# 算法分析与设计

# 第4章 动态规划

主讲人：甘文生 PhD

Email: wsgan001@gmail.com

暨南大学网络空间安全学院

Fall 2021

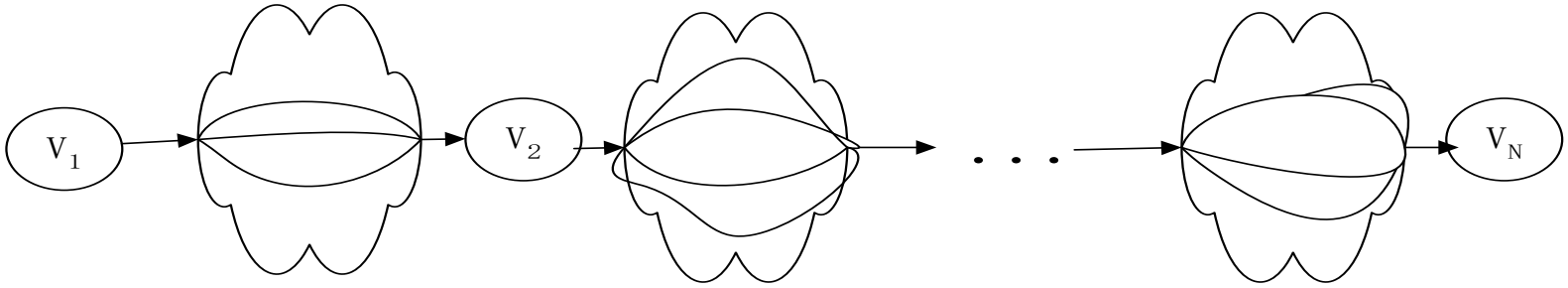
Jinan University, China

# 第四章 动态规划

**内容提要：**

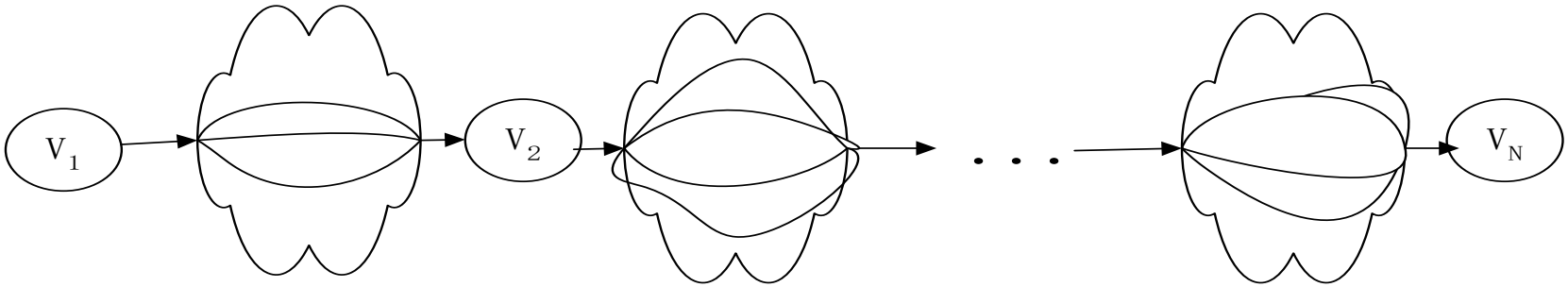
- 理解动态规划算法的由来、概念、定义
- 掌握动态规划算法的要素
- 掌握设计动态规划算法的步骤
- 通过范例学习动态规划算法的设计策略

# 1. 多阶段决策问题



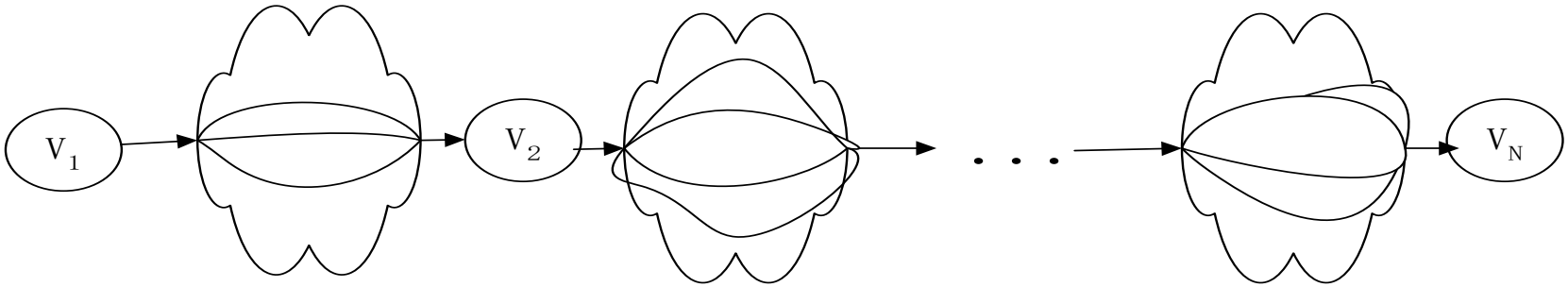

- **多阶段决策过程**：问题的活动过程分为若干相互联系的阶段，任一阶段i以后的行为仅依赖于i阶段的过程状态，而与i阶段之前的过程如何达到这种状态的方式无关。在每一个阶段都要做出决策，这一系列的决策称为多阶段决策过程 (Multistep Decision Process, MDP) 。
- **最优化问题**：问题的每一阶段可能有多种可供选择的决策，必须从中选择一种决策。各阶段的决策构成一个决策序列。决策序列不同，所导致的问题的结果可能不同。
- **多阶段决策的最优化问题**：求能够获得问题最优解的决策序列——最优决策序列。

# 2. 多阶段决策问题的求解策略



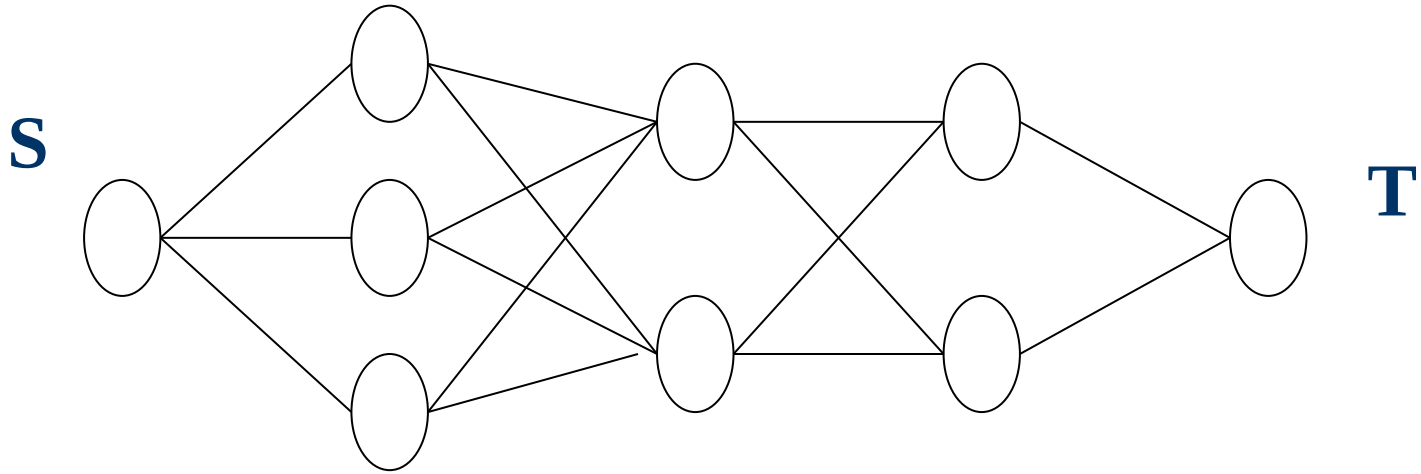

- 1）枚举法：穷举所有可能的决策序列，从中选取能获得最优解的决策序列。
- 2）动态规划：20世纪50年代初美国数学家R. E. Bellman等人在研究多阶段决策过程的优化问题时，提出了著名的最优化原理(principle of optimality)，把多阶段过程转化为一系列单阶段问题，创立了解决这类过程优化问题的新方法——动态规划。1957年出版的个人专著《Dynamic Programming》，这是该领域的第一本著作。
- 思考：如果有面值为1元、3元和5元的硬币若干枚，如何用最少的硬币凑够11元？如何用最少的硬币凑够i元(i<11)？

# 3. 最优性原理(Principle of Optimality)




- 过程的**最优决策序列**具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。
- **利用动态规划求解问题的前提**
  - ✓ 1）证明问题满足最优性原理。如果对所求解问题证明满足最优性原理，则说明用动态规划方法有可能解决该问题。
  - ✓ 2）获得问题状态的递推关系式。获得各阶段间的递推关系式是解决问题的关键。

# 最短路问题 (Shortest Path Problem)




特点：多阶段决策 - 子决策仍然最优 - 动态规划技术

**动态规划 – R.E. Bellman (1950’s)**

许多网络优化问题要用到动态规划技术

# 数塔问题 (Leetcode)

## 120. Triangle

Description | Hints | Submissions | Discuss | Solution

Discuss | Pick One

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

用最短路径(数字之和最小),
从塔顶走到塔底
而且不能使用额外的空间,

The minimum path sum from top to bottom is `11` (i.e., 2 + 3 + 5 + 1 = 11).

**Note:**

Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle.

特点：多阶段决策 - 子决策仍然最优 - 动态规划技术

# 历史由来

- 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, MDP)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。

# 动态规划的意义

- **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。

# 问题与概念

- 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。

# 多阶段决策模型

- **决策(Decision Making)**，是为了达到一定的目的，从若干个可能的策略(Policy)（如行动、方案）中选取最好的策略的过程。一般来说，一个决策模型包含三个最基本的因素：
  - ✓ **（1）自然状态（或简称状态, State）**：这是指决策活动中决策者无法控制的一些因素，即决策时客观对象所具备的基本条件。状态的集合称为状态集合或状态空间。
  - ✓ **（2）策略**：这是指决策活动中决策者可以采取的行动方案. 策略的集合称为策略集合或策略空间。
  - ✓ **（3）益损值**：这是指决策活动中决策者可以采取不同的策略，在不同的自然状态下所获得的收益或损失值。它是策略和状态的函数，也是决策活动的目标和基础。

- **战略决策(高层决策)、战术决策(中层决策)、操作决策(基本决策)**
- **单目标决策、多目标决策**
- **单阶段决策（一次决策）、多阶段决策**
- **确定型决策、非确定型决策或风险型决策（随机决策、模糊决策）**

# 无后效性的多阶段决策过程

动态规划中，多阶段决策问题具有无后效性（马尔科夫性质），即当某阶段的状态一旦确定, 则此后过程的演变不再受此前各状态和决策的影响, 或者说“未来与过去无关”. 即由状态$x_k$出发的后部子过程可以看成一个以$x_k$为初始状态的独立过程.

状态转移方程(equation of state) $x_{k+1} = T_k(x_k, u_k)$

由所有各阶段的决策组成的决策序列称为全过程策略，或简称策略，记为$p_{1,n}(x_1)$. 可供选择的所有全过程策略的集合构成允许策略集合，记为$P_{1,n}(x_1)$ .其中能使总体性能达到最优的策略称为最优策略，一般记为

$$p_{1,n}^{*} = (u_1^{*}, u_2^{*}, \cdots, u_n^{*})$$

相应于后部子过程（$k$子过程）的决策序列称为子策略，记为$p_{k,n}(x_k)$，所有允许子策略的集合记为$P_{k,n}(x_k)$.

# 无后效性的多阶段决策过程
## - 准则函数及可分性

准则函数/指标函数（Criterion Function）是衡量策略好坏的尺度(益损值).

- 定义在全过程上的准则函数相当于目标函数，一般记为 $V_{1,n}(x_1; p_{1,n})$，或简记为$V_{1,n}$
- 定义在$k$子过程上的准则函数，记为$V_{k,n}(x_k; p_{k,n})$,简记为$V_{k,n}$
- 准则函数在第$k$阶段一个阶段内的取值称为第$k$阶段的准则函数，记为$v_k(x_k; u_k)$

最优性原理中，准则函数具有（阶段）可分性，即

$$V_{k,n} = v_k(x_k, u_k) \oplus v_{k+1}(x_{k+1}, u_{k+1}) \oplus \cdots \oplus v_n(x_n, u_n),$$

其中$\oplus$是满足单调性的某种运算，如加法或乘法等。

一般记为 $$V_{k,n} = \sum_{j=1}^{n} v_j(x_j, u_j) = v_k(x_k, u_k) + V_{k+1,n}$$

# 最优性原理

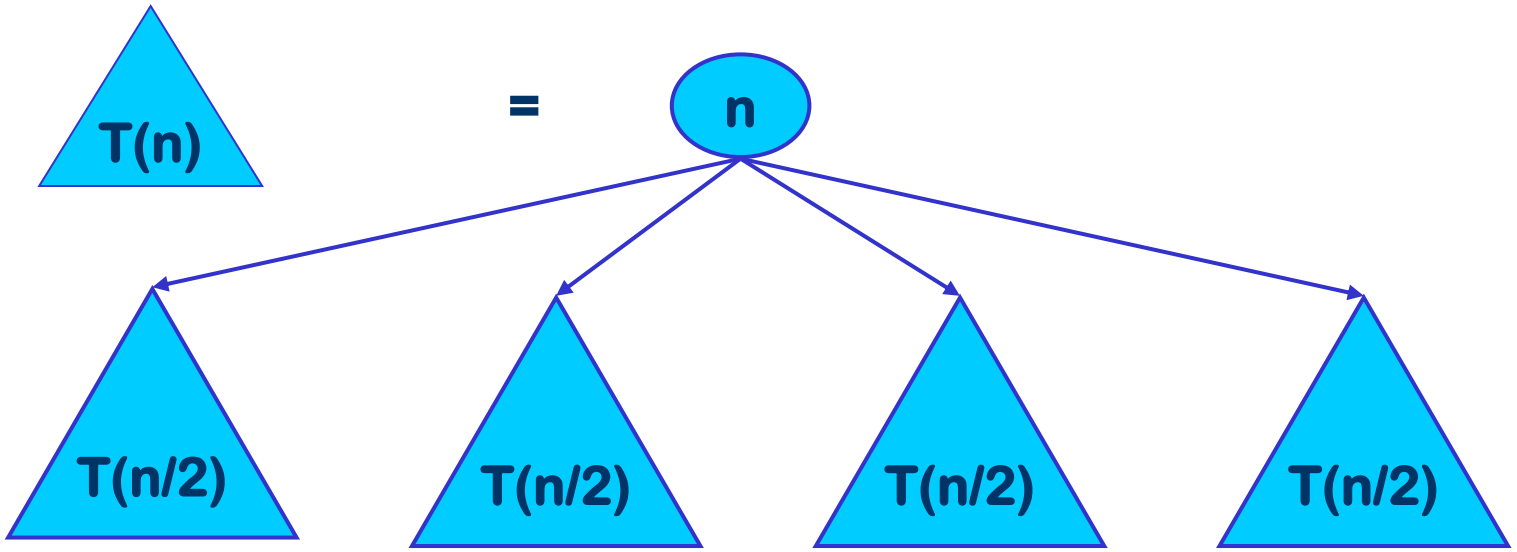
- Bellman最优性原理：
  - 求解问题的一个最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。
  - 注：对具有最优性原理性质的问题而言，如果有一决策序列包含有非最优的决策子序列，则该决策序列一定不是最优的。

“全过程的最优策略具有这样的性质：不管该最优策略上某状态以前的状态和决策如何，对该状态而言，余下的所有决策必定构成最优子策略.”

即：最优策略的任一后部子策略都是最优的。这只是最优性定理的一个推论，即最优策略的必要条件。

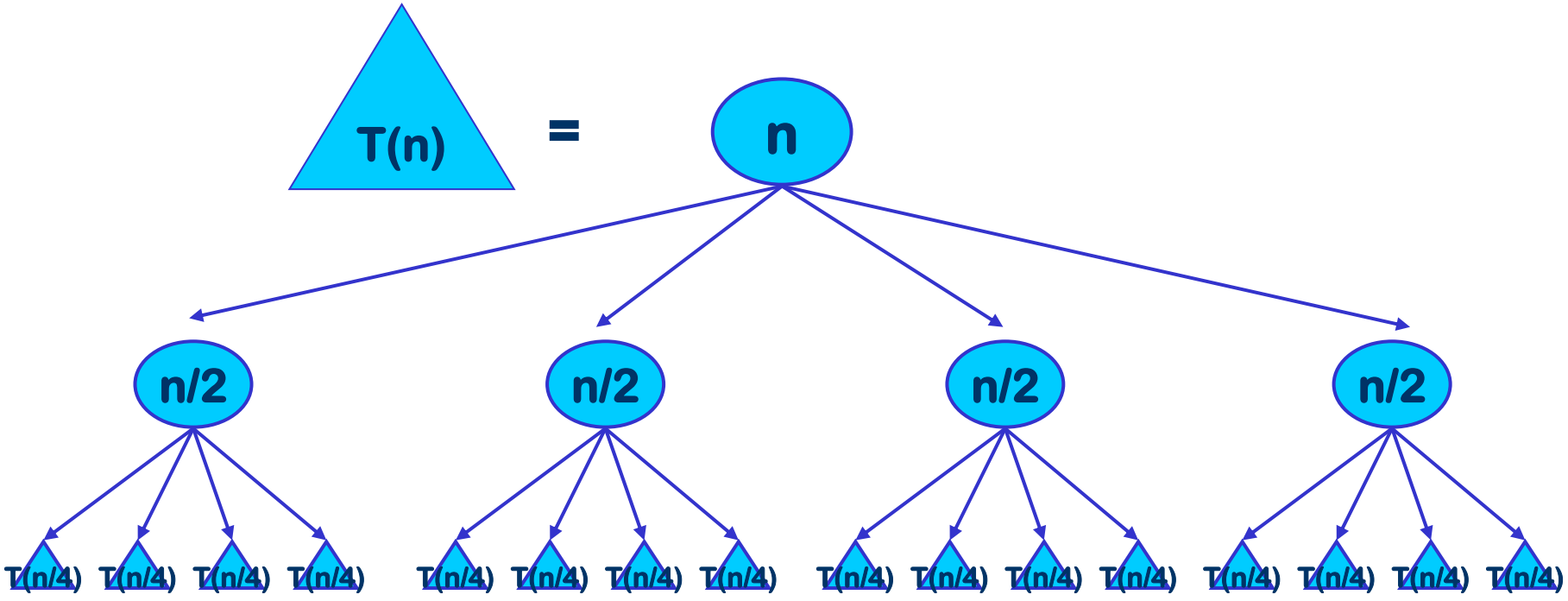
# 总体思想

- 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



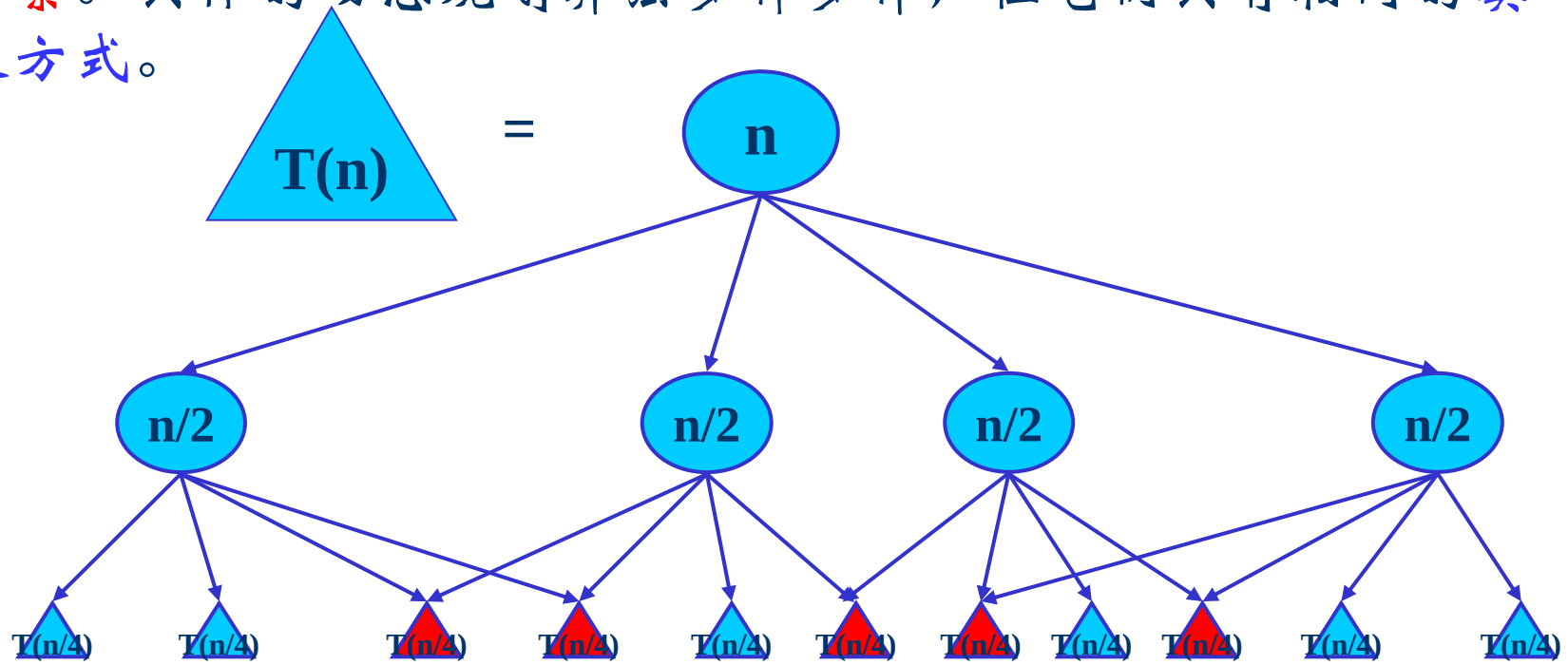
# 总体思想

- 但是经过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。



# 总体思想

- 如果能够保存已解决的子问题的答案，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多种多样，但它们具有相同的填表方式。



# 动态规划的基本思想

（1）动态规划方法的关键在于正确写出基本的递推关系式和恰当的边界条件。要做到这一点，必须将问题的过程分成几个相互联系的阶段，恰当选择状态变量，决策变量和定义最优值函数，从而把一个大问题化成一簇同类型的子问题，然后逐个求解。即从边界条件开始，逐段递推寻优，在每一个子问题的求解中，均利用了它前面的子问题的最优化结果，依次进行，最后一个子问题的最优解，就是整个问题的最优解。

（2）在多阶段决策过程中，动态规划方法是既把当前一段和未来各段分开，又把当前的效益和未来效益综合起来考虑的一种最优化方法。因此，每段决策的选取是从全局来考虑的，与该段的最优选择答案一般是不同的。

（3）在求整个问题的最优策略时，由于初始状态是已知的，而每段的决策都是该段状态的函数，故最优策略所经的各段状态便可逐次变换得到，从而确定最优路线。

# 动态规划的基本思想

进一步说明

- 采用枚举法：若问题的决策序列由n次决策构成，而每次决策有p种选择，则可能的决策序列将有$p^n$个。
- 利用动态规划策略的求解过程中保存了所有子问题的最优解，而舍去了所有不能导致问题最优解的次优决策序列。
- 动态规划：可能有多项式的计算复杂度。

# 总体步骤

- 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；

# 总体步骤 (数学思想)

- **基本步骤：**

（1）正确划分阶段，选择阶段变量$k$.

（2）对每个阶段，正确选择状态变量$x_k$. 选择状态变量时应当注意两点：一是要能够正确描述受控过程的演变特性，二是要满足无后效性.

（3） 对每个阶段，正确选择决策变量$u_k$ .

（4） 列出相邻阶段的状态转移方程： $x_{k+1}= T_k(x_k, u_k)$.

（5）列出按阶段可分的准则函数$V_{1,n}$ .

# 矩阵连乘问题

- **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**。在数学中，矩阵Matrix是一个按照长方阵列排列的**复数或实数**集合。

$$A = \begin{bmatrix} a[1,1] & a[1,2] & \cdots & a[1,m-1] & a[1,m] \\ a[2,1] & a[2,2] & \cdots & a[2,m-1] & a[2,m] \\ \vdots & \vdots & & \vdots & \vdots \\ a[n,1] & a[n,2] & \cdots & a[n,m-1] & a[n,m] \end{bmatrix}$$

**注意：**

① 设$A_{p \times q}$，$B_{q \times r}$两矩阵相乘得到矩阵C，普通乘法次数为$p \times q \times r$

② 加括号对乘法次数的影响：

$A_{10\times100} \times B_{100\times5} \times C_{5\times50}$

((AB)C): 7500次　　(A(BC)): 75000次

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

# 矩阵连乘问题

- **矩阵连乘问题：**
- 存在两个矩阵A和B，如果AB能够计算乘积，则BA不一定能够计算乘积。
  - ✓ 例如：A=3*2， B=2*4，则按照AB的乘积计算顺序能够计算(A的列数2等于B的行数2)，但按照BA的乘积计算顺序却不能够计算(B的列数4不等于A的行数3)
- 存在两个矩阵A和B，满足AB能相乘，BA也能相乘，但结果却可能不一样。
- 矩阵连乘可以被递归为：

$$A_1A_2A_3\cdots A_{s-1}A_s = A_1(A_2(A_3\cdots(A_{s-1}A_s))).$$



- 矩阵连乘无论按照什么样的乘积顺序，最后计算出来的矩阵虽然“样子”不一样，但是计算出来的值都是一样的。

# 矩阵连乘问题

- 矩阵连乘的计算次数与计算顺序的关系
- 假设有一个p*q规模的矩阵A，一个q*r规模的矩阵B，再有一个规模为r*s的矩阵C，那么这三个矩阵的乘积ABC，有两种计算顺序：(AB)C 或 A(BC)

$$mult[(AB)C] = pqr + prs$$

$$mult[A(BC)] = qrs + pqs$$

- 假设p=5, q=4, r=6并且s=2，则mult[(AB)C] = 180，mult[A(BC)] = 88
- 计算结果完全一样，但计算的次数不同！！

$$\begin{aligned} A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) \\ &= A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\ &= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4) \end{aligned}$$

# 矩阵连乘问题

- **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。

# 矩阵连乘问题

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]

# 矩阵连乘问题

- 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

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

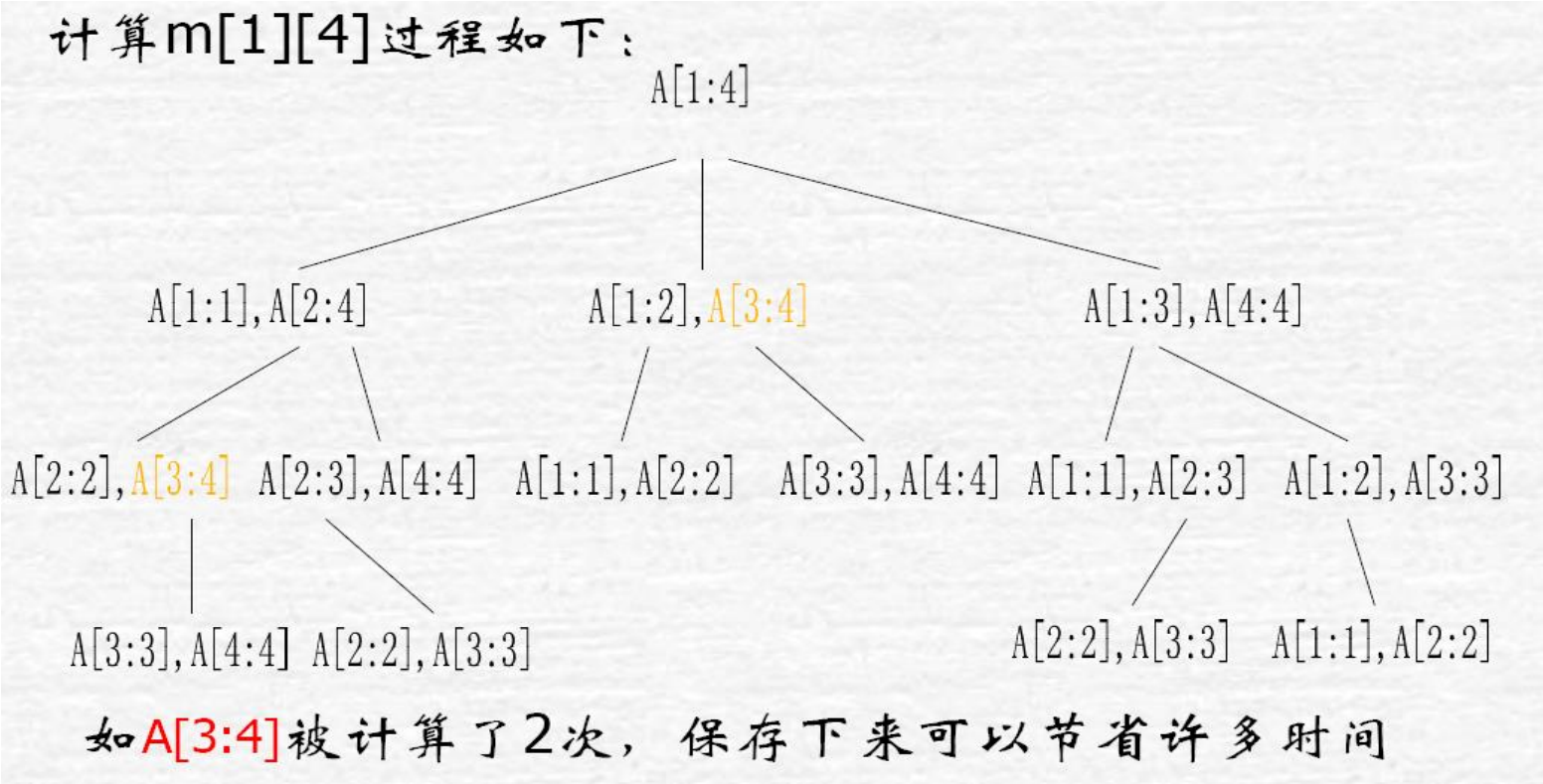
这里，

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

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。

# 矩阵连乘问题

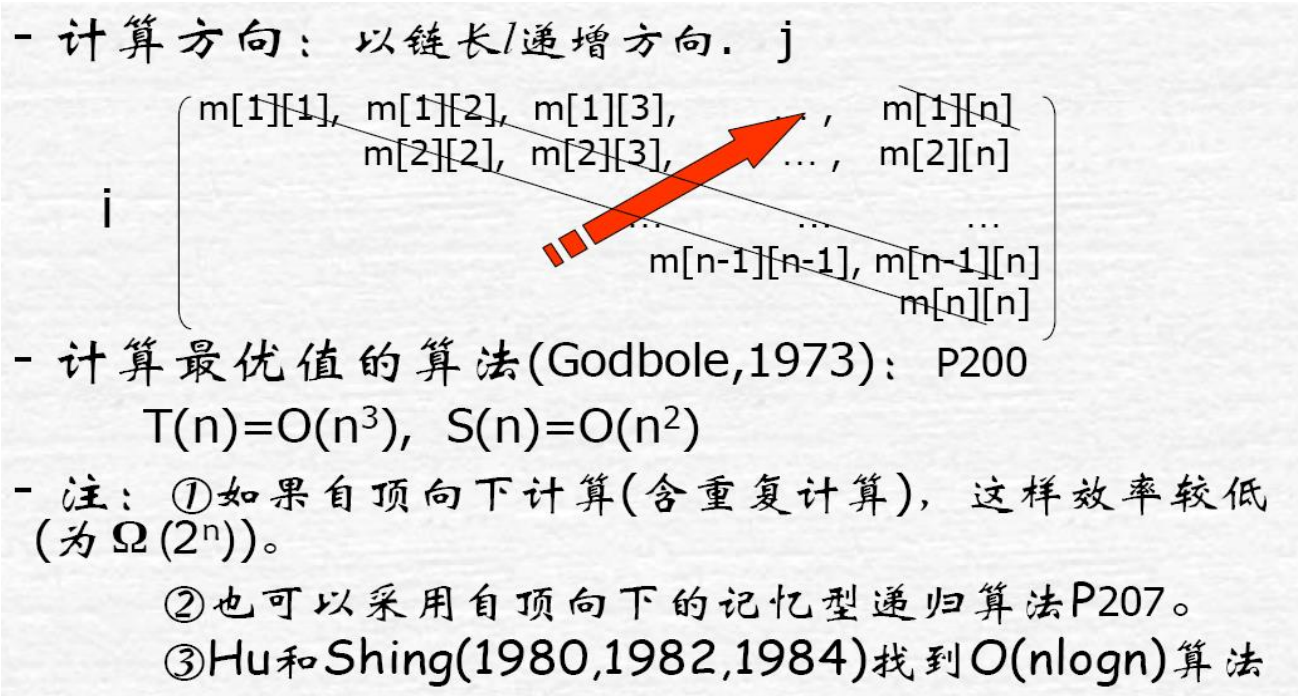



- 在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

# 矩阵连乘问题

- 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。



# 矩阵连乘问题

- 步骤3：计算最优代价，自底向上记忆化方式求解m[i][j]

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j)$$

$$\begin{aligned} m[1,3] &= \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3) \\ &= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\} \\ &= 88. \end{aligned}$$

# 矩阵连乘问题

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

# 矩阵连乘问题

## 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 ")";
   }
}
```

# 矩阵连乘问题

- 计算示例：

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

| S | j=1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| i=1 | | 1 | 1 | 3 | 3 | 3 |
| 2 | | | 2 | 3 | 3 | 3 |
| 3 | | | | 3 | 3 | 3 |
| 4 | | | | | 4 | 5 |
| 5 | | | | | | 5 |
| 6 | | | | | | |

| m | j=1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| i=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**

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

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

# 适用条件

- 适合采用动态规划方法的最优化问题中的两要素：
  - ✓ 最优子结构
  - ✓ 重叠子问题

## 一、 最优子结构

- 如果问题的最优解是由其子问题的最优解来构造的，则称该问题具有最优子结构。
- 在动态规划中，我们利用子问题的最优解来构造问题的一个最优解，因此必须确保在我们所考虑的子问题范围中，包含了用于一个最优解的那些子问题。

# 最优子结构

- **寻找最优子结构的模式：**
  - 问题的一个解可以是做一个选择(e.g.：选择一个下标以在该位置分裂矩阵链)
  - 假设一个给定的问题，已知一个可以导致最优解的选择(不必关心如何确定该选择，只需假定它是已知的)
  - 在已知该选择后，要确定哪些子问题会随之发生，以及如何最好地描述所得到的子问题空间
  - 利用剪贴(cut-and-paste)技术，来证明在问题的一个最优解中，使用的子问题的解本身也必须是最优的。
    - 方法：假设每一个子问题的解都不是最优解,导出矛盾即可。
    - 通过“剪除”非最优的子问题再“贴上”最优解，即得到原问题的一个更好的解，从而与假设已得到一个最优解矛盾。

# 最优子结构

- **子问题空间的描述：**
  - 遵循规则：尽量保持这个空间简单，需要时再扩充它
  - 最优子结构在问题域中的变化方式：
    - 有多少个子问题在原问题的一个最优解中被使用
    - 在决定一个最优解中使用哪些子问题时有多少个选择。

**例如：** 子链$A_i A_{i+1} \cdots A_j$的矩阵链乘法有2个子问题，$j$-$i$个选择

  - 一个动态规划算法的运行时间依赖于两个因素的乘积：子问题的总个数和每个子问题中有多少种选择

例如：矩阵链乘法，有$\Theta(n^2)$个子问题，每个子问题中又至多有$n$-$1$个选择，因此执行时间为$\Theta(n^3)$

# 最优子结构

- 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。
- 寻找问题的一个最优解需要在子问题中做出选择，即选择将用哪一个来求解问题
- 问题解的代价 = 子问题的代价 + 选择带来的开销
- 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)
- 注：贪心算法适用的问题也具有最优子结构，但它是以自顶向下的方式使用最优子结构；先做选择再求解一个结果子问题。

# 最优子结构

- 注意：在不能应用最优子结构的时候，一定不能假设它能够应用
- 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。

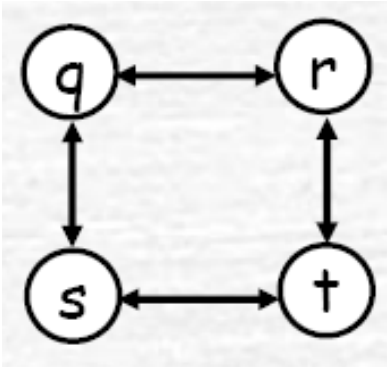
例1：设G是一个有向加权图，则G从顶点i到顶点j之间的最短路径问题满足最优性原理。

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。

# 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。

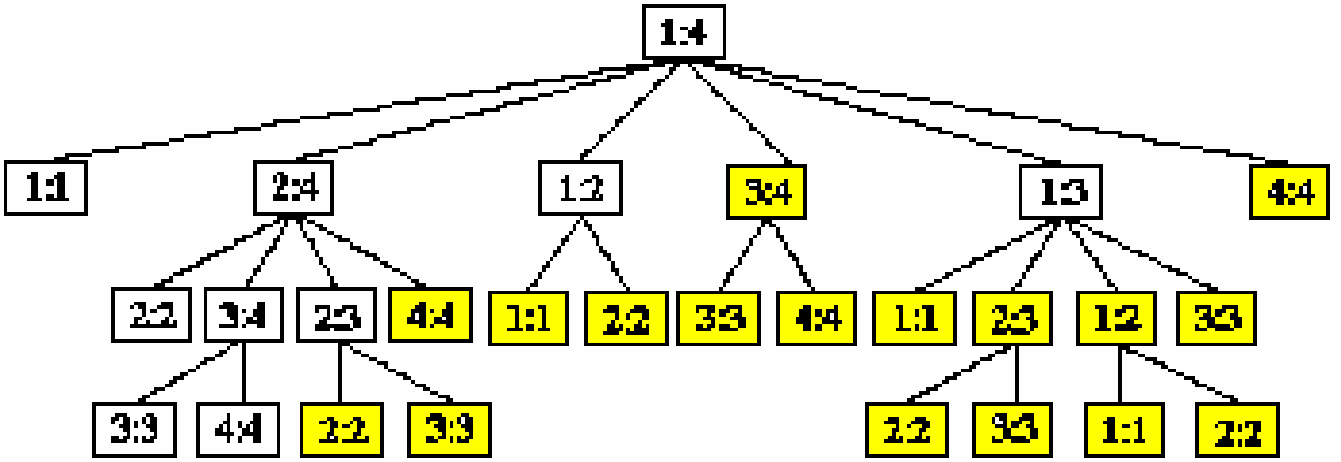
**所以，原问题并不满足最优性原理。**

**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。**

# 重叠子问题

## 二、重叠子问题

- 适用于动态规划求解的最优化问题的第二个要素是子问题的空间要小，使用来解原问题的递归算法可反复解同样的子问题，而不总在产生新的子问题。
- 不同的子问题数是输入规模的一个多项式。
- 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，该性质称为子问题的重叠性质

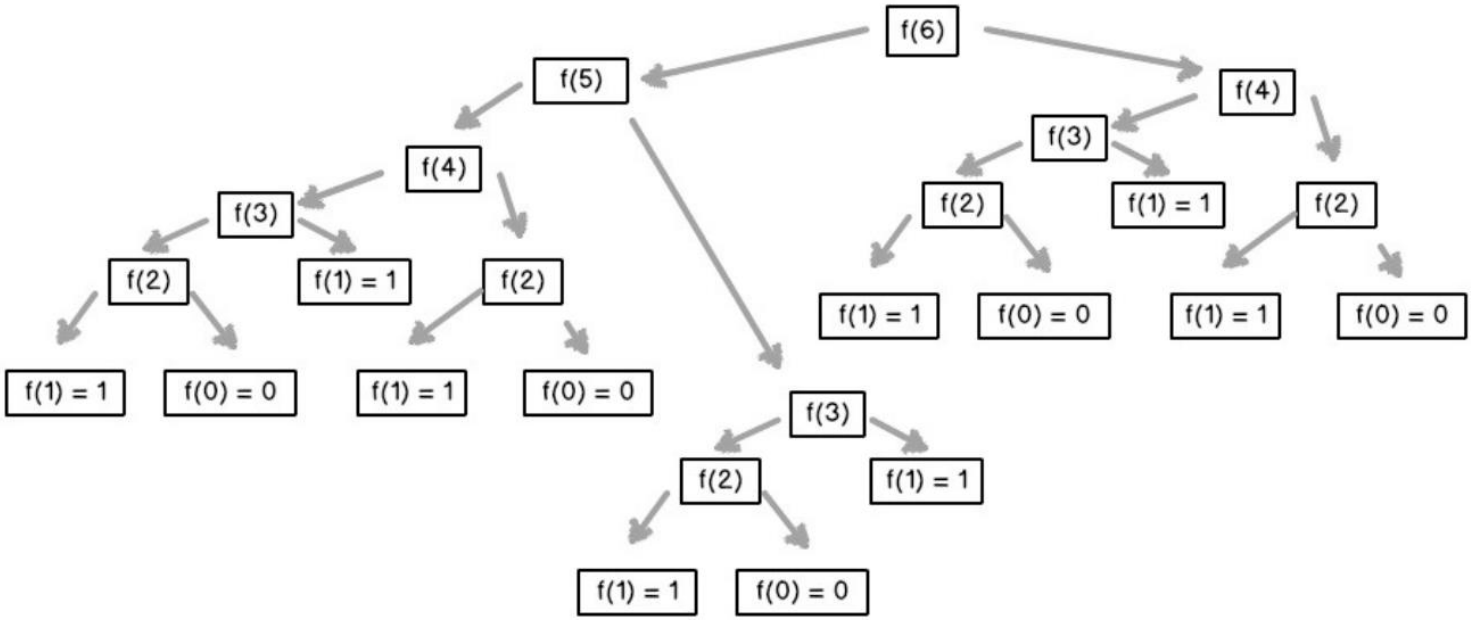


# 重叠子问题

- 斐波那契数列Fibonacci array 1,1,2,3,5,8,13,21,34，...
- 递归方法：

```
int fib(int n) {
   if (n < 2) return n;
   return fib(n-1) + fib(n-2);
}
```

时间复杂度：**递归算法是 O(n!)**，呈指数级增长，而采用动态规划思想的算法只有 **O(n)**，其空间复杂度为 O(n)。



# 重叠子问题

- 动态规划算法，充分利用重叠子问题，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。
- 通常不同的子问题其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。

# 做备忘录

- 动态规划的一种变形，既具有通常的动态规划方法的效率，又采用了一种自顶向下的策略

- 思想：备忘(memorize)原问题的简单直接但低效的递归算法，维护一个记录了子问题解的表，但有关填表动作的控制结构更像递归算法。
  - ✓ 方法：加了备忘的递归算法为每一个子问题的解在表中记录一个表项。
    - 每个表项最初包含一个特殊的值，表示该表项有待填入；
    - 在递归算法执行中第一次遇到一个子问题，计算它的解并填入表中；
    - 以后再遇到该子问题，只要查看并返回表中先前填入的值即可。

- 自顶向下的做备忘录算法和自底向上的动态规划算法都利用重叠子问题性质
  - ✓ 如果所有子问题都至少要被计算一次，则后者比前者多出一个常数因子，因为后者无需递归的代价，维护表格的开销也小些。
  - ✓ 有些问题可以用动态规划算法的表存取模式来进一步减少时间或空间上的需求。
  - ✓ 如果某些子问题没有必要求解，做备忘录方法具有只需要求解那些肯定要求解的子问题的优点。

# 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。

# 动态规划法的总结

与非线性规划相比，动态规划的优点：

（1）易于确定全局最优解。动态规划方法是一种逐步改善法，它把原问题化为一系列结构相似的最优化子问题，而每个子问题的变量个数比原问题少的多，约束集合也要简单得多。

（2）能得到一簇解，有利于分析结果

（3）能利用经验，提高求解的效率。动态规划方法反映了过程逐段演变的前后联系，较之非线性规划与实际过程联系得更紧密。

不足之处：

（1）没有一个统一的标准模型可供应用。

（2）应用的局限性。要求状态变量满足“无后效性”条件，不少实际问题在取其自然特征作为状态变量往往不能满足这条件。

（3）在数值求解中，存在“维数障碍”。在计算机中，每递推一段，必须把前一段的最优值函数在相应的状态集合上的全部值存入内存中。当维数增大时，所需的内存量成指数倍增长。

# 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**。

# 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

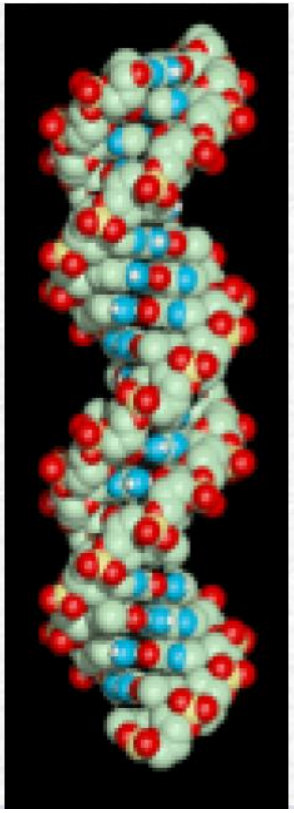
  $S_1$ = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

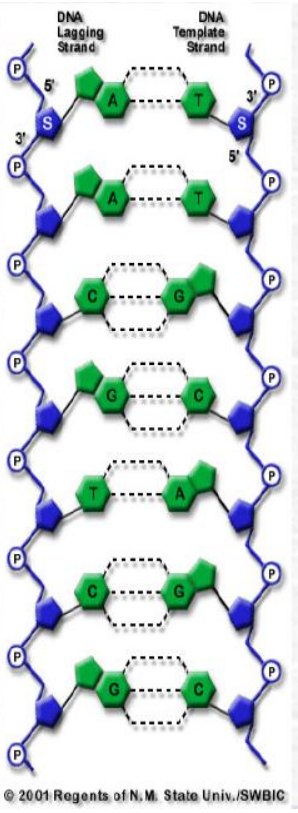
  and

  $S_2$ = 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.





# 最长公共子序列（LCS）

- Step 1: LCS最优解的结构特征

  定义X的第*i*个前缀：$X_i = (x_1, x_2, \cdots, x_i), i = 1 \sim m$

  $X_0 = \phi$， $\phi$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。

# 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

# 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。 □

# 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

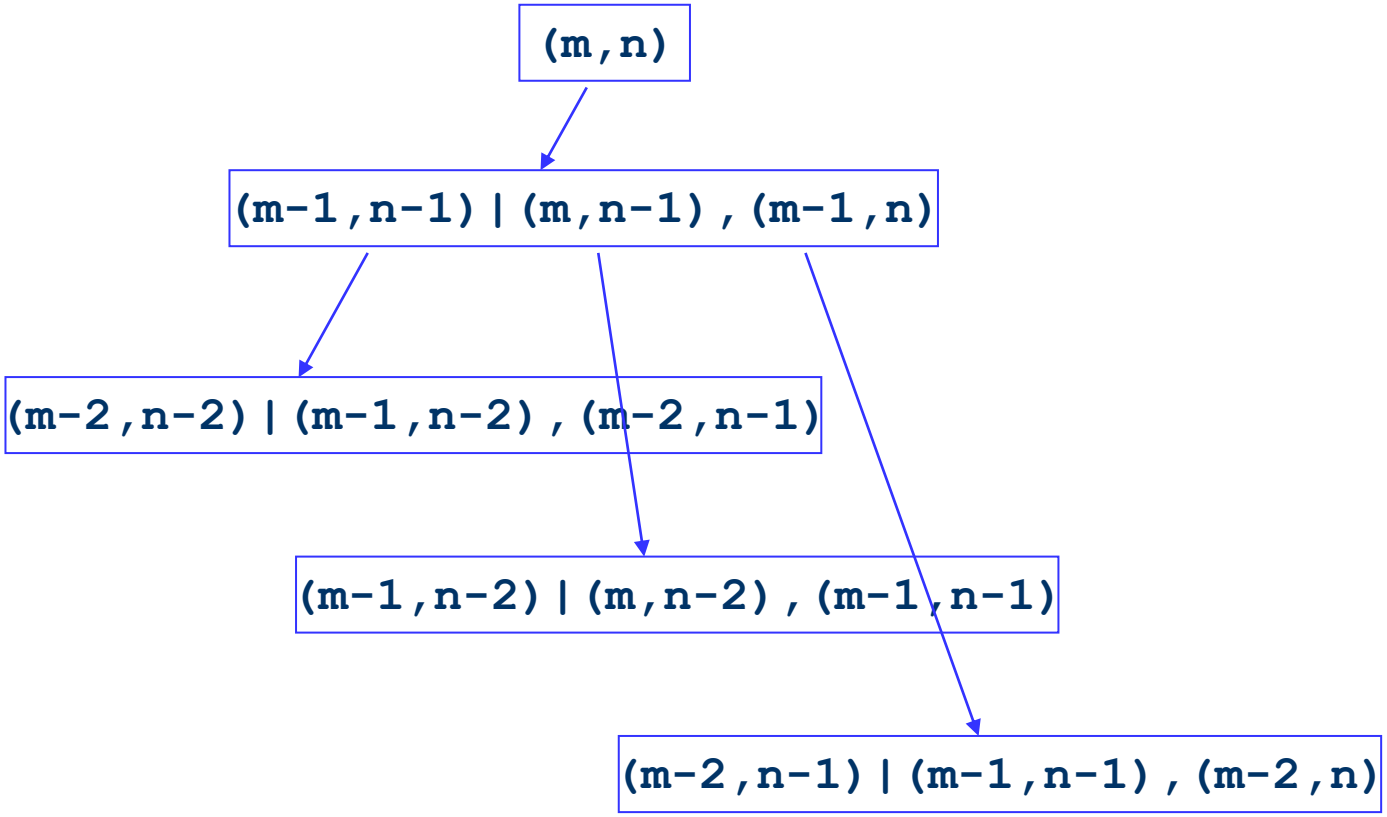
找$X_{m-1}$和Y的LCS；找X和$Y_{n-1}$的LCS；

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 ~ m, j = 0 ~ n；

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

# 最长公共子序列（LCS）



# 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)

# 最长公共子序列（LCS）

一算法

```
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 xi=yj 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Xi-1和Yj确定
                else
                {  c[i, j]←c[i, j-1]; b[i, j] ← "←"; }   //由Xi和Yj-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。

# 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)

# 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

# 最长公共子序列（LCS）

改进代码：

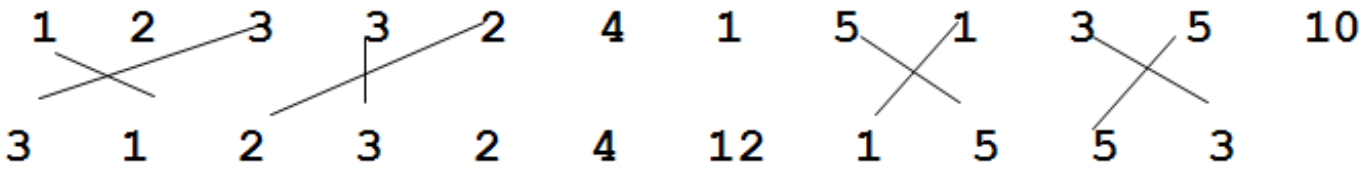
- 在算法lcsLength和LCS中，可进一步将数组b省去。事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。对于给定的数组元素c[i][j]，可以不借助于数组b而仅借助于c本身在常数时间内确定c[i][j]的值是由c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的。
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。

# 最长公共子序列（LCS）

**思考题：**

**□ 交错匹配（最长公共子串的改编）：**

给定两排数字，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？



# 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)；

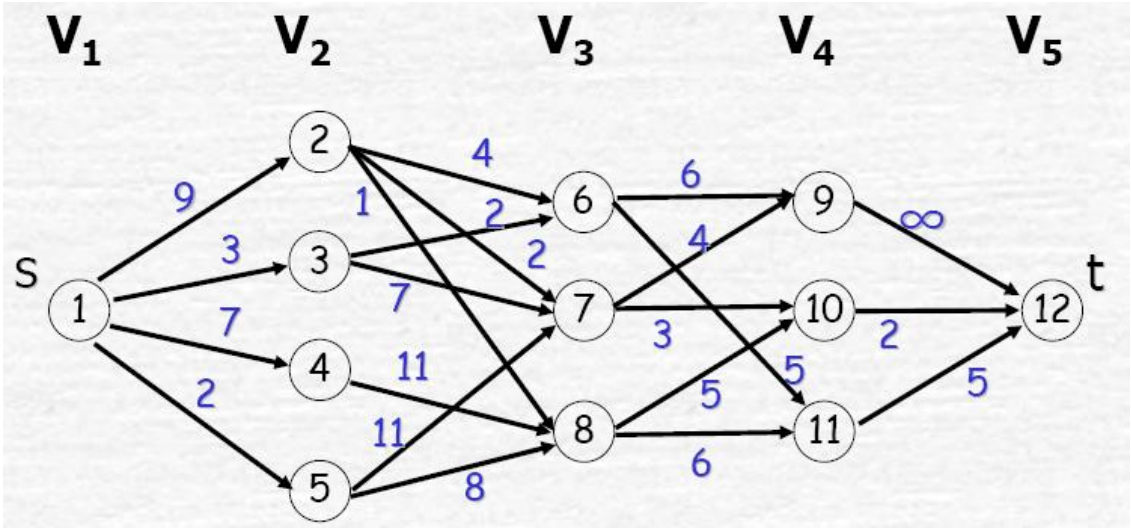
(3) 若<u, v> ∈E(G)，u∈$V_i$，则v∈$V_{i+1}$,边上成本记c(u,v)；

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。

# 多段图规划

- 举例：一个5-段图




求从s到t的最短路径。

# 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)

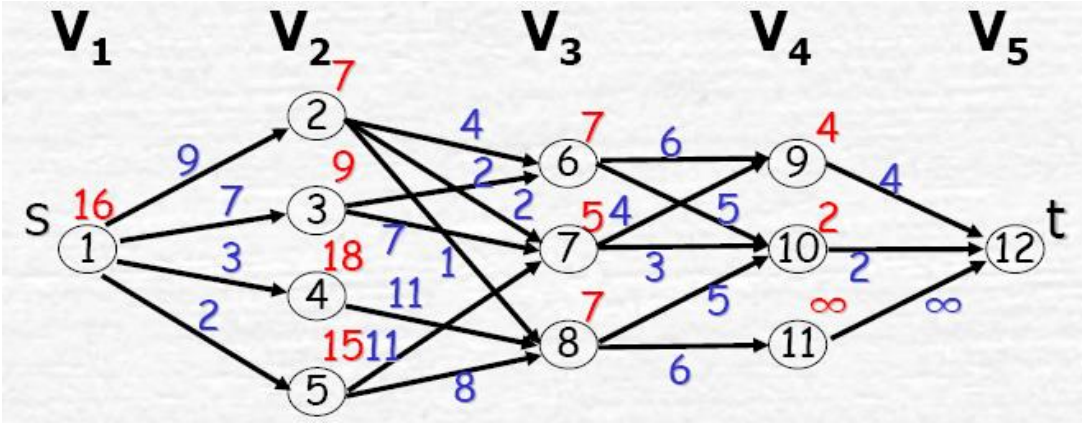
## 递归式推导

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：

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

# 多段图规划

- 计算过程(以5-段图为例)




$V_4$时, cost(4, 9)=4, cost(4, 10)=2, cost(4, 11)=∞

$V_3$时, cost(3, 6)=7, cost(3, 7)=5, cost(3, 8)=7

$V_2$时, cost(2, 2)=7, cost(2, 3)=9, cost(2, 4)=18, cost(2, 5)=15

$V_1$时, 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)

# 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)
```

# 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：

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

# 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$O(n^2)$；

# 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。

# 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
```

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

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

# 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

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

# 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？

# 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。

# 0-1背包问题

- 问题描述：给定n种物品和一个背包。物品i的体积/权重是$w_i$，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。

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

- 根据动态规划解题步骤(问题抽象化、建立模型、寻找约束条件、判断是否满足最优性原理、找大问题与小问题的递推关系式、填表、寻找解组成)，找出0-1背包问题的最优解以及解组成，然后编写代码实现。

# 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=($v_1$, $v_2$, $v_3$)=(1, 2, 5), 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(n$2^n$)

# 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。

# 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。

# 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$x_i=0$，$\therefore m(i,j)=m(i+1,j)$；

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

- 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。

# 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];
}
```

# 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;
}
```

# 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会被利用，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。

# 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);
}
```

# 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，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求解呢？

# 谢谢！

## Q & A