

算法设计与分析

3. 动态规划 (Part 1)

学习要点

- 理解动态规划算法的概念与基本思想。
- 掌握动态规划算法的两个基本要素
 - (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方法
 - ❖ 求解步骤-4步
 - ❖ 自上而下自下而上求解方式；备忘录（打表）的应用
- 动态规划典型问题
 - ❖ 矩阵连乘、最长公共子序列、0-1背包问题、最大子段和、流水作业调度（Johnson调度法则）等等

引言 几个典型问题

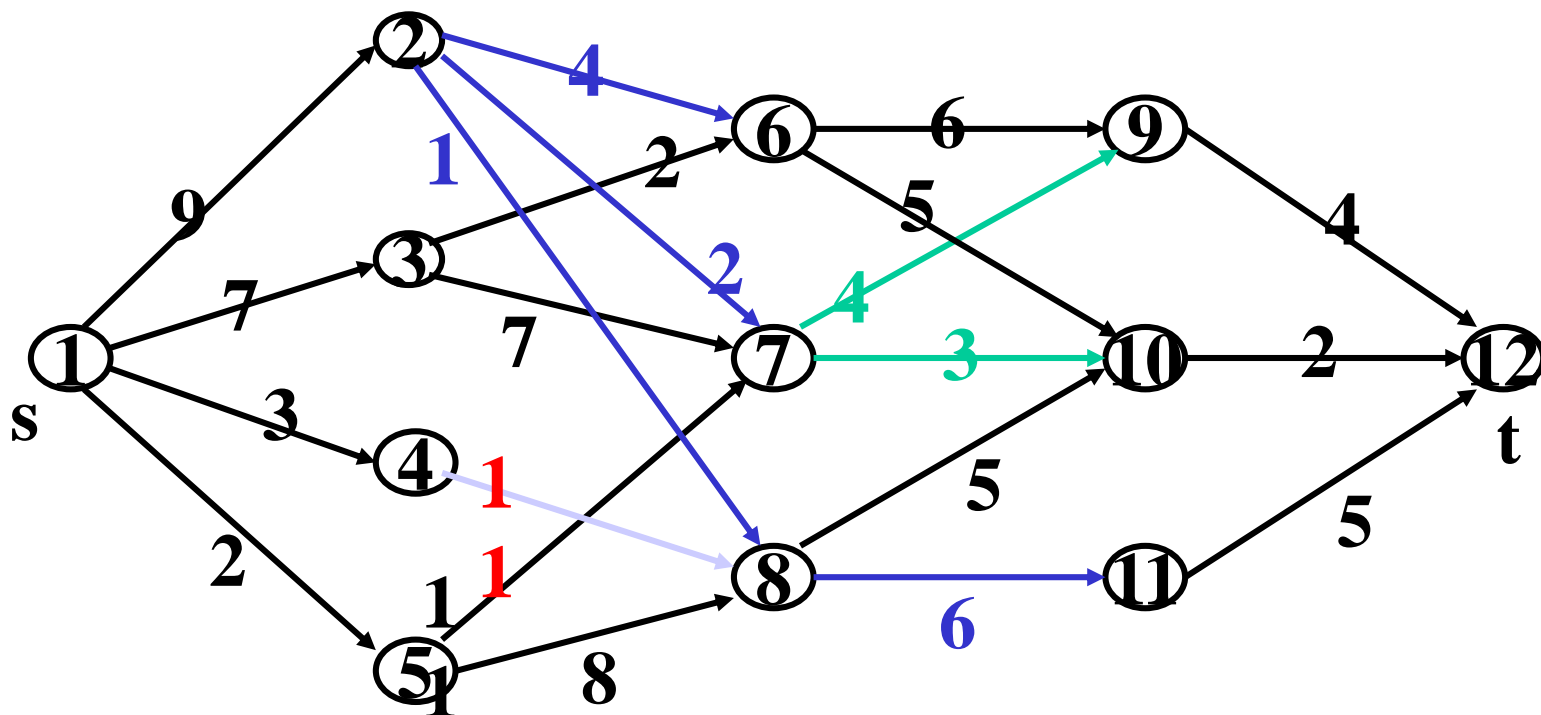
问题1：最短路径

- 有一个棋盘形街道，某人从西南角O走到东北角U，想选择一条路线使所走的路径最短，如何走法？
总路径数 = $C_7^3 = 35$

F	2	K	3	P	1	S	2	U
	2		1		2		2	3
C	2	G	3	L	4	Q	5	T
	3		4		1		2	3
A	2	D	2	H	1	M	4	R
	2		2		1		3	4
O	1	B	3	E	2	I	3	N

问题2：最短路径

- 对有向无环图，求从s到t的一条最短路径



问题3：背包问题

- 有一旅行者要从 n 种物品中选取体积不超过 C 的行李随身携带，要求包装得最满。
- 例：设有 $n=8$ 个体积分别为54，45，43，29，23，21，14，1的物体和一个容积为 $C=110$ 的背包，问选择哪几个物体装入背包可以使其装的最满。

问题4：最长公共子序列问题

给定两个序列 $x=\langle x_1, x_2, x_3, \dots, x_n \rangle$ 和
 $y=\langle y_1, y_2, \dots, y_m \rangle$, 要求找出 x 和 y 的一个最长
公共子序列

3.1 矩阵连乘问题

3.1 矩阵连乘问题

一、问题叙述

- 给定 n 个矩阵 A_1, A_2, \dots, A_n , 其中, A_i 与 A_{j+1} 是可乘的, $i=1, 2, \dots, n-1$, 现要计算出这 n 个矩阵的连乘积 $A_1A_2\dots A_n$ 。
- **矩阵连乘问题:** 确定一种运算次序, 使总的运算次数达到最少。

矩阵乘法

- $A=(a_{ij})_{m \times k}$, $B=(b_{ij})_{k \times n}$, $C=(c_{ij})_{m \times n}$

$$c_{ij} = \sum_{t=1}^k a_{it}b_{tj}$$

- 计算C：每个元素需k次乘法，k-1次加法
- C有mn个元素：计算C，需mnk次乘法，mn(k-1)次加法
- 举例：设 A_1 , A_2 , A_3 ，分别为 10×100 , 100×5 , 5×50 的矩阵，求连乘积 $A_1A_2A_3$ 时需多少次乘法运算？

矩阵连乘方法数与添括号方式数一一对应

- 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) 若矩阵连乘积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 $A=(BC)$

矩阵连乘积与加括号

- ◆ 设有四个矩阵 **A,B,C,D**，它们的维数分别是：

$$A = 50 \times 10 \quad B = 10 \times 40 \quad C = 40 \times 30 \quad D = 30 \times 5$$

- ◆ 总共有五种完全加括号的方式

$$\begin{array}{lll} (A((BC)D)) & (A(B(CD))) & ((AB)(CD)) \\ (((AB)C)D) & ((A(BC))D) & \end{array}$$

乘法运算数 **16000, 10500, 36000, 87500, 34500**

二、算法分析--穷举法行吗？

列举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

- 所有可能的计算次序有多少？

设矩阵序列的加括号方式为 $b(n)$ ，可得：

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

$b(n)$ 是Catalan数 $\rightarrow b(N)=\Omega(4^n/n^{3/2})$

二、算法分析--新思路

采用一个新方法

(1) 分析最优解的结构

记 $A[i: j]$ 为 $A_i A_{i+1} \dots A_j$ ，记 $m[i][j]$ 是计算 $A_i A_{i+1} \dots A_j$ 时的最少乘法次数

显然， $A[i: i] = A_i$ ， $m[i][i] = 0$

问题变为：计算 $A[1: n]$ 、求 $m[1][n]$

二、算法分析--新思路

(2) 建立递归关系

假定计算 $A[1:n]$ 的一个最优次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开, $1 \leq k < n$

$$m[1][n] = m[1][k] + m[k+1][n] + p_0 p_k p_n$$

- 一般情况

假定计算 $A[i:j]$ 的一个最优次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开, $i \leq k < j$

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

分析最优解的特征和结构

- 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这一算法求解的显著特征。

矩阵链问题的一般递推关系

$$m[i][j] = \begin{cases} 0 & i=j \\ \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1}p_kp_j\} & i \neq j \end{cases}$$

(3) 计算最优值

$$m[i][j] = \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1}p_kp_j\} \quad i < j$$

对于 $1 \leq i \leq j \leq n$ 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

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

1、简单递归

2、动态规划的求解方法

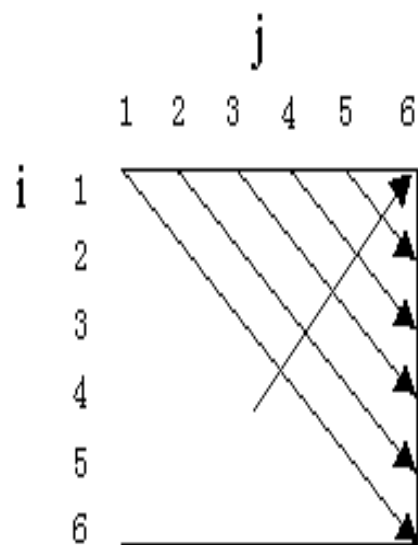
按自底向上方式求解

计算最优值程序

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

算法过程

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



(a) 计算次序

	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

(b) $m[i][j]$

	j	1	2	3	4	5	6
i	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 $O(n^2)$ 。

(4) 算法分析--构造最优解

引入分割点标记 $s[i][j]$ ，确定加括号方式，构造最优解

```
void Traceback(int i, int j, int **s){
    if (i == j) return;
    Traceback(i, s[i][j], s);
    Traceback(s[i][j] + 1, j, s);
    cout << "Multiply A " << i << ", " << s[i][j];
        cout << " and A " << (s[i][j] + 1) << " , " << j
            << endl
}
```

3.2 动态规划算法的基本要素

动态规划的基本思想

- 算法目标：求解具有某种最优性质的问题。它可能有许多可行解，希望找到具有最优值的解。
- 算法思想：
 1. 动态规划算法将待求解问题分解成若干个子问题，先求解子问题，
 2. 从这些子问题的解得到原问题的解。这些子问题往往不互相独立。
 3. 分解时得到的子问题数目可能很多，有些子问题被重复计算了很多次。

求解方法

- 自底向上方式、自上而下方式
- 采用备忘录方法：求解过程中需保存已解决的子问题的解，而在需要时再找出已求得的解，就可避免大量的重复计算，节省时间。动态规划法用表记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。

动态规划中的概念、名词术语(1)

- 1.阶段：** 把问题分成几个相互联系的有顺序的几个环节，这些环节即称为阶段。
- 2.状态：** 某一阶段的出发位置称为状态。通常一个阶段包含若干状态。
- 3.决策：** 从某阶段的一个状态演变到下一个阶段某状态的选择。
特点：前一阶段的终点是后一阶段的起点，前一阶段的决策影响后一阶段的状态。
- 4.策略：** 由开始到终点的全过程中，由每段决策组成的决策序列。

动态规划中的概念、名词术语(2)

- 5. **状态转移方程**：描述由 k 阶段到 $k+1$ 阶段状态的演变规律称为状态转移方程（用数学形式表达）。
- 6. **目标函数与最优化概念**：目标函数是衡量多阶段决策过程优劣的准则。最优化概念是在一定条件下找到一个途径，经过按题目具体性质所确定的运算以后，使全过程的总效益达到最优。
- 7. **动态规划**：在多阶段决策问题中，各阶段采取的决策依赖于目前状态，并引起状态的转移，以期求得最优化的过程

最佳原理

- 一个最优化策略的子策略总是最优的。

动态规划的求解步骤

- 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（写出动态规划方程）
- 3、以自底向上（或自顶向下）的方式计算出最优值
- 4、根据计算最优值时得到的信息，构造一个最优解

注意： 1、步骤1-3：是动态规划算法的基本步骤。

2、如只求最优值，步骤4可以省略；

3、若需求出问题的一个最优解，则必须执行步骤4，步骤3中记录的信息必须足够多以便构造最优解。

动态规划的求解方法

按自底向上方式求解

- 是动态规划方法的一种
- 所有子问题计算一次，无需递归代价
- 效率较高

程序 (next page)

```

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;
                }
            }
        }
    }
}

```

备忘录方法

动态规划算法的基本要素

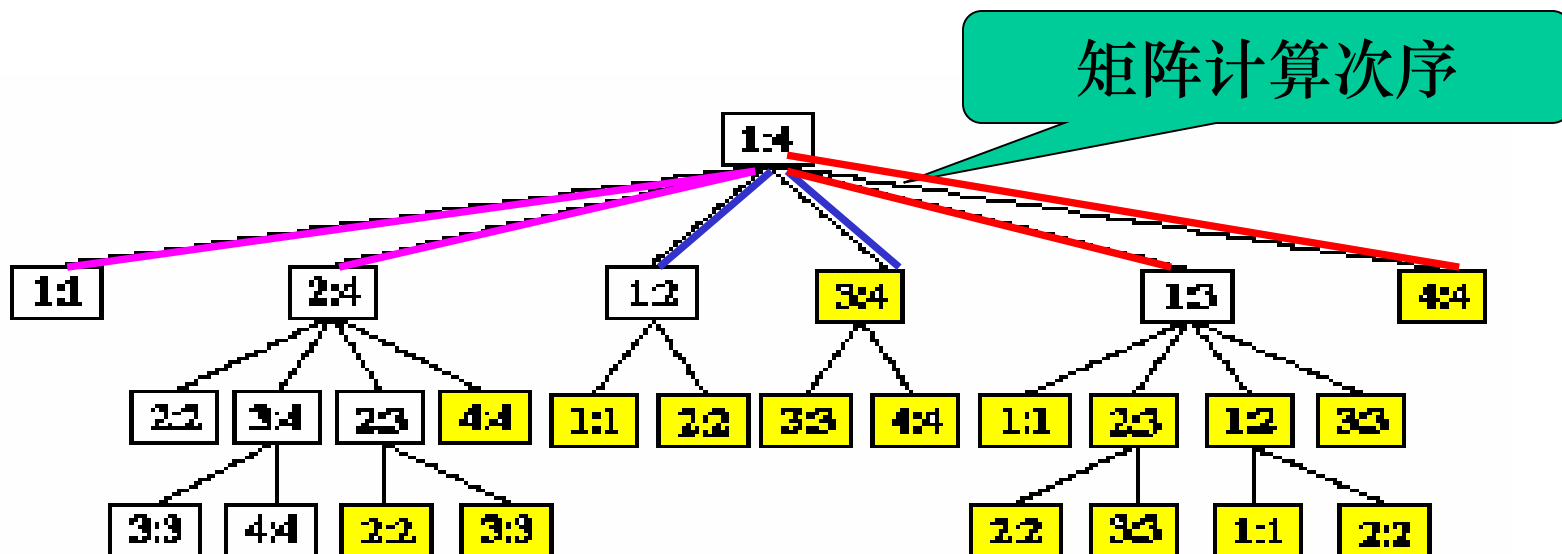
- 动态规划算法的有效性依赖于问题本身所具有的两个重要性质：最优子结构性性质和子问题重叠性质。
- 1. 最优子结构：当问题的最优解包含了其子问题的最优解时，称该问题具有最优子结构性性质。

如何说明问题具有最优子结构性性质？

用反证法：先假设由问题的最优解导出的子问题的解不是最优的，然后再设法证明在这个假设下可构造出一个比原问题最优解更好的解，从而导致矛盾。

动态规划算法的基本要素

2. **重叠子问题**：在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为子问题的重叠性质。



重叠子结构性质

动态规划算法利用子问题的重叠性质，对每一个子问题只解一次，并将解保存在一个表格中，在以后尽可能多地利用这些子问题的解。

特征：不同的子问题个数随问题的大小呈多项式增长，而不是指数增长。

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。

动态规划法与分治策略

共性：都通过子问题求解原问题

方法：分治法是把一个规模为 n 的问题分成多个与原问题类型相同的较小的子问题，通过对子问题的求解，并把子问题的解合并起来，构造出整个问题的解；

动态规划法也先求子问题的解，通过求解子问题，构造原问题的解。

动态规划法与分治策略（续）

差异：

1. **独立性**：分治法各子问题互相独立，动态规划法的各子问题可不独立
2. **子问题数目**：动态规划法中涉及的子问题，不独立的有很多，而独立的应只有**多项式级**；分治法涉及的子问题数一般达**指数级**
3. 动态规划法把问题分成许多子问题，每个子问题的解都是**局部最优**；分治法未必考虑最优性
4. 动态规划法可采用**备忘录方法**。

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。

```
int t,k, m=0;
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];
//记初值
    s[i][j] = i;
    for (k = i+1; k < j;
k++)
```

```
    {
        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;
}
```

动态规划小结

- 特征：最优子结构性质和子问题重叠性质。
- 最佳原理
- 动态规划法与分治法的联系与区别
- 动态规划法的使用方法

上机题

上机实验2:

修改3.1节的算法，找出计算 n 个矩阵之积的最少乘法数，并完成添括号过程。

动态规划解题举例1

1、单侧跳马问题

给定 $m*n$ 棋盘，求棋盘上一只马从左上角 $(1, 1)$ 到达右下角位置的最短路径长。

注意：在本问题中马只能向右侧、向下走“日”字形的，但马不能向所在位置的左边、向上走“日”字形。

输入

2个整数 m 、 n ， $(1 \leq m, n \leq 19)$ 之间用一个空格隔开，表示棋盘大小。

输出

输出马从棋盘左上角跳到右下角所需的最短路径长，如果不能跳到，那么直接输出“Impossible”。

分析

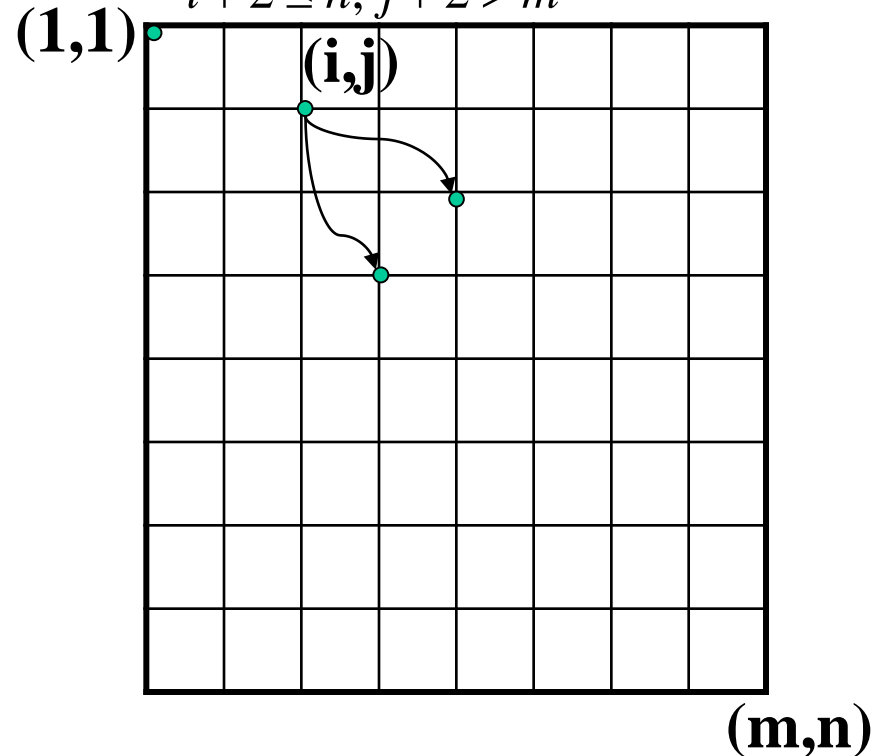
- 设 $f(i,j)$ 为从 (i,j) 到 (m,n) 的最短距离。从 (i,j) 开始可走2个位置 $(i+1,j+2)$ 与 $(i+2,j+1)$ 。

$$f(i, j) = \begin{cases} 1 + \min\{f(i+1, j+2), f(i+2, j+1)\} & i+2 \leq n, j+2 \leq m \\ 1 + f(i+1, j+2) & i+2 > n, j+2 \leq m \\ 1 + f(i+2, j+1) & i+2 \leq n, j+2 > m \end{cases}$$

初值:

$$f(m, n) = 0$$

$$f(i, j) = +\infty, \quad i \neq n, j \neq n$$



动态规划解题举例2

- 复制书稿
- 以前人们靠抄写员复制书籍。一次，有个剧院要排演一场戏剧,希望将剧本复制一份。已知剧本由 m 册组成，每册有 p_i 页($1 \leq i \leq m$)。雇佣了 k 个抄写员($1 \leq k \leq m$)。每个抄写员只能被分配一个任务，即抄写连续的若干册书稿。假定每个抄写员的抄写速度一样。该任务完成所需的总时间由工作量最大的抄写员决定，即该抄写员要抄写的总书页码。
- 求一种分配方案，使得完成任务所需的总时间最少。

输入

整数 m 及 k 。接着输入 m 个整数 p_1 、 p_2 、 p_3 、...、 p_m ，之间用一个空格隔开。

输出

输出完成任务所需的最少总时间。

分析

- 设 $F[i][j]$ 为前 i 个抄写员复制前 j 本书的最小完成时间数。考察第 i 人的抄写情况。设前面 t 本书由前 $i-1$ 人抄写，而从第 $t+1, t+2, \dots, j$ 本书由第 i 人抄写，于是便有状态转移方程：

$$F[i][j] = \min_{i-1 \leq t < j} \{ \max \{ F[i-1][t], p_{t+1} + p_{t+2} + \dots + p_j \} \}$$

其中， $1 \leq i \leq k, i \leq j \leq m$ 。

初始时， $F[1][1]=p_1$ 。又当 $m \leq k$ 时， $F[k][m] = \max_{1 \leq t \leq m} \{ p_t \}$

3.3 最长公共子序列

概念

- 1、子序列：**若给定序列 $X=\{x_1, x_2, \dots, x_m\}$ ，则另一序列 $Z=\{z_1, z_2, \dots, z_k\}$ 是 X 的子序列，是指存在一个严格递增下标序列 $\{i_1, i_2, \dots, i_k\}$ 使得对于所有 $j=1, 2, \dots, k$ 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 $\{2, 3, 5, 7\}$ 。

2、公共子序列

- 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

最长公共子序问题

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

一、算法分析

- 用穷举法：X有 2^m 个子集， $m=|X|$ ，逐一验证X的每个子集是否合要求，找出Z需指数时间
- 可否用动态规划法？

前提：问题是否具有

- 1、最优子结构性质；
- 2、子问题重叠性质？

最长公共子序列的结构

设序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的最长公共子序列为

$Z=\{z_1, z_2, \dots, z_k\}$ ，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。

结论

- 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。

二、子问题的递归结构

- 用 $c[i][j]$ 记录序列 X_i 和 Y_j 的最长公共子序列的长度。

其中, $X_i = \{x_1, x_2, \dots, x_i\}$; $Y_j = \{y_1, y_2, \dots, y_j\}$ 。

- 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:

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

- 为什么是这种形式?

三、计算最优值

- 子问题空间中，总共有 $\theta(mn)$ 个不同的子问题，用动态规划算法自底向上计算最优值能提高算法的效率。

```
lcsLength(x,y,b)
m=x.length-1; n=y.length-1;
c[i][0]=0; c[0][i]=0;
for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
        if (x[i]==y[j]) {
            c[i][j]=c[i-1][j-1]+1;
            b[i][j]='0';} //指↖
        else if (c[i-1][j]>=c[i][j-1])
            { c[i][j]=c[i-1][j];
              b[i][j]='1';} //指↑
        else
            { c[i][j]=c[i][j-1];
              b[i][j]='';} //指←
```

构造最长公共子序列

```
lcs(int i,int j,char
*x,int **b)
{
    if (i ==0 || j==0)
return;
    if (b[i][j]== '0' ){
        lcs(i-1,j-1,x,b);
        printf(x[i]);
    }
    else if (b[i][j]==
'1') lcs(i-1,j,x,b);
        else lcs(i,j-1,x,b);
}
```

求最长公共子序列问题的重叠子结构性性质很明显

举例

- 设 $X=\{A,B,C,B,D,A,B\}$ 和 $Y=\{B,D,C,A,B,A\}$, 找出 X 和 Y 的最长公共子序列 $LCS(X,Y)$ 。
- 先通过 $LCSLength$, 求 $C[i][j]$, $b[i][j]$
- 再调用 LCS , 求得 $LCS(X, Y) = \{B, C, B, A\}$

算法的改进

- 在算法lcsLength和lcs中，可进一步将数组b省去。

```
void lcsPrint(i,j,x) {  
    if (i==0)||(j==0) return;  
    if (c[i][j]==c[i-1][j-1]+1) {  
        lcsPrint(i-1,j-1,x) ; printf(x[i]);}  
    else if (c[i][j]>c[i-1][j]+1)  
        lcsPrint(i-1,j,x);  
    else    lcsPrint(i, j-1,x)  
}
```

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 $O(\min(m,n))$ 。

银币问题—补充

- **银币问题：**用天平秤银币，找出不合格的银币，且在最坏情况下秤银币的次数最少。
- 设有 n 个银币中有一个是不合格的，不合格的银币比合格银币要轻。
- 设 T_n 为最坏情况下秤 n 个银币的最少次数，需建立动态规划方程

补充：银币问题-续

- n块银币C1、C2、C3、...、Cn中有一块不合格，较重，用天平找出这块不合格的银币，问天平要称量多少次，如何称？
- 若从n中取出2k块，每边k块，在天平上称，设 T_n 为从n块中找出不合格银币的称量次数，显然， T_n 是n的单调增（不是严格单调）函数。

$$T_n = 1 + \min_{1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor} \{ \max \{ T_{n-2k}, T_k \} \}$$

3.4 最大子段和

心情愉快指数

- 问题描述：某人驾车旅行，一些道路使他心情很舒畅，如高速公路，但一些道路使他心情很沮丧，如乡村泥泞的道路。于是他对路过的这些给出心情愉快指数，并统计道路上若干路段的心情愉快指数之和，他想知道在道路上从哪个起点开始到哪个位置结束的路段上心情愉快指数最大。
- 您是计算机程序编程高手，请您帮助他解决这问题。

最大子段和问题

- 问题描述：给定由n个整数组成的序列 $a_1, a_2, a_3, \dots, a_n$ ，求该序列形如 $\sum a_k$ 的子段和的最大值。

注：当n个整数均为负数时，规定其最大子段和为0。即，所求最大子段和为：

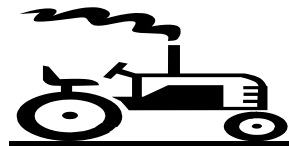
$$\max \left\{ 0, \max_{1 \leq i \leq j \leq n} \sum_{k=i}^j a_k \right\}$$

举例

- 设 $(a_1, a_2, a_3, a_4, a_5, a_6) = (-2, 11, -4, 13, -5, -2)$, 求该序列的子段和的最大值。

算法分析

- 简单算法
- 改进算法
- 分治策略
- 动态规划



简单算法--枚举

```
int MaxSum(int n, int *a, int &besti,
           int &bestj)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++){
            int thissum = 0;
            for (int k = i; k <= j; k++)
                thissum += a[k];
            if (thissum > sum) {
                sum = thissum;
                besti = i; bestj = j;
            }
        }
    return sum;
}
```

取2个位置i,j

复杂度 $O(n^3)$

返回 ■

改进算法—递推

- 利用 $\sum_{k=i}^j a_k = a_j + \sum_{k=i}^{j-1} a_k$

```
int MaxSum(int n, int *a, int &besti, int &bestj)
```

```
{    int sum = 0, thissum;
    for (int i = 1; i <= n; i++) {
        thissum = 0;
        for (int j = i; j <= n; j++) {
            thissum += a[j];
            if (thissum > sum)
                {    sum = thissum;    besti
= i; bestj = j;    }
        }
    }
    return sum;
}
```

复杂性 $O(n^2)$

返回 ■

分治策略

- 将序列 $a[1:n]$ 分为长度相等的两段 $a[1:n/2]$ 和 $a[n/2+1:n]$ ，分别求出这两段的最大子段和 S_1, S_2
- $S_1 = \max_{\{1 \leq i \leq j \leq n/2\}} \left\{ \sum_{k=i}^j a_k \right\}$
- $S_2 = \max_{\{n/2+1 \leq i \leq j \leq n\}} \left\{ \sum_{k=i}^j a_k \right\}$
- $a[1:n]$ 的最大子段和 S 有三种可能：
- $S = S_1$, $S = S_2$ 或 $S = \sum_{k=i}^j a_k$, $i \leq n/2$, $n/2+1 \leq j$

第三种情况的处理： $S = \sum_{k=i}^j a_k$

- $S'_1 = \max_{\{1 \leq i \leq n/2\}} \left\{ \sum_{k=i}^{n/2} a_k \right\}$
- $S'_2 = \max_{\{n/2+1 \leq j \leq n\}} \left\{ \sum_{k=n/2+1}^j a_k \right\}$

确定i需 $O(n)$ ，确定j需 $O(n)$

$$S = S'_1 + S'_2$$

- 时间复杂度递推式

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

解上述递推式，得： $T(n) = O(n \log n)$

返回 ■

动态规划方法

- 思想：对 $a[1:j]$ ，记 $b[j]$ ， $1 \leq j \leq n$

$$b[j] = \max_{\{1 \leq i \leq j\}} \left\{ \sum_{k=i}^j a_k \right\}$$

$$a[1:n] \text{ 的最大子段和 } S = \max_{\{1 \leq j \leq n\}} \max_{\{1 \leq i \leq j\}} \sum_{k=i}^j a_k$$

$$= \max_{\{1 \leq j \leq n\}} \{b[j]\}$$

$$\text{而 } b[j] = \max\{b[j-1] + a[j], a[j]\}$$

$b[j-1] > 0$ 时

动态规划-程序

```
int MaxSum(int n, int *a)
{
    int    s = 0, b = 0;
    for (int i = 1; i <= n; i ++ )
    {
        if (b > 0) b += a[i];
        else b = a[i];
        if (b > sum) sum = b;
    }
    return sum;
}
```

复杂性 $O(n)$

返回 ■

最大子段和的推广

- 2维--最大子矩阵和问题的求解算法
- 1维--最大 m 子段和问题的求解算法

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。

$$A = \begin{pmatrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{pmatrix}$$

算法

- 直接枚举：需4重循环，复杂性 $O(m^2n^2)$

- 改进：记 $S(i_1, i_2, j_1, j_2) = \sum_{i=i_1, j=j_1}^{i_2, j_2} a_{ij}$

$$\text{要求 } S = \max_{\{1 \leq i_1 \leq i_2 \leq m\}} \max_{\{1 \leq j_1 \leq j_2 \leq n\}} S(i_1, i_2, j_1, j_2)$$

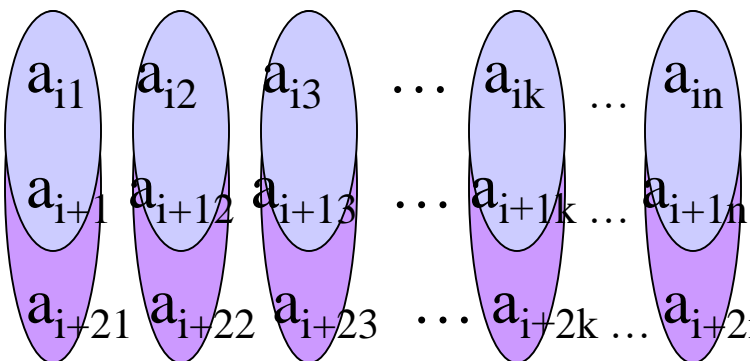
$$= \max_{\{1 \leq i_1 \leq i_2 \leq m\}} t(i_1, i_2)$$

其中 $t(i_1, i_2) = \max_{\{1 \leq j_1 \leq j_2 \leq n\}} S(i_1, i_2, j_1, j_2)$

$$= \max_{\{1 \leq j_1 \leq j_2 \leq n\}} \sum_{j=j_1}^{j_2} b_j$$

b_j 是j列上从 i_1 行到 i_2 行的元素之和，显然 $t(i_1, i_2)$ 是求 $b[1..n]$ 的最大子段和

算法图示

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1k} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2k} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{ik} & \dots & a_{in} \\ a_{i+11} & a_{i+12} & a_{i+13} & \dots & a_{i+1k} & \dots & a_{i+1n} \\ a_{i+21} & a_{i+22} & a_{i+23} & \dots & a_{i+2k} & \dots & a_{i+2n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mk} & \dots & a_{mn} \end{bmatrix}$$


The diagram illustrates a matrix A with rows indexed from 1 to m and columns indexed from 1 to n . The matrix is represented as a large bracketed array. The i -th row is highlighted with a blue oval, and the $i+1$ -th and $i+2$ -th rows are highlighted with purple ovals. The columns are labeled $a_{i1}, a_{i2}, a_{i3}, \dots, a_{ik}, \dots, a_{in}$ for the i -th row, and $a_{i+11}, a_{i+12}, a_{i+13}, \dots, a_{i+1k}, \dots, a_{i+1n}$ for the $i+1$ -th row, and $a_{i+21}, a_{i+22}, a_{i+23}, \dots, a_{i+2k}, \dots, a_{i+2n}$ for the $i+2$ -th row. The matrix is shown with ellipses indicating continuation of rows and columns.

复杂性 $O(m^2n)$

```

int MaxSum(int m, int n, int **a){
    int sum = 0, *b = new int [n + 1];
    for (int i = 1; i <= m; i++) {
        for (int k = 1; k <= n; k++) b[k] =
a[i][k];
        for (int j = i + 1; j <= m; j++) {
            for (int k = 1; k <= n; k++) b[k] +=
a[j][k];
            int max = MaxSum(j, n, b);
            if (max > sum) sum = max;
        }
    }
    return sum;
}

```

复杂性 $O(n)$

总复杂性 $O(m^2n)$