算法基础

主讲人：　庄连生

Email: { *lszhuang@ustc.edu.cn* }

Spring 2018，USTC

University of Science and Technology of China

第九讲 动态规划

内容提要：

- 理解动态规划算法概念
- 掌握动态规划算法要素
- 掌握设计动态规划算法的步骤
- 通过范例学习动态规划算法设计策略

15.1 历史及研究问题

- 动态规划（dynamic programming）是运筹学的一个分支，20世纪50年代初美国数学家R.E.Bellman等人在研究多阶段决策过程(Multistep decision process)的优化问题时，提出了著名的最优性原理，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。

- 多阶段决策问题：求解的问题可以划分为一系列相互联系的阶段，在每个阶段都需要做出决策，且一个阶段决策的选择会影响下一个阶段的决策，从而影响整个过程的活动路线，求解的目标是选择各个阶段的决策是整个过程达到最优。

15.1 历史及研究问题

- 动态规划主要用于求解以时间划分阶段的动态过程的优化问题，但是一些与时间无关的静态规划（如线性规划、非线性规划），可以人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。
- 动态规划是考察问题的一种途径，或是求解某类问题的一种方法。
- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比其它方法求解更为方便。

15.1 历史及研究问题

- 基本概念：

① 状态：表示每个阶段开始时，问题或系统所处的客观状况。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。

  - 状态无后效性：如果某个阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有马尔科夫性。
  - 适于动态规划法求解的问题具有状态的无后效性

② 策略：各个阶段决策确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为子过程，其对应的某个策略称为子策略。

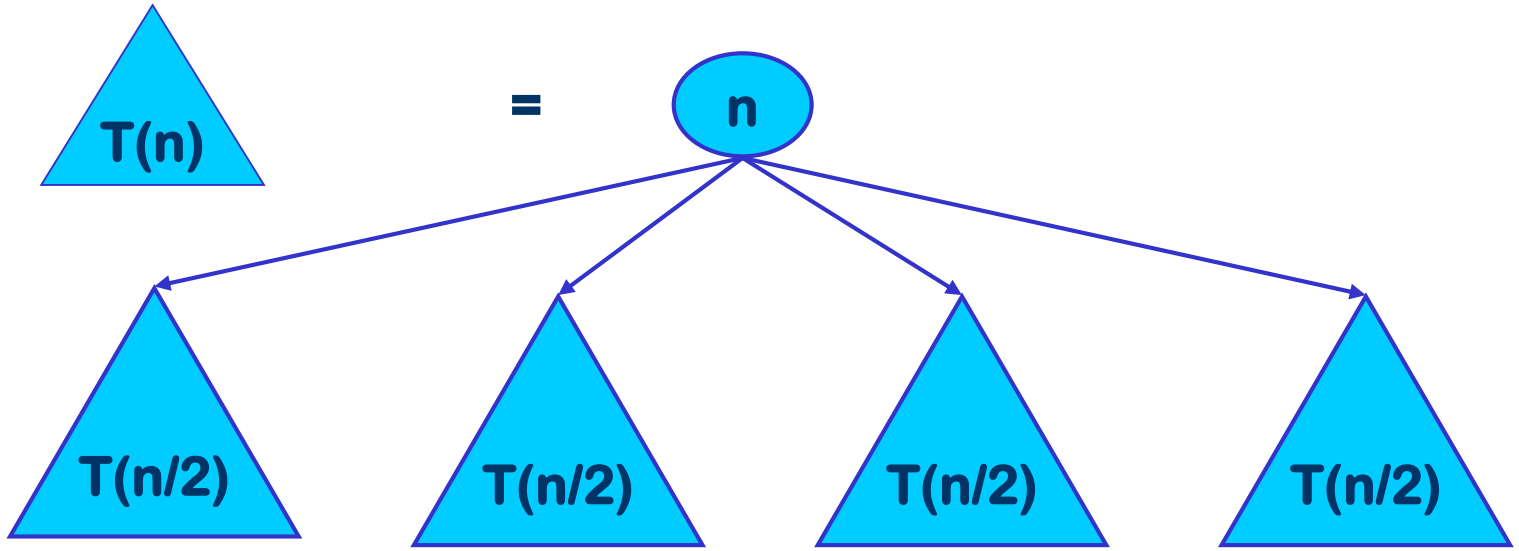
15.2 最优性原理

- Bellman的定义如下：
  - ✓ An Optimal policy has the property that whatever the initial state and initial decision are, then remaining decisions must constitute an optimal policy with regard to the state resulting from first decision.

- Bellman最优性原理：
  - ➢ 求解问题的一个最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。
  - ➢ 注：对具有最优性原理性质的问题而言，如果有一决策序列包含有非最优的决策子序列，则该决策序列一定不是最优的。

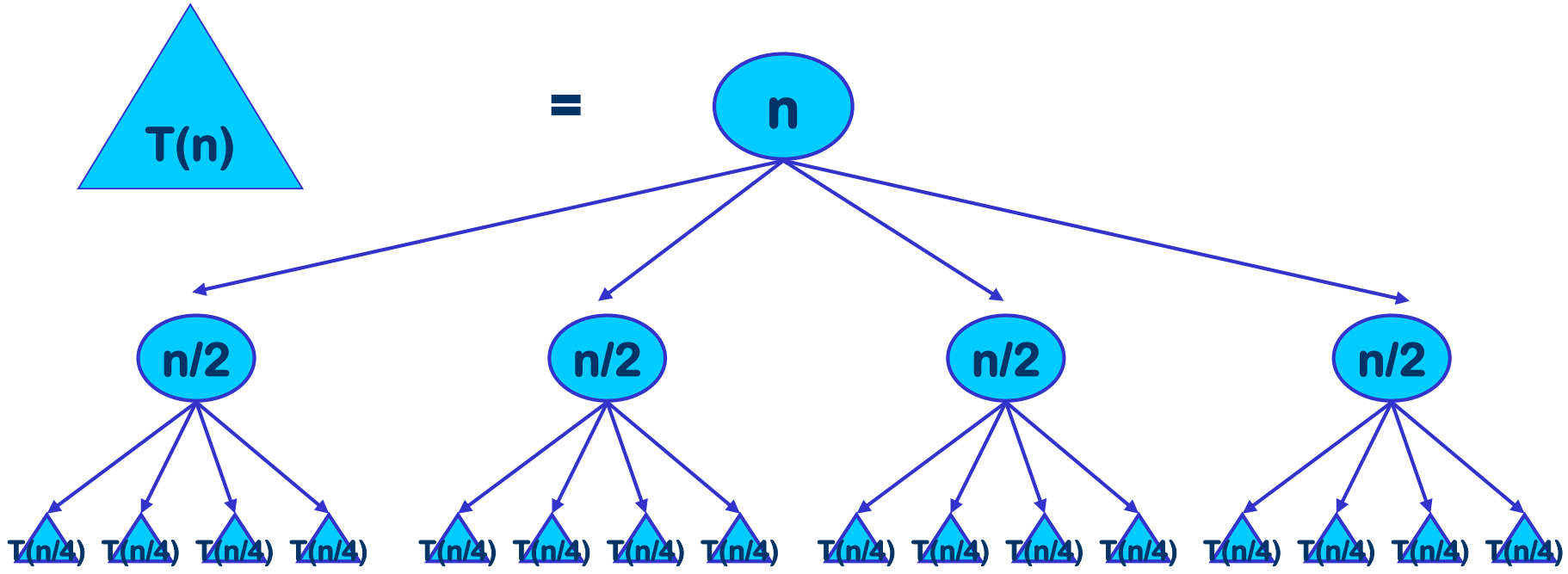
15.3 总体思想

- 动态规划的思想实质是分治思想和解决冗余
- 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



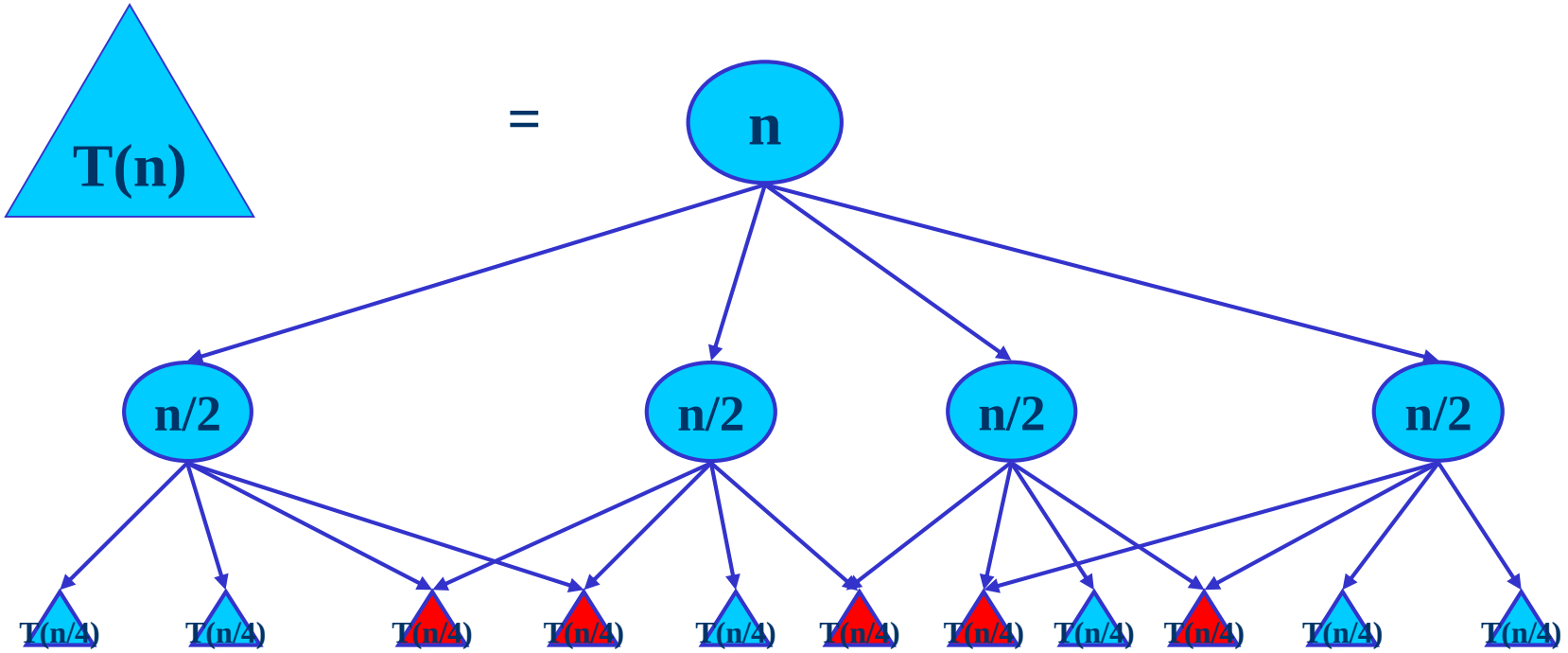
15.3 总体思想

- 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。



15.3 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。动态规划法用一个表来记录所有已解决的子问题的答案。具体的动态规划算法尽管多种多样，但它们具有相同的填表方式。



15.3 总体思想

- **基本步骤：**

① 找出最优解的性质，并刻划其结构特征。 ---》划分子问题

② 递归地定义最优解的值。 ----》给出最优解的递归式

③ 按自底向上的方式计算最优解的值。

④ 由计算出的结果构造一个最优解。

- **注：**

- 步骤①~③是动态规划算法的基本步骤。如果只需要求出最优值的情形，步骤④可以省略；
- 若需要求出问题的一个最优解，则必须执行步骤④，步骤③中记录的信息是构造最优解的基础；

15.4 矩阵链乘法问题

问题描述：

给定n个矩阵$A_1, A_2, \cdots, A_n$，A_i的维数为$p_{i-1} \times p_i$ $(1 \leqslant i \leqslant n)$，以一种最小化标量乘法次数的方式进行完全括号化。

注意：

① 设$A_{p\times q}, A_{q\times r}$两矩阵相乘，普通乘法次数为$p \times q \times r$；

② 加括号对乘法次数的影响：

$A_{10\times 100}\times B_{100\times 5}\times C_{5\times 50}$

((AB)C): 7500次　　(A(BC)): 75000次

15.4 矩阵链乘法问题

- **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。

- **复杂性分析**：

用p(n)表示n个矩阵链乘的穷举法计算成本，如果将n个矩阵从第k和第k+1处隔开，对两个子序列再分别加扩号，则可以得到下面递归式：

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

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

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

因此，穷举法不是一个有效算法。

15.4 矩阵链乘法问题

用动态规划法来求解：

- 步骤1：分析最优解的结构

1.矩阵链乘问题满足最优性原理

记A[i:j]为$A_iA_{i+1}\ldots A_j$链乘的一个最优括号方案，设A[i:j]的最优次序中含有二个子链A[i:k]和A[k+1:j]，则A[i:k]和A[k+1:j]也是最优的。（反证可得）

2.矩阵链乘的子问题空间：A[i:j]，$1 \leqslant i \leqslant j \leqslant n$

A[1:1], A[1:2], A[1:3], … , A[1:n]
A[2:2], A[2:3], … , A[2:n]
… … …
A[n-1:n-1], A[n-1:n]
A[n:n]

15.4 矩阵链乘法问题

- 步骤2：递归求解最优解的值

记m[i][j]为计算A[i:j]的最少乘法数，则原问题的最优值为m[1][n]，那么有

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

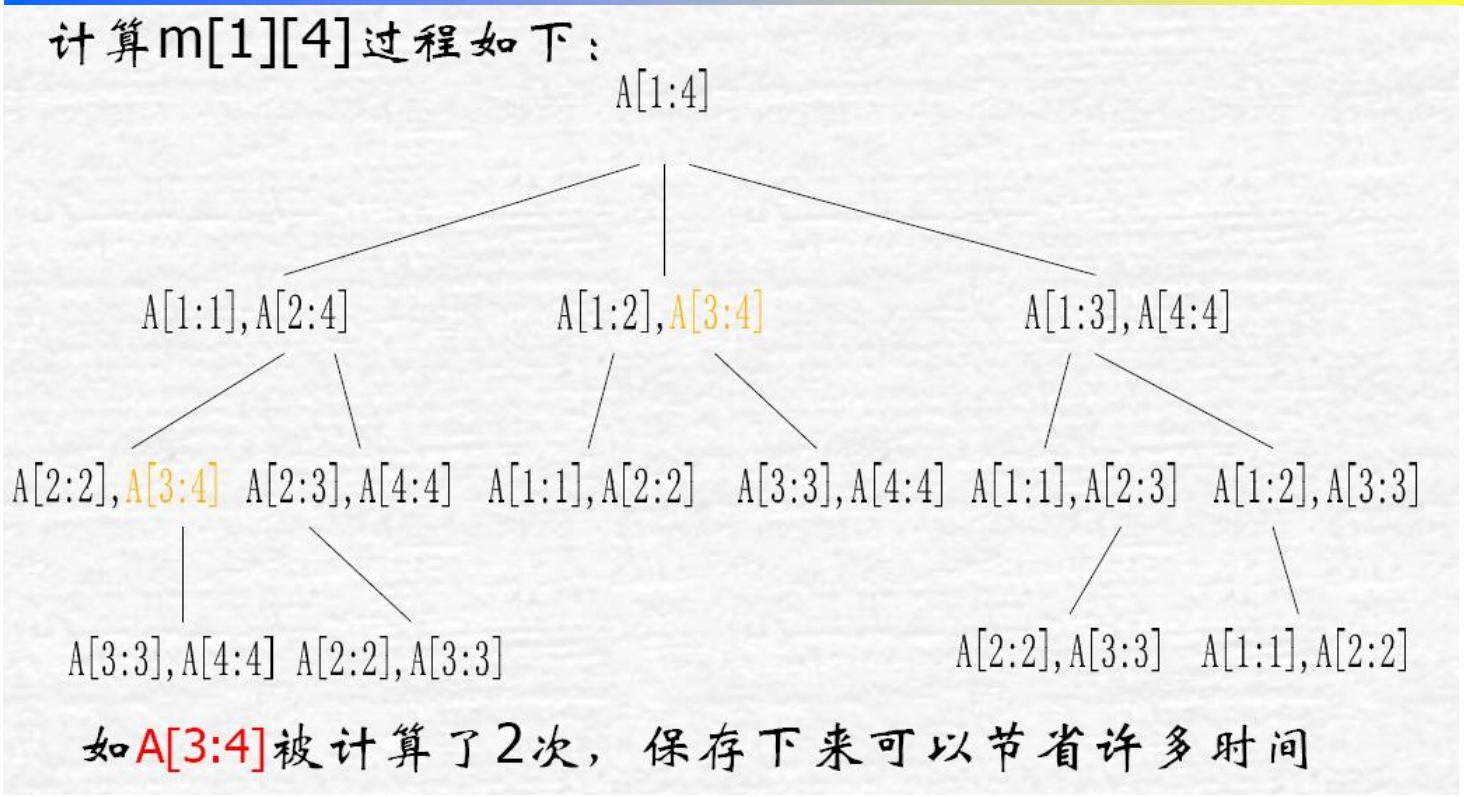
这里，

$$(A_iA_{i+1}\cdots A_k)_{p_{i-1}\times p_k}\times(A_{k+1}A_{k+2}\cdots A_j)_{p_k\times p_j}$$

取得的k为A[i:j]最优次序中的断开位置，并记录到表s[i][j]中，即s[i][j]←k。

注：m[i][j]实际是子问题最优解的解值，保存下来避免重复计算。

15.5 矩阵链乘法问题



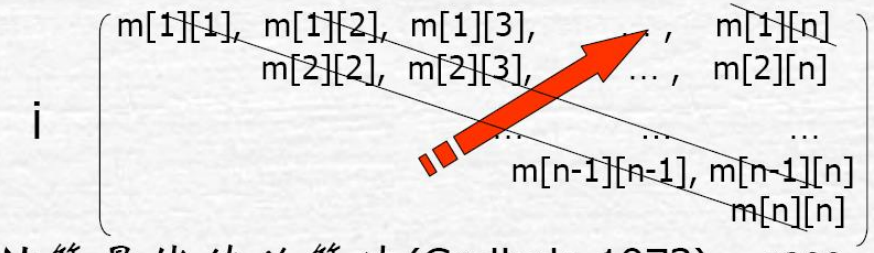
- 在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

15.5 矩阵链乘法问题

- 步骤3：计算最优代价，自底向上记忆化方式求解m[i][j]

-- 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。

- 计算方向：以链长l递增方向，j



- 计算最优值的算法(Godbole,1973)：P200

$T(n)=O(n^3)$, $S(n)=O(n^2)$

- 注：①如果自顶向下计算(含重复计算)，这样效率较低(为$\Omega(2^n)$)。

②也可以采用自顶向下的记忆型递归算法P207。

③Hu和Shing(1980,1982,1984)找到O(nlogn)算法

15.5 矩阵链乘法问题

```
void MatrixChain(int *p, int n, int **m, int **s)
{
      for (int i = 1; i <= n; i++) m[i][i] = 0;
      for (int r = 2; r <= n; r++)
        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;}
            }
        }
}
```

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$O(n^2)$。

15.5 矩阵链乘法问题

步骤4：构造最优解

- 利用s[i][j]中保存的k，进行对A[i:j]的最佳划分，加括号为$(A_iA_{i+1}\cdots A_k)\times(A_{k+1}A_{k+2}\cdots A_j)$
- 构造最优解的算法：P201

```
PrintOptimalParens(s, i, j)
{  if i=j then
       print "A"i;
   else
   { print "(";
     PrintOptimalParens(s, i, s[i,j]);
     PrintOptimalParens(s, s[i,j]+1, j);
     print ")";
   }
}
```

15.5 矩阵链乘法问题

- 计算示例：

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

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						

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

Result

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

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

15.6 适用条件

一、最优子结构性质

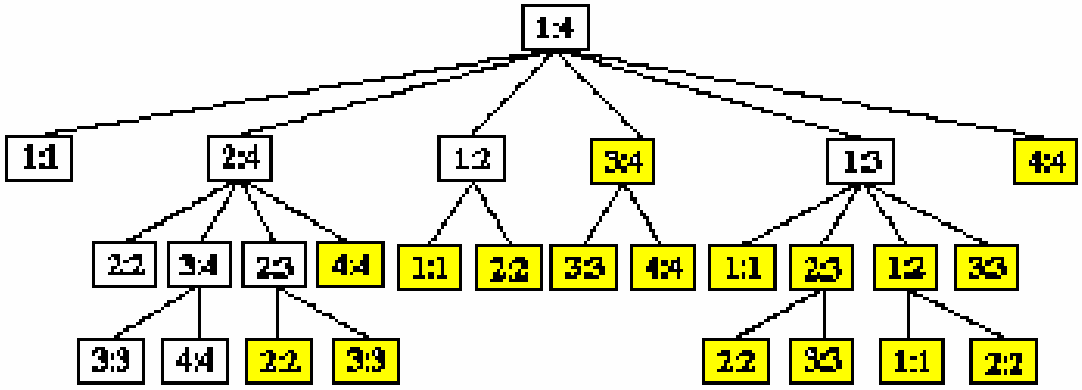
- 如果问题的最优解是由其子问题的最优解来构造的，则称该问题具有**最优子结构性质**。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

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）

15.6 适用条件

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。



15.7 设计技巧

- 动态规划的设计技巧：阶段的划分、状态的表示和存储表的设计；
- 在动态规划的设计过程中，阶段的划分和状态的表示是其中最重要的两步，这两步会直接影响该问题的计算复杂性和存储表设计，有时候阶段划分或状态表示的不合理还会使得动态规划法不适用。
- 问题的阶段划分和状态表示，需要具体问题具体分析，没有一个清晰明朗的方法；
- 在实际应用中，许多问题的阶段划分并不明显，这时如果刻意地划分阶段法反而麻烦。一般来说，只要该问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优性原理），则可以考虑用动态规划解决。

15.7 设计技巧

- 如何判断问题满足最优性原理？

思路：利用反证法，通过假设每一个子问题的解都不是最优解，然后导出矛盾，即可做到这一点。

- 例1：设G是一个有向加权图，则G从顶点i到顶点j之间的最短路径问题满足最优性原理。

证明：（反证法）

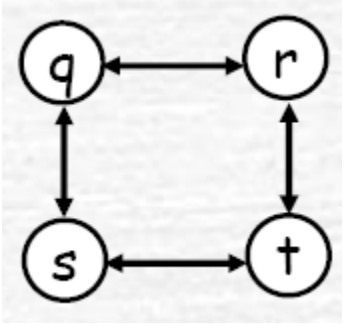
设i～ip～iq～j是一条最短路径，但其中子路径ip～iq～j不是最优的。

假设最优的子路径为ip～iq'～j，则我们可以重新构造一条路径：i～ip～iq'～j，显然该路径长度小于i～ip～iq～j，与i～ip～iq～j是顶点i到顶点j的最短路径相矛盾。

所以原问题满足最优性原理。

15.7 设计技巧

例2：有向图的最长路径问题不满足最优性原理。



证明：

如右图所示，q→r→t是q到t的最长路径，
而q→r的最长路径是q→s→t→r，
r→t的最长路径是r→q→s→t，
但q→r和r→t的最长路径合起来并不是q到t的最长路径。
所以，原问题并不满足最优性原理。

注：因为q→r和r→t的子问题都共享路径s→t，组合成原问题解时，有重复的路径对原问题是不允许的。

15.8最长公共子序列（LCS）

- 子序列定义

给定序列$X=\{x_1,x_2,\cdots,x_m\}$，序列$Z=\{z_1,z_2,\cdots,z_k\}$是X的子序列，是指：存在一个严格递增下标序列$\{i_1,i_2,\cdots,i_k\}$，使得对于所有$j=1,2,\cdots,k$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}。

- 两个序列的公共子序列定义

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。

15.8最长公共子序列（LCS）

- **问题描述：**

  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的最大长度公共子序列。

- Example:

  In biological application, given two DNA sequences, for instance

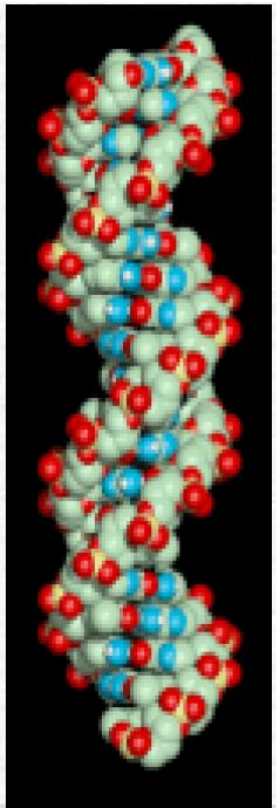
  S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

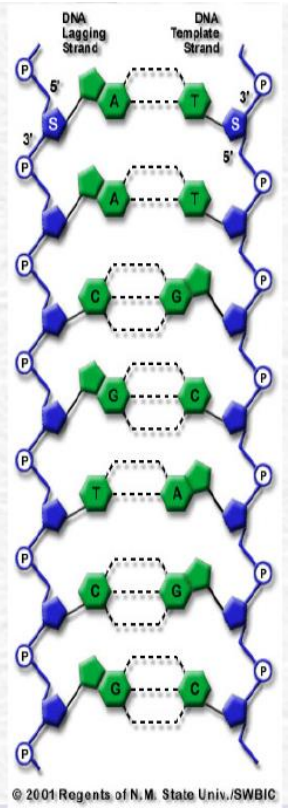
  and

  S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA,

  how to compare them?

  We have various standards of similarity for distinct purposes. While, the LCS of S_1 and S_2 is S_3 = GTCGTCGGAAGCCGGCCGAA.





15.8最长公共子序列（LCS）

- **Step 1: LCS最优解的结构特征**

  定义X的i^{th}前缀：$X_i = (x_1, x_2, \cdots, x_i), i = 1 \sim m$

  $X_0 = \phi$， ϕ为空集

- **定理15.1（一个LCS的最优子结构）**

  设序列$X = (x_1, x_2, \cdots, x_m)$和$Y = (y_1, y_2, \cdots, y_n)$， $Z = (z_1, z_2, \cdots, z_k)$是X和Y的任意一个LCS，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}的一个LCS；

  (2) 若 $x_m \neq y_n$且$z_k \neq x_m$，则Z是X_{m-1}和Y的一个LCS；

  (3) 若 $x_m \neq y_n$且$z_k \neq y_n$，则Z是X和y_{n-1}的一个LCS；

- 该定理两个序列X和Y的一个LCS包含了两个序列的前缀的一个LCS，即最长公共子序列问题具有最优子结构性质。

15.8最长公共子序列（LCS）

- Th15.1 的证明

(1)若$x_m=y_n$, ==> $z_k=x_m=y_n$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS;

（应用反证法）

先证：$z_k=x_m=y_n$。

若$z_k<>x_m$（也有$z_k<>y_n$），则将x_m加到Z后，于是获得X和Y的长度为k+1的CS，与Z是X和Y的LCS矛盾。

==> $z_k=x_m=y_n$

再证：Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS。

由Z的定义 ==> 前缀Z_{k-1}是X_{m-1}和Y_{n-1}的CS(长度为k-1)

若Z_{k-1}不是X_{m-1}和Y_{n-1}的LCS，则存在一个X_{m-1}和Y_{n-1}的公共子序列W，W的长度>k-1，于是将z_k加入W之后，则产生的公共子序列长度>k，与Z是X和Y的LCS矛盾。

==> Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS

15.8最长公共子序列（LCS）

- Th15.1 的证明

(2)若$x_m <> y_n$且$z_k <> x_m$, ==> Z是X_{m-1}和Y的一个LCS;

∵ $z_k <> x_m$，则Z是X_{m-1}和Y的一个CS

下证：Z是X_{m-1}和Y的LCS

（反证）若不然，则存在长度>k的CS序列W，

显然，W也是X和Y的CS，但其长度>k，矛盾。

(3)若$x_m <> y_n$且$z_k <> y_n$, ==> Z是X和Y_{n-1}的一个LCS;

(3)与(2)对称，类似可证。

综上，定理15.1证毕。 □

15.8最长公共子序列（LCS）

Step 2: 子问题的递归解

— 定理15.1将X和Y的LCS分解为2种情况：

(1) if $x_m=y_n$ then　　//解一个子问题

找X_{m-1}和Y_{n-1}的LCS；

(2) if $x_m \neq y_n$ then　//解二个子问题

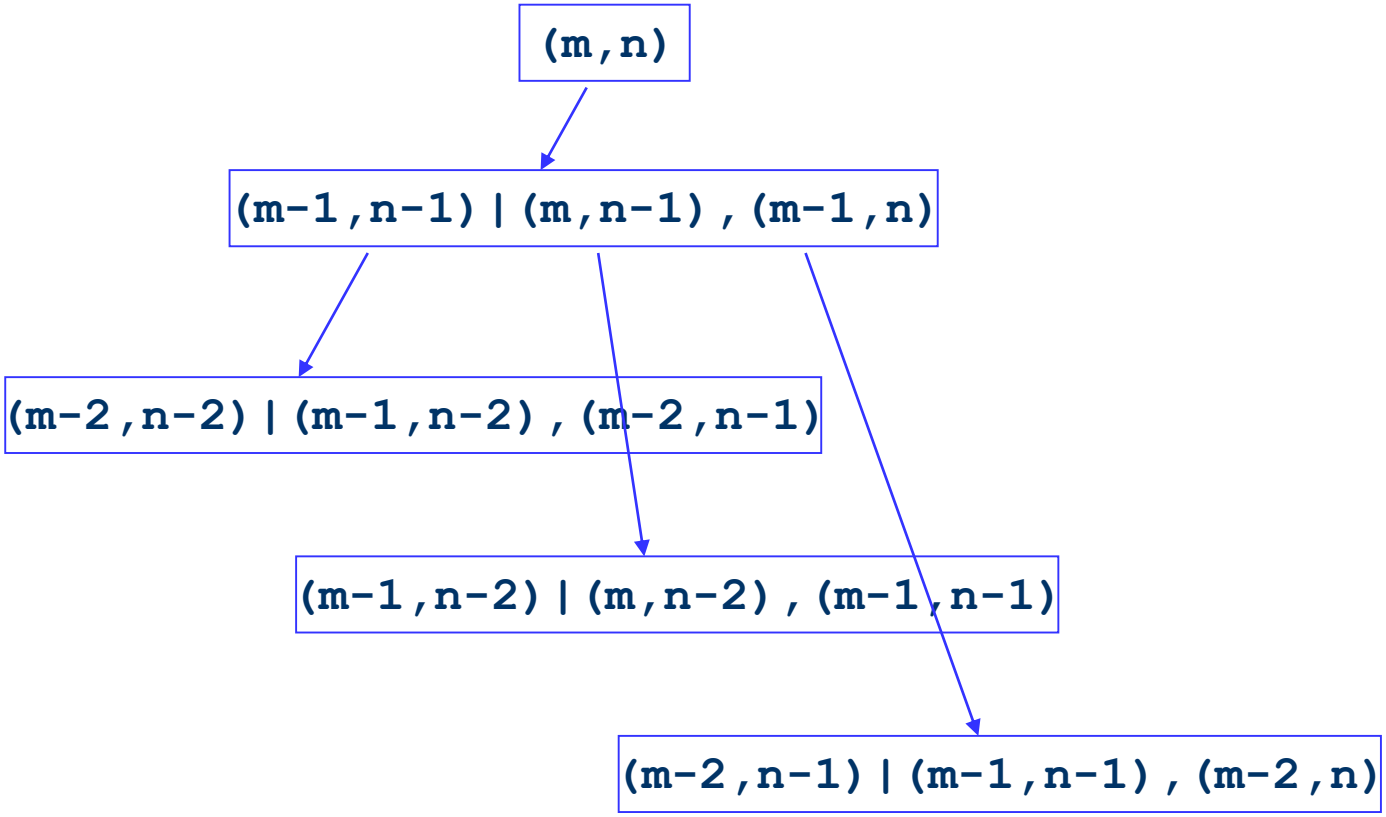
找X_{m-1}和Y的LCS；找X和Y_{n-1}的LCS；

取两者中的最大的；

— c[i,j]定义为X_i和Y_j的LCS长度，$i = 0 \sim m, j = 0 \sim n$；

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

15.8最长公共子序列（LCS）



15.8最长公共子序列（LCS）

Step 3：计算最优解值

— 数据结构设计

c[0..m, 0..n]　　//存放最优解值，计算时行优先

b[1..m, 1..n]　　//解矩阵，存放构造最优解信息

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

当构造解时，从b[m,n]出发，上溯至i=0或j=0止

上溯过程中，当b[i,j]包含"↖"时打印出xi(yj)

15.8最长公共子序列（LCS）

- 算法

```
LCS_Length(X, Y)
{   m ←length[X];  n ←length[Y];
    for i←0 to m do c[i,0] ←0;    //0列
    for j←0 to n do c[0,j] ←0;    //0行
    for i←1 to m do
        for j←1 to n do
            if x_i=y_j then
             {  c[i, j]←c[i-1, j-1] +1; b[i, j] ← "↖"; }
            else
                if c[i-1, j]>=c[i, j-1] then
                 {  c[i, j]←c[i-1, j]; b[i, j] ← "↑"; }  //由X_{i-1}和Y_j确定
                else
                 {  c[i, j]←c[i, j-1]; b[i, j] ← "←"; }  //由X_i和Y_{j-1}确定
     return b and c;
   }
```

- 过程LCS-Length以两个序列X和Y为输入，它把c[i,j]的值填入一个按行计算表项的表c[0..m, 0..n]中，并维护表b[1..m, 1..n]以简化最优解的构造。b[i,j]指向一个表项，对应于在计算c[i,j]时所选择的最优子问题的解。程序最后返回b和c。c[m,n]包含X和Y的一个LCS的长度。

15.8最长公共子序列（LCS）

Step 4：计算最优解值

—算法：

```
Print_LCS(b, X, i, j)
{  if i=0 or j=0 then return;
   if b[i,j]="↖" then
    {  Print_LCS(b, X, i-1, j-1);
       print  xi;
    }
    else
       if b[i,j]="↑" then Print_LCS(b, X, i-1, j);
       else  Print_LCS(b, X, i, j-1);
}
```

说明：

—每当在表项b[i,j]中遇到一个"↖"时，即意味着xi=yj是LCS的一个元素；

—时间复杂度： θ (m+n)

15.8最长公共子序列（LCS）

- 若X = <A, B, C, B, D, A, B>， Y=<B, D, C, A, B, A>， 则

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
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- 最优解: BCAB 或 BCBA

15.8最长公共子序列（LCS）

改进代码：

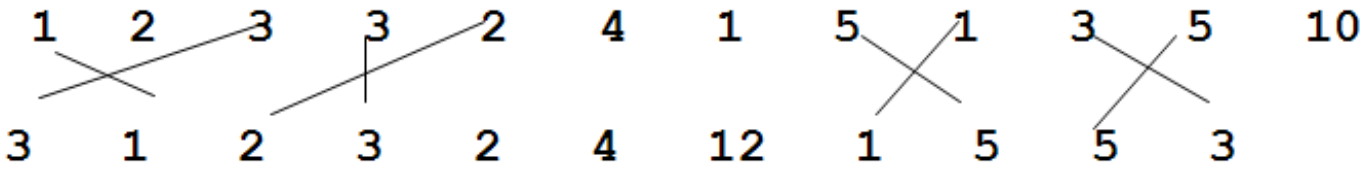
- 在算法lcsLength和lcs中，可进一步将数组b省去。事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。对于给定的数组元素c[i][j]，可以不借助于数组b而仅借助于c本身在常数时间内确定c[i][j]的值是由c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的。
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。

15.8最长公共子序列（LCS）

思考题:

- 交错匹配（最长公共子串的改编）:

给你两排数字,只能将两排中数字相同的两个位置相连,而每次相连必须有两个匹配形成一次交错,交错的连线不能再和别的交错连线有交点.问这两排数字最多能形成多少个交错匹配?



15.9 多段图规划

- 问题描述

多段图G=(V,E)是一个有向图，且具有以下特征：

(1) 划分为k≥2个不相交的集合Vi, 1≤i≤k;

(2) V_1和V_k分别只有一个结点s(源点)和t(汇点)；

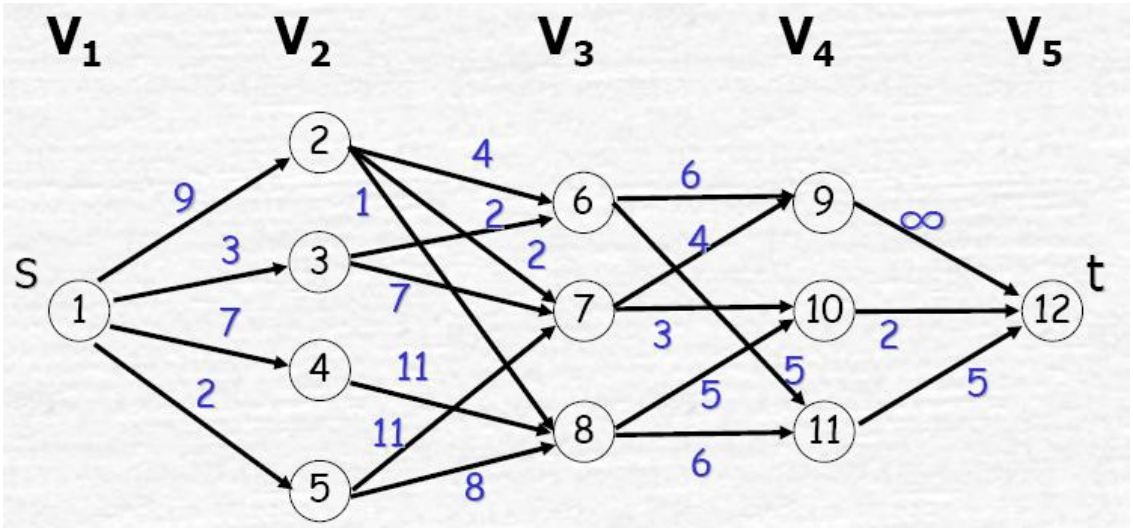
(3) 若<u, v> ∈E(G)，u∈V_i，则v∈V_{i+1},边上成本记c(u,v)；

若<u,v>不属于E(G)，边上成本记c(u,v)=∞。

求由s到t的最小成本路径。

15.9 多段图规划

- 举例：一个5-段图



求从s到t的最短路径。

15.9 多段图规划

多段图问题满足最优性原理

设s,…,v_{ip},…,v_{iq},…,t是一条由s到t的最短路径，则v_{ip},…, v_{iq},…,t也是由vip到t的最短路径。(反证即可)

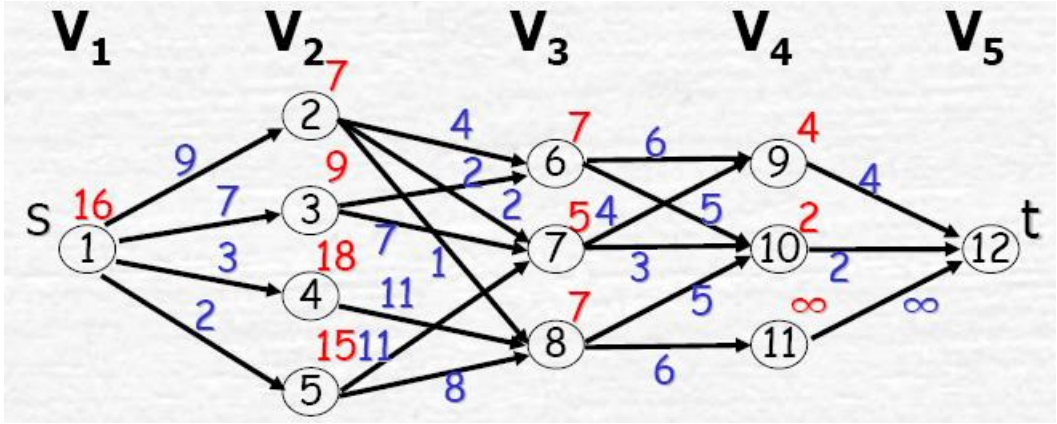
递归式推导

设cost(i, j)是V_i中结点v_j到汇点t的最小成本路径的成本，递归式为：

$$cost(i,j)=\begin{cases} c(j,t) & i=k-1 \\ \min\limits_{\substack{v_l\in V_{i+1}\\ <j,l>\in E(G)}}\{c(j,l)+cost(i+1,l)\} & 1\le i<k-1 \end{cases}$$

15.9 多段图规划

- 计算过程(以5-段图为例)



cost(4, 9)=4, cost(4, 10)=2, cost(4, 11)=∞

cost(3, 6)=7, cost(3, 7)=5, cost(3, 8)=7

cost(2, 2)=7, cost(2, 3)=9, cost(2, 4)=18, cost(2, 5)=15

cost(1, 1)=min{9+cost(2,2), 7+cost(2,3), 3+cost(2,4), 2+cost(2,5)} = 16

构造解：解1(1, 2, 7, 10, 12)，解2(1, 3, 6, 10, 12)

15.9 多段图规划

```
MultiStageGraph( G, k, n, p[] )
{//输入n个结点的k段图，假设顶点按段的顺序编号
 //E(G)是边集，p[1..k]是最小成本路径
   new cost[n];    //生成数组cost, cost[j]相当于前面的cost(i,j)
   new d[n];       //生成数组d, d[j]保存v_j与下一阶段的最优连接点
   cost[n]=0;
   for i=n-1 dwonto 1 do  //计算cost[i]和d[i]
   {  cost[i]=∞;
      while( 任意<i, r>∈E(G) )  //r是下一阶段中的顶点
        if( c(i, r)+cost[r]<cost[i] )
        {   cost[i]=c(i, r)+cost[r]; d[i]=r;
        }
   }                                          } O(n+e)
   p[1]=1; p[k]=n;   //以下是找一条最小成本路径（构造解）
   for i=2 to k-1 do   p[i]=d[p[i-1]];        } O(k)
}                  ∴ T(n)=O(n+e)
```

15.10 最大子段和

- 问题描述：

给定整数序列$a_1,a_2,\cdots,a_n$，求形如$\sum_{k=i}^{j} a_k$的子段和的最大值。规定子段和为负整数时，定义其最大子段和为0，即

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

如：整数序列$(a_1, a_2, a_3, a_4, a_5, a_6) = (-2, 11, -4, 13, -5, -2)$

最大子段和为：

$$\sum_{k=2}^{4} a_k = 20$$

15.10 最大子段和

- 直接算法：

```
MaxSubSum1(n, a[], besti, bestj)
{ //数组a[]存储ai，返回最大子段和，保存起止位置到Besti,Bbestj中
    sum=0;
    for i=1 to n do
       for j=i to n do
       {  thissum=0;
          for k=i to j do        //可以改进，省略此循环
             thissum += a[k];
          if(thissum>sum)
          {  sum=thissum;
             besti=i; bestj=j;
          }
       }
    return sum;
}
```

- 分析：时间复杂度为$O(n^3)$；
- 思考：对k循环可以省略，改进后的算法时间复杂度为$O(n^2)$；

15.10 最大子段和

分治法求解

将A[1..n]分为a[1..n/2]和a[n/2+1..n]，分别对两区段求最大子段和，这时有三种情形：

- ✓ case 1: a[1..n]的最大子段和的子段落在a[1..n/2]；
- ✓ case 2: a[1..n]的最大子段和的子段落在a[n/2..n]；
- ✓ case 3: a[1..n]的最大子段和的字段跨在a[1..n/2]和a[n/2+1..n]之间

此时，

- 对Case 1和Case 2可递归求解；
- 对Case 3，可知a[n/2]和a[n/2+1]一定在最大和的子段中，因此，

① 在a[1..n/2]中计算：$S_1 = \max_{1 \le i \le n/2} \sum_{k=i}^{n/2} a_k$

② 在a[n/2+1..n]中计算：$S_2 = \max_{n/2+1 \le i \le n} \sum_{k=n/2+1}^{i} a_k$

易知，$S_1 + S_2$ 是Case 3的最大值。

15.10 最大子段和

- 算法

```
MaxSubSum2(a[], left, right)
{ //返回最大子段和
   sum=0;
   if( left=right )
     sum=a[left]>0?a[left]:0;
   else
   {  center=(left+right)/2;
     leftsum=
          MaxSubSum2(a, left,center);
     rightsum=
          MaxSubSum2(a, center+1, right);
     s1=0; leftmidsum=0;
     for i=center to left do
     {    leftminsum += a[i];
          if (leftmidsum>s1)  then
               s1=leftmidsum;
      }
     s2=0; rightmidsum=0;
     for i=center+1 to right do
     {    rightminsum += a[i];
          if(rightmidsum>s2) then
               s2=rightmidsum;
      }
      sum=s1+s2;
      if(sum<leftsum) then sum=leftsum;
      if(sum<rightsum) then sum=rightsum;
   } //end if
   return sum;
}//end
```

$$T(n)=\begin{cases} O(1) & n=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

$$\Rightarrow \quad T(n)=O(n\log n)$$

15.10 最大子段和

动态规划法求解

（1）描述最优解的结构

- 子问题定义：考虑所有下标以j结束的最大子段和b[j]，即

$$b[j]=\max_{1\le i\le j}\left\{\max\left\{0,\ \sum_{k=i}^{j}a_k\right\}\right\}\quad j=1,2,\ldots,n$$

- 原问题与子问题的关系：

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

（2）递归定义最优解的值

$$b[j]=\begin{cases}\max\{a_1,0\} & j=1\\ \max\{b[j-1]+a_j,0\} & j>1\end{cases}$$

15.10 最大子段和

- 算法：

```
int MaxSubSum3(int n, int a[])
{
    int sum=0, b=0;   //sum存储当前最大的b[j], b存储b[j]
    for(int j=1; j<=n; j++)
    {   b += a[j];
        if(b<0) then b=0;        // b[j]
        if(b>sum) then sum=b;
     }
    return sum;
}
```

- 运行时间：O(n)
- 思考：如果要记录最大子段对应的区间，该如何修改程序？

15.10 最大子段和

- 思考题(最大子段和问题的推广):

(1)最大子矩阵和问题：给定一个m行n列的整数矩阵A，试求矩阵A的一个子矩阵，使其各元素之和为最大。

(2)最大m子段和问题：给定由n个整数（可能为负整数）组成的序列$a_1, a_2, \cdots, a_n$，以及一个正整数m，要求确定序列$a_1, a_2, \cdots, a_n$的m个不相交子段，使这m个子段的总和达到最大。

15.11 0-1背包问题

问题描述：

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。

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

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

15.11 0-1背包问题

- Knap(1, n, c)定义如下：

$$\max\sum_{i=l}^{n} v_i x_i \qquad v_i > 0$$

$$\begin{cases} \sum_{i=l}^{n} w_i x_i \le c & w_i > 0 \\ x_i \in \{0,1\} & l \le i \le n \end{cases}$$

求$(x_l, x_{l+1}, \ldots, x_n)$使目标函数最大

- 例如：$w = (w_1, w_2, w_3) = (2, 3, 4)$, $v=(v_1, v_2, v_3)=(2, 3, 4)$,
  求Knap(1, 3, 6)

取x = (1, 0, 1)时，

Knap(1,3,6) = (v1x1+v2x2+v3x3) = 1*1 + 2*0 + 5*1 = 6最大

用穷举法求解，时间复杂度为$O(n2^n)$

15.11 0-1背包问题

0-1背包问题Knap(1, n, c)满足最优性原理

证明：(反证法)

设$(y_1,y_2,\cdots,y_n)$是Knap(1,n,c)的一个最优解，下证$(y_2,\cdots,y_n)$是Knap(2,n,c-w_1y_1)子问题的一个最优解。

若不然，设(z2,…,zn)是Knap(2,n,c-w1y1)的最优解，因此有

$$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i \text{ 且 } \sum_{i=2}^{n} w_i z_i \le c - w_1 y_1$$

$$\Rightarrow \quad v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=1}^{n} v_i y_i \text{ 又有 } w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le c$$

说明$(y_1,z_2,\cdots,z_n)$是Knap(1,n,c)的一个更优解，矛盾。

15.11 0-1背包问题

子问题定义

设所给0-1背包问题的子问题记为Knap(i, n, j)，j≤c(假设c, w_i取整数)，其定义为：

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

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

- 其最优值为m(i, j)，即m(i, j)是背包容量为j，可选物品为i, i+1, …, n的0-1背包问题的最优值。

注：子问题的背包容量j在不断地发生变化。

15.11 0-1背包问题

- 最优值的递归式如下：

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

说明：当$j<w_i$时，只有$x_i=0$，$\therefore m(i,j)=m(i+1,j)$；

当$j\geq w_i$时，$\begin{cases}\text{取}x_i=0\text{时}, & \text{为}m(i+1,j)\\ \text{取}x_i=1\text{时}, & \text{为}m(i+1,j-w_i)+v_i\end{cases}$

- 临界条件：

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

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。

15.11 0-1背包问题

- 代码：

```
Knapsack( v[], w[], c, n, m[][] )
{//输出m[1][c]
   jMax=min(w[n]-1, c); //j≤jMax, 即0≤j<w_n; j>jMax, 即j≥ w_n
   for j=0 to jMax do  m[n][j]=0;      //0≤j<w_n, (4)式
   for j=w[n] to c do   m[n][j]=v[n];  //j≥ w_n, (3)式
   for i=n-1 dwonto 2 do //i>1表示对i=1暂不处理，i=1时只需求m[1][c]
   {  jMax=min(w[i]-1, c);
      for j=0 to jMax do  //0≤j<w_i, (2)式
          m[i][j]=m[i+1][j];
      for j=w[i]  to c  do    //j≥ w_i, (1)式
          m[i][j]=max(m[i+1][j], m[i+1][j-w[i]]+v[i]);
    }
   if c>=w[1]  then  m[1][c]=max(m[2][c], m[2][c-w[1]]+v[1]);
   else m[1][c]=m[2][c];
}
```

15.11 0-1背包问题

- 构造最优解：

```
Traceback( w[], c, n, m[][], x[])
{//输出解x[1..n]
   for i=0 to n do
      if(m[i][c]=m[i+1][c])  x[i]=0;
      else
       {   x[i]=1;
           c -= w[i];
       }
   x[n]=(m[n][c])?1:0;
}
```

15.12 备忘录动态规划算法

- 通常，动态规划算法都是由底向上求解，逐一求解子问题，最终得到原问题的解。无论所求解的子问题在后面是否利用到，动态规划法都要记录所有子问题的解。这种方法不够直观。
- 备忘录动态规划法，不仅具有通常动态规划方法的效率，同时还采取了一种自顶向下的策略。其思想是备忘原问题的自然但是低效的递归算法。像在通常的动态规划算法中一样，维护一个记录了子问题解得表，但有关填表动作的控制结构更像递归算法。

15.12 备忘录动态规划算法

- 矩阵链乘问题的备忘录动态规划算法：

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];
    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;
    return u;
}
```

```
int (Memorized-Matrix-Chainp)
{
  n = length[p]-1;
  for( int i = 1; i <=n ; i++)
      for( j=i; j<=n; j++ )
            m[i,j] = -1;     //表示无穷大
  return  LookupChain(p, 1, n);
}
```

15.13 其它问题

- 动态规划算法除了可以用于求解最优化问题之外，还可以用于非最优化问题。如计算二项式系数：

- **二项式系数**，或**组合数**，是定义为形如$(1+x)$的二项式n次幂展开后x的系数（其中n为自然数，k为整数），通常记为。从定义可看出二项式系数的值为整数。

$$(a+b)^n = C(n,0)a^n + \cdots + C(n,k)a^{n-k}b^k + \cdots + C(n,n)b^n$$

二项式系数：

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

- 如何利用动态规划技术来求解呢？

谢谢！

Q & A