算法设计与分析

第三章 动态规划

学习要点

理解动态规划算法的概念。

掌握动态规划算法的基本要素

（1）最优子结构性质

（2）重叠子问题性质

掌握设计动态规划算法的步骤。

(1)找出最优解的性质，并刻划其结构特征。

(2)递归地定义最优值。

(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。

通过应用范例学习动态规划算法设计策略。

（1）矩阵连乘问题；

（2）最长公共子序列；

（3）凸多边形最优三角剖分；

（4）多边形游戏；

（5）图像压缩；

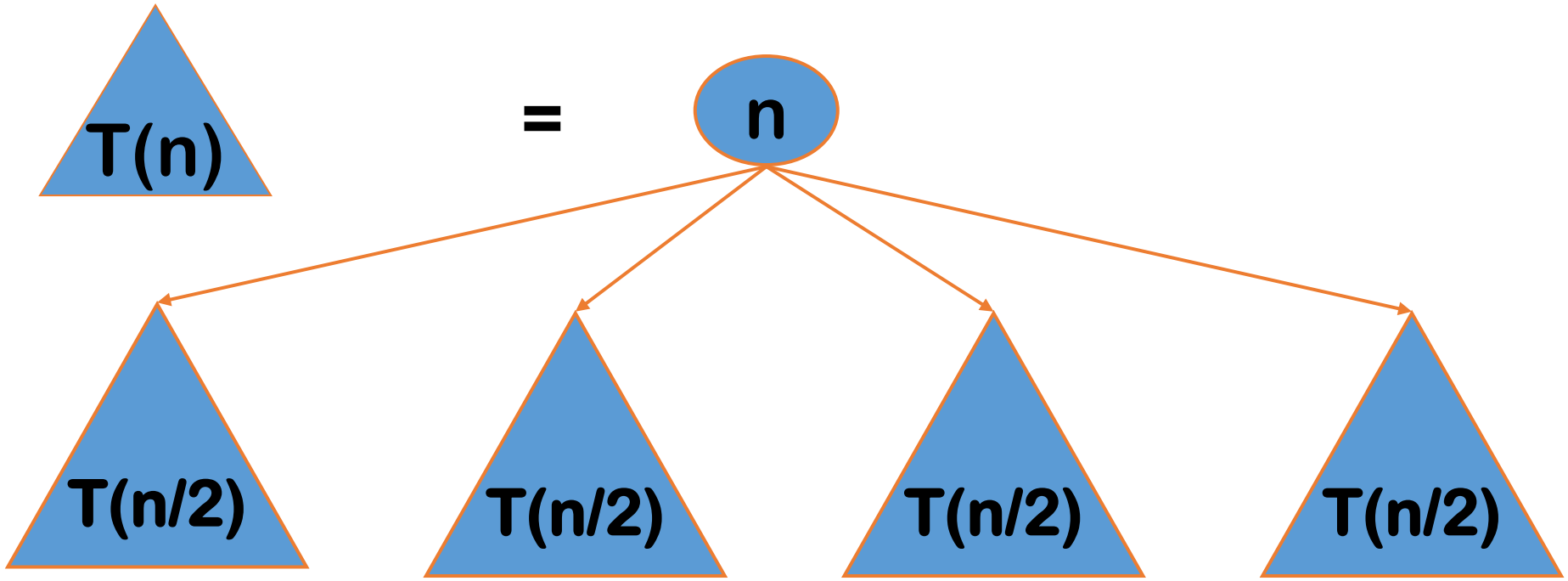
（6）流水作业调度；

（7）0-1背包问题；

（8）最优二叉搜索树。

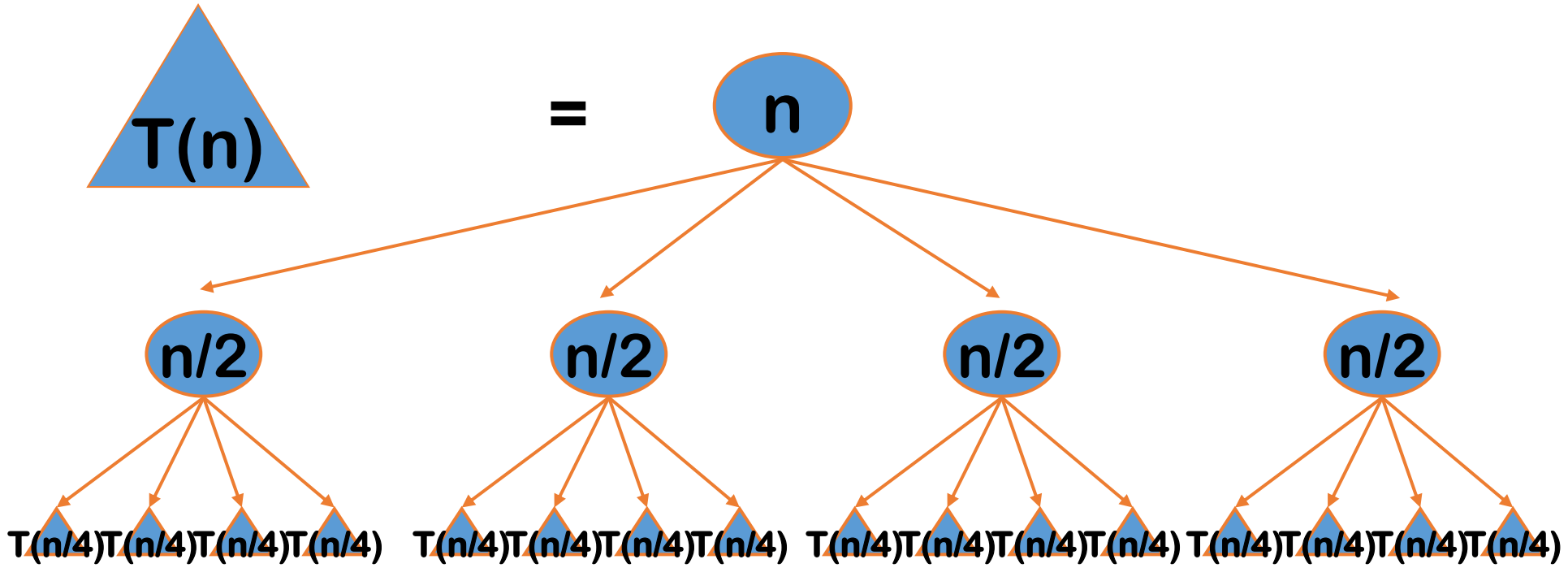
算法总体思想

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



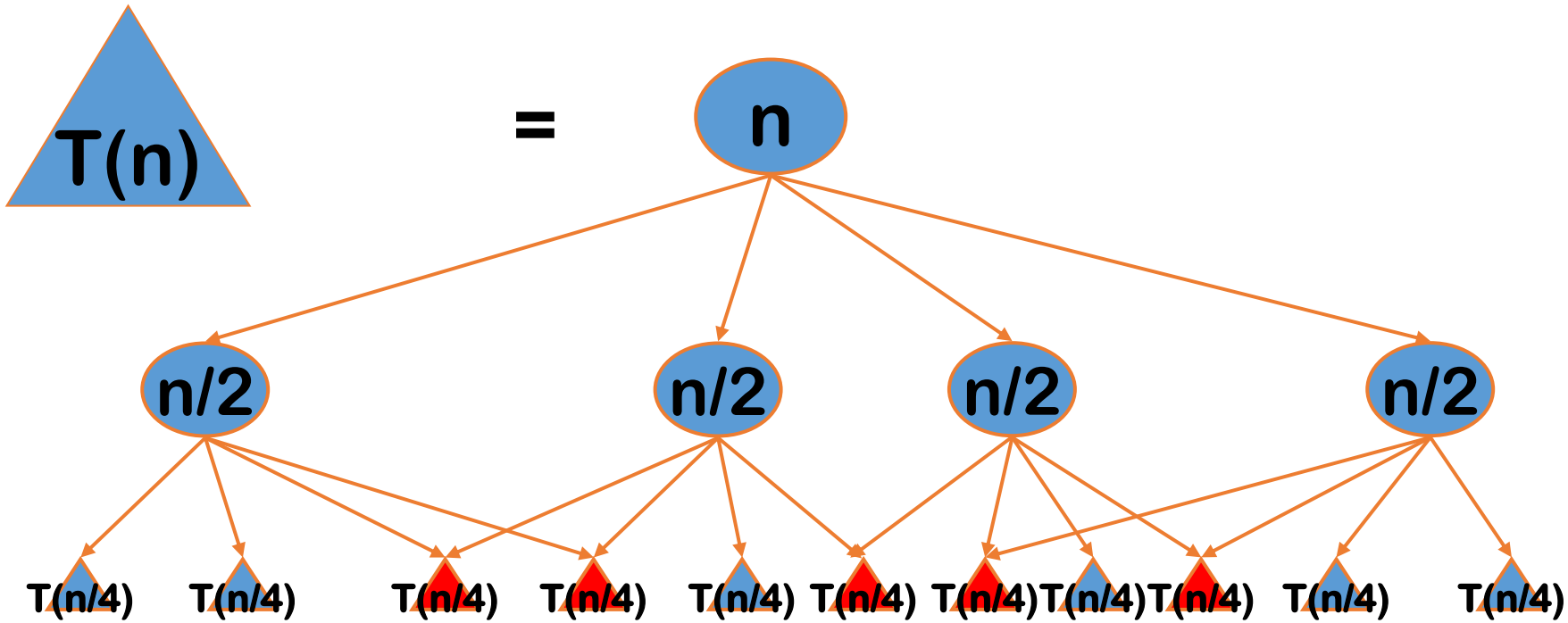
算法总体思想

但是在动态规划问题中，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。如果使用分治法求解，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。



动态规划基本步骤

① 找出最优解的性质，并刻画其结构特征。
② 递归地定义最优值。
③ 计算出最优值，通常采用自底向上的方式。
④ 根据计算最优值时得到的信息，构造最优解。

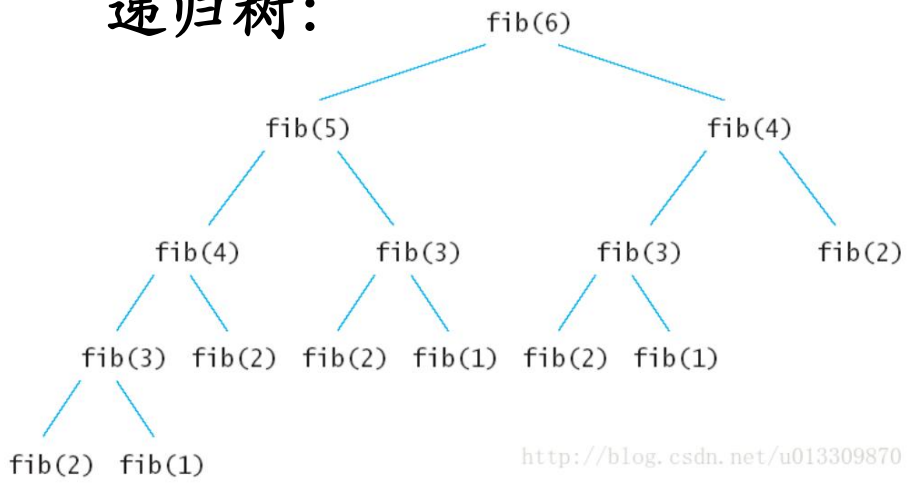
简单示例——斐波那契数列

Fibonacci (n) = 1; n = 0
Fibonacci (n) = 1; n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

自顶向下的递归版本：

```
public int fib(int n) {
    if(n<=0) return 0;
    if(n==1) return 1;
    return fib( n-1)+fib(n-2);
}//测试fib(6)
```

递归树：



上面的递归树中的每一个子节点都会执行一次，很多重复的节点被执行，fib(2)被重复执行了5次。由于调用每一个函数的时候都要保留上下文，所以空间上开销也不小。这么多的子节点被重复执行，如果在执行的时候把执行过的子节点保存起来，后面要用到的时候直接查表调用的话可以节约大量的时间。

自顶向下的备忘录方法


```
public static int Fibonacci(int n){
        if(n<=0)            return n;
        int []Memo=new int[n+1];
        for(int i=0;i<=n;i++)            Memo[i]=-1; //memo数组赋初值-1
        return fib(n, Memo);
}
public static int fib(int n,int []Memo)    {
        if(Memo[n]!=-1)            return Memo[n]; //不等于-1表示该值已计算过
        if(n<=2)            Memo[n]=1;
        else  Memo[n]=fib( n-1,Memo)+fib(n-2,Memo);
        return Memo[n];
}
```

创建了一个n+1大小的数组来保存求出的斐波拉契数列中的每一个值，在递归的时候如果发现前面fib（n）的值计算出来了就不再计算，如果未计算出来，则计算出来后保存在Memo数组中，下次在调用fib（n）的时候就不会重新递归了。

自底向上的方法


```
public static int fib(int n){
        if(n<=0)            return n;
        int []Memo=new int[n+1];
        Memo[0]=0;
        Memo[1]=1;
        for(int i=2;i<=n;i++) {
            Memo[i]=Memo[i-1]+Memo[i-2];
        }
        return Memo[n];
}
```

先计算子问题，再由子问题计算父问题。

自底向上的方法——改进

参与循环的只有i，i-1，i-2三项，因此该方法的空间可以进一步的压缩：

```
public static int fib(int n)    {
        if(n<=1)            return n;
        int Memo_i_2=0;
        int Memo_i_1=1;
        int Memo_i=1;
        for(int i=2;i<=n;i++){
            Memo_i=Memo_i_2+Memo_i_1;
            Memo_i_2=Memo_i_1;
            Memo_i_1=Memo_i;
        }
        return Memo_i;
}
```

一般来说由于备忘录方式的动态规划方法使用了递归，递归的时候会产生额外的开销，使用自底向上的动态规划方法要比备忘录方法好。

3.1 矩阵连乘问题

给定n个矩阵 $\{A_1, A_2, ;.., A_n\}$ 其中 A_i 与 A_{i+1}是可乘的， $i=1,2,\ldots,n-1$。
考察这n个矩阵的连乘积

$$A_1 A_2 \ldots A_n$$

由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有许多不同的计算次序。这种计算次序可以用给矩阵加括号的方式来确定。

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为：

（1）单个矩阵是完全加括号的；

（2）矩阵连乘积A是完全加括号的，则A可表示为2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）

$$B=10\times40 \quad C=40\times30$$

则 A=（BC）=10*30，使用到的乘法次数为 10*40*30 次

共12000次

完全加括号的矩阵连乘积

设有四个矩阵 A,B,C,D ，它们的维数分别是:

- A=50×10 B=10×40 C=40×30 D=30×5

总共有五种完全加括号的方式

$(A((BC)D))$ $(A(B(CD)))$ $((AB)(CD))$

$(((AB)C)D)$ $((A(BC))D)$

所需数乘次数分别为16000, 10500, 36000, 87500, 34500次

3.1 矩阵连乘问题

给定n个矩阵 $\{A_1,A_2,\cdots,A_n\}$ ，其中A_i与A_{i+1}是可乘的，i=1，2…，n-1。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，即计算量最少。

◆穷举法：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。

穷举法

算法复杂度分析：

对于n个矩阵的连乘积，设其不同的计算次序为P(n)种。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：$(A_1...A_k)(A_{k+1}\cdots A_n)$可以得到关于P(n)的递推式如下：

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

因此，计算次序的数量与n呈指数关系，通过穷举所有可能的方案来寻找最优方案，是一个糟糕的策略。

注意区分这里的计算次序数量与上文的数乘次数

3.1 矩阵连乘问题

- 穷举法
- 动态规划

将矩阵连乘积 $A_iA_{i+1}...A_j$ 简记为 $A[i:j]$，这里 i≤j 。

考察计算 $A[i:j]$ 的最优计算次序。假设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，其相应完全加括号方式为

$$(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$$

计算量（数乘次数）：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

分析最优解的结构

$$(A_1A_2\cdots\cdots A_k)\,(A_{k+1}\cdots\cdots A_n)$$

特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。

反证法：如果A[i:k]和A[k+1:j]的计算次序不是最优的，则原矩阵连乘的计算次序也不可能是最优的。

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。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

所谓的最优子结构，是指一个问题的解，可以由他的子问题的解来确定，而要想达到当前状态的最优解，则子问题的解也应该是最优的，即全局最优解包含局部最优解

建立递归关系

设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数m[i,j]，则原问题的最优值为m[1,n]。

用表s[i,j]记录m[i,j]对应的分割点k

假设Ai的维数为

- 当i=j时，A[i:j]=Ai，不需要数乘，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j$

新构成的两个子矩阵的大小分别为$p_{i-1}*p_k$和p_k*p_j，这两个矩阵相乘，需要的数乘次数为$p_{i-1}p_kp_j$

可以递归地定义m[i,j]为：

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

计算最优值

A[i:j]：对于1≤i≤j≤n不同的有序对(i, j)对应于不同的子问题。因此，不同子问题的个数最多只有

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

在[1,n]内任取两个不相同的数，共这些种方案

在[1,n]内任取两个相同的数，共n种方案

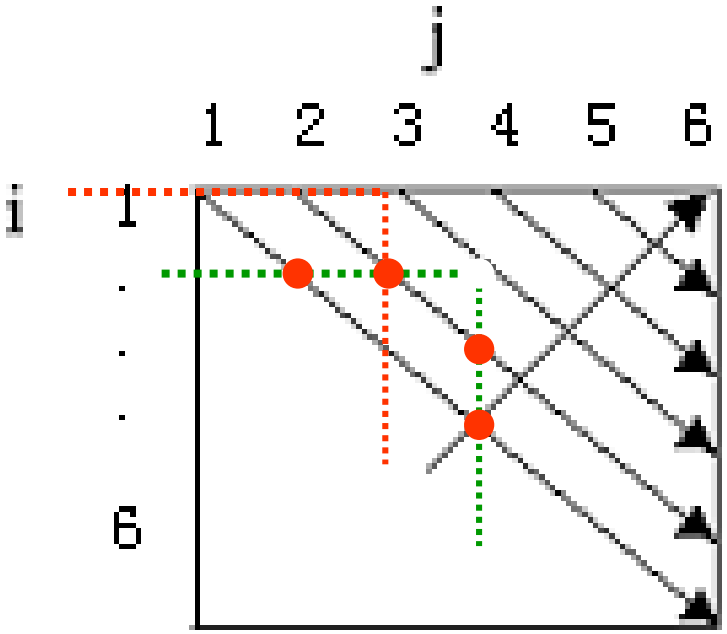
由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，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

求解过程

- 为避免重复计算，自底向上求解

在求解长度为r的矩阵链前，先把所有长度r-1的矩阵链都求完。以此类推。



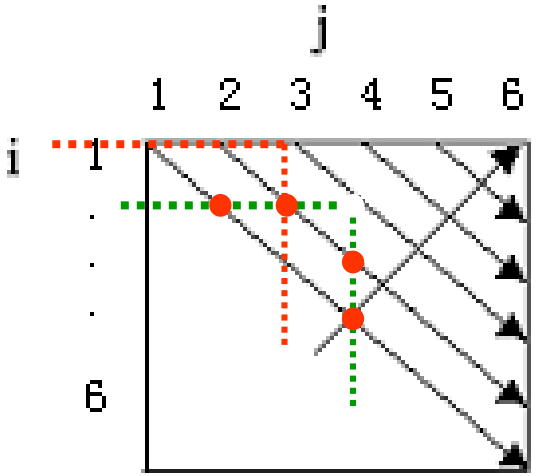
用动态规划法求最优解

```
void MatrixChain(int *p，int n，int **m，int **s){
    for i=1 to n    m[i][i]=0
    for r = 2 to n
        for i=1 to n-r+1
            j=i+r-1;
            m[i][j]=∞
            for k=i to j-1
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if (q < m[i][j])
                  m[i][j] = q;
                  s[i][j] = k;
}
```

先对m[i][i]置零(单矩阵)

在这个循环里，第一次循环，对i=1...n-1计算m[i][i+1]；
第二次循环，对i=1...n-2计算m[i][i+2]；即每次循环里，
计算长度为r的矩阵链的最小计算代价

k依次取值，寻找使m[i][j]最小的切割方案



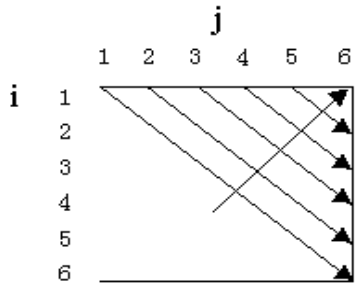
用动态规划法求最优解

例：6个矩阵，其大小分别为：

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

在算m[2][5]时，其过程为：

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$



(a) 计算次序

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

(b) m[i][j]

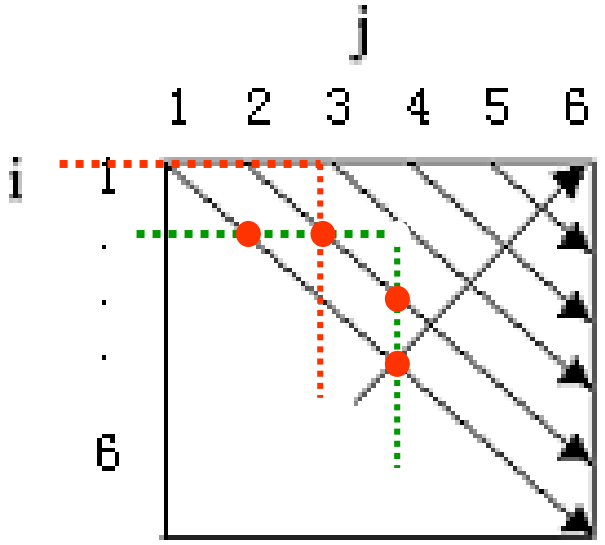
i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

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$O(n^2)$。

计算过程演示



- r=2:
- m[1][2]=P0*P1*P2=30*35*15=15750;
- m[2][3]=P1*P2*P3=35*15*5=2625;
- m[3][4]=P2*P3*P4=15*5*10=750;
- m[4][5]=P3*P4*P5=5*10*20=1000;
- m[5][6]=P4*P5*P6=10*20*25=5000;

- r=3:

$$m[1][3]=\min\begin{cases} m[1][2]+m[3][3]+P_0P_2P_3 \\ \underline{m[1][1]+m[2][3]+P_0P_1P_3} \end{cases}=\min\begin{cases} 18000 \\ \underline{7875} \end{cases}=7875$$

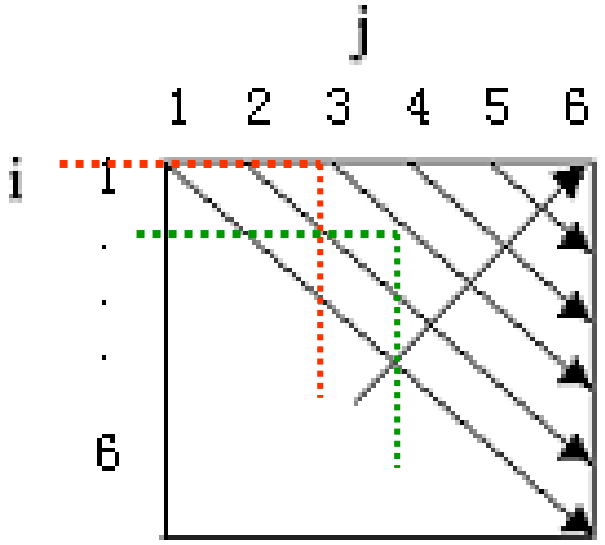
- s[1][3]=1;

$$m[2][4]=\min\begin{cases} m[2][3]+m[4][4]+P_1P_3P \\ m[2][2]+m[3][4]+P_1P_2P_4 \end{cases}^{4}=\min\begin{cases} 4375 \\ 6000 \end{cases}=4375$$

- s[2][4]=3;

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

$$m[3][5]=\min\begin{cases}m[3][4]+m[5][5]+P_2P_4P_5\\ \underline{m[3][3]+m[4][5]+P_2P_3P_5}\end{cases}=\min\begin{cases}3750\\ \underline{2500}\end{cases}=2500$$



- s[3][5]=3;

$$m[4][6]=\min\begin{cases}m[4][5]+m[6][6]+P_3P_5P_6\\ m[4][4]+m[5][6]+P_3P_4P_6\end{cases}=\min\begin{cases}\underline{3500}\\ 6250\end{cases}=3500$$

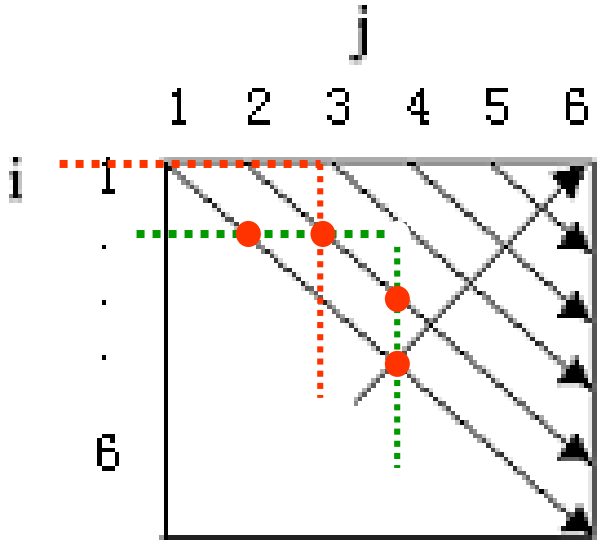
- s[4][6]=5;

- r=4:

$$m[1][4]=\min\begin{cases}m[1][1]+m[2][4]+P_0P_1P_4\\ m[1][2]+m[3][4]+P_0P_2P_4\\ \underline{m[1][3]+m[4][4]+P_0P_3P_4}\end{cases}=\min\begin{cases}14875\\ 21000=9375\\ \underline{9375}\end{cases}$$

- s[1][4]=3;

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+P_1P_2P_5 \\ \underline{m[2][3]+m[4][5]+P_1P_3P_5} \\ m[2][4]+m[5][5]+P_1P_4P_5 \end{cases}=\min\begin{cases} 13000 \\ \underline{7125} \\ 11375 \end{cases}=7125$$



- s[2][5]=3;

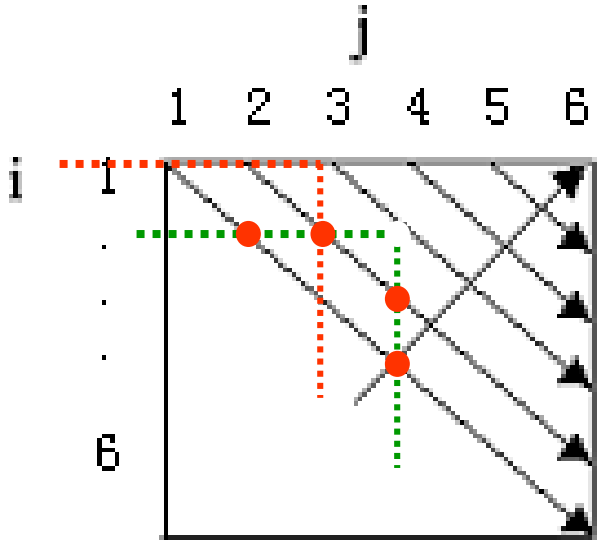
$$m[3][6]=\min\begin{cases} \underline{m[3][3]+m[4][6]+P_2P_3P_6} \\ m[3][4]+m[5][6]+P_2P_4P_6 \\ m[3][5]+m[6][6]+P_2P_5P_6 \end{cases}=\min\begin{cases} \underline{5375} \\ 9500 \\ 10000 \end{cases}=5375$$

- s[3][6]=3;
- r=5:

$$m[1][5]=\min\begin{cases} m[1][1]+m[2][5]+P_0P_1P_5=28125 \\ m[1][2]+m[3][5]+P_0P_2P_5=27250 \\ \underline{m[1][3]+m[4][5]+P_0P_3P_5=11875} \\ m[1][4]+m[5][5]+P_0P_4P_5=15375 \end{cases}=11875$$

- s[1][5]=3;

$$m[2][6]=\min\begin{cases} m[2][2]+m[3][6]+P_1P_2P_6=18500 \\ \underline{m[2][3]+m[4][6]+P_1P_3P_6=10500} \\ m[2][4]+m[5][6]+P_1P_4P_6=18125 \\ m[2][5]+m[6][6]+P_1P_5P_6=24625 \end{cases}=10500$$



- s[2][6]=3;
- r=6：

$$m[1][6]=\min\begin{cases} m[1][1]+m[2][6]+P_0P_1P_6=36750 \\ m[1][2]+m[3][6]+P_0P_2P_6=32375 \\ \underline{m[1][3]+m[4][6]+P_0P_3P_6=15125}=15125 \\ m[1][4]+m[5][6]+P_0P_4P_6=21875 \\ m[1][5]+m[6][6]+P_0P_5P_6=26875 \end{cases}$$

- s[1][6]=3;

s[1][6]=3

- 从而可知最优乘积次序为: [A1(A2A3)][(A4A5)A6]

s[1][3]=1　　s[4][6]=5

计算过程演示

- 验证数乘次数如下:
- A2A3数乘P1*P2*P3=2625次,矩阵(A2A3)的维数:P1*P3;
- A1(A2A3)数乘P0*P1*P3=5250次,矩阵A1(A2A3)的维数:P0*P3;
- A4A5数乘P3*P4*P5=1000次,矩阵A4A5的维数:P3*P5;
- (A4A5)A6数乘P3*P5*P6=2500次,矩阵(A4A5)A6的维数:P3*P6;
- (A1(A2A3))((A4A5)A6)数乘P0*P3*P6=3750次,其维数:P0*P6
- 故总的数乘次数=2625+5250+1000+2500+3750=15125.

3.2 动态规划算法的基本要素

一、最优子结构

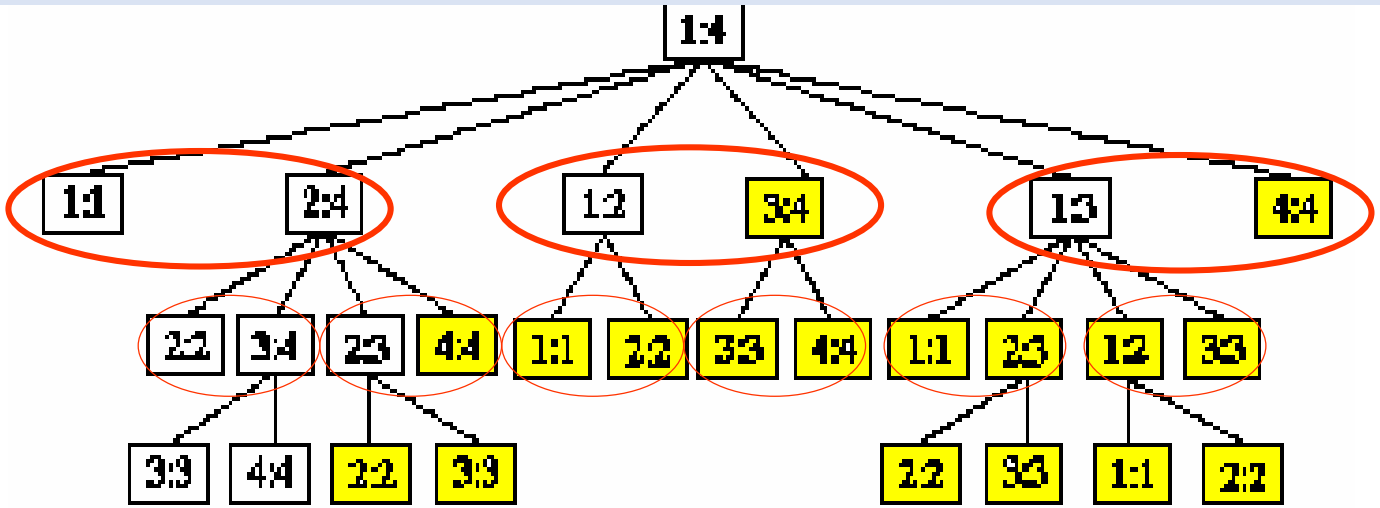
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

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）

3.2 动态规划算法的基本要素

二、重叠子问题

- 递归算法求解问题时，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。这种性质称为子问题的重叠性质。
- 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
- 通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。



3.2 动态规划算法的基本要素

三、除了自底向上方法，也可以用带有备忘录的自顶向下方法

•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，避免了相同子问题的重复求解。

```
int LookupChain(int i，int j)
{
    if (m[i][j] > 0) return m[i][j]; //子问题已解
    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; //记录最优分解位置
    for (int k = i+1; k < j; k++) {   //遍历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; //记录最优分解位置
        }
    }
    m[i][j] = u;    return u;
}
```

动态规划法 .VS. 分治法

- 动态规划法的实质也是将较大问题分解为较小的同类子问题，这一点上它与分治法类似。

- 分治法的子问题相互独立，相同的子问题被重复计算。

- 动态规划法利用问题的最优子结构特征，设计自底向上的计算过程，通过从子问题的最优解逐步构造出整个问题的最优解，避免重复计算。

动态规划基本步骤

设计一个动态规划算法，通常可以按以下几个步骤进行：

1. 找出最优解的性质，并刻划其结构特征。
2. 递归地定义最优值。
3. 计算出最优值，通常采用自底向上的方式。
4. 根据计算最优值时得到的信息，构造最优解。

3.3 最长公共子序列(Longest Common Subsequence, LCS)

- 若给定序列X={x1, x2,…, xm}，则另一序列Z={z1, z2, …, zk}是X的子序列是指存在一个严格递增下标序列{$i_1, i_2, \cdots, i_k$}, 使得对于所有j=1, 2, …, k, $z_j = x_{i\_j}$

- 例如，给定序列X={A，B，C，B，D，A，B}，序列Z={B，C，D，B}是X的子序列，相应的递增下标序列为{2，3，5，7}。

- 给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。

- 问题：给定2个序列X={$x_1, x_2, \cdots, x_m$}和Y={$y_1, y_2, \cdots, y_n$}，找出X和Y的最长公共子序列。

最长公共子序列的结构

设序列$X_m=\{x_1,x_2,\cdots,x_m\}$和$Y_n=\{y_1,y_2,\cdots,y_n\}$的最长公共子序列为$Z_k=\{z_1,z_2,\cdots,z_k\}$，从后向前推理，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}的最长公共子序列。
(2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$，则Z_k是X_{m-1}和Y_n的最长公共子序列。
(3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$，则Z_k是X_m和Y_{n-1}的最长公共子序列。

- 如果Z=LCS(X,Y)，则Z的任意前缀，是X的前缀与Y的前缀的LCS

2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有最优子结构性质。

子问题的递归结构

由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。用c[i][j]记录序列Xi和Yj的最长公共子序列的长度。其中，Xi={x1,x2,…,xi}；Yj={y1,y2,…,yj}。当i=0或j=0时，空序列是Xi和Yj的最长公共子序列。故此时c[i][j]=0。其它情况下，由最优子结构性质可建立递归关系如下：

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

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。

$$\begin{bmatrix} c[0][0] & c[0][1] & \cdots & c[0][n] \\ c[1][0] & c[1][1] & \cdots & c[1][n] \\ \vdots & \vdots & \vdots & \vdots \\ c[m][0] & c[m][1] & \cdots & c[m][n] \end{bmatrix}$$

要得到c[i][j]，需要其左侧、上侧和左上方的值，因此，先对c[i][0]和c[0][j]赋初值0

然后计算c[1][1]...c[m][1]，再计算c[1][2]...c[m][2]，以此类推，最终得到c[m][n]

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; //(1) “左上角”
      }
      else if (c[i-1][j]>=c[i][j-1]) {
        c[i][j]=c[i-1][j];
        b[i][j]=2; //(2) “上侧”
      }
      else {
        c[i][j]=c[i][j-1];
        b[i][j]=3; //(3) “左侧”
      }
  }}
```

构造最优解

```
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);
}
```

b[i][j]表示c[i][j]取值的三种情况，用来构造最优解

LCS的回溯构造最优解过程

c[i][j]存储最长公共子序列的长度

b[i][j]记录c[i][j]是由哪个子问题的解得到的

没有单独考虑c[i-1][j]==c[i][j-1]的情况，对b[i][j]二维数组的取值添加一种可能，等于4，来表示c[i-1][j]==c[i][j-1]。这样通过路径回溯可以找出所有最长公共子序列。不做这项更改则只能找出一个最优解。

构造LCS所有最优解的过程

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

指c[i][j]=4,
b[i][j]=1

```
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; //(1) “左上角”
        }
        else if (c[i-1][j]>c[i][j-1]) {
            c[i][j]=c[i-1][j];
            b[i][j]=2; //(2) “上侧”
        }
        else if (c[i-1][j]<c[i][j-1]) {
            c[i][j]=c[i][j-1];
            b[i][j]=3; //(3) “左侧”
        }
        else{                    // c[i-1][j]==c[i][j-1])
            c[i][j]=c[i-1][j];
            b[i][j]=4; //(4) “上侧和左侧相等”
        }
    }
```

比如序列 X 为 ABCBDAB，序列 Y 为 BDCABA，则
其公共最长子序列为 BCBA，或者 BCAB，或者BDAB

算法的改进

•在算法lcsLength和lcs中，可进一步将数组b省去。事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。对于给定的数组元素c[i][j]，可以不借助于数组b而仅借助于三个值的关系在O(1)时间内来确定c[i][j]的值是由c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的。

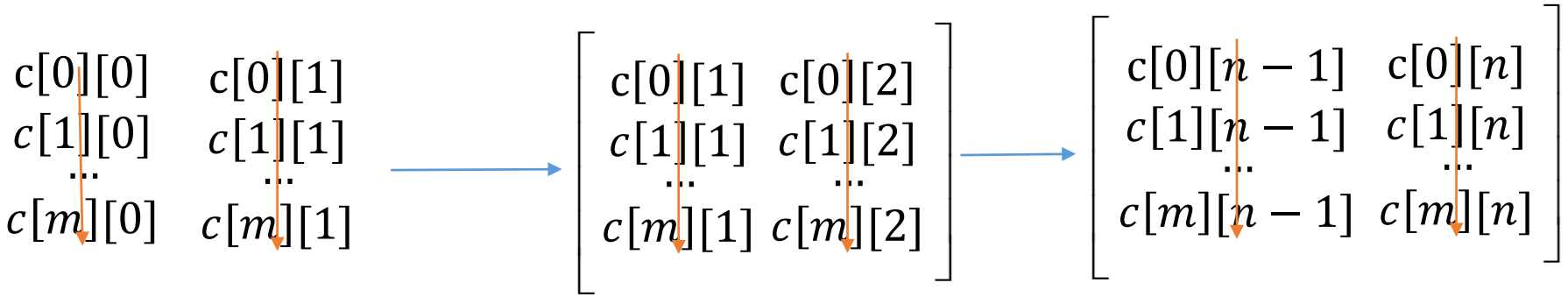
•如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

算法的改进

比如：

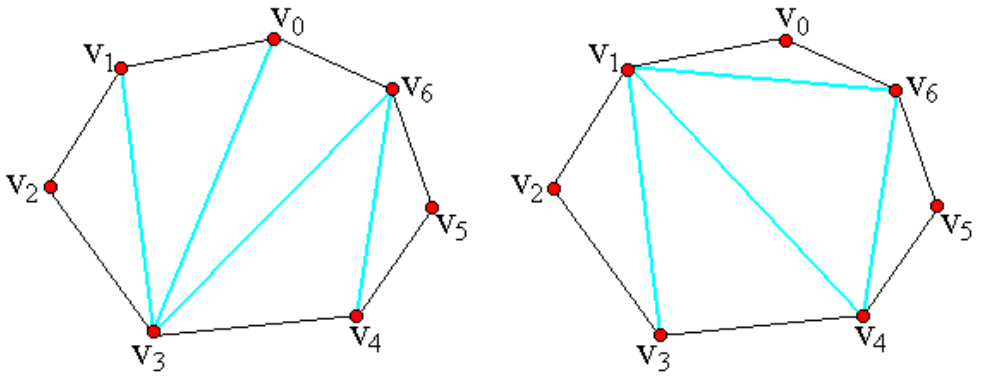
$$\begin{bmatrix} c[0][0] & c[0][1] & \cdots & c[0][n] \\ c[1][0] & c[1][1] & \cdots & c[1][n] \\ \vdots & \vdots & \vdots & \vdots \\ c[m][0] & c[m][1] & \cdots & c[m][n] \end{bmatrix}$$

只准备(m+1)*2大小的空间



3.5 凸多边形最优三角剖分

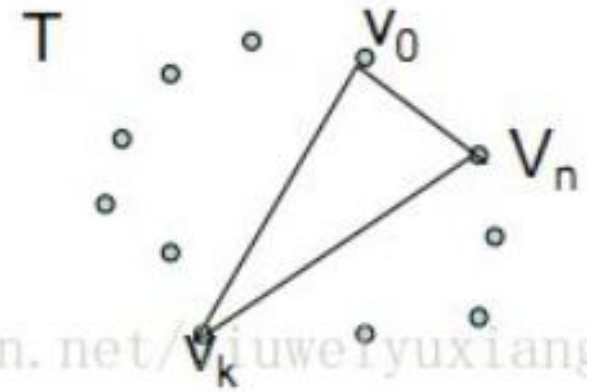
- 用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\cdots,v_{n-1}\}$表示具有n条边的凸多边形。
- 若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦。弦将多边形分割成2个多边形$\{v_i,v_{i+1},\cdots,v_j\}$和$\{v_j,v_{j+1},\cdots v_i\}$。
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合T。
- 给定凸多边形P，以及定义在由多边形的三个顶点组成的三角形上的权函数$w(v_iv_kv_j)$。要求确定该凸多边形的三角剖分，使得在该三角剖分中诸三角形上权之和为最小。



最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质。

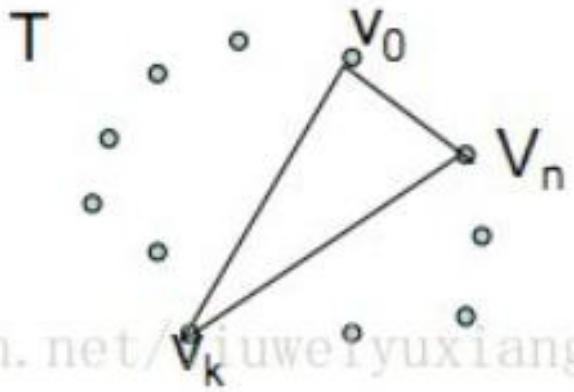
- 事实上，若凸(n+1)边形$P=\{v_0,v_1,\cdots,v_n\}$的最优三角剖分T包含三角形$v_0v_kv_n$，$1\leqslant k\leqslant n-1$，则T的权为3个部分权的和：三角形$v_0v_kv_n$的权，子多边形$\{v_0,v_1,\cdots,v_k\}$和$\{v_k,v_{k+1},\cdots,v_n\}$的权之和。可以断言，由T所确定的这2个子多边形的三角剖分也是最优的。因为若有$\{v_0,v_1,\cdots,v_k\}$或$\{v_k,v_{k+1},\cdots,v_n\}$的更小权的三角剖分将导致T不是最优三角剖分的矛盾。



最优三角剖分的递归结构

•定义t[i][j]，1≤i<j≤n为凸子多边形{v_{i-1},vi,…,v_j}的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形{v_{i-1},v_i}具有权值0。据此定义，要计算的凸(n+1)边形P的最优权值为t[1][n]。

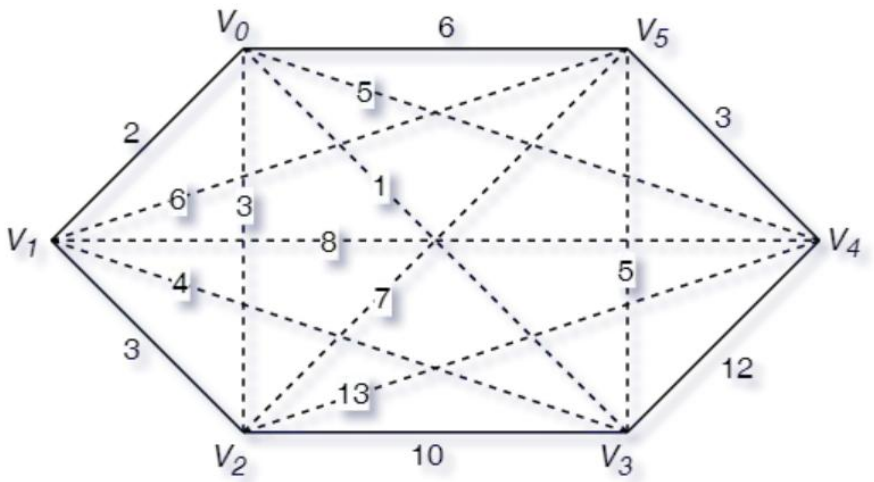
•t[i][j]的值可以利用最优子结构性质递归地计算。当j-i≥1时，凸子多边形至少有3个顶点。由最优子结构性质，t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中i≤k≤j-1。由于在计算时还不知道k的确切位置，而k的所有可能位置只有j-i个，因此可以在这j-i个位置中选出使t[i][j]值达到最小的位置。

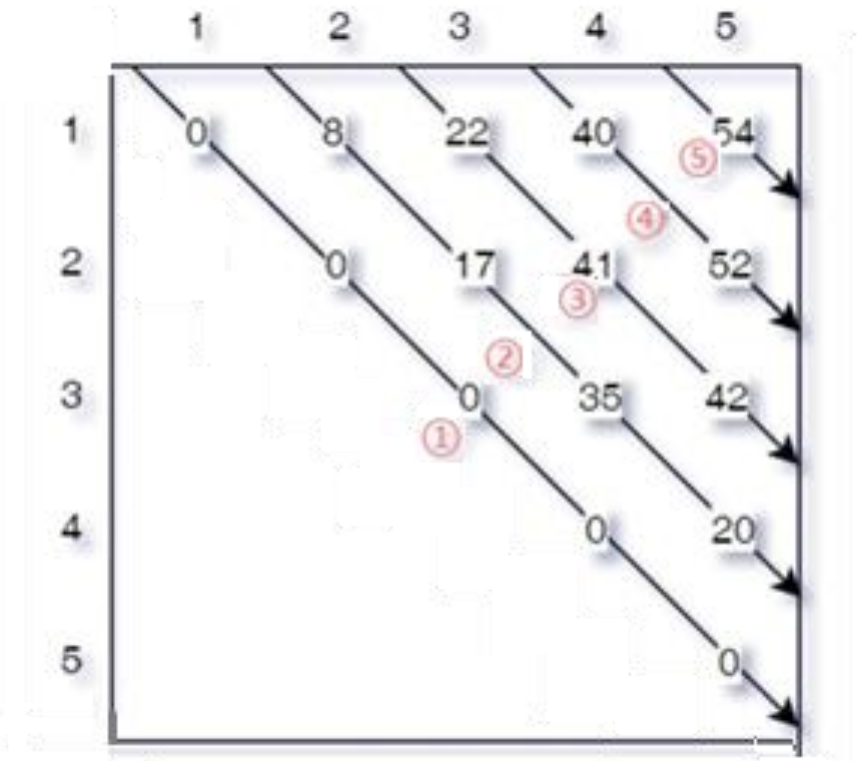


最优三角剖分的递归结构

- 由此，t[i][j]可递归地定义为：

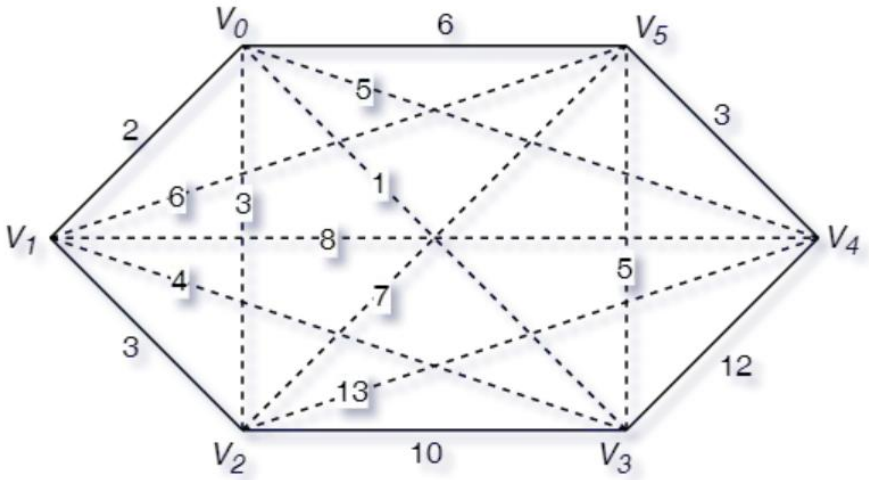
$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

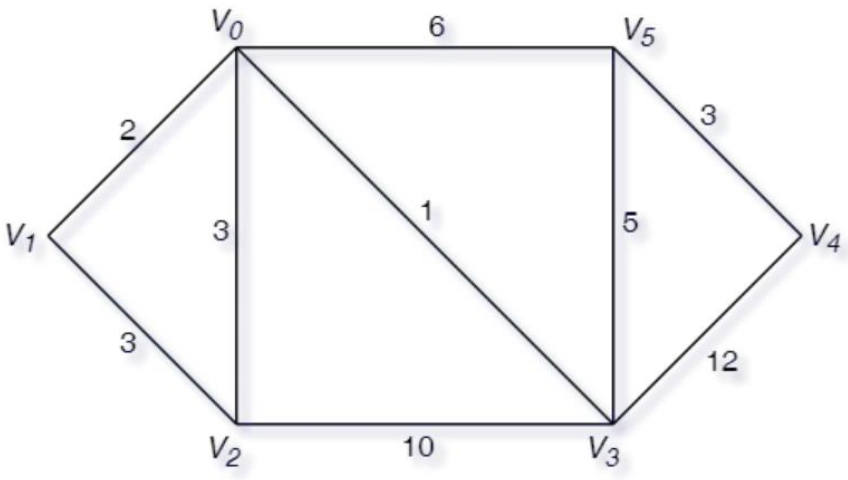




得到该递归式就可以写出动态规划算法

最优三角剖分





- 此凸 6 边形的最优三角剖分的权值之和为
  $(2 + 3 + 3) + (3 + 10 + 1) + (1 + 5 + 6) + (5 + 12 + 3) = 54$ 。

```
public static void minWeightTriangulation(int n) {
  for (int i = 1; i <= n; i++) {
     t[i][i] = 0;
  }
  for (int r = 2; r <= n; r++) {//i与j的差值
     for (int i = 1; i <= n - r + 1; i++) {
        int j = i + r - 1;
        m[i][j] = m[i + 1][j] + w(i-1,i,j);//k==i的情况
        s[i][j] = i;
        for (int k = i + 1; k < i+r-1; k++) {
           int u = m[i][k] + m[k + 1][j] + w(i-1,k,j);
           if (u < m[i][j]) {
              m[i][j] = u;
              s[i][j] = k; }}}}}
```

3.6 多边形游戏 (课堂不讲)

多边形游戏是一个单人玩的游戏，开始时有一个由n个顶点构成的多边形。每个顶点被赋予一个整数值，每条边被赋予一个运算符“+”或“*”。所有边依次用整数从1到n编号。

游戏第1步，将一条边删除。

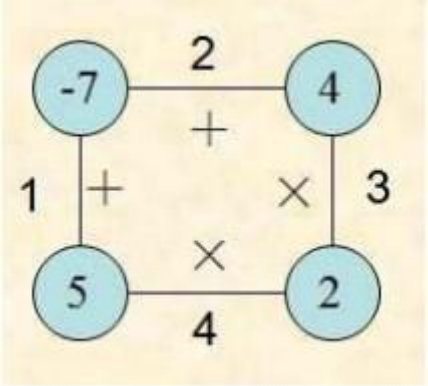
随后n-1步按以下方式操作：

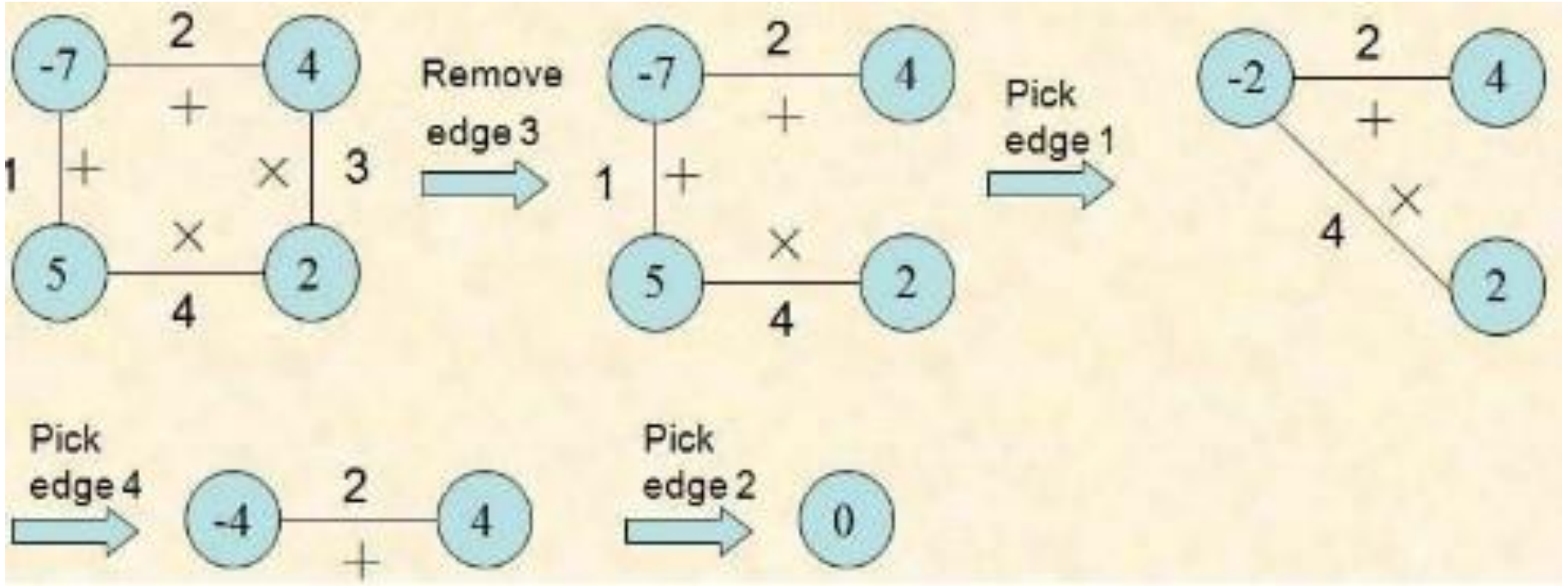
(1)选择一条边E以及由E连接着的2个顶点V_1和V_2；

(2)用一个新的顶点取代边E以及由E连接着的2个顶点V_1和V_2。将由顶点V_1和V_2的整数值通过边E上的运算得到的结果赋予新顶点。

最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值。

问题: 对于给定的多边形，计算最高得分。




-7
2
4
1
+
×
3
5
×
2
4
Remove
edge 3
-7
2
4
1
+
5
×
2
4
Pick
edge 1
-2
2
4
+
4
×
2
Pick
edge 4
-4
2
4
+
Pick
edge 2
0

多边形游戏规则

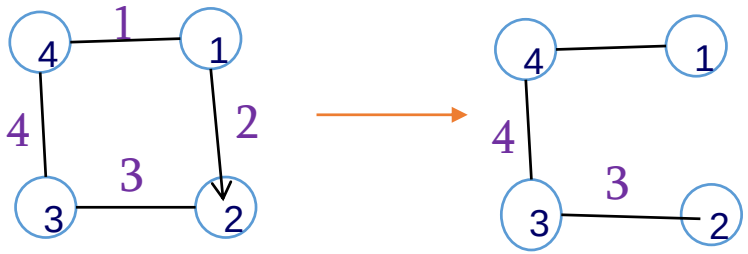
最优子结构性质

- 设所给的多边形的边和顶点的顺时针序列为op[1], v[1], op[2], v[2], op[3],…,op[n],v[n] 其中，op[i]表示第i条边所对应的运算符，v[i]表示第i个顶点上的数值，i=1~n。

- 在所给多边形中，从顶点i(1≤i≤n)开始，长度为j (链中有j个顶点)的顺时针链p(i,j) 可表示为v[i]，op[i+1]，…，v[i+j-1]。则将一条边op[i]删去后，得到的链为p(i,j)。

- 如果这条链的最后一次合并运算在op[i+s]处发生(1≤s≤j-1)，则可在op[i+s]处将链分割为2个子链p(i， s)和p(i+s，j-s)。
- 这里的加法需要进行取模运算，即mod n

j-s为长度，非下标



p(2,4)=v(2),op(3),....,v(1)

最优子结构性质

子链合并?

- 设m_1是对子链p(i，s)的任意一种合并方式得到的值，而a和b分别是在所有可能的合并中得到的最小值和最大值。m_2是p(i+s，j-s)的任意一种合并方式得到的值，而c和d分别是在所有可能的合并中得到的最小值和最大值。依此定义有$a \leqslant m_1 \leqslant b$，$c \leqslant m2 \leqslant d$

(1) 当op[i+s]='+'时，显然有$a+c \leqslant m \leqslant b+d$
(2) 当op[i+s]='*'时，有$\min\{ac，ad，bc，bd\} \leqslant m \leqslant \max\{ac，ad，bc，bd\}$

- 换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。

多边形游戏的递归求解

- 设m[i, j, 0]是链p(i, j)合并的最小值，而m[i, j, 1]是最大值。若最优合并在op[i+s]处将p(i, j)分为两个长度小于j的子链p(i, s)和p(i+s, j-s)的最大值和最小值均已计算出。即：

a = m[i, s, 0]
b = m[i, s, 1]
c = m[i+s, j-s, 0]
d = m[i+s, j-s, 1]

(1) 当op[i+s]='+'时

m[i, j, 0] = a + c
m[i, j, 1] = b + d

(2) 当op[i+s]='*'时

m[i, j, 0] = min{ac, ad, bc, bd}
m[i, j, 1] = max{ac, ad, bc, bd}

多边形游戏的递归求解

综合 (1)和(2),将p(i, j)在op[i+s]处断开的最大值记为maxf(i, j, s)， 最小值记为minf(i, j, s)

$$\text{minf}(i,j,s)=\begin{cases} a+c, & op[i+s]='+' \\ min\{ac,ad,bc,bd\}, & op[i+s]='*' \end{cases}$$

$$\text{maxf}(i,j,s)=\begin{cases} b+d, & op[i+s]='+' \\ \max\{ac,ad,bc,bd\}, & op[i+s]='*' \end{cases}$$

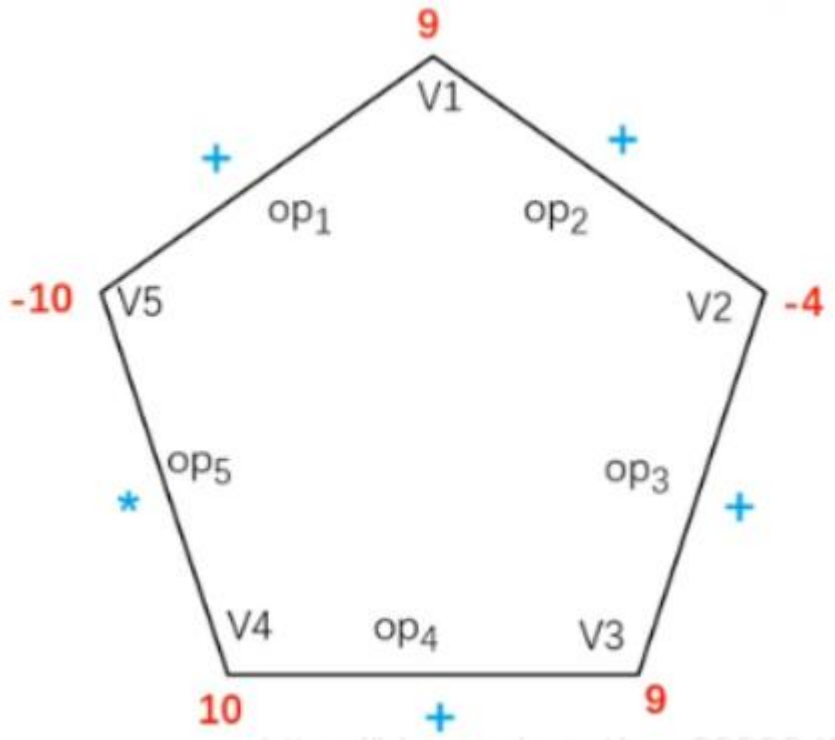
由于最优断开位置s有1 ≤ s ≤ j-1的j-1中情况。

$$m[i,j,0]=\min\{minf(i,j,s)\},\ 1\le i,j\le n \qquad 1\le s\le j$$
$$m[i,j,1]=\max\{maxf(i,j,s)\},\ 1\le i,j\le n \qquad 1\le s\le j$$

初始边界值为 m[i, 1, 0] = v[i] m[i, 1, 1] = v[i] 1 ≤ i ≤ n

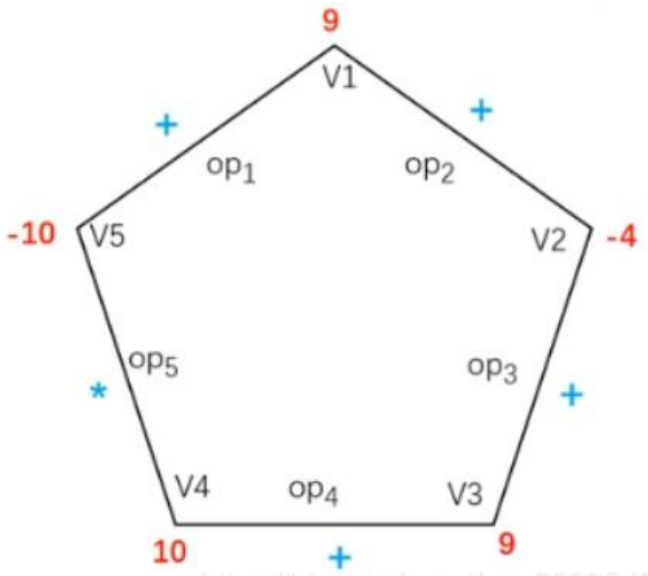
多边形是封闭的， 当i+s>n时， 顶点i+s实际编号为(i+s)%n。m[i,n,1]记为游戏首次删除第i条边后得到的最大得分。

多边形游戏样例

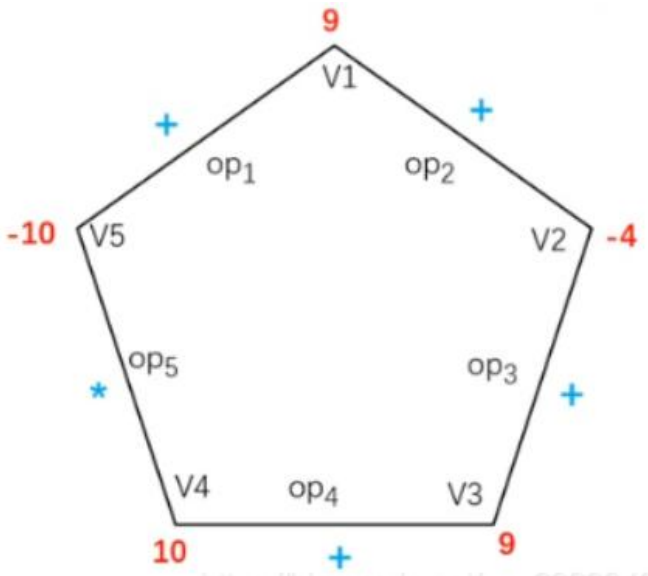


m(1,1,1) = 9
m(1,1,0) = 9
m(2,1,1) = -4
m(2,1,0) = -4
m(3,1,1) = 9
m(3,1,0) = 9
m(4,1,1) = 10
m(4,1,0) = 10
m(5,1,1) = -10
m(5,1,0) = -10

多边形游戏样例



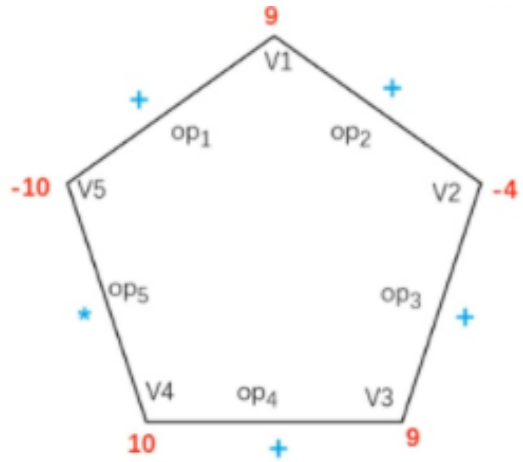
m(1,2) = m(1,1)op(2)m(2,1); op(2) = “+”
m(1,2,1) = m(1,1,1) + m(2,1,1) = 5
m(1,2,0) = m(1,1,0) + m(2,1,0) = 5
m(2,2) = m(2,1)op(3)m(3,1); op(3) = “+”
m(2,2,1) = m(2,1,1) + m(3,1,1) = 5
m(2,2,0) = m(2,1,0) + m(3,1,0) = 5
m(3,2) = m(3,1)op(4)m(4,1); op(4) = “+”
m(3,2,1) = m(3,1,1) + m(4,1,1) = 19
m(3,2,0) = m(3,1,0) + m(4,1,0) = 19
m(4,2) = m(4,1)op(5)m(5,1); op(5) = “*”
m(4,2,1) = max{m(4,1,1)*m(5,1,1), m(4,1,1)*m(5,1,0), m(4,1,0)*m(5,1,1), m(4,1,0)*m(5,1,0)} = -100
m(4,2,0) = min{m(4,1,1)*m(5,1,1), m(4,1,1)*m(5,1,0), m(4,1,0)*m(5,1,1), m(4,1,0)*m(5,1,0)} = -100
m(5,2) = m(5,1)op(1)m(1,1)
m(5,2,1) = m(5,1,1) + m(1,1,1) = -1
m(5,2,0) = m(5,1,0) + m(1,1,0) = -1



m(1,3,1) = max{m(1,1,1)+m{2,2,1}, m(1,2,1)+m(3,1,1)} = 14
(m(1,1,1)+m(2,2,1))
m(1,3,0) = min{m(1,1,0)+m{2,2,0}, m(1,2,0)+m(3,1,0)} = 14
(m(1,1,0)+m(2,2,0))
m(2,3,1) = max{m(2,1,1)+m(3,2,1), m(2,2,1)+m(4,1,1)} = 15 (m(2,1,1)+m(3,2,1))
m(2,3,0) = min{m(2,1,0)+m(3,2,0), m(2,2,0)+m(4,1,0)} = 15 (m(2,1,0)+m(3,2,0))
m(3,3,1) = max{m(3,1,1)+m(4,2,1), max{m(3,2,1)*m(5,1,1), m(3,2,1)*m(5,1,0),
m(3,2,0)*m(5,1,1), m(3,2,0)*m(5,1,0)}} = -91 (m(3,1,1)+m(4,2,1))
m(3,3,0) = min{m(3,1,0)+m(4,2,0), min{m(3,2,1)*m(5,1,1), m(3,2,1)*m(5,1,0),
m(3,2,0)*m(5,1,1), m(3,2,0)*m(5,1,0)}} = -190 (m(3,2,1)*m(5,1,1))
m(4,3,1) = max{max{m(4,1,1)*m(5,2,1), m(4,1,1)*m(5,2,0), m(4,1,0)*m(5,2,1),
m(4,1,0)*m(5,2,0)}, m(4,2,1)+m(1,1,1)} = -10 (m(4,1,1)*m(5,2,1))
m(4,3,0) = min{min{m(4,1,1)*m(5,2,1), m(4,1,1)*m(5,2,0), m(4,1,0)*m(5,2,1),
m(4,1,0)*m(5,2,0)}, m(4,2,0)+m(1,1,0)} = -91 (m(4,2,0)+m(1,1,0))
m(5,3,1) = max{m(5,1,1)+m(1,2,1), m(5,2,1)+m(2,1,1)} = -5 (m(5,1,1)+m(1,2,1))
m(5,3,0) = min{m(5,1,0)+m(1,2,0), m(5,2,0)+m(2,1,0)} = -5 (m(5,1,0)+m(1,2,0))

max = max{m(1,5,1), m(2,5,1), m(3,5,1), m(4,5,1), m(5,5,1)} = 40
min = min{m(1,5,0), m(2,5,0), m(3,5,0), m(4,5,0), m(5,5,0)} = -240

	1	2	3	4	5
1	9 9	5 5	14 14	24 24	-86 -240
2	-4 -4	5 5	15 15	-95 -194	-5 -185
3	9 9	19 19	-91 -190	-1 -181	-5 -185
4	10 10	-100 -100	-10 -91	-14 -95	40 -86
5	-10 -10	-1 -1	-5 -5	4 4	14 14



所以

max = 40
min=-240

3.7 图像压缩

在计算机中，常用像素点的灰度值序列{p1,p2,……pn}表示图像。其中整数pi, 1<=i<=n，表示像素点i的灰度值。通常灰度值的范围是0~255。因此最多需要8位表示一个像素。

一张分辨率为640 x 480的图片，那它的像素就达到了307200，也就是人们常说的30万像素

3.7 图像压缩

图像的变位压缩存储格式将所给的像素点序列$\{p_1,p_2,\cdots,p_n\}$, $0\leqslant p_i\leqslant 255$分割成m个连续段$S_1,S_2,\cdots,S_m$。

第i个像素段S_i中$(1\leqslant i\leqslant m)$，有l[i]个像素,且该段中每个像素都只用b[i]位表示。

分段的过程就是要找出断点，让一段里面的像素的最大灰度值比较小，那么这一段像素(本来需要8位)就可以用较少的位(比如7位)来表示，从而减少存储空间。

设 $t[i]=\sum_{k=1}^{i-1} l[k]$ ，则第i个像素段S_i为 $S_i=\{p_{t[i]+1},\cdots,p_{t[i]+l[i]}\}$

比如某个片段为：

p_i=10, p_{i+1}=15, p_{i+2}=100, p_{i+3}=55, p_{i+4}=200, p_{i+5}=255

可以分成p_i-p_{i+3}和p_{i+4}-p_{i+5}两段，而第一段的每个像素只需要7个比特位就可以表示。

图像压缩

本题中，$0 \leqslant p_i \leqslant 255$，因此$b[i] \leq 8$，即需要用3位表示b[i]，如果限制每段不超过255个像素，即$1 \leq l[i] \leq 255$，则需要用8位表示l[i]。因此，第i个像素段所需的存储空间为$l[i]*b[i]+8+3= l[i]*b[i]+11$位。

按此格式存储像素序列$\{p_1,p_2,\cdots,p_n\}$，需要 $\sum_{i=1}^{m} l[i]*b[i]+11m$ 位的存储空间。

图像压缩问题要求确定像素序列$\{p_1, p_2,\cdots, p_n\}$的最优分段，使得依此分段所需的存储空间最少。每个分段的像素不超过256个。

图像压缩

设l[i]，b[i]， $1 \leqslant i \leqslant m$ ，是$\{p_1,p_2,\cdots,p_n\}$的最优分段。显而易见，l[1]，b[1]是$\{p_1,\cdots,p_{l[1]}\}$的最优分段，且l[i]，b[i]， $2 \leqslant i \leqslant m$是$\{p_{l[1]+1},\cdots,p_n\}$ 的最优分段。即图像压缩问题满足最优子结构性质。

设s[i]，$1 \leqslant i \leqslant n$，是像素序列$\{p_1,\cdots,p_i\}$的最优分段所需的存储位数，则s[i]为前i-k个的存储位数（s[i-k]已算过）加上后k个的存储位数（需再计算）。由最优子结构性质易知：

$$s[i] = \min_{1 \le k \le \min\{i,256\}} \{s[i-k] + k * \mathrm{bmax}(i-k+1, i)\} + 11$$

其中 $\mathrm{bmax}(i,j) = \left\lceil \log\left(\max_{i \le k \le j} \{p_k\} + 1 \right) \right\rceil$ 为pi到pj中，最大的值需要的比特位数。

i-k

求像素序列 4，6，5，7，129，138，1 的最优分段

4	6	5	7	129	138	1
1	*2*	*3*	*4*	*5*	*6*	*7*

$s[1]=3+11=14 \quad l[1]=1 \quad b[1]=3$

$$s[2]=\min\begin{cases} s[0]+2\times 3+11 \\ s[1]+3+11 \end{cases}=\min\{17,28\}=17$$

$l[2]=2 \quad b[2]=3$

$$s[3]=\min\begin{cases} s[0]+3\times 3+11 \\ s[1]+2\times 3+11 \\ s[2]+3+11 \end{cases}=\min\{20,31,31\}=20$$

$l[3]=3 \quad b[3]=3$

$$s[4]=\min\begin{cases} s[0]+4\times 3+11 \\ s[1]+3\times 3+11 \\ \cdots \end{cases}=\min\{23,34,34,34\}=23$$

$l[4]=4 \quad b[4]=3$

$$s[i]=\min_{1\le k\le \min\{i,256\}}\{s[i-k]+k*\mathrm{b}\max(i-k+1,i)\}+11$$

求像素序列 4，6，5，7，129，138，1 的最优分段

4	6	5	7	129	138	1
1	*2*	*3*	*4*	*5*	*6*	*7*

s[5] = min{51, 57, 52, 47, 42} = 42
l[5] = 1 b[5] = 8

s[6] = min{59, 65, 60, 55, 50, 61} = 50
l[6] = 2 b[6] = 8

s[7] = min{67, 73, 68, 63, 58, 69, 62} = 58
l[7] = 3 b[7] = 8

$$s[i] = \min_{1 \le k \le \min\{i,256\}} \{s[i-k] + k * \mathrm{b}\max(i-k+1, i)\} + 11$$

算法复杂度分析：

由于算法**compress**中对k的循环次数不超过256，故对每一个确定的i，可在时间O(1)内完成的计算。因此整个算法所需的计算时间为O(n)。

```
void Compress(int n, int p[], int s[], int l[], int b[]) { //令s[i]为前i个段最优合并的存储位数
    int Lmax = 256, header = 11;
    s[0] = 0;
    for(int i=1; i<=n; i++) {  //i表示前几段
        b[i] = length(p[i]); //计算像素点p需要的存储位数
        int bmax = b[i];
        s[i] = s[i-1] + bmax+header;  //最后一段有一个像素
        l[i] = 1;
        for(int j=2; j<=i && j<=Lmax; j++){  // 最后一段有两个像素，三个像素，…,全部i个像素（段内像素的数量强制不超过256）
        //递推关系:s[i]=  min {s[i-j]+ lsum(i-j+1,i)*bmax(i-j+1,i) } + 11
            if(bmax < b[i-j+1])
                bmax = b[i-j+1];
            if(s[i] > s[i-j] + j*bmax+header) {
            //因为一开始所有序列并没有分段,所以可以看作每一段就是一个数,故lsum(i-j+1, i) = j;
                s[i] = s[i-j] + j*bmax+header;
                l[i] = j;  //记录最优断开位置，便于构造最优解
            }
        }
    }
}
```

3.9 流水作业调度(课堂不讲)

n个作业{1，2，…，n}要在由2台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业i所需的时间分别为ai和bi。

流水作业调度问题要求确定这n个作业的最优加工顺序，使得从第一个作业在机器M1上开始加工，到最后一个作业在机器M2上加工完成所需的时间最少。

3.9 流水作业调度

分析：

- 直观上，一个最优调度应使机器M_1没有空闲时间，且机器M_2的空闲时间最少。在一般情况下，机器M_2上会有机器空闲和作业积压2种情况。

- 设全部作业的集合为N={1，2，…，n}。S⊆N是N的作业子集。在一般情况下，机器M1开始加工S中作业时，机器M2还在加工其它作业，要等时间t后才可利用。将这种情况下完成S中作业所需的最短时间记为T(S,t)。流水作业调度问题的最优值为T(N,0)。

3.9 流水作业调度

设有3个作业

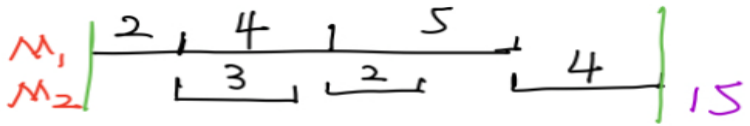
i	1	2	3
a_i	2	4	5
b_i	3	2	4

那么根据排列组合知识 按作业序号有

123 . 132 . 213 . 231 , 312 . 321 共6种情况

选择 123 . 321这两种情况来说明

123：

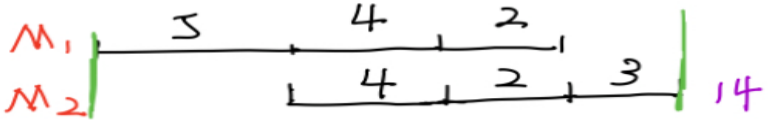


T(1,{2,3},0)

T(2,{3},3)

T(3,{},2) =9

321：



T(3,{2,1},0)

T(2,{1},4)

T(1,{},2) = 5

3.9 流水作业调度

设π是所给n个流水作业的一个最优调度，它所需的加工时间为$a_{\pi(1)}+T'$。其中T'是在机器M2的等待时间为$b_{\pi(1)}$时，安排作业π(2)，…，π(n)所需的时间。
记S=N-{π(1)}，则有$T'=T(S,b_{\pi(1)})$。

证明：事实上，由T的定义知$T'\geq T(S,b_{\pi(1)})$。若$T'>T(S,b_{\pi(1)})$，设π'是作业集S在机器M2的等待时间为$b_{\pi(1)}$情况下的一个最优调度。则π(1)，π'(2)，…，π'(n)是N的一个调度，且该调度所需的时间为$a_{\pi(1)}+T(S,b_{\pi(1)})<a_{\pi(1)}+T'$。这与π是N的最优调度矛盾。故$T'\leq T(S,b_{\pi(1)})$。从而$T'=T(S,b_{\pi(1)})$。这就证明了流水作业调度问题具有最优子结构的性质。

由流水作业调度问题的最优子结构性质可知，

$$T(N,0)=\min_{1\leq i\leq n}\{a_i+T(N-\{i\},b_i)\}$$

$$T(S,t)=\min_{i\in S}\{a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})\}$$

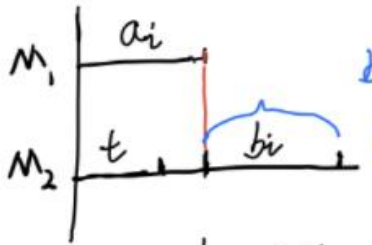
选定作业i为S中第一个加工作业之后，在机器M2上开始对S-{i}中的作业进行加工之前，所需要的等待时间为bi + max{t-ai,0}。

原因：若M2在开始加工S中的作业之前需等待t个时间单位，且t > ai，则作业i在M1上加工完毕（需时ai）之后，还要再等t-ai个时间单位才能开始在M2上加工；若t≤ai，则作业i在M1上加工完毕之后，立即可以在M2上加工，等待时间为0。故M2在开始对S-{i}中的作业进行加工之前，所需要的等待时间为t’ = bi + max{t-ai,0}。（bi是作业i在M2上加工所需的时间）。所以，假定ai为已知的使得T(S,t)值最小的第一个执行的作业，可以得到

$$T(S,t) = ai + T(S-\{i\}, bi + \max\{t-ai,0\})$$

$T(S,t)$：M_1开始加工S中的作业i时，M_2要等待t时后可利用

情形1：$a_i > t$

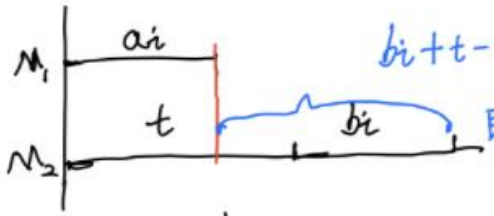


b_i

即 下一个作业开始时 M_2要等b_i时间后才可以利用

则 $T(S,t) = \{a_i + T(S-\{i\}, b_i)\}$，$i \in S$ ①

情形2：$a_i < t$



$b_i + t - a_i$

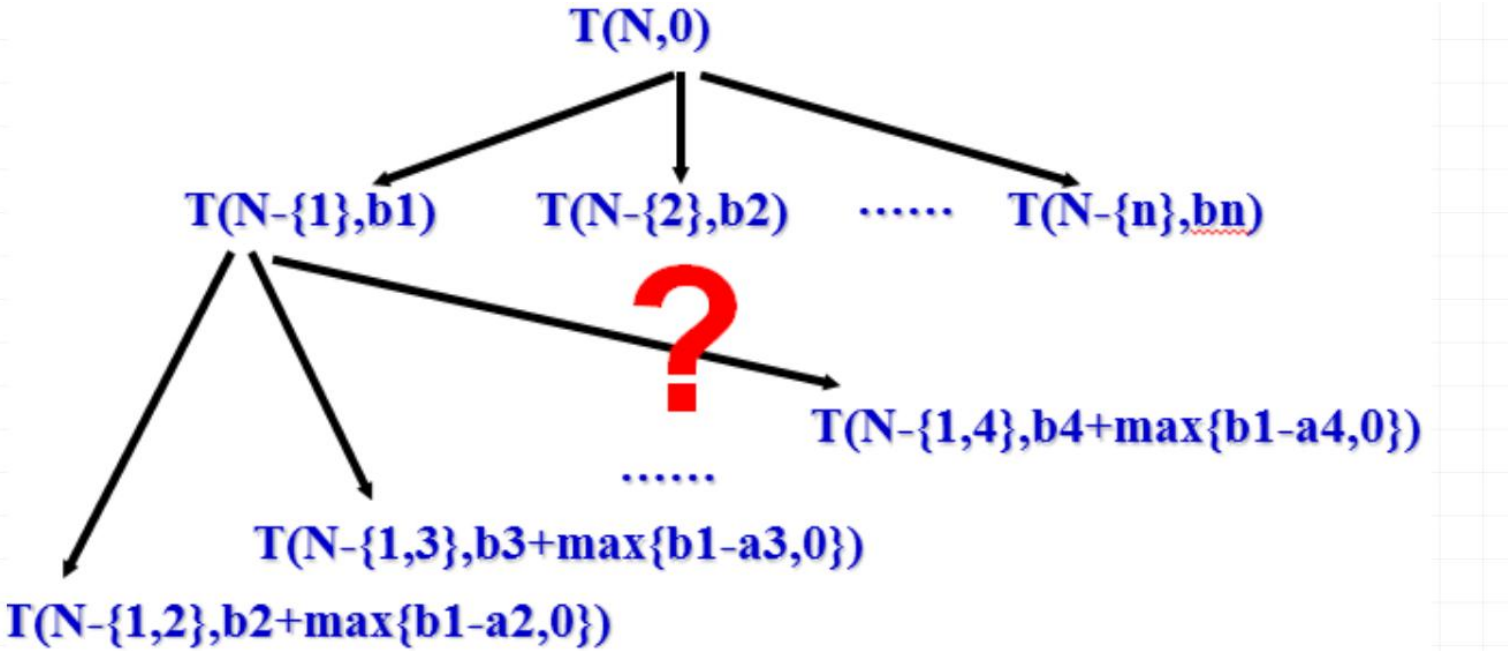
即 下一个作业开始，M_2要等 $b_i + t - a_i$时间后才可以利用

则 $T(S,t) = \{a_i + T(S-\{i\}, b_i + t - a_i)\}$ $i \in S$ ②

综合①②得

$$T(S,t) = \{a_i + T(S-\{i\}, b_i + \max(t-a_i, 0))\}$$

子问题重叠性质



虽然满足最优子结构性质，也在一定程度满足子问题重叠性质。N的每个非空子集都计算一次，共2^n-1次，指数级的。

为了解决这个问题引入Johnson不等式

Johnson不等式

对递归式的深入分析表明，算法可进一步得到简化。

设π是作业集S在机器M2的等待时间为t时的任一最优调度。若π(1)=i, π(2)=j。则由动态规划递归式可得:

T(S,t)=ai+T(S-{i},bi+max{t-ai,0})=ai+aj+T(S-{i,j},tij)

其中，$t_{ij} = b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\}$

$= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\}$

$= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\}$

$= b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}$

流水作业调度的Johnson法则

$t_{ij}=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_i,a_i\}$　　T(S,t)=ai+T(S-{i},bi+max{t-ai,0})=ai+aj+T(S-{i,j},t_{ij})

交换作业i和作业j的加工顺序，得到作业集S的另一调度，它所需的加工时间为

T’(S,t)=ai+aj+T(S-{i,j},t_{ji})

其中，$t_{ji}=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_j,a_j\}$

当作业i和j满足Johnson不等式时，有

$$\max\{-b_i,-a_j\}\le\max\{-b_j,-a_i\}$$

$$a_i+a_j+\max\{-b_i,-a_j\}\le a_i+a_j+\max\{-b_j,-a_i\}$$

$$\max\{a_i+a_j-b_i,a_i\}\le\max\{a_i+a_j-b_j,a_j\}$$

$$\max\{t,a_i+a_j-b_i,a_i\}\le\max\{t,a_i+a_j-b_j,a_j\}$$

如果作业i和j满足min{bi,aj}≥min{bj,ai}，则称作业i和j满足Johnson不等式。

由此可见当作业i和作业j满足Johnson不等式时，交换它们的加工顺序后，会增加加工时间。对于流水作业调度问题，必存在最优调度π，使得任意i，作业π(i)和π(i+1)满足Johnson不等式。进一步还可以证明，调度满足Johnson法则当且仅当对任意i<j有

$$\min\{b_{\pi(i)},a_{\pi(j)}\}\ge\min\{b_{\pi(j)},a_{\pi(i)}\}$$

由此可知，所有满足Johnson法则的调度均为最优调度。

算法描述

流水作业调度问题的Johnson算法

(1)令 $N_1 = \{i | a_i < b_i\}, N_2 = \{j | a_j \geq b_j\}$;

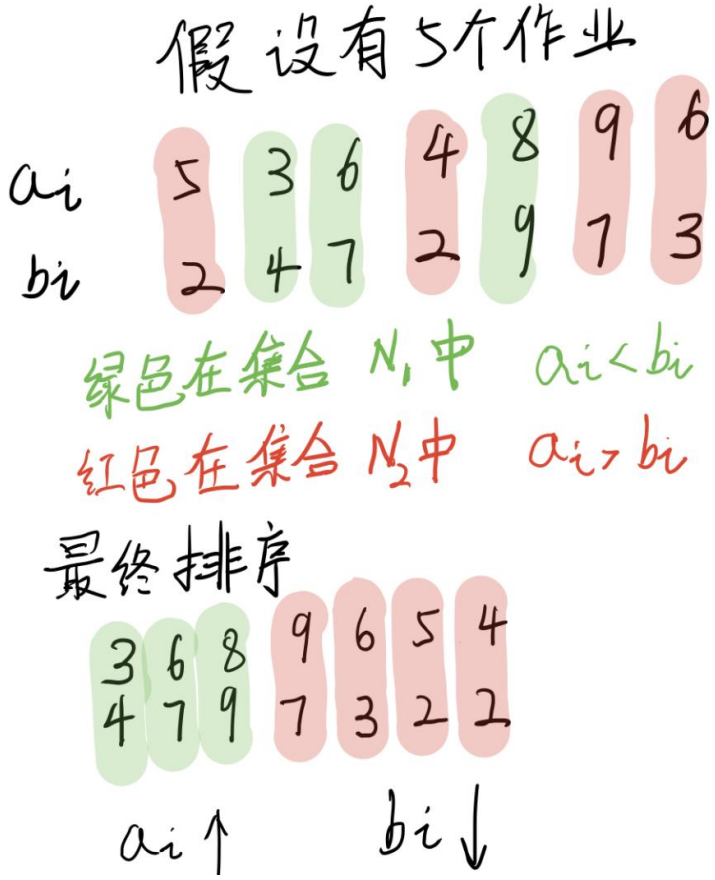
(2)将N1中作业依ai的非减序排序；将N2中作业依bj的非增序排序；

(3)N1中作业接N2中作业构成满足Johnson法则的最优调度。

$\min\{bi,aj\} \geqslant \min\{bj,ai\}$

算法复杂度分析:

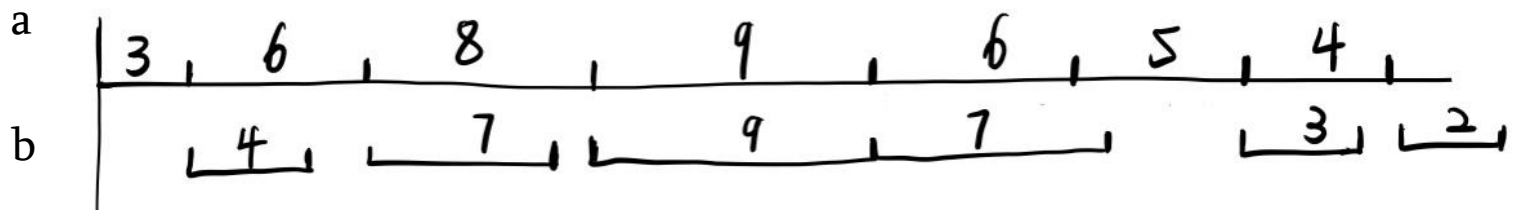
算法的主要计算时间花在对作业集的排序。因此，在最坏情况下算法所需的计算时间为O(nlogn)。所需的空间为O(n)。



推测一下这个Johson法则为什么能够得到最小的作业时间？

Johson法则分出的第一组都是b加工时间大于a的，且按a时间递增；分出的第二组都是a加工时间大于b的，且按b时间递减。

由于a加工是无间断的，决定时间长短的只是b。按照Johson法则会发现，中间部分都是一些b耗时大的作业，两头都是一些耗时小的作业，这样安排会很好填充b中的时间空隙。



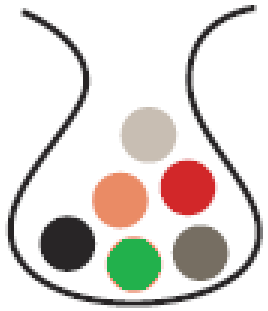
3.10 0-1背包问题

给定n种物品和一背包。物品i的重量是wi，其价值为vi，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个特殊的整数规划问题。

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

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



3.10 0-1背包问题

设所给0-1背包问题 $\max\sum_{i=2}^{n} v_i x_i$

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

利用反证法，设(X2,…Xn)不是上述子问题的一个最优解，而(y2,…yn)是，则后者目标函数值要大于前者

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

又因为前者满足上述约束条件，说明(x1,y2,…,yn)是原问题的一个解

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

说明(x1,x2,…xn)非原问题最优解。这与(x1,x2,…,xn)是原问题最优解相悖。所以(x2,…,xn)是子问题最优解，最优子结构性质得证。

3.10 0-1背包问题

设所给0-1背包问题的子问题 $\max\sum_{k=i}^{n} v_k x_k$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

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

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

若背包剩余容量大于第i个物品重量，则分别考虑装与不装两种情况的重量，取最大值

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

若背包剩余容量小于第i个物品重量，则不考虑这个物品

```
void Knapsack(int v[],int w[],int c,int n,int m[][10])
{
   int jMax = min(w[n]-1,c);//背包剩余容量上限 范围[0~w[n]-1]
   for(int j=0; j<=jMax;j++) { //第n个物品无法装入背包
      m[n][j]=0;
   }
   for(int j=w[n]; j<=c; j++)  {//限制范围[w[n]~c]
      m[n][j] = v[n];  //第n个物品装入背包
   }
   for(int i=n-1; i>1; i--) {
      jMax = min(w[i]-1,c);
      for(int j=0; j<=jMax; j++) { //背包不同剩余容量j<=jMax<c
         m[i][j] = m[i+1][j];//第i个物品没法装入，没产生任何效益
      }
      for(int j=w[i]; j<=c; j++) {  //背包不同剩余容量j-wi >c
         m[i][j] = max(m[i+1][j],m[i+1][j-w[i]]+v[i]);//效益值增长vi
      }
   }
   m[1][c] = m[2][c];
   if(c>=w[1]) {
      m[1][c] = max(m[1][c],m[2][c-w[1]]+v[1]);
   }
}
```

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

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

n= 4 c=8

w[]={1,4,2,3} v[]={2,1,4,3}

i \ j	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8
i=4	0	0	3	3	3	3	3	3
i=3	0	4	4	4	7	7	7	7
i=2	0	4	4	4	7	7	7	7
i=1								9

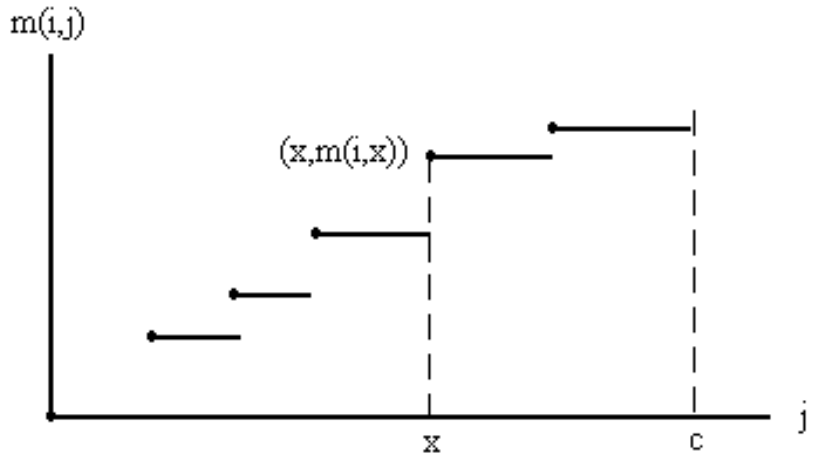
绿色框为方法 Traceback 的回溯过程 x[]={1,0,1,1}

算法复杂度分析：

从Knapsack容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当c>2n时，算法需要Ω(n2n)计算时间。

算法改进

由m(i,j)的递归式容易证明，在一般情况下，对每一个确定的i(1≤i≤n)，函数m(i,j)是关于变量j的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数m(i,j)由其全部跳跃点唯一确定。如图所示。



对每一个确定的i(1≤i≤n)，用一个表p[i]存储函数m(i，j)的全部跳跃点。表p[i]可根据计算m(i，j)的递归式来递归地由表p[i+1]计算，初始时p[n+1]={(0，0)}。

算法改进

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

函数m(i,j)是由函数m(i+1,j)与函数m(i+1,j-wi)+vi作max运算得到的。因此，函数m(i,j)的全部跳跃点包含于函数m(i+1，j)的跳跃点集p[i+1]与函数m(i+1，j-wi)+vi的跳跃点集q[i+1]的并集中。易知，(s,t)∈q[i+1]当且仅当wi≤s≤c且(s-wi,t-vi)∈p[i+1]。

因此，容易由p[i+1]确定跳跃点集q[i+1]如下

q[i+1]=p[i+1]⊕(wi,vi)={(j+wi,m(i,j)+vi)|(j,m(i,j))∈p[i+1]}

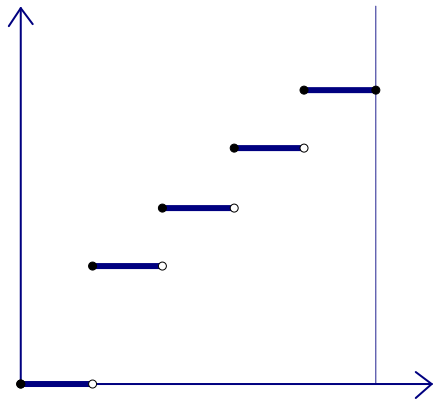
另一方面，设(a，b)和(c，d)是p[i+1]∪q[i+1]中的2个跳跃点，则当c≥a且d<b时，(c，d)受控于(a，b)，从而(c，d)不是p[i]中的跳跃点。除受控跳跃点外，p[i+1]∪q[i+1]中的其它跳跃点均为p[i]中的跳跃点。

由此可见，在递归地由表p[i+1]计算表p[i]时，可先由p[i+1]计算出q[i+1]，然后合并表p[i+1]和表q[i+1]，并清除其中的受控跳跃点得到表p[i]。

一个例子

n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。

初始时p[6]={(0,0)}，(w5,v5)=(4,6)。因此，
q[6]=p[6]⊕(w5,v5)={(4,6)}。
p[5]={(0,0),(4,6)}。
q[5]=p[5]⊕(w4,v4)={(5,4),(9,10)}。从跳跃点集p[5]与q[5]的并集p[5]∪q[5]={(0,0),(4,6),(5,4),(9,10)}中看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到p[4]={(0,0),(4,6),(9,10)}
q[4]=p[4]⊕(6，5)={(6，5)，(10，11)}
p[3]={(0，0)，(4，6)，(9，10)，(10，11)}
q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}
p[2]={(0，0)，(2，3)，(4，6)，(6，9)，(9，10)，(10，11)}
q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}
p[1]={(0，0)，(2，6)，(4，9)，(6，12)，(8，15)}
p[1]的最后的那个跳跃点(8,15)给出所求的最优值为m(1,c)=15。



算法复杂度分析

上述算法的主要计算量在于计算跳跃点集p[i](1≤i≤n)。由于q[i+1]=p[i+1]⊕(wi，vi)，故计算q[i+1]需要O(|p[i+1]|)计算时间。合并p[i+1]和q[i+1]并清除受控跳跃点也需要O(|p[i+1]|)计算时间。从跳跃点集p[i]的定义可以看出，p[i]中的跳跃点相应于xi,…,xn的0/1赋值。因此，p[i]中跳跃点个数不超过2n-i+1。由此可见，算法计算跳跃点集p[i]所花费的计算时间为

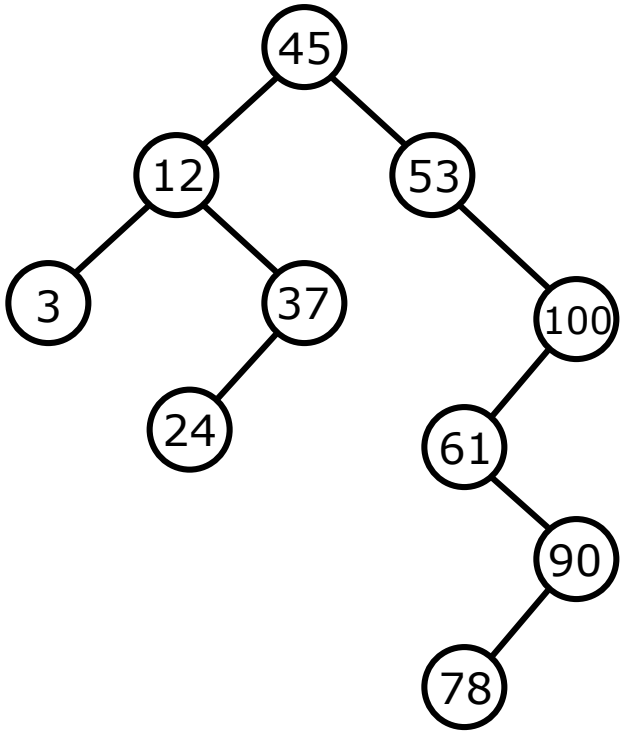
$$O\left(\sum_{i=2}^{n}|\,p[i+1]\,|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O\left(2^{n}\right)$$

从而，改进后算法的计算时间复杂性为O(2n)。当所给物品的重量wi(1≤i≤n)是整数时，|p[i]|≤c+1，(1≤i≤n)。在这种情况下，改进后算法的计算时间复杂性为O(min{nc,2n})。

3.11 最优二叉搜索树

二叉搜索树

(1) 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
(2) 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
(3) 它的左、右子树也分别为二叉搜索树



在随机的情况下，二叉查找树的平均查找长度和 $\log n$ 是等数量级的

二叉查找树的期望耗费

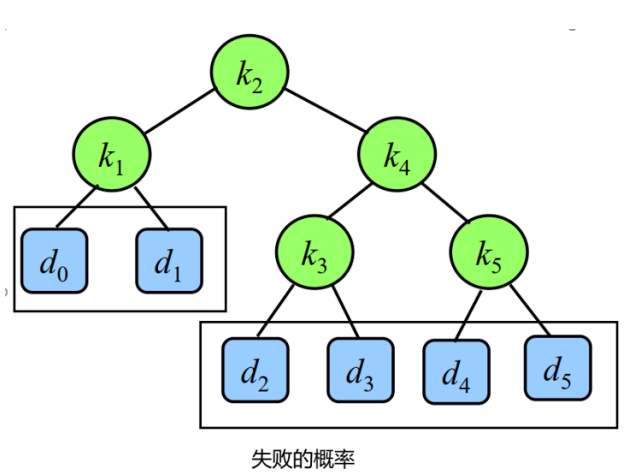
给定n个互异的关键字组成的序列K=<k_1,k_2,...,k_n>，且关键字有序（k_1<k_2<...<k_n），从这些关键字中构造一棵二叉查找树。对每个关键字k_i，一次搜索搜索到的概率为bi(查找成功)。可能有一些搜索的值不在K内（查找不成功），因此还有n+1个“虚拟键”$d_0, d_1,..., d_n$，他们代表不在K内的值。具体：d_0代表所有小于k_1的值，d_n代表所有大于k_n的值。而对于i = 1,2,...,n-1,虚拟键d_i代表所有位于k_i和k_{i+1}之间的值。对于每个虚拟键，一次搜索对应于d_i的概率为a_i。
要使得查找一个节点的期望代价最小，就需要建立一棵最优二叉查找树。

查找成功与不成功的概率，bi为成功搜索，ai未成功

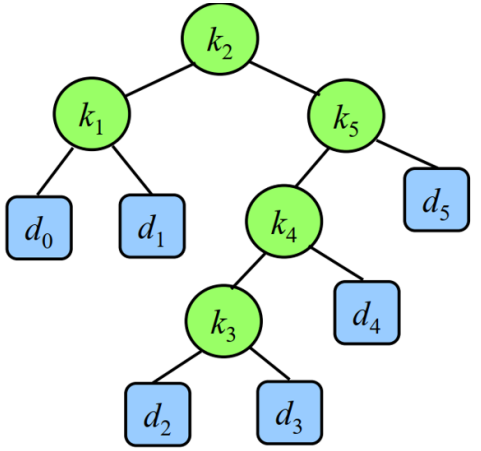
$$\sum_{i=1}^{n} a_i + \sum_{i=0}^{n} b_i = 1$$

找到元素ki的概率为bi

在(k_i,k_{i+1})区间的概率为ai



失败的概率

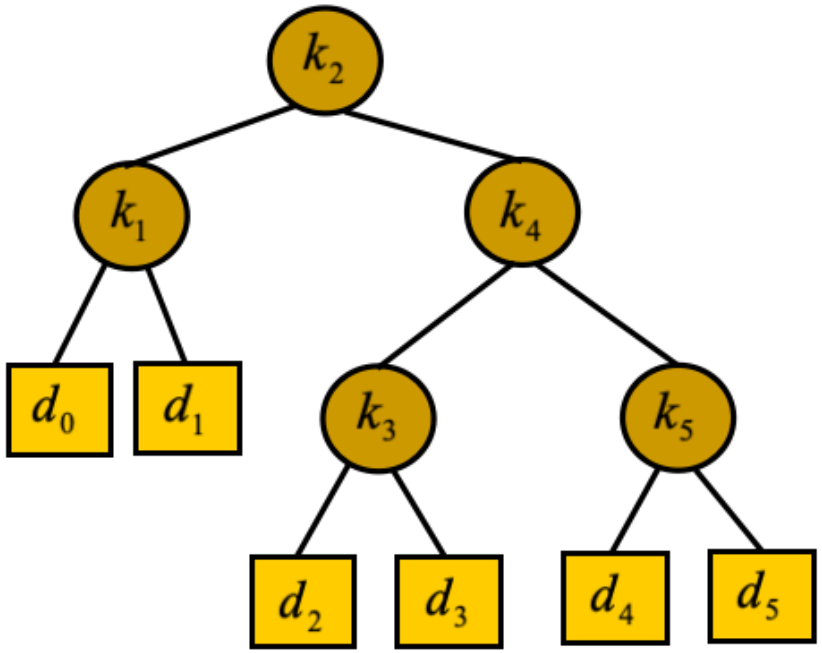


二叉查找树的期望耗费（平均路长）

$$p = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot b_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot a_i$$

最优二叉搜索树问题是对干有序集 S 及其概率分布 (a_0,b_1.a_1.b_n.a_n), 在所有表示有序集S的二叉搜索树中找出一棵有最小平均路长的二叉搜索树。

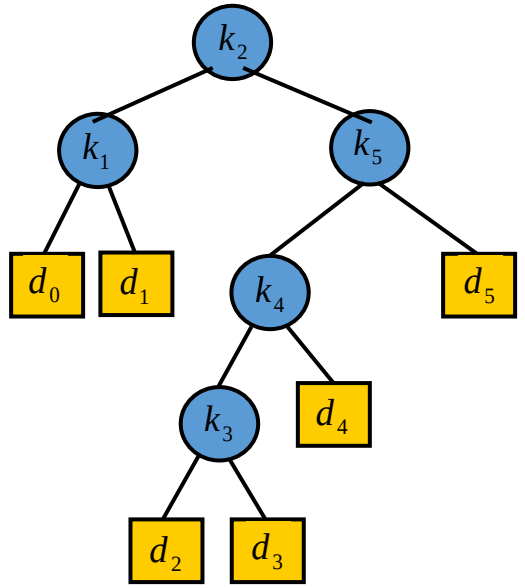
二叉查找树的期望耗费



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.10
d_1	2	0.10	0.20
d_2	3	0.05	0.15
d_3	3	0.05	0.15
d_4	3	0.05	0.15
d_5	3	0.10	0.30
Total			2.40

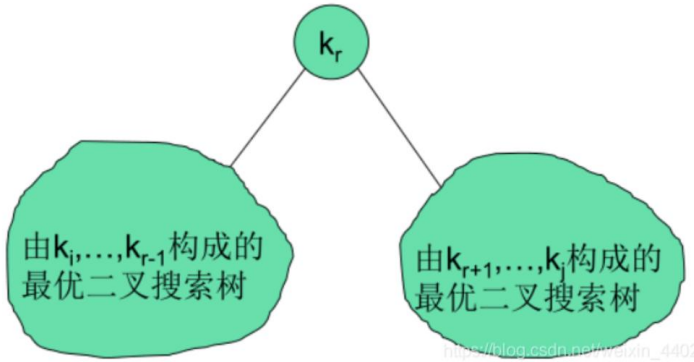
二叉查找树的期望耗费

$$p=\sum_{i=1}^{n}(\text{depth}_T(k_i)+1)\cdot b_i+\sum_{i=0}^{n}\text{depth}_T(d_i)\cdot a_i$$



假设有一棵$a_{i-1},b_i....,b_j,a_j$的树，接到另一个结点形成一颗新树，增加的代价（每个结点在新树中的深度都增加了1，所以其实就算所有结点频率之和）为：

$$w_{i,j}=a_{i-1}+b_i+\cdots+b_j+a_j$$



二叉查找树的期望耗费

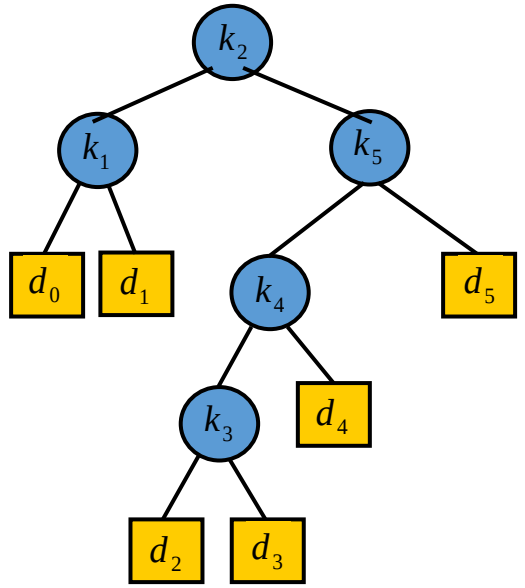
$$p=\sum_{i=1}^{n}(\text{depth}_T(k_i)+1)\cdot b_i+\sum_{i=0}^{n}\text{depth}_T(d_i)\cdot a_i$$

节点k_i至k_j的查找概率之和

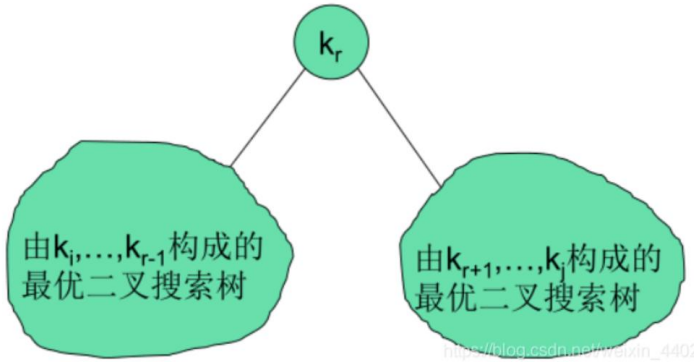
$$w_{i,j}=a_{i-1}+b_i+\cdots+b_j+a_j$$

$$w_{i,j}p_{i,j}=w_{i,j}+w_{i,m-1}p_l+w_{m+1,j}p_r$$

左子树　右子树



n个节点的二叉树的个数为：$\Omega(4^n/n^{3/2})$
穷举搜索法的时间复杂度为指数级



最优子结构性质证明

- 二叉搜索树T 的一棵含有顶点$x_i, \cdots, x_j$和叶顶点 $(x_{i-1}, x_i), \cdots, (x_j, x_{j+1})$ 的子树可以看作是有序集$\{x_i, \cdots, x_j\}$关于全集为 $\{x_{i-1}, x_{j+1}\}$的一棵二叉搜索树（T 自身可以看作是有序集)。

根据S 的存取分布概率，在子树的顶点处被搜索到的概率是:

$$w_{ij} = \sum_{i-1 \le k \le j} a_k + \sum_{i \le k \le j} b_k$$

最优子结构性质

- $\{x_i, \cdots, xj\}$的存储概率分布为$\{\overline{a_{i-1}}, \overline{b_i}, \ldots, \overline{b_j}, \overline{a_j}\}$，其中，$\overline{a_h}$，$\overline{b_k}$分别是下面的条件概率：

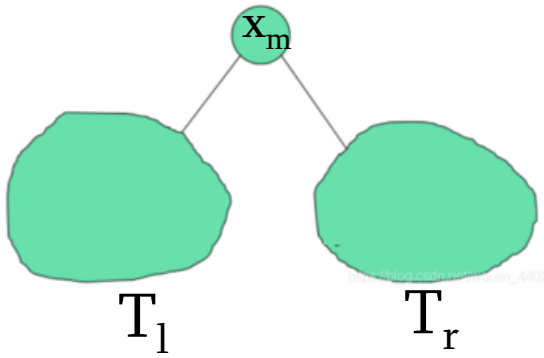
$$\overline{b_k} = b_k / w_{ij}, \quad i \le k \le j; \qquad \overline{a_h} = a_h / w_{ij}, \quad i-1 \le h \le j$$

- 设T_{ij}是有序集$\{x_i, \cdots, xj\}$关于存储概率分布为$\{a_{i-1}, b_i, \ldots, b_j, a_j\}$的一棵最优二叉搜索树，其平均路长为$p_{ij}$，$T_{ij}$的根顶点存储的元素$x_m$，其左子树$T_l$和右子树$T_r$的平均路长分别为$p_l$和$p_r$。由于$T_l$和$T_r$中顶点深度是它们在$T_{ij}$中的深度减1，所以得到

左子树的搜索概率　　右子树的搜索概率

$$w_{ij} p_{ij} = w_{i,m-1}(p_l + 1) + w_{m,m} + w_{m+1,j}(p_r + 1)$$

$$= w_{ij} + w_{i,m-1} p_l + w_{m+1,j} p_r \qquad i \le j$$



最优子结构性质

由于T_l是有序集$\{x_i,\cdots,x_{m-1}\}$的一棵二叉搜索树，故$p_l \geq p_{i,m-1}$.若$p_l > p_{i,m-1}$，则$T_{i,m-1}$替换T_l可得到平均路长比T_{ij}更小的二叉搜索树。这与T_{ij}是最优二叉搜索树矛盾。同理可证，T_r也是一棵最优二叉搜索树。因此，最优二叉搜索树问题具有最优子结构性质。

构造最优二叉搜索树时，可以选择先构造其左右子树，使其左右子树最优，然后构造整棵树。

递归计算最优值

- 最优二叉搜索树T_{ij}的平均路长为p_{ij}，则所求的最优值为$p_{1,n}$。由二叉树的代价公式

$$w_{ij}\,p_{ij} = w_{i,k-1}(p_{i,k-1}+1)+w_{k,k}+w_{k+1,j}(p_{k+1,j}+1)$$

$$= w_{ij}+w_{i,k-1}\,p_{i,k-1}+w_{k+1,j}\,p_{k+1,j} \qquad i \le j$$

- 根据最优二叉搜索树问题的最优子结构性质可建立计算p_{ij}的递归式如下

$$w_{ij}p_{ij} = w_{ij} + \min_{i\le k\le j}\{w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j}\}, \quad i \le j$$

初始时 $p_{i,i-1} = 0, \quad 1 \le i \le n$

递归计算最优值

- 记 $w_{i,j}p_{i,j}$ 为 $m(i,j)$

$m(1,n)=w_{1,n}p_{1,n}=p_{1,n}$ 为所求最优值。

记 $w_{ij}p_{ij}$ 为 $m(i,j)$ ，得关于 $m(i,j)$

$$m(i,j)=w_{ij}+\min_{i\le k\le j}\{m(i,k-1)+m(k+1,j)\},\quad i\le j$$

$$m(i,i-1)=0,\quad i=1,2,\cdots,n$$

例

给出标识符集{1, 2, 3}＝{do, if, stop}存取概率

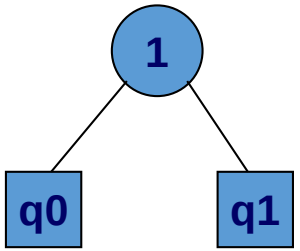
若　　b_1=0.5,　　b_2=0.1,　　b_3=0.05,

a_0=0.15,　　a_1=0.1,　　a_2=0.05,　　a_3=0.05

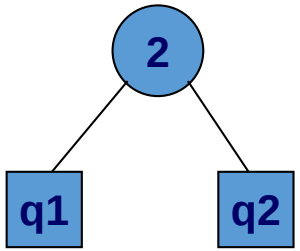
构造一棵最优二叉搜索树

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

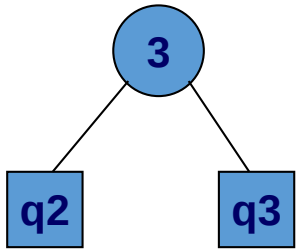
$$m(i,j)=w_{ij}+\min_{i\le k\le j}\{m(i,k-1)+m(k+1,j)\},\quad i\le j$$



T[1][1]
w[1][1]=0.75
m[1][1]=0.75

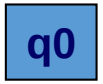


T[2][2]
w[2][2]=0.25
m[2][2]=0.25

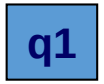


T[3][3]
w[3][3]=0.15
m[3][3]=0.15

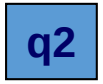
$$m(i,i-1)=0,\quad i=1,2,\cdots,n$$



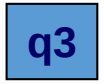
T[1][0]
w[1][0]=0.15
m[1][0]=0



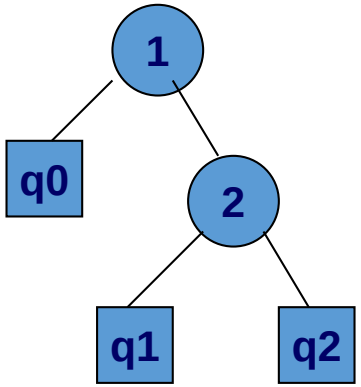
T[2][1]
w[2][1]=0.1
m[2][1]=0



T[3][2]
w[3][2]=0.05
m[3][2]=0

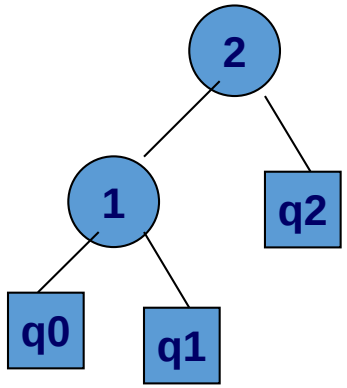


T[4][3]
w[4][3]=0.05
m[4][3]=0



w[1][2]=0.9
m[1][2]=0.9+
m[1][0]+m[2][2]
=1.15

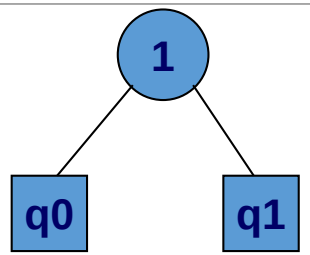
T[1][2]



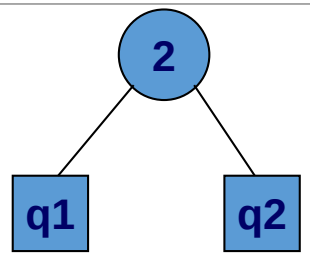
w[1][2]=0.9
m[1][2]=0.9+
m[1][1]+m[3][2]
=1.65

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

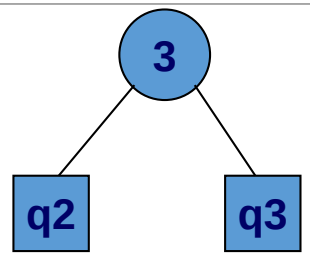
$$m(i,j) = w_{ij} + \min_{i \le k \le j} \{m(i,k-1) + m(k+1,j)\}, \quad i \le j$$



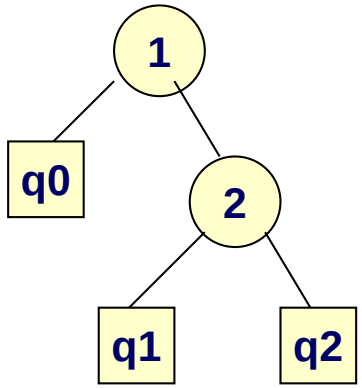
T[1][1]
w[1][1]=0.75
m[1][1]=0.75



T[2][2]
w[2][2]=0.25
m[2][2]=0.25

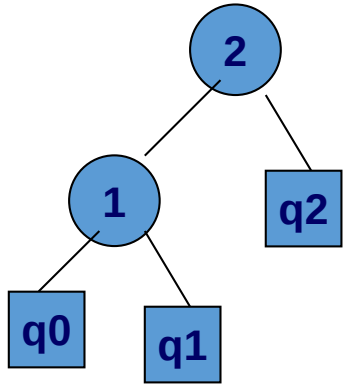


T[3][3]
w[3][3]=0.15
m[3][3]=0.15

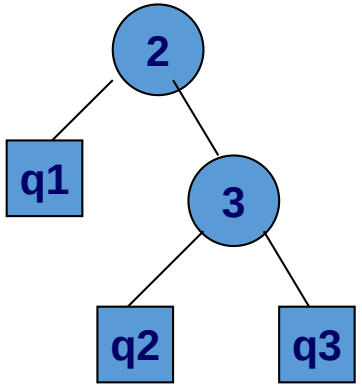


w[1][2]=0.9
m[1][2]=0.9+
m[1][0]+m[2][2]
=1.15

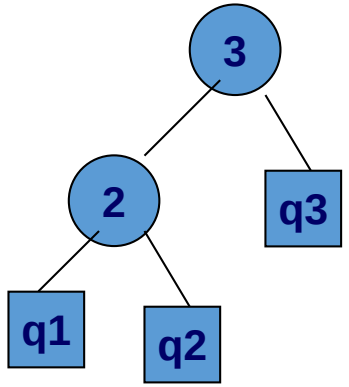
T[1][2]



w[1][2]=0.9
m[1][2]=0.9+
m[1][1]+m[3][2]
=1.65



m[2][3]=0.5

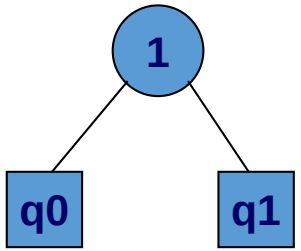


T[2][3]
w[2][3]=0.35

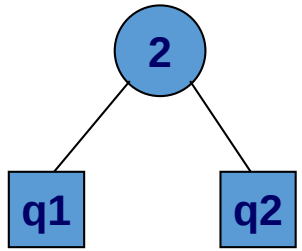
m[2][3]=0.6

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

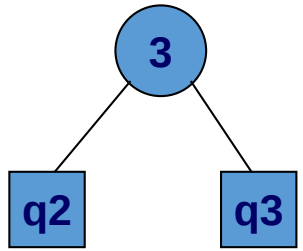
$$m(i,j)=w_{ij}+\min_{i\le k\le j}\{m(i,k-1)+m(k+1,j)\},\quad i\le j$$



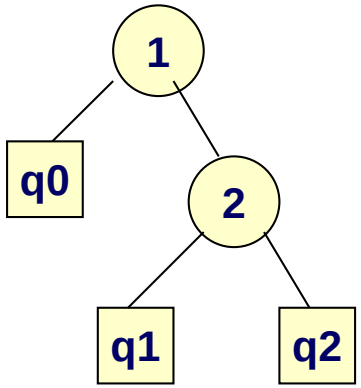
T[1][1]
w[1][1]=0.75
m[1][1]=0.75



T[2][2]
w[2][2]=0.25
m[2][2]=0.25

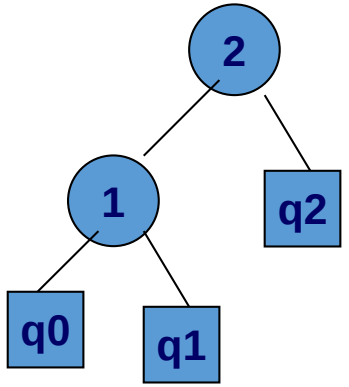


T[3][3]
w[3][3]=0.15
m[3][3]=0.15

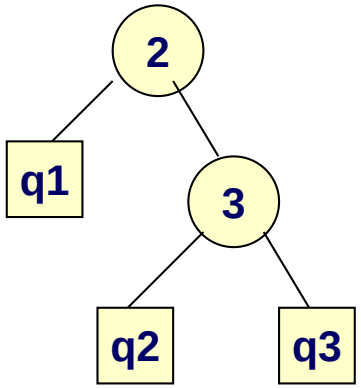


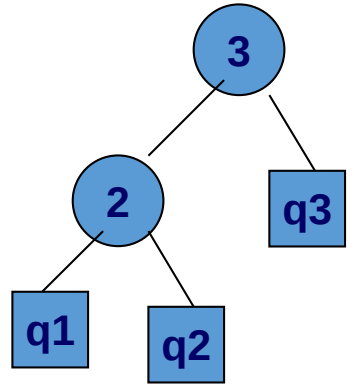
w[1][2]=0.9
m[1][2]=0.9+
m[1][0]+m[2][2]
=1.15

T[1][2]



w[1][2]=0.9
m[1][2]=0.9+
m[1][1]+m[3][2]
=1.65





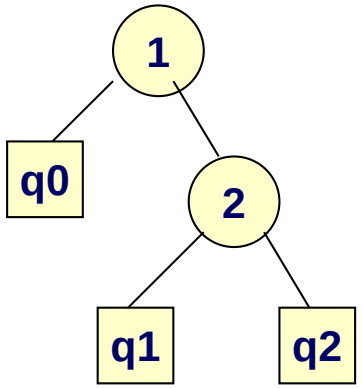
T[2][3]
w[2][3]=0. 35

m[2][3]=0.5

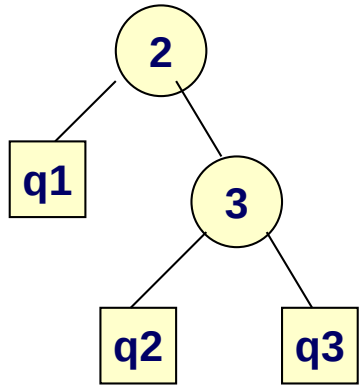
m[2][3]=0.6

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

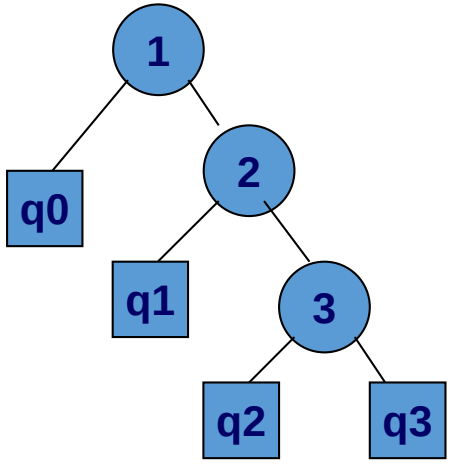
$$m(i,j) = w_{ij} + \min_{i \le k \le j} \{m(i,k-1) + m(k+1,j)\}, \quad i \le j$$



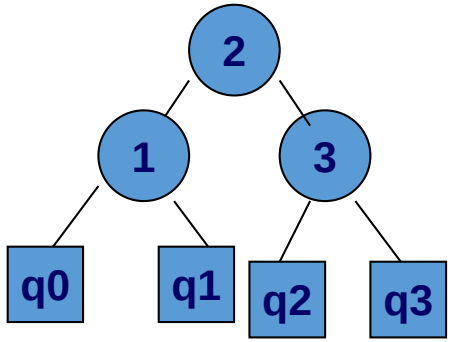
T[1][2]
m[1][2]=1.15



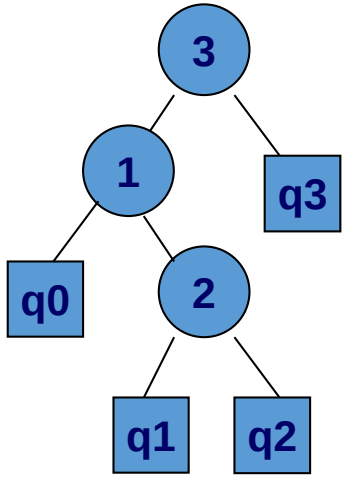
T[2][3]
m[2][3]=0.5



m[1][3]=1.5



m[1][3]=1.9

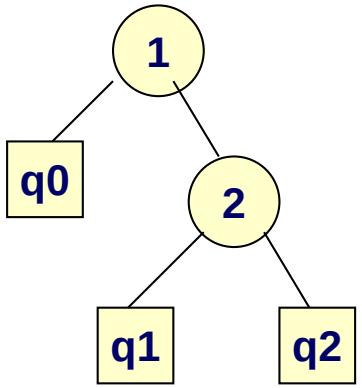


m[1][3]=2.15

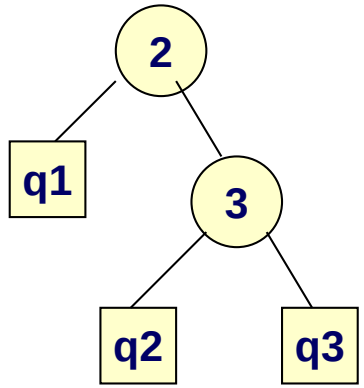
T[1][3]
W[1][3]=1

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

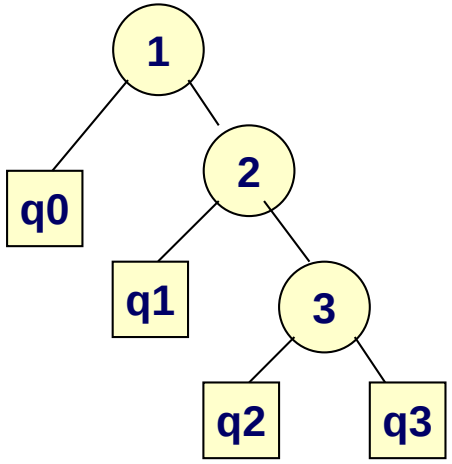
$$m(i,j) = w_{ij} + \min_{i \le k \le j} \{m(i,k-1) + m(k+1,j)\}, \quad i \le j$$



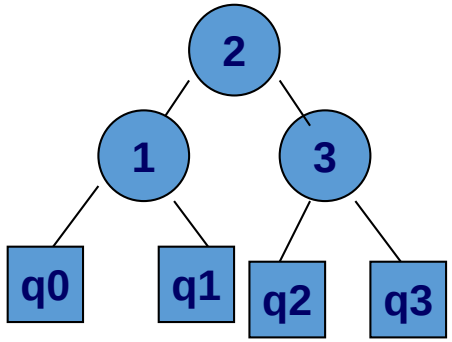
T[1][2]
m[1][2]=1.15



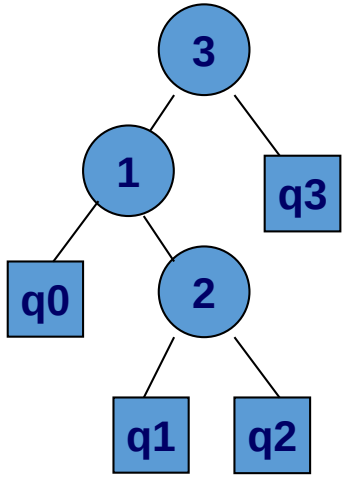
T[2][3]
m[2][3]=0.5



m[1][3]=1.5



m[1][3]=1.9



m[1][3]=2.15

T[1][3]
W[1][3]=1

a0=0.15, b1=0.5, a1=0.1, b2=0.1, a2=0.05, b3=0.05, a3=0.05

$$m(i,j) = w_{ij} + \min_{i \le k \le j} \{m(i,k-1) + m(k+1,j)\}, \quad i \le j$$

同济大学 TONGJI UNIVERSITY

	0	1	2	3
1	0.15	0.75	0.9	1
2		0.1	0.25	0.35
3			0.05	0.15
4				0.05

W(i, j)

	0	1	2	3
1	0	0.75	1.15	1.5
2		0	0.25	0. 5
3			0	0.15
4				0

m(i, j)

	0	1	2	3
0	1	1	1	
	0	2	2	
		0	3	
			0	

s(i, j)