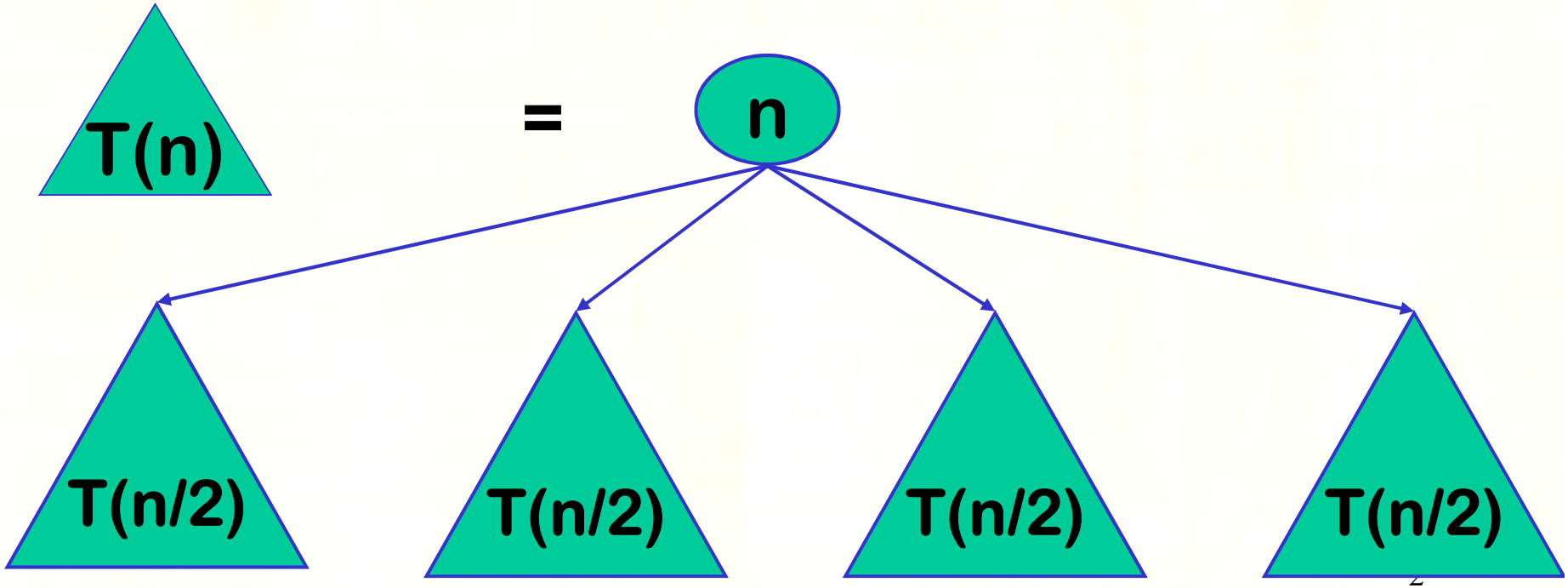
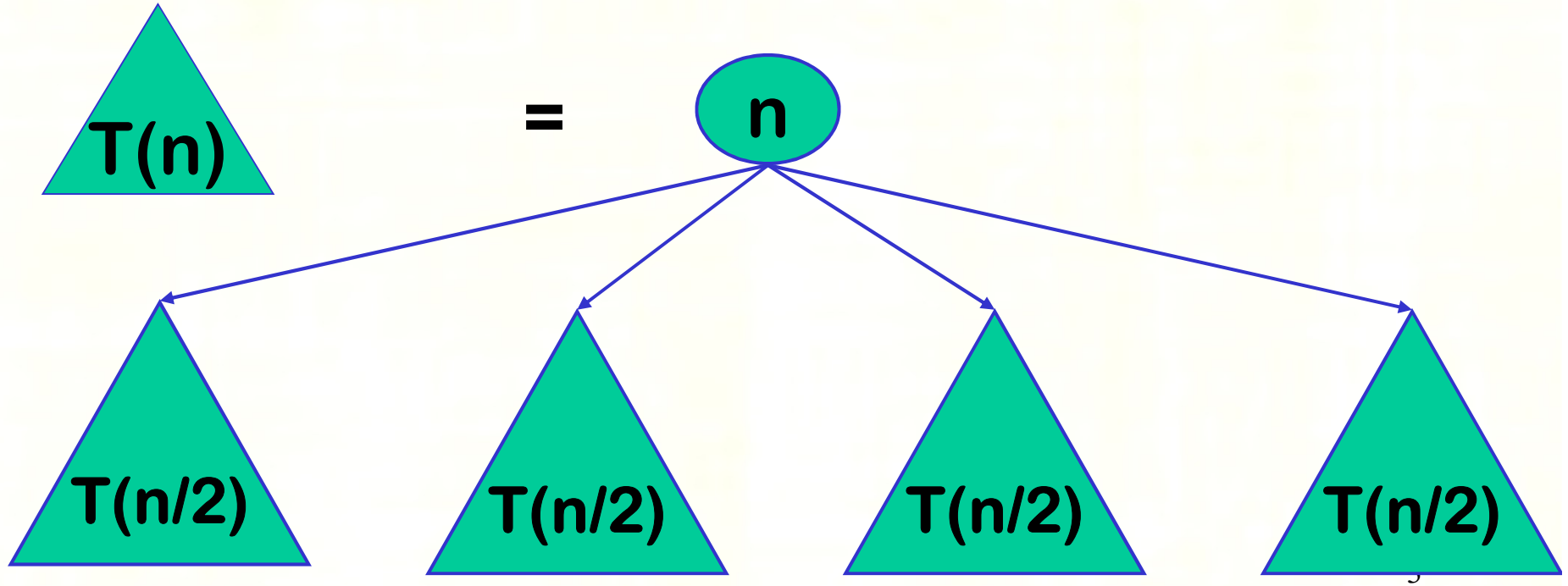
第3章　动态规划

- 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



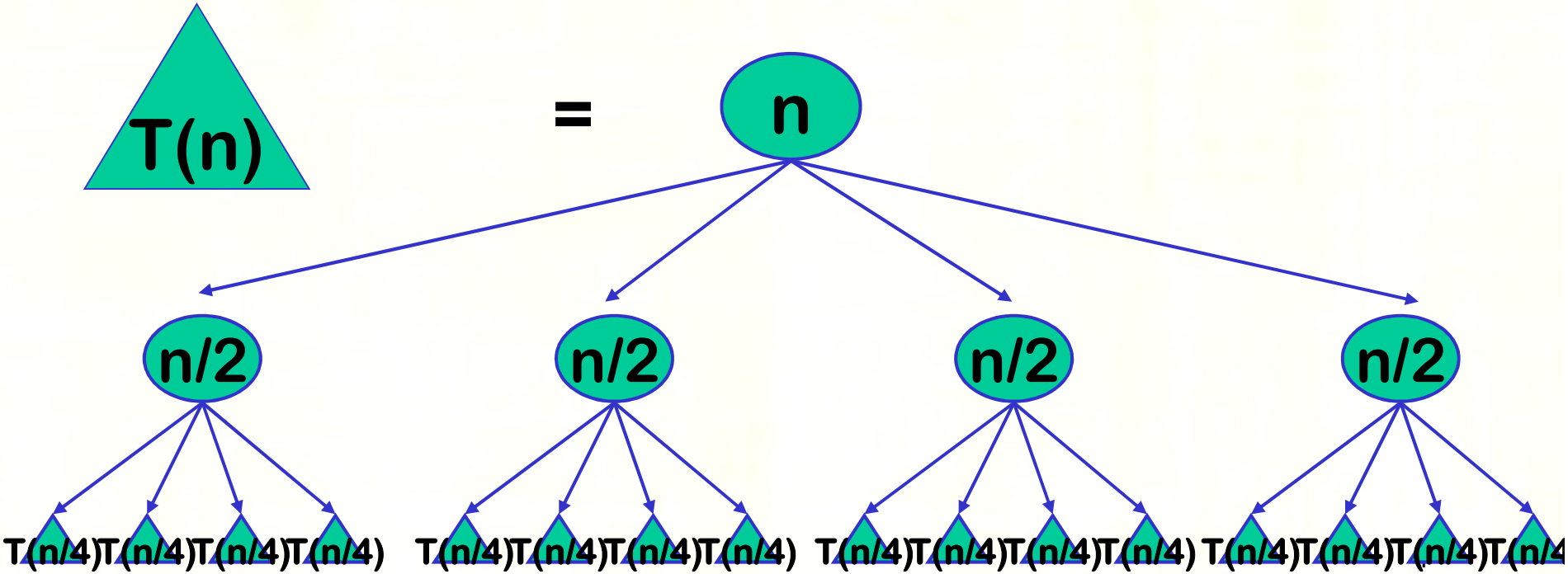
算法总体思想

- 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。

Those who cannot remember the past are doomed to repeat it.

-----George Santayana, The life of Reason, Book I: Introduction and Reason in Common Sense (1905)

动态规划基本步骤

- 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。

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 $A=(BC)$

- 设有四个矩阵 A, B, C, D，它们的维数分别是：

  $A=50\times10$ $B=10\times40$ $C=40\times30$ $D=30\times5$

- 总共有五中完全加括号的方式

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

  16000, 10500, 36000, 87500, 34500

矩阵连乘问题

- 给定n个矩阵 $\{A_1, A_2, \ldots, A_n\}$，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加括号的方式来确定。
- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积

矩阵连乘问题

给定n个矩阵 $\{A_1,A_2,\ldots,A_n\}$ ，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。

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)。

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：

$(A_1\ldots A_k)(A_k+1\ldots A_n)$可以得到关于P(n)的递推式如下：

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

矩阵连乘问题

- 穷举法
- 动态规划

将矩阵连乘积 $A_i A_{i+1} \ldots A_j$ 简记为A[i:j]，这里i≤j

考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵Ak和Ak+1之间将矩阵链断开，i≤k<j，则其相应完全加括号方式为 $(A_i A_{i+1} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j)$

计算量：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

- 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。
- 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。

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]
- 当i=j时，A[i:j]=Ai，因此，m[i,i]=0，i=1,2,…,n
- 当i<j时，

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

这里 A_i 的维数为 $p_{i-1}\times p_i$

- 可以递归地定义m[i,j]为：

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

k 的位置只有 $j-i$ 种可能

计算最优值

- 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

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

- 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，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

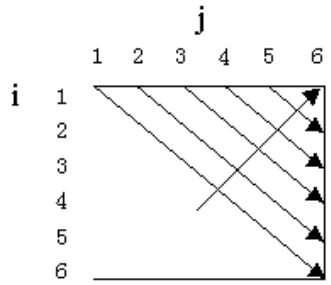
用动态规划法求最优解

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
  {
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          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;}
          }
        }
  }
```

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

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

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$O(n^2)$。



(a) 计算次序

j

i	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]

j

i	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]

3.2 动态规划算法的基本要素

一、最优子结构

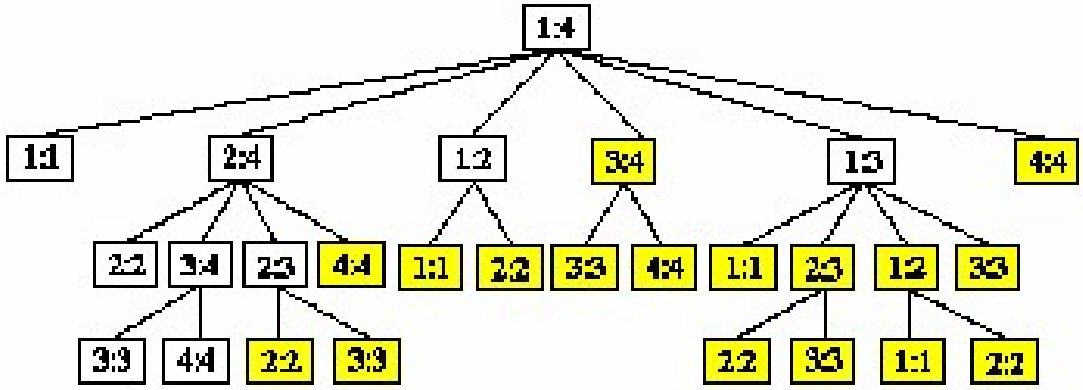
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**最优子结构性质**。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

注意：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、备忘录方法

- 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。

```
m←0
private static int lookupChain(int i, int j)
  {
    if (m[i][j] > 0) return m[i][j];
    if (i == j) return 0;
    int u = 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;
    return u;
  }
```

3.3 最长公共子序列

- 若给定序列X={x_1,x_2,…,x_m}，则另一序列Z={z_1,z_2,…,z_k}，是X的子序列是指存在一个严格递增下标序列{i_1,i_2,…,i_k}使得对于所有j=1,2,…,k有：$z_j=x_{i_j}$。例如，序列Z={B，C，D，B}是序列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}。
- 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。
- 给定2个序列X={x_1, x_2, …, x_m}和Y={y_1, y_2, …, y_n}，找出X和Y的最长公共子序列。

3.3 最长公共子序列

最长公共子序列的结构

设序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z=\{z_1,z_2,\ldots,z_k\}$，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}的最长公共子序列。

(2)若$x_m\neq y_n$且$z_k\neq x_m$，则Z是x_{m-1}和Y的最长公共子序列。

(3)若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和y_{n-1}的最长公共子序列。

由此可见，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。

3.3 最长公共子序列

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]记录序列和的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\ldots,x_i\}$；$Yj=\{y_1,y_2,\ldots,y_j\}$。当i=0或j=0时，空序列是$X_i$和$Y_j$的最长公共子序列。故此时C[i][j]=0。其他情况下，由最优子结构性质可建立递归关系如下：

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

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。

```
Algorithm lcsLength(x,y,b)
1: m←x.length-1;
2: n←y.length-1;
3: c[i][0]=0; c[0][i]=0;
4: for (int i = 1; i <= m; i++)
5:    for (int j = 1; j <= n; j++)
6:       if (x[i]==y[j])
7:          c[i][j]=c[i-1][j-1]+1;
8:          b[i][j]=1;
9:       else if (c[i-1][j]>=c[i][j-1])
10:         c[i][j]=c[i-1][j];
11:         b[i][j]=2;
12:      else
13:         c[i][j]=c[i][j-1];
14:         b[i][j]=3;
```

构造最长公共子序列

```
Algorithm 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);
      System.out.print(x[i]);
      }
    else if (b[i][j]== 2) lcs(i-1,j,x,b);
      else lcs(i,j-1,x,b);
  }
```

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c本身在时间内确定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))。

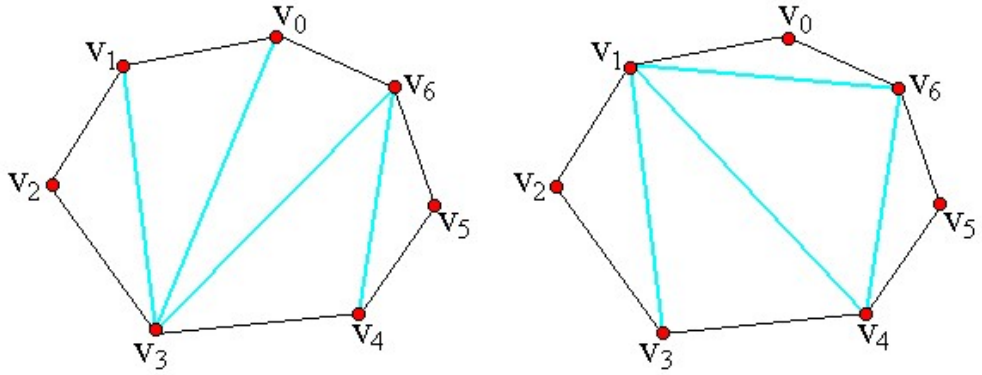
3.4 凸多边形最优三角剖分

•用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\ldots,v_{n-1}\}$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$\{v_i,v_{i+1},\ldots,v_j\}$和$\{v_j,v_{j+1},\ldots v_i\}$。

•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，使得即该三角剖分中诸三角形上权之和为最小。

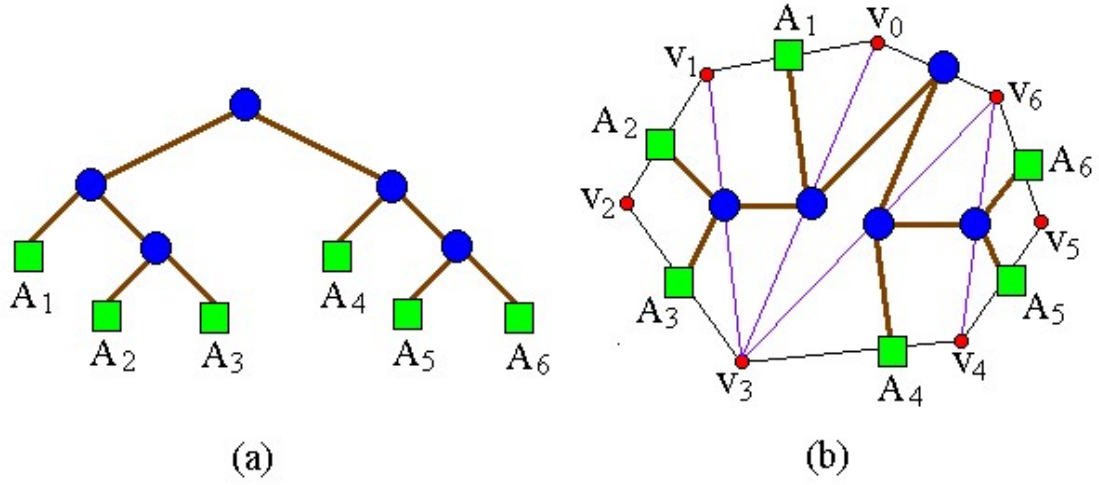


三角剖分的结构及其相关问题

•一个表达式的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$所相应的语法树如图 (a)所示。

•凸多边形$\{v_0,v_1,\ldots v_{n-1}\}$的三角剖分也可以用语法树表示。例如，图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示。

•矩阵连乘积中的每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$。三角剖分中的一条弦v_iv_j，i<j，对应于矩阵连乘积A[i+1:j]。



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$P=\{v_0,v_1,\ldots,v_{n-1}\}$的最优三角剖分T包含三角形$v_0v_kv_n$，$1\le k\le n-1$，则T的权为3个部分权的和：三角形v0vkvn的权，子多边形$\{v_0,v_1,\ldots,v_k\}$和$\{v_k,v_{k+1},\ldots,v_n\}$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$\{v_0,v_1,\ldots,v_k\}$或$\{v_k,v_{k+1},\ldots,v_n\}$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{vi-1,vi,…,vj}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{vi-1,vi}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。由此，t[i][j]可递归地定义为：

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

3.4 凸多边形最优三角剖分

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

```
public static void matrixChain(int [] p, int [][] m, int [][] s)  //矩阵连乘
  {
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          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;}
          }
        }
  }
```

```
public static void minWeightTriangulation(int n, int [][] t, int [][] s)  //最优三角形剖分
  {
    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;}
          }
        }
  }
```

3.9 0-1背包问题

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

0-1背包问题是一个特殊的整数规划问题。

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

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

3.9 背包问题

- **0-1背包问题**
  - 一个小偷在洗劫一家商店时找到了n个物品:其中第i个物品价值v_i并且重w_i(v_i, w_i都是整数)
  - 小偷的背包只能承受W的重量
  - 物品要么被带走要么留下
  - 带那些物品可以在指定的重量下达到价值最大呢?
- 可带碎片的背包问题
  - 与上述基本相同

  --小偷可以带走一个物品的一部分

0-1背包问题

- 小偷有一个可承受W的背包
- 有n件物品: 第i个物品价值v_i且重w_i
- 目标:
  - 找到x_i使得对于所有的$x_i = \{0, 1\}, \mathrm{i} = 1, 2, .., n$

    $\sum w_i x_i \leq W$ 并且$\sum x_i v_i$ 最大

最优子结构

- 考虑最多重W的物品且价值最高
- 如果我们把j物品从背包中拿出来

$\Rightarrow$剩下的装载一定是取自n-1个物品使得不超过载重量$W - w_j$并且所装物品价值最高的装载

0-1背包问题的动态规划

- $V[i, w]$ –考虑前i件物品所能获得的最高价值，其中w是背包的承受力
  - 第1种情况:物品i的重量w_i<=w，小偷对物品i可拿或者不拿

$$V[i,w] = \max\{V[i\text{-}1, w], V[i\text{-}1,w\text{-}w_i] + v_i\}$$

  - 第2种情况:物品i的重量w_i>w,即小偷不拿物品i

$$V[i, w] = V[i - 1, w]$$

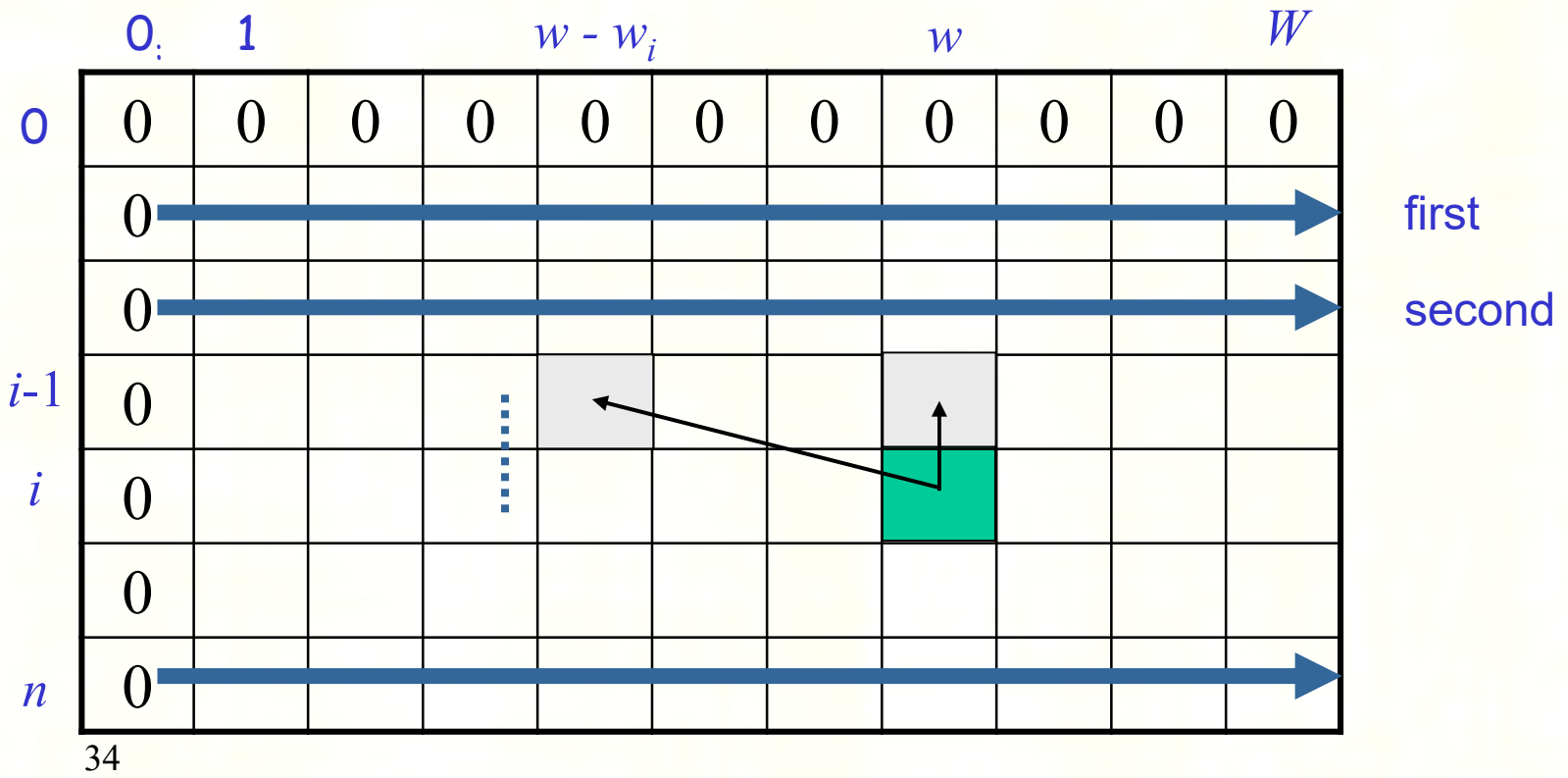
```
DPKnapsack(S, W)
1   for w ← 0 to w_1 - 1 do V[1, w] ← 0
2   for w ← w_1 to W do V[1, w] ← v1
3   for i ← 2 to n do
4       for w ← 0 to W do
5           if w_i > w then
6               V[i, w] ← V[i-1, w]
7               b[i, w] ← "↑"
8           else if V[i-1, w]> V[i-1, w-w_i] + v_i  then
9                       V[i, w] ← V[i-1, w]
10                      b[i, w] ← "↑"
11              else
12                      V[i, w] ←V[i-1, w-w_i] + vi
13                      b[i, w] ← "↖"
14  return V and b
```

运行时间: $\Theta(nW)$

0-1背包问题的动态规划

带走物品i　　不带走物品i

$$V[i, w] = \max \{v_i + V[i - 1, w\text{-}w_i], V[i - 1, w] \}$$



例子

$W = 5$

物品	重量	价值
1	2	12
2	1	10
3	3	20
4	2	15

$$V[i, w] = \max \{v_i + V[i - 1, w\text{-}w_i], V[i - 1, w] \}$$

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$V[1, 1] = V[0, 1] = 0$

$V[1, 2] = \max\{12+0, 0\} = 12$

$V[1, 3] = \max\{12+0, 0\} = 12$

$V[1, 4] = \max\{12+0, 0\} = 12$

$V[1, 5] = \max\{12+0, 0\} = 12$

$V[2, 1] = \max\{10+0, 0\} = 10$

$V[2, 2] = \max\{10+0, 12\} = 12$

$V[2, 3] = \max\{10+12, 12\} = 22$

$V[2, 4] = \max\{10+12, 12\} = 22$

$V[2, 5] = \max\{10+12, 12\} = 22$

$V[3, 1] = V(2,1) = 10$

$V[3, 2] = V(2,2) = 12$

$V[3, 3] = \max\{20+0, 22\}=22$

$V[3, 4] = \max\{20+10,22\}=30$

$V[4, 5] = \max\{20+12,22\}=32$

$V[4, 1] = P[3,1] = 10$

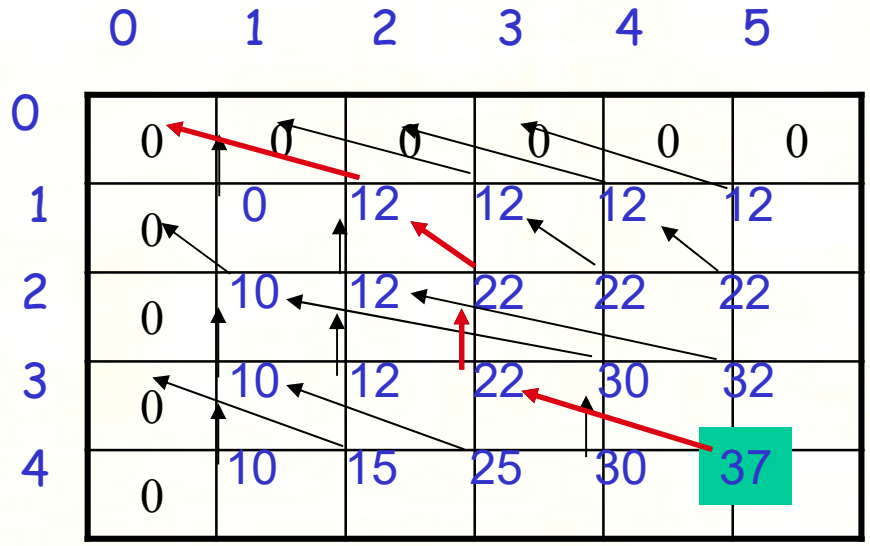
$V[4, 2] = \max\{15+0, 12\} = 15$

$V[4, 3] = \max\{15+10, 22\}=25$

$V[4, 4] = \max\{15+12, 30\}=30$

$V[4, 5] = \max\{15+22, 32\}=37$

构造最优解法



•物品4

•物品2

•物品1

- 从 $V[n, W]$开始
- 当往左上走⇒物品i被带走
- 当直往上走⇒物品i未被带走

子问题的重复

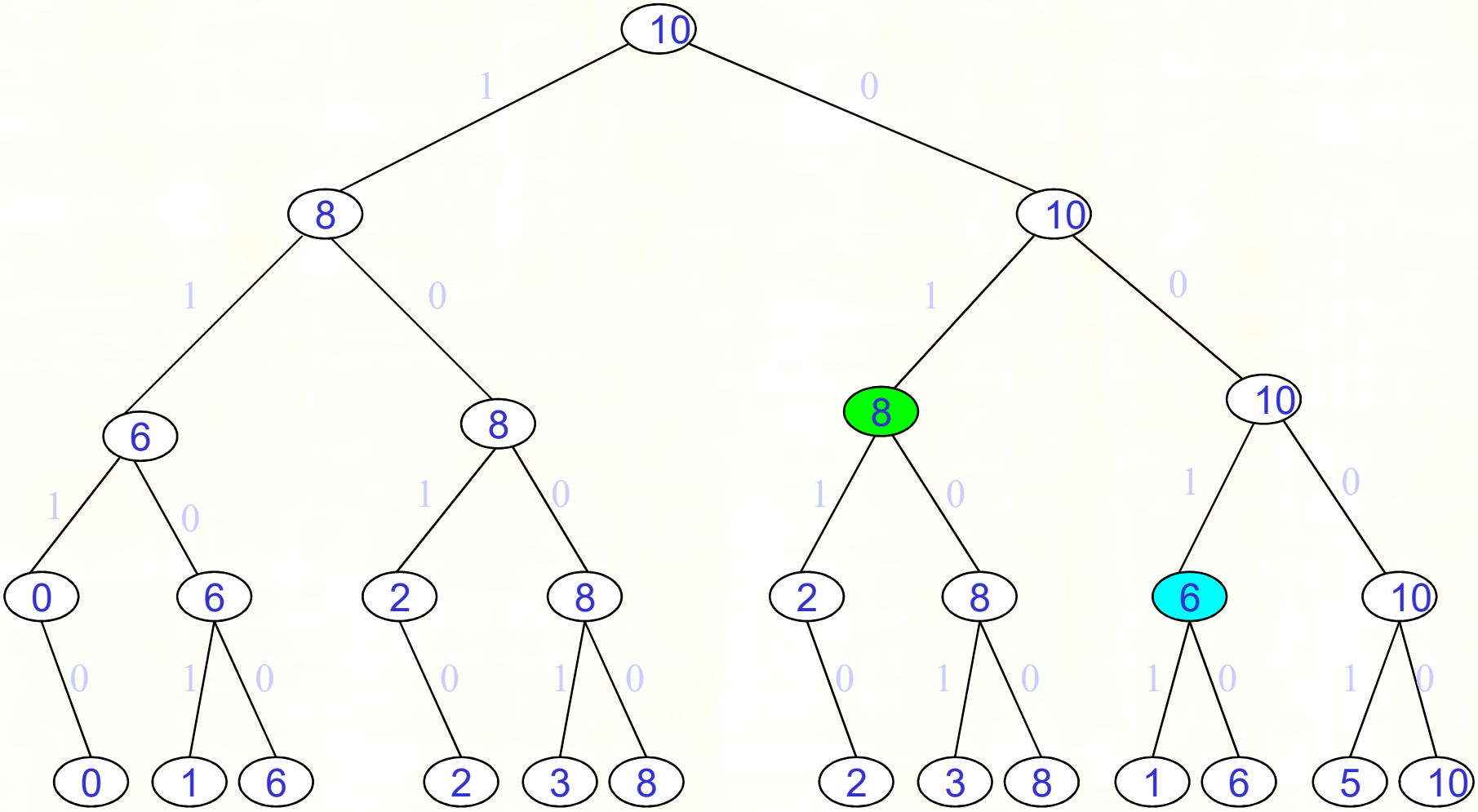
$$V[i, w] = \max \{v_i + V[i - 1, w\text{-}w_i], V[i - 1, w] \}$$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

例如: 所有用灰色表示的子问题都取决于$V[i\text{-}1, w]$

子问题的重复

例子: n=5, p=[6,3,5,4,6], w=[2,2,6,5,4], W=10



3.9 0-1背包问题

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$

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

算法复杂度分析：

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

的 品为i，

i+ 最优子

结构性质，可以建立计算m(i，j)的递归式如下。

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

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