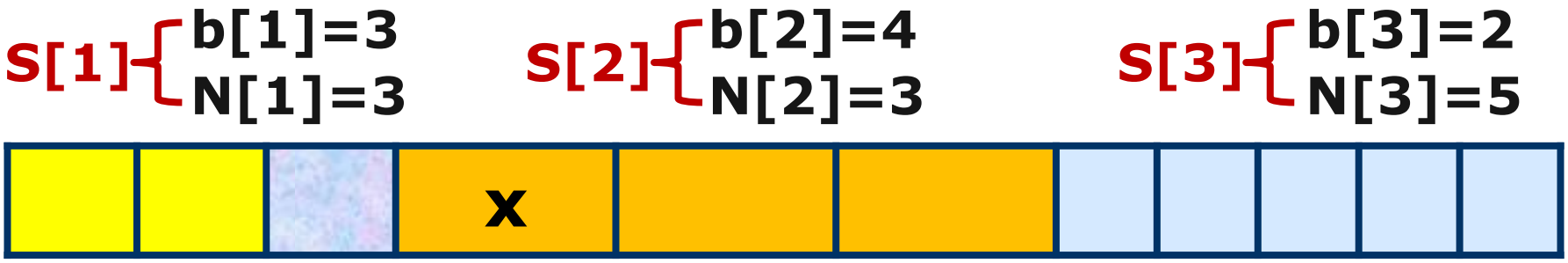
图像压缩问题



- **图像压缩问题的最优子结构性质**
  - 设：**N[i]，b[i]** 是 **{p_1, p_2, … ,p_n}** 的最优分段
    - 其中： **1 ≤ i ≤ m**
  - 则：**N[1]，b[1]** 是 **{p_1, p_2, … ,$p_{N[1]}$}**的最优分段
  - 且：**N[i]，b[i]** 是 **{$p_{N[1]+1}$, … ,p_n}** 的最优分段
    - 其中： **2 ≤ i ≤ m**
  - 即：图像压缩问题满足最优子结构性质

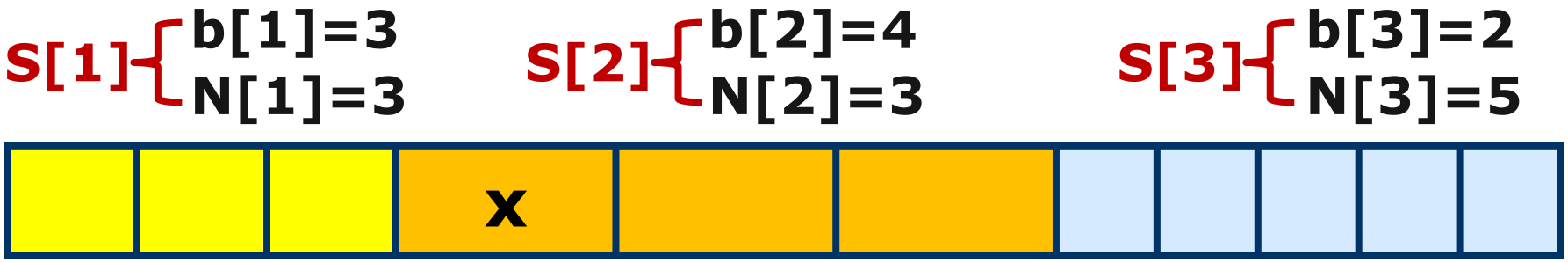
思考：这个最优子结构性质对于解决问题有何意义？

图像压缩问题的最优子结构性质



- **图像压缩问题的最优子结构性质**
  - 设：S[1] 不是 $\{p_1, p_2, \cdots, p_{N[1]}\}$ 的最优分段
  - 则：存在如下两种可能性（可以提出如下两个问题）
    - S[1] 是否可以被划分？ ——这个矛盾很明显
      - 若分段长度之和小于S[1]，则与基本假设矛盾
    - S[1] 是否可以被延长？ —— 这个问题很关键！
      - 不妨设 x 并入 S[1] 后，同样得到最优分段结果
      - 注意由基本假设可知： $b[1] \neq b[2]$

图像压缩问题的最优子结构性质



- **图像压缩问题的最优子结构性质**
  - 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：矛盾（0<b[1]-b[2] ≤7<11，总长度减少）
  - 若： b[1]<b[2]（复杂情况）
    - 若N[2]=1：这种情况不可能出现（b[1]长度容纳不了x）
    - 若N[2]>1，且b[1]可容纳x：显然与基本假设矛盾
    - 若N[2]>1，且b[1]无法容纳x：需扩展b[1]，矛盾?
  - **思考：讨论S[1]的边界，意义何在?**

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 p_i 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											
S	15	19										
S	15	19	23									
S	15	19	23	42								

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

■ 计算最优值

```
public void compress (int[] p, int[] s,int[] l,int[] b)
    {
        header=11;  lmax=256;
        int n=p.length-1;  s[0]=0;
        for(int i=1;i<=n;i++) {
            b[i]=length(p[i]);
            int bmax=b[i];
            s[i]=s[i-1]+bmax;
            l[i]=1;
            for ( int k=2;k<=i&&k<=lmax;k++ ) {
                if ( bmax<b[i-k+1] ) bmax=b[i-k+1];
                if ( s[i]>s[i-k]+k*bmax ) {
                    s[i]=s[i-k]+k*bmax;
                    l[i]=k;
                }
            }
            s[i]+=header;
        }
    }
```

```
public void length(int i)
{
    int k=1;
    i=i/2;
    while ( i>0) {
        k++;
        i=i/2;
    }
    return ;
}
```

header：每个分段的附加存储位数，11=8+3

程序执行到这一行，所分的分段中，最后一段只有**p[i]**一个像素点

每除一次**2**，**k**增加**1**，最后算出**2**的多少次方

i从**1**到**n**，计算出所有的**s[i]**.因为 $1 \le m \le n$

最后一个分段的像素点的个数为**2…i**或者**256**，依次计算**bmax(i-k+1,i)**,和**s[i-k]**相加，选择最少的位数

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
  - 思考：对每个像素点 **i**，求解S[i]的计算复杂度?
    - 求解最优分段的算法中对k的循环次数不超过256
    - 故对每一个确定的 **i**，可在O(1)时间内完成对S[i]的计算
  - 因此：整个算法的时间复杂度为：**O(n)**
  - 算法的空间复杂度为：**O(n)**

图像压缩问题

构造最优解

- 算法compress中用l[i]和b[i]记录了最优分段的信息。
- 最优分段的最后一段的段长度和像素位数存储于l[n]和b[n]当中。
- 前一分段的段长度和像素位数存储于l[n-l[n]]和b[n-l[n]]当中。
- 以此类推，可以在O(n)时间内构造出最优解。

图像压缩问题

■ 实例

P = <10,12,15,255,1,2>，像素灰度值序列

$S[i]$: 对于灰度序列<$P[1]$, …, $P[i]$>最佳分组的所需位数

b: 最后一段表示每个灰度值所需要的二进制位数

l: 最后一段的像素个数

i	0	1	2	3	4	5	6
P		10	12	15	255	1	2
S	0	15	19	23	42	50	57
b		4	4	4	8	8	2
l		1	2	3	1	2	2
策略		1	1-2	1-3	1-3,4	1-3,4-5	1-3,4,5-6

图像压缩问题

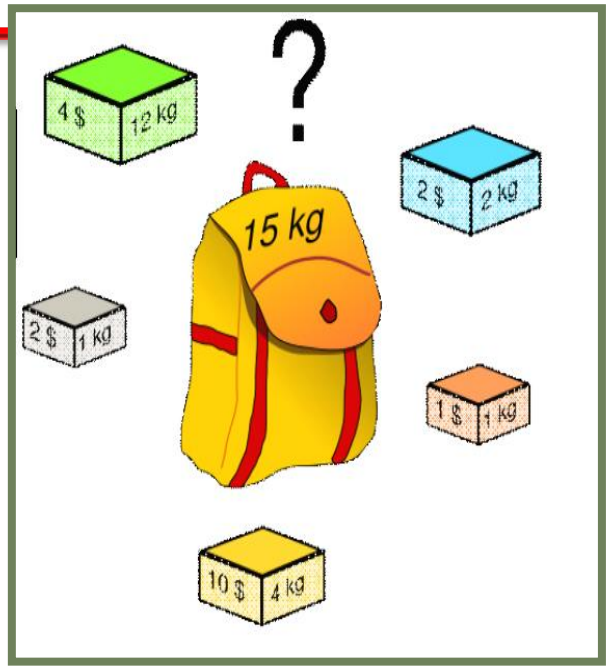
```
public void output(int[] p, int[] s, int[] l, int[] b)
{
    int n=s.length-1;
    System.out.println( “The optimal value is” +s[n]);
    m=0;
    traceback(n,s,l);
    S[m]=n;
    System.out.println( “Decomposed into” +m+
        “segments” );
    for( int j=1; j<=m; j++) {
        l[j]=l[s[j]];
        b[j]=b[s[j]];
    }
    for( int j=1; j<=m; j++)
        System.out.println(l[j]+” ,” +b[j]);
    }
```

```
public void traceback
(int n, int[] s, int[] l)
{
    if( n==0) return;
    traceback( n-l[n],s,l);
    s[m++]=n-l[n];
}
```

3.10 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：$\text{maxmize}\left(\sum_{i=1}^{n} v_i x_i\right)$
- 其中：$\mathbf{x} = (x_1, x_2, \ldots, x_n)$ 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

- 思考：
  1. 0/1背包问题能用什么方法解决？这些方法怎么样?
  2. 0/1背包问题能否用动态规划法解决?
  3. 0/1背包问题如何用动态规划法解决?
  4. 0/1背包问题用动态规划法，与其他方法相比，效率如何，效果如何?

0/1背包问题

1. **0/1背包问题能用什么方法解决？这些方法怎么样？**

- **穷举法：**
  - 列出所有物品的组合
  - 逐一计算这些物品组合所能获得的价值及所需的容量
  - 在不超过背包容量的物品组合中，选择能获得最大价值的物品组合
  - 时间复杂度为$O(2^n)$——效率过低
- **贪心法：**
  - 挑价值越大、重量越轻的（价值/重量 最大）
  - 但不一定能得到最优解

0/1背包问题

2. 0/1背包问题能否用动态规划法解决？

怎样的问题才能用动态规划法解决?

问题必须满足最优子结构性质

(1)最优子结构性质是什么?

(2)验证0/1背包问题是否满足最优子结构性质

。

(1)最优子结构：原问题的最优解包含其子问题的最优解。对背包问题而言：

如果$(x_1, x_2, \ldots, x_n)$是所给0/1背包问题的一个最优解，

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\} \quad (1 \le i \le n) \end{cases}$$

$$\max \sum_{i=1}^{n} v_i x_i$$

若能证明$(x_2, \ldots, x_n)$是下面子问题的最优解：

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le C - w_1 x_1 \\ x_i \in \{0,1\} \quad (2 \le i \le n) \end{cases}$$

$$\max \sum_{i=2}^{n} v_i x_i$$

则0/1背包问题具有最优子结构性质。

(2)证明0/1背包问题是最优子结构（反证）。

设$(x_1, x_2, \ldots, x_n)$是所给0/1背包问题的一个最优解，则$(x_2, \ldots, x_n)$是下面一个子问题的最优解：

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le C - w_1 x_1 \\ x_i \in \{0,1\} \quad (2 \le i \le n) \end{cases}$$

$$\max \sum_{i=2}^{n} v_i x_i$$

如若不然，设$(y_2, \ldots, y_n)$是上述子问题的一个最优解，则

$$\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i$$

$$w_1 x_1 + \sum_{i=2}^{n} w_i y_i \le C$$

因此，

$$v_1 x_1 + \sum_{i=2}^{n} v_i y_i > v_1 x_1 + \sum_{i=2}^{n} v_i x_i = \sum_{i=1}^{n} v_i x_i$$

这说明$(x_1, y_2, \ldots, y_n)$是所给0/1背包问题比$(x_1, x_2, \ldots, x_n)$更优的解，从而导致矛盾。

0/1背包问题

3. 0/1背包问题如何用动态规划法解决？

关键问题：找出动态规划函数

- 0/1背包问题可以看作是决策一个序列$(x_1, x_2, \ldots, x_n)$，对任一变量x_i的决策是决定$x_i=1$还是$x_i=0$。在对x_{i-1}决策后，已确定了$(x_1, \ldots, x_{i-1})$，在决策x_i时，问题处于下列两种状态之一：

（1）背包容量不足以装入物品i，则$x_i=0$，背包不增加价值；

（2）背包容量可以装入物品i。

在（2）的状态下，物品i有两种情况，装入（则$x_i=1$）或不装入（则$x_i=0$）。在这两种情况下背包价值的最大者应该是对x_i决策后的背包价值。

令$V(i, j)$表示在前$i(1\le i\le n)$个物品中能够装入容量为j（$1\le j\le C$）的背包中的物品的最大值，则可以得到如下动态规划函数：

$$V(i, 0)= V(0, j)=0 \qquad \text{(式1)}$$

$$V(i, j)=\begin{cases} V(i-1, j) & j < w_i \\ \max\{V(i-1, j),\ V(i-1, j-w_i)+v_i\} & j \ge w_i \end{cases} \qquad \text{(式2)}$$

式1表明：把前面i个物品装入容量为0的背包和把0个物品装入容量为j的背包，得到的价值均为0。

$$V(i,j)=\begin{cases} V(i-1,j) & j<w_i \\ \max\{V(i-1,j),\ V(i-1,j-w_i)+v_i\} & j\geq w_i \end{cases}$$ **(式2)**

（1）式2的第一个式子表明：如果第*i*个物品的重量大于背包的容量，则物品*i*不能装入背包，则装入前*i*个物品得到的最大价值和装入前*i*−1个物品得到的最大价值是相同的。

$$V(i,j)=\begin{cases}V(i-1,j) & j<w_i\\ \max\{V(i-1,j),\ V(i-1,j-w_i)+v_i\} & j\geq w_i\end{cases}$$ (式2)

（2）式2的第二个式子表明：如果第*i*个物品的重量小于背包的容量，则会有以下两种情况：

①如果第*i*个物品没有装入背包，则背包中物品的价值就等于把前*i*−1个物品装入容量为*j*的背包中所取得的价值。

②如果把第*i*个物品装入背包，则背包中物品的价值等于把前*i*−1个物品装入容量为$j-w_i$的背包中的价值加上第*i*个物品的价值v_i；

显然，取二者中价值较大者作为把前*i*个物品装入容量为*j*的背包中的最优解。

实例：有5个物品，其重量分别是{2, 2, 6, 5, 4}，价值分别为{6, 3, 5, 4, 6}，背包的容量为10。

根据动态规划函数，用一个(n+1)×(C+1)的二维表V，V[i][j]表示把前i个物品装入容量为j的背包中获得的最大价值。

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

		0	1	2	3	4	5	6	7	8	9	10	
	0	0	0	0	0	0	0	0	0	0	0	0	x_1=1
w_1=2 v_1=6	1	0	0	6	6	6	6	6	6	6	6	6	x_2=1
w_2=2 v_2=3	2	0	0	6	6	9	9	9	9	9	9	9	x_3=0
w_3=6 v_3=5	3	0	0	6	6	9	9	9	9	11	11	14	x_4=0
w_4=5 v_4=4	4	0	0	6	6	9	9	9	10	11	13	14	x_5=1
w_5=4 v_5=6	5	0	0	6	6	9	9	12	12	15	15	15	

按下述方法来划分阶段：第一阶段，只装入前1个物品，确定在各种情况下的背包能够得到的最大价值；第二阶段，只装入前2个物品，确定在各种情况下的背包能够得到的最大价值；依此类推，直到第n个阶段。最后，$V(n,C)$便是在容量为C的背包中装入n个物品时取得的最大价值。

为了确定装入背包的具体物品，从$V(n,C)$的值向前推，如果$V(n,C)>V(n-1,C)$，表明第n个物品被装入背包，前n-1个物品被装入容量为$C-w_n$的背包中；否则，第n个物品没有被装入背包，前n-1个物品被装入容量为C的背包中。依此类推，直到确定第1个物品是否被装入背包中为止。由此，得到如下函数：

$$x_i=\begin{cases}0 & V(i,j)=V(i-1,j)\\ 1, \quad j=j-w_i & V(i,j)>V(i-1,j)\end{cases}$$

（式3）

算法实现

设n个物品的重量存储在数组w[n]中，价值存储在数组v[n]中，背包容量为C，数组V[n+1][C+1]存放迭代结果，其中V[i][j]表示前i个物品装入容量为j的背包中获得的最大价值，数组x[n]存储装入背包的物品，动态规划法求解0/1背包问题的算法如下：

C++描述

算法——0/1背包问题

```
int KnapSack(int n, int w[ ], int v[ ]) {
    for (i=0; i<=n; i++)   //初始化第0列
        V[i][0]=0;
    for (j=0; j<=C; j++)   //初始化第0行
        V[0][j]=0;
```

C++描述

算法——0/1背包问题

```
    for (i=1; i<=n; i++)  //计算第i行，进行第i次迭代
        for (j=1; j<=C; j++)
            if (j<w[i])  V[i][j]=V[i−1][j];
            else  V[i][j]=max(V[i−1][j], V[i−1][j−w[i]]+v[i]);

    j=C;   //求装入背包的物品
    for (i=n; i>0; i−−){
        if (V[i][j]>V[i−1][j]) {
            x[i]=1;
            j=j−w[i];
        }
        else x[i]=0;
    }
    return V[n][C];   //返回背包取得的最大价值
}
```

0/1背包问题

4. 0/1背包问题用动态规划法，与其他方法相比，效率如何，效果如何?

- 在算法中，第一个for循环的时间性能是$O(n)$，第二个for循环的时间性能是$O(C)$，第三个循环是两层嵌套的for循环，其时间性能是$O(n\times C)$，第四个for循环的时间性能是$O(n)$，所以，算法的时间复杂性为$O(n\times C)$，一定能求得最优解。

- 穷举法：$O(2^n)$，一定能求得最优解

- 贪心法：$O(n)$，不一定能求得最优解

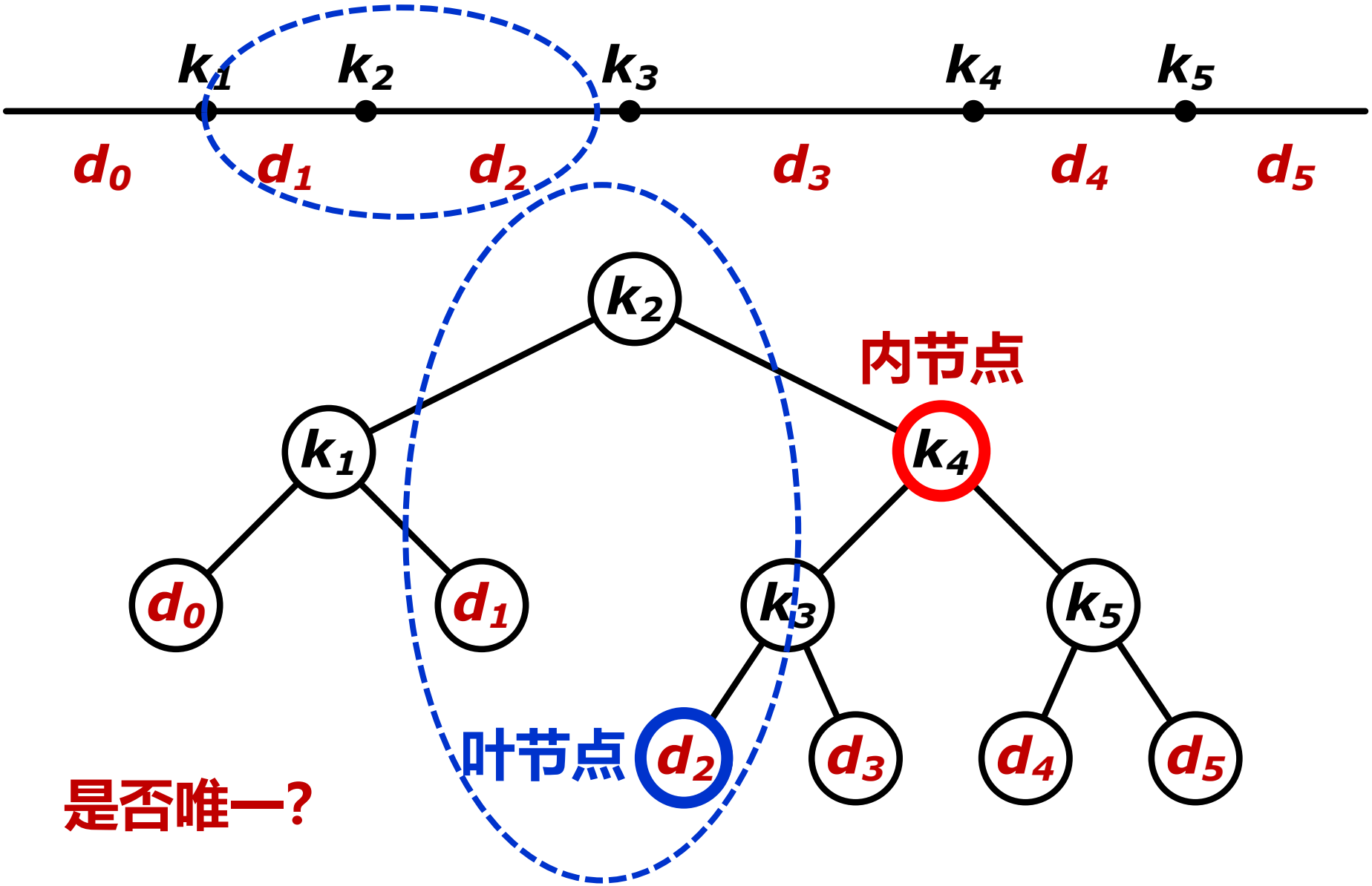
3.11 最优二叉查找树

(Optimal Binary Search Tree)

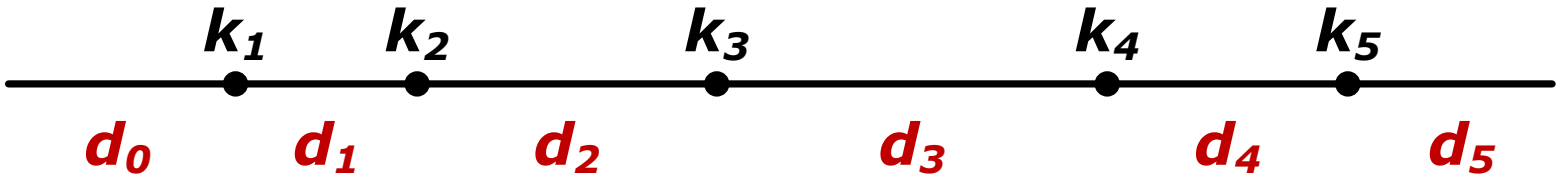
二叉查找树

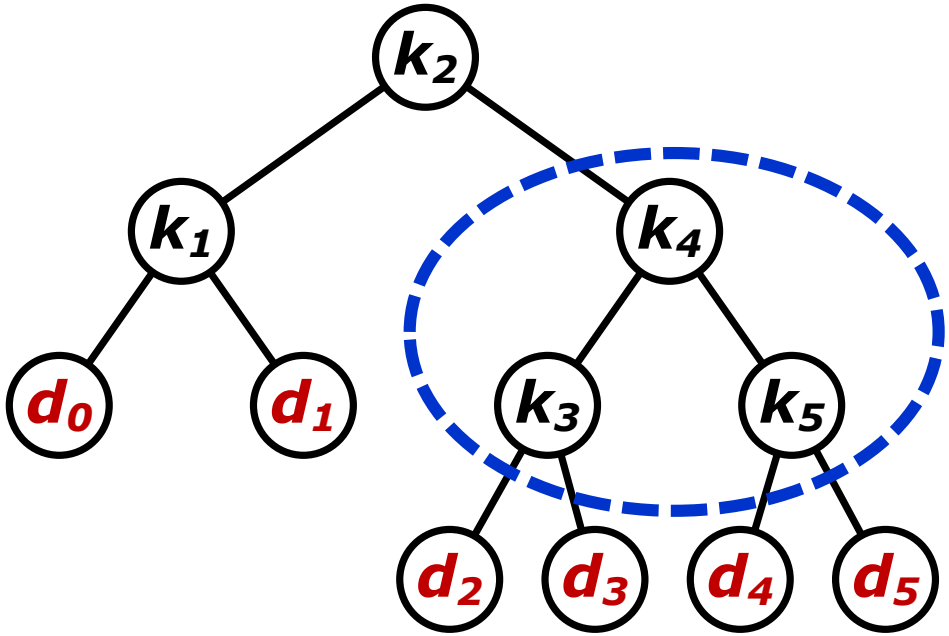
- **二叉查找树（也成为二叉排序树，或二叉搜索树）**
  - 设：$S = \{k_1, k_2, \ldots k_n\}$是n个互异的关键字组成的有序集
  - 且：$k_1 < k_2 < \ldots < k_n$
  - 可利用二叉树的节点存储有序集S中的元素，称为二叉查找树
- **二叉查找树的性质**
  - 二叉查找树中任意节点
    - 大于其左子树中任意节点；小于等于其右子树中任意节点
  - 规定：二叉查找树的**叶节点**是形如**(d_i, d_{i+1})**的开区间
    - 以符号d_i表示虚拟的叶节点，约定：$d_0 = -\infty$; $d_{n+1} = \infty$
  - 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：**$k \in (k_i, k_{i+1})$**

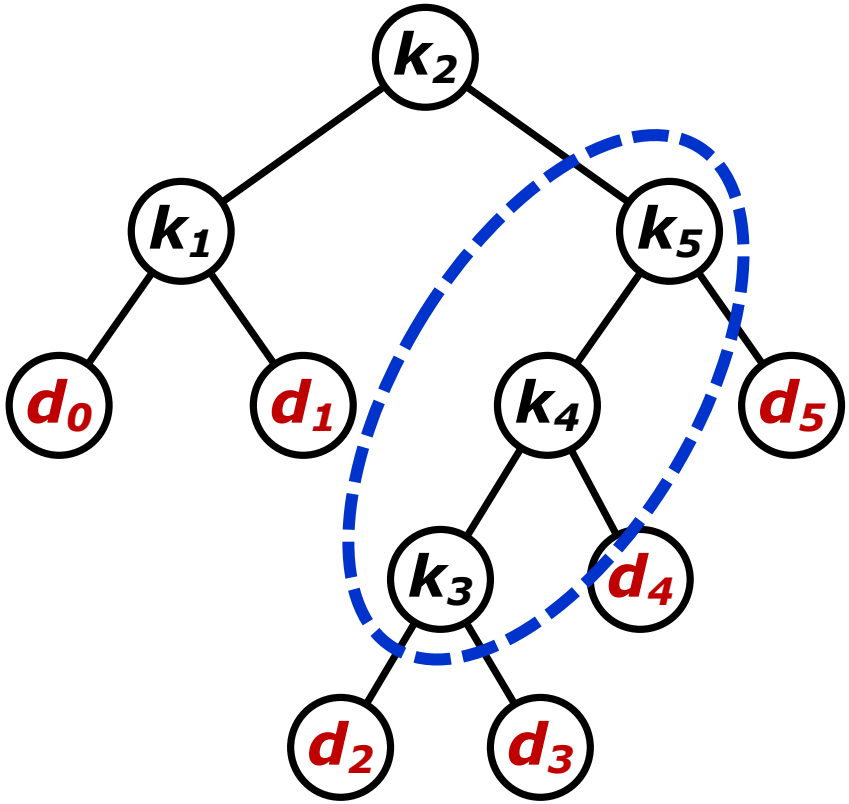
二叉查找树


k_1 k_2 k_3 k_4 k_5
d_0 d_1 d_2 d_3 d_4 d_5
k_2
k_1 k_4
d_0 d_1 k_3 k_5
d_2 d_3 d_4 d_5
内节点
叶节点
是否唯一?

二叉查找树

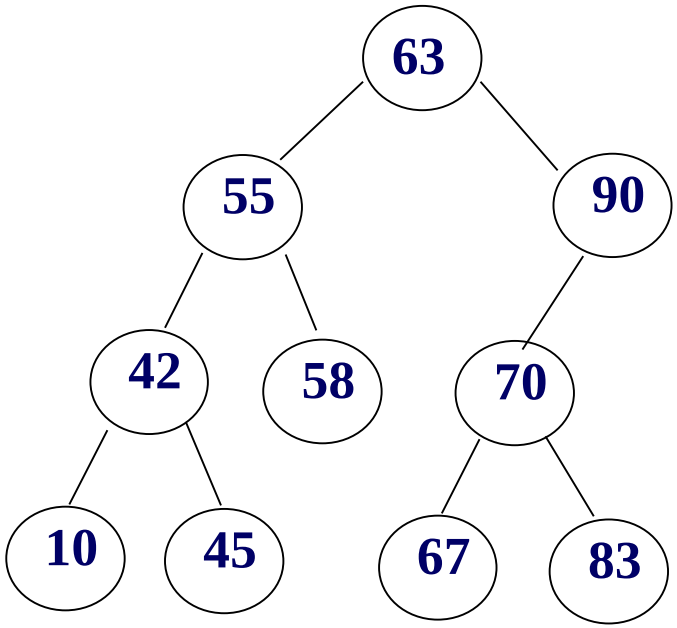




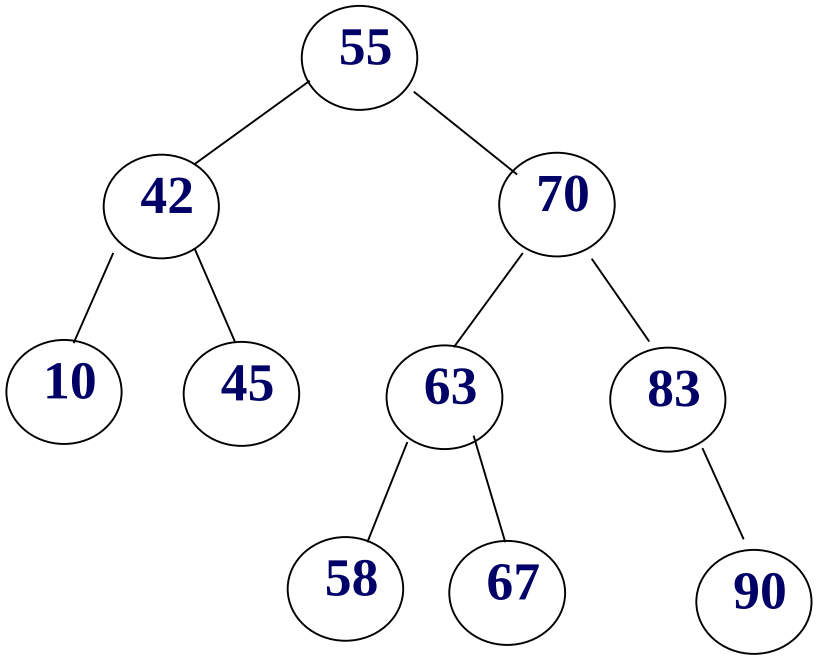


在两棵树上对某些元素进行查找，比较的次数是不同的

示例



(a) 按63,90,55,58,70,42,10,45,83,67 的顺序构造的二叉排序树

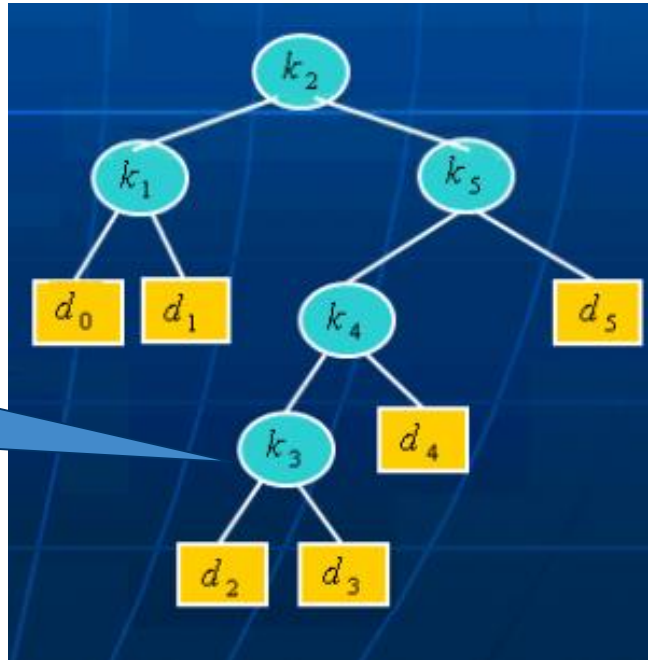


(b) 按55,42,10,70,63,58,83,67,90,45 的顺序构造的二叉排序树

二叉查找树

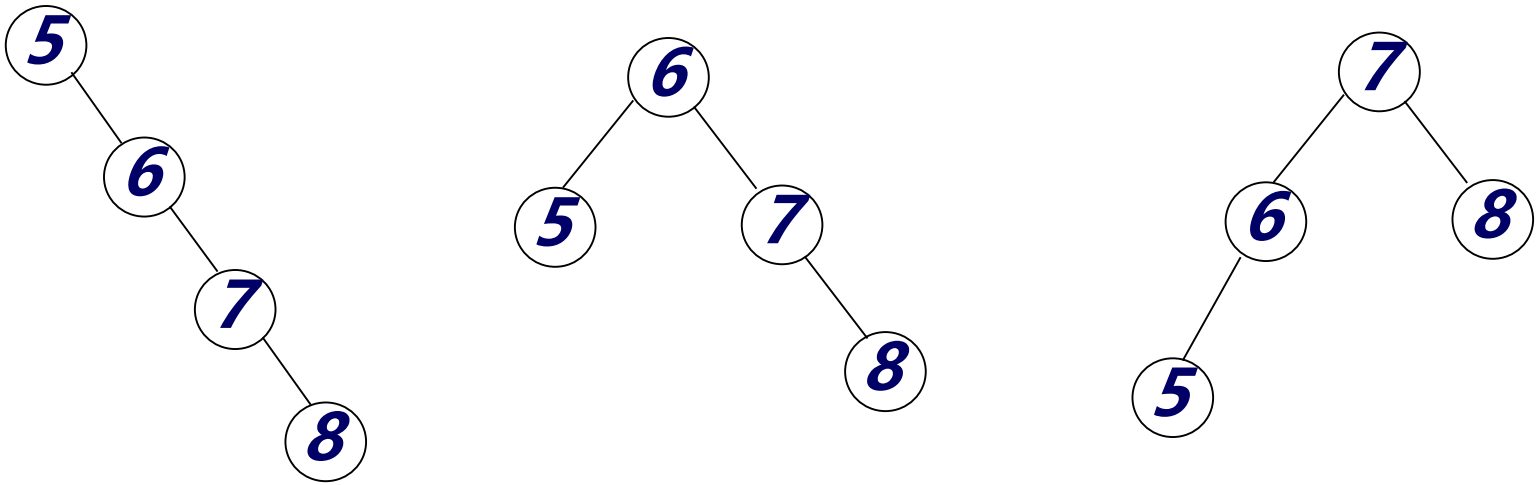
二叉搜索树的搜索代价

- 已知了每个关键字和虚拟键被搜索的概率，可以确定在给定二叉搜索树T内一次搜索的期望代价。假设一次搜索的实际代价为检查的结点的个数，亦即，在T内搜索所发现的“结点的深度”加“1”。



比如：k_3的结点深度为3，要找到结点k_3,需要检查3+1=4个结点的个数 k_2 k_5 k_4 k_3

例如，集合{*5, 6, 7, 8*}的查找概率是{0.1, 0.2, 0.4, 0.3}，
(a)的平均比较次数是0.1×1＋0.2×2+0.4×3+0.3×4=2.9，
(b)的平均比较次数是0.1×2＋0.2×1+0.4×2+0.3×3=2.1，
(c)的平均比较次数是0.1×3＋0.2×2+0.4×1+0.3×2=1.7。



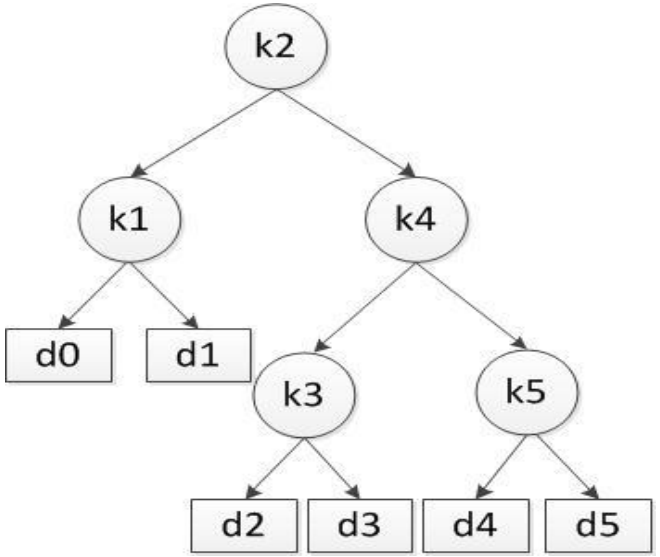
(a)　　(b)　　(c)

二叉查找树示例

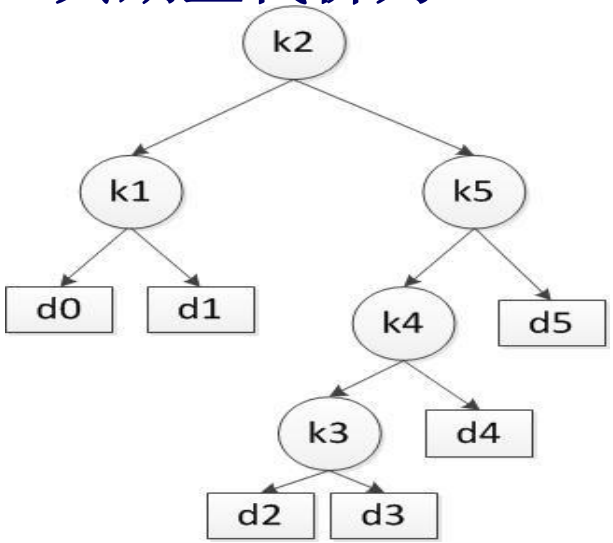
二叉查找树的期望搜索代价

- **对于有序集S = {k_1, k_2, ... k_n}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== k_i** 的概率为p_i
    - 设：返回**k ∈ (k_i, k_{i+1})** 的概率为q_i
    - 则：$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$
    - 即：查找成功和查找不成功的概率之和为1
  - 设：**T**中内节点**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

二叉查找树的期望搜索代价实例



- 其期望代价为**2.8**



- 其期望代价为**2.75**

node	depth	probabilit y	contributi on
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

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, … 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 , …, p_n, q_n）
  - 目标：在集合{T}中找出一棵具有最小期望代价的二叉查找树
- **最优二叉查找树问题具有最优子结构性质**
  - 为描述最优子结构，首先来考察它的子树
  - 若：最优二叉查找树T有一棵包含关键字{k_i,…,k_j} 的子树T′
    - 相应的最优二叉树叶节点为：{ d_{i-1}, d_i …, d_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,…,k_j } （1≤i≤j≤n)
- 假设：**k_r** 是包含该序列的一棵最优子树的根（i≤r≤j)
  - 根 **k_r** 的左子树包含：k_i, …, k_{r-1} （以及d_{i-1},…,d_{r-1}）
  - 根 **k_r** 的右子树包含：k_{r+1},…, 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 **k_r** ）
  - 确定包含关键字k_i,…,k_{r-1}和k_{r+1},…,k_j 的所有二叉查找树
  - 其中期望搜索代价最小是就最优二叉查找树
  - 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 } 的最优二叉查找树（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{k_i,…,k_j }中选择一个根**k_r**
  - 用关键字k_i, …, k_{r-1}构造最优二叉查找树作为 **k_r** 的左子树
  - 用关键字k_{r+1},…, k_j构造最优二叉查找树作为 **k_r** 的右子树

最优二叉查找树

最优值的递归表达式

- 当一棵树T′成为根节点K_r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k_r是一棵包含关键字$\{k_i,\ldots,k_j\}$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d_{i-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]**?

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

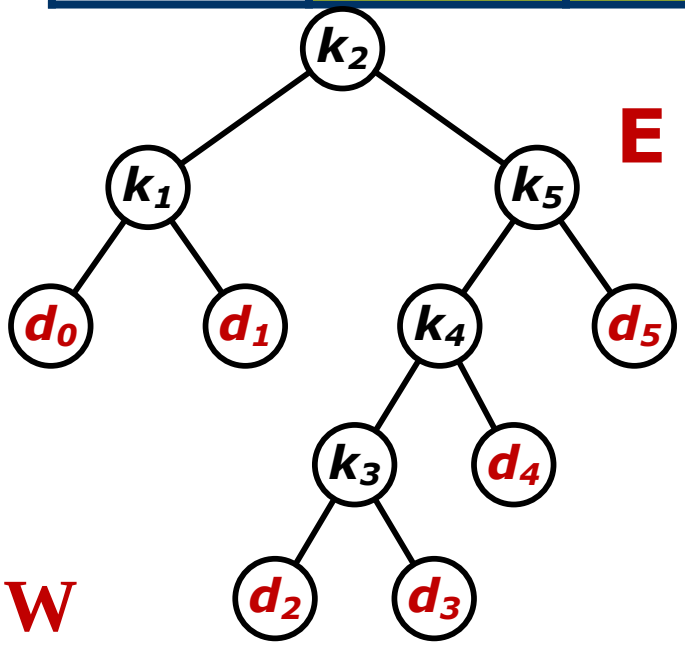
最优二叉查找树

构造最优解

- 采用**R[i][j]**保存以K_r为根的最优子树**T[i,j]**的根节点
- 若：**R[1][n]=r**，则：k_r 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 k_s 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

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



E

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	0.006	0.00	0.00	0.10

W

0.05	0.30	0.45	0.55	0.70	1.00
0.00	0.10	0.25	0.35	0.50	0.80
0.00	0.00	0.05	0.15	0.30	0.60
0.00	0.00	0.00	0.05	0.20	0.50
0.00	0.00	0.00	0.00	0.05	0.35
0.00	0.00	0.006	0.00	0.00	0.10

R

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

算法——最优二叉查找树

设n个字符的查找概率存储在数组p[n]中，动态规划法求解最优二叉查找树的算法如下：

```
double OptimalBST(int n, double p[ ], double E[ ][ ], int R[ ][ ] )
{
   for (i=1; i<=n; i++)  //初始化
   {
      E[i][i-1]=0;
      E[i][i]=p[i];
      R[i][i]=i;
   }
   E [n+1][n]=0;
```

```
for (d=1; d<n; d++)    //按对角线逐条计算
   for (i=1; i<=n-d; i++)
   {
      j=i+d;
      min=∞; mink=i; sum=0;
      for (k=i; k<=j; k++)
      {
         sum=sum+p[k];
         if (E[i][k-1]+E [k+1][j]<min) {
            min=E [i][k-1]+E [k+1][j];
            mink=k;
         }
      }
      E [i][j]=min+sum;
      R[i][j]=mink;
   }
   return E [1][n];
}
```

时间复杂度$O(n^3)$

最优二叉查找树

- **算法的时间复杂度： $O(n^3)$**
- **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($1 \le i \le j \le n$)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：**$O(n^2)$**

本章小结

- **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

小结——适用条件

动态规划法的有效性依赖于问题本身所具有的两个重要的性质：

1.最优子结构。如果问题的最优解是由其子问题的最优解来构造，则称 该问题具有最优子结构性质。

2.重叠子问题。在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。动态规划 算法正是利用了这种子问题的重叠性质，对每一个子问 题只解一次，而后将其解保存在一个表格中，在以后该 子问题的求解时直接查表。

小结——与其他算法比较

动态规划的思想实质是分治思想和解决冗余。

●与分治法类似的是将原问题分解成若干个子问题，先求解子问题，然后从 这些子问题的解得到原问题的解。

●与分治法不同的是经分解的子问题往往不是互相独立的。若用分治法来解，有些共同部分（子问题或子子问题）被重复计算。

小结——其他应用领域

生产与存储问题

某工厂每月需供应市场一定数量的产品。供应需求所剩余产品应存入仓库，一般地说，某月适当增加产量可降低生产成本，但超产部分存入仓库会增加库存费用，要确定一个每月的生产计划，在满足需求条件下，使一年的生产与存储费用之和最小。

小结——其他应用领域

投资决策问题

某公司现有资金Q亿元，在今后5年内考虑给A、B、C、D四个项目投资，这些项目的投资期限、回报率均不相同，问应如何确定这些项目每年的投资额，使到第五年末拥有资金的本利总额最大。

小结——其他应用领域

设备更新问题

企业使用设备都要考虑设备的更新问题，因为设备越陈旧所需的维修费用越多，但购买新设备则要一次性支出较大的费用。现在某企业要决定一台设备未来8年的更新计划，已预测到第j年购买设备的价格为K_j，G_j为设备经过j年后的残值，C_j为设备连续使用j-1年后在第j年的维修费用(j=1,2...8)，问应在哪年更新设备可使总费用最小。