

数据结构与算法 (五)' 动态规划基础补充例题

杜育根

Ygdu@sei.ecnu.edu.cn

目录

- 钱币问题
- 0/1背包 滚动数组
- 最长公共子序列
- 最长递增子序列

最少钱币问题

- 有多个不同面值的钱币（任意面值）；
 - 数量不限；
 - 输入金额 S ，输出最少钱币组合。
-
- 回顾前面用贪心求解的钱币问题。贪心算法不能求出任意面值的最优解，要满足对任何一个面值钱币，要大于比它小的所有钱币面值之和。
 - 任意面值钱币问题的求解：**动态规划**。

动态规划思路

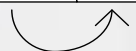
`int type[] = {1, 5, 10, 25, 50}; //5种面值钱币`

定义数组 `int Min[]` 记录最少钱币数量：对输入的某个金额 `i`, `Min[i]` 是**最少**的钱币数量。

- 把 `Min[]` 叫做 “状态”

第一步：只考虑1元面值的钱币


金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1								



- `i=1`元时，**等价于**：对 `i = 1`, `i-1 = 0`元需要的钱币数量，加上1个1元钱币。 `Min[1]=Min[0]+1`
- 把 `Min[]` 的变化叫做 “状态转移”

继续，所有金额仍然都只用1元钱币

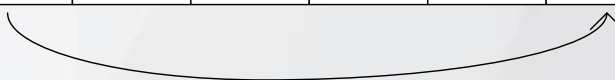
金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	5	6	7	8	...



- $i=2$ 元时，等价于： $i-1 = 1$ 元需要的钱币数量，加上1个1元钱币。
- $i=3$ 元时...
- $i=4$ 元时...
- 所以，只用一种1元钱币，最少钱币数量 $\text{Min}[i]=i$

- 在1元钱币的计算结果基础上，再考虑**加上5元**钱币的情况。从*i*=5开始就行了：

金额 <i>i</i> :	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	1	6	7	8	...



***i*=5元时**，等价于：

(1) $i-5 = 0$ 元需要的钱币数量，加上1个5元钱币。
 $\text{Min}[5]=1$ 。

(2) 原来的 $\text{Min}[5]=5$ 。

➤ 取 (1) (2) 的最小值，所以 $\text{Min}[5]=1$ 。

金额 <i>i</i> :	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	1	2	3	4	...

- ***i*=6元时**，等价于：

(1) $i - 5 = 1$ 元需要的钱币数量，加上1个5元钱币。
 $\text{Min}[6] = 2$.

(2) 原来的 $\text{Min}[6] = 6$

➤ 取 (1) (2) 的最小值，所以 $\text{Min}[6] = 2$

- ***i*=7元时**， ...

- ***i*=8元时**， ...

○用1元和5元钱币，结果：

金额i:	0	1	2	3	4	5	6	7	8	9 ...
硬币数量Min[]:	0	1	2	3	4	1	2	3	4	...



递推关系：

$$\text{Min}[i] = \min(\text{Min}[i], \text{Min}[i - 5] + 1);$$

继续处理其它面值钱币。


```
void solve(){  
    for(int k = 0; k < MONEY; k++)    //初始值为无穷大  
        Min[k] = INT_MAX;  
    Min[0] = 0;  
  
    for(int j = 0; j < VALUE; j++)  
        for(int i = type[j]; i < MONEY; i++)  
            Min[i] = min(Min[i], Min[i - type[j]] + 1);    //递推式  
}
```

复杂度 $O(\text{money} * \text{value})$, money 输入的总金额, value 是钱币种类数

假如 $\text{money} = \text{value} = 1000$, 那 $O(\text{money} * \text{value}) = 100\text{万}$ 。

```

void solve(int s){
    for(int k = 0; k <= s; k++) //初始值为无穷大
        Min[k] = INT_MAX;
    Min[0] = 0;

    for(int j = 0; j < VALUE; j++)
        for(int i = type[j]; i <= s; i++)
            Min[i] = min(Min[i], Min[i - type[j]] + 1);
            //递推式
}

int main(){
    int s;
    while(cin >> s){
        solve(s);
        cout << Min[s] << endl;
    }
    return 0;
}

```

这样写main()
行吗？

- solve()计算100万次。
- 如果OJ有1万组测试数据，那么main()需要计算：100万*1万次=100亿次。
- 这样不行。

此题应该打表：提前计算所有的情况，然后根据输入的金额，直接打印结果。

```
int main()
{
    int s;
    solve();           //提前计算
    while(cin >> s)    //再读输入
        cout << Min[s] << endl;
    return 0;
}
```

- 如果OJ有1万组测试数据，那么只需要计算：100万 + 1万次。

扩展1：打印最少钱币的组合

- 增加一个记录表 **Min_path[i]**，记录金额i需要的最后一个钱币。利用Min_path[]逐步倒推，就能得到所有的钱币。

金额i:	0	1	2	3	4	5	6	7	8	9 ...
Min[]:	0	1	2	3	4	1	2	3	4	...
Min_Path[]:	0	1	1	1	1	5	5	5	5	...

- Min_Path[6]=5：最后一个钱币是5元的；
- Min_Path[6-5]=Min_Path[1]=1，1元钱币；
- Min_Path[1-1]=Min_Path[0]=0，结束；
- 输出：钱币组合是“5元 + 1元”

完整代码

```
#include <bits/stdc++.h>
using namespace std;
const int MONEY = 1001; //定义最大金额
const int VALUE = 5;    //5种硬币
int type[VALUE] = {1,5,10,25,50}; //5种面值
int Min[MONEY]; //每个金额对应最少的硬币数量
int Min_path[MONEY]={0}; //记录最小硬币的路径
void solve(){
    for(int k=0; k< MONEY;k++)
        Min[k] = INT_MAX;
    Min[0]=0;
    for(int j = 0;j < VALUE;j++)
        for(int i = type[j]; i < MONEY; i++){
            if(Min[i] > Min[i - type[j]]+1){
                Min_path[i] = type[j];
                //在每个金额上记录路径，即某个硬币的面值
                Min[i] = Min[i - type[j]] + 1; //递推式
            }
        }
}
```

//打印硬币组合

```
void print_ans(int *Min_path, int s) {  
    while(s){  
        cout << Min_path[s] << " ";  
        s = s - Min_path[s];  
    }  
}  
  
int main() {  
    int s;  
    solve();  
    while(cin >> s){  
        cout << Min[s] << endl; //输出最少硬币个数  
        print_ans(Min_path,s); //打印硬币组合  
    }  
    return 0;  
}
```

扩展2：硬币组合方案有多少？

hdu 2069题

- 有5种面值的硬币：1分、5分、10分、25分、50分。输入一个钱数 S ，输出组合方案的数量。
- 例如11分，有4种组合方案：11个1分、2个5分 + 1个1分、1个5分 + 6个1分、1个10分 + 1个1分。
- $S \leq 250$ ，硬币数量 $NUM \leq 100$ 。

方法一：暴力法

逐个枚举各种面值的硬币个数，判断每种情况是否合法。

```
int main() {  
    int s;  
    while (scanf("%d", &s) != EOF) {  
        int cnt = 0, a = s / 50, b = s / 25, c = s / 10, d = s / 5;  
        for (int i = 0; i <= a; ++i)  
            for (int j = 0; j <= b; ++j)  
                for (int k = 0; k <= c; ++k)  
                    for (int u = 0; u <= d; ++u) {  
                        int v = s - i*50 - j*25 - k*10 - u*5;  
                        if (v >= 0 && i + j + k + u + v <= 100) ++cnt;  
                    }  
        printf("%d\n", cnt);  
    }  
    return 0;  
}
```


方法二：动态规划法

定义状态为 $dp[i][j]$ ，建立一个“转移矩阵”，如下表。
 横向是金额（题目中 $i \leq 250$ ），纵向是硬币数（题目中最多用100个硬币， $j \leq 100$ ）。

[illegible]

矩阵元素 $dp[i][j]$ 的含义是：用 j 个硬币实现金额 i 的方案数量。

	0	1	2	3	4	5	6	7	8	9	10	...
0	1											
1		1				1						
2			1				1				1	
3				1				1				...
4					1				1			
5						1				1		
6							1				1	
7								1				
...									...			
100												

$dp[6][2] = 1$ ，表示用2个硬币凑出6分钱，只有1种方案（5分+1分）。

矩阵元素 $dp[i][j]$

- 例如表中 $dp[6][2] = 1$ ，表示用2个硬币凑出6分钱，只有1种方案：5分+1分。表中的空格为0，即没有方案，例如 $dp[6][1]=0$ ，用1个硬币凑6分钱，不存在这样的方案。
- 表中列出了 $dp[10][7]$ 以内的方案数。

- 第一步：只用1分硬币实现。 初始化： $dp[0][0] = 1$
- $dp[1][1] = dp[1][1] + dp[1-1][1-1] = dp[1][1] + dp[0][0] = 1$ 。
- $dp[1-1][1-1]$ 的意思是：在1分金额中减去1分硬币的金额；原来1个硬币也减少1个。在这次状态转移中， $dp[1][1]$ 与 $dp[1-1][1-1]$ 等价。

[illegible]

- 第二步：加上5分硬币，继续进行组合。
- $dp[i][j]$, $i < 5$ 时，组合中不可能有5分硬币。
- 当 $i \geq 5$ 时，金额为 i ，硬币为 j 个的组合数量，等价于在 i 中减去5分钱，而且硬币数量也减去1个（即这个面值5的硬币）的情况。 $dp[i][j] = dp[i][j] + dp[i-5][j-1]$ 。

	0	1	2	3	4	5	6	7	8	9	10	...
0	1											
1		1				1						
2			1				1				1	
3				1				1				...
4					1				1			
5						1				1		
6							1				1	
7								1				...
...									...			
100												

- 第三步：陆续加上10分、25分、50分硬币，同理有 $dp[i][j] = dp[i][j] + dp[i - \text{type}[k]][j - 1]$, $k=2, 3, 4$ 。

01背包问题

- 在01背包问题中，物品*i*或者被装入背包，或者不被装入背包，设 x_i 表示物品*i*装入背包的情况，则当 $x_i=0$ 时，表示物品*i*没有被装入背包， $x_i=1$ 时，表示物品*i*被装入背包。根据问题的要求，有如下约束条件和目标函数：

$$\begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\} \quad (1 \leq i \leq n) \end{cases}$$

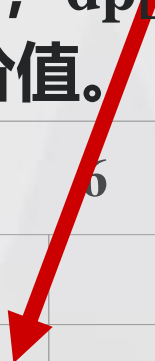
$$\max \sum_{i=1}^n v_i x_i$$

步骤解析

有5个物品，其重量分别是{2, 2, 6, 5, 4}，价值分别为{6, 3, 5, 4, 6}，背包的容量为10。

用一个 $(n+1) \times (C+1)$ 的二维表 $dp[][]$ ， $dp[i][j]$ 表示把前 i 个物品装入容量为 j 的背包中获得的最大价值。

		0	1	2	3	4	5	6	7	8	9	10
	0											
$w_1=2 \ v_1=6$	1											
$w_2=2 \ v_2=3$	2											
$w_3=6 \ v_3=5$	3											
$w_4=5 \ v_4=4$	4											
$w_5=4 \ v_5=6$	5											



填表的过程，是按只放第1个物品、只放前2个、只放前3个.....一直到放完，这样的顺序考虑。(从小问题扩展到大问题)

1、只装第1个物品。（横向是递增的背包容量）

[illegible]

2、只装前2个物品。如果第2个物品重量比背包容量大，那么不能装第2个物品，情况和只装第1个一样。如果第2个物品重量小于背包容量，那么：（1）如果把物品2装进去(重量是2)，那么相当于只把1装到(容量-2)的背包中。（2）如果不装2，那么相当于只把1装到背包中。 - - 取（1）和（2）的最大值。

[illegible]

2、只装前2个物品。如果第2个物品重量比背包容量大，那么不能装第2个物品，情况和只装第1个一样。如果第2个物品重量小于背包容量，那么：（1）如果把物品2装进去(重量是2)，那么相当于只把1装到(容量-2)的背包中。（2）如果不装2，那么相当于只把1装到背包中。 - - 取（1）和（2）的最大值。

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
$w_1=2 \ v_1=6$	1	0	0	6	6	6	6	6	6	6	6	6
$w_2=2 \ v_2=3$	2	0	0	3+0	6	9	9	9	9	9	9	9
$w_3=6 \ v_3=5$	3											
$w_4=5 \ v_4=4$	4											
$w_5=4 \ v_5=6$	5											

需要用到前面的结果，即已经解决的子问题的答案。

3、只装前3个物品。如果第3个物品重量比背包大，那么不能装第3个物品，情况和只装第1、2个一样。如果第3个物品重量小于背包，那么：（1）如果把物品3装进去(重量是6)，那么相当于只把1、2装到(容量-6)的背包中。（2）如果不装3，那么相当于只把1、2装到背包中。 - 取（1）和（2）的最大值。

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
$w_1=2 \ v_1=6$	1	0	0	6	6	6	6	6	6	6	6	6
$w_2=2 \ v_2=3$	2	0	0	6	6	9	9	9	9	9	9	9
$w_3=6 \ v_3=5$	3	0	0	6	6	9	9	9	9	11	11	14
$w_4=5 \ v_4=4$	4											
$w_5=4 \ v_5=6$	5											

3、只装前3个物品。如果第3个物品重量比背包大，那么不能装第3个物品，情况和只装第1、2个一样。如果第3个物品重量小于背包，那么：（1）如果把物品3装进去(重量是6)，那么相当于只把1、2装到(容量-6)的背包中。（2）如果不装3，那么相当于只把1、2装到背包中。 - 取（1）和（2）的最大值。

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
$w_1=2 \ v_1=6$	1	0	0	6	6	6	6	6	6	6	6	6
$w_2=2 \ v_2=3$	2	0	0	6	6	9	9	9	9	9	9	9
$w_3=6 \ v_3=5$	3	0	0	6	6	9	9	9	9	6+5	11	14
$w_4=5 \ v_4=4$	4											
$w_5=4 \ v_5=6$	5											

按这样的规律一行行填表，直到结束。现在回头考虑，装了哪些物品。

看最后一列， $15 > 14$ ，说明装了物品5，否则价值不会变化

°

		0	1	2	3	4	5	6	7	8	9	10	
	0	0	0	0	0	0	0	0	0	0	0	0	$x_1=1$
$w_1=2 \ v_1=6$	1	0	0	6	6	6	6	6	6	6	6	6	$x_2=1$
$w_2=2 \ v_2=3$	2	0	0	6	6	9	9	9	9	9	9	9	$x_3=0$
$w_3=6 \ v_3=5$	3	0	0	6	6	9	9	9	9	11	11	14	$x_4=0$
$w_4=5 \ v_4=4$	4	0	0	6	6	9	9	9	10	11	13	14	$x_5=1$
$w_5=4 \ v_5=6$	5	0	0	6	6	9	9	12	12	15	15	15	

- 复杂度： $n \times C$

- 思考：如何输出背包方案？

例题 hdu 2602

“骨头收集者”带着体积 V 的背包去捡骨头，已知每个骨头的体积和价值，求能装进背包的最大价值。 $n \leq 1000$, $V \leq 1000$ 。

输入：第1行是测试数量，第2行是骨头数量和背包体积，第3行是每个骨头的价值，第4行是每个骨头的体积。

1

5 10

1 2 3 4 5

5 4 3 2 1

输出：最大价值。

14

解题思路

经典的01背包问题

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
struct BONE{
```

```
    int val;
```

```
    int vol;
```

```
}bone[1011];
```

```
int T,N,V;
```

```
int dp[1011][1011];
```

```
int ans(){
```

```
    memset(dp,0,sizeof(dp));
```

```
    for(int i=1; i<=N; i++)
```

```
        for(int j=0; j<=V; j++){
```

```
            if(bone[i].vol > j) //第i个骨头太大，装不了
```

```
                dp[i][j] = dp[i-1][j];
```

```
            else //第i个骨头可以装
```

```
                dp[i][j]=max(dp[i-1][j],dp[i-1][j-bone[i].vol]+bone[i].val);
```

```
        }
```

```
    return dp[N][V];
```

```
}
```

```
int main(){
```

```
    cin>>T;
```

```
    while(T--){
```

```
        cin >> N >> V;
```

```
        for(int i=1;i<=N;i++) cin>>bone[i].val;
```

```
        for(int i=1;i<=N;i++) cin>>bone[i].vol;
```

```
        cout << ans() << endl;
```

```
    }
```

```
    return 0;
```

```
}
```

空间优化：滚动数组

- 处理`dp[][]`状态数组的时候，有个小技巧：把它变成一维的`dp[]`，以节省空间。
- 观察二维表`dp[][]`，可以发现，每一行是从上面一行算出来的，只跟上面一行有关系，跟更前面的行没有关系。
- 那么用新的一行覆盖原来的一行就好了。

```
int ans(){  
    memset(dp, 0, sizeof(dp));  
    for(int i=1; i<=N; i++){  
        for(int j=V; j>=bone[i].vol; j--) //反过来循环  
            dp[j]=max(dp[j],dp[j-bone[i].vol]+bone[i].val);  
    return dp[V];  
}
```

最长公共子序列

- 一个给定序列的**子序列**是在该序列中删去若干元素后得到的序列。
- **例如:** $X = (A, B, C, B, D, A, B)$
- X 的子序列: 所有 X 的子集(集合中元素按原来在 X 中的顺序排列), 例如 (A, B, D) , (B, C, D, B) , 等等。

- 给定两个序列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：公共子序列中长度最长的子序列。

最长公共子序列问题：给定两个序列X和Y，找出X和Y的一个最长公共子序列。

- $X = (A, B, C, B, D, A, B)$ $X = (A, B, C, B, D, A, B)$
- $Y = (B, D, C, A, B, A)$ $Y = (B, D, C, A, B, A)$
- (B, C, B, A)和(B, D, A, B)都是X和Y的最长公共子序列(长度为4)
- 但是,(B, C, A)就不是X和Y的最长公共子序列

蛮力法

- 对于每一个 $X=\{x_1, x_2, \dots, x_m\}$ 的子序列，验证它是否是 $Y=\{y_1, y_2, \dots, y_n\}$ 的子序列。
- X 有 2^m 个子序列。
- 每个子序列需要 $O(n)$ 的时间来验证它是否是 Y 的子序列：从 Y 的第一个字母开始扫描，如果不是则从第二个开始。
- 运行时间: $O(n2^m)$ 。

- 不用蛮力法。

- 是否有一种可以逐步缩小问题规模，从而减少复杂度的办法？

动态规划

- $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的最长公共子序列，递推：
 - 1、**当 $x_m=y_n$ 时**，找出 X_{m-1} 和 Y_{n-1} 的最长公共子序列，然后在其尾部加上 x_m 即可得到 X 和 Y 的最长公共子序列；
 - 2、**当 $x_m \neq y_n$ 时**，求解两个子问题：（1）找出 X_{m-1} 和 Y 的最长公共子序列；（2） X_m 和 Y_{n-1} 的最长公共子序列。这两个公共子序列中的较长者即为 X 和 Y 的最长公共子序列。

用 $L[i][j]$ 表示子序列 X_i 和 Y_j 的最长公共子序列的长度，动态规划函数：

$$L[0][0]=L[i][0]=L[0][j]=0(1\leq i\leq m, 1\leq j\leq n)$$

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

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

$$L[1][1] = L[0][0] + 1 \quad (x_1 = y_1)$$

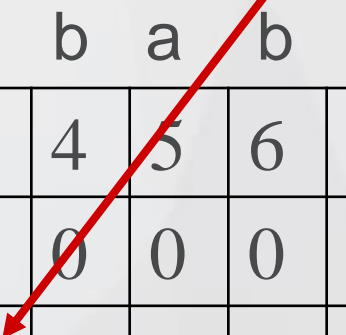
Y = **a** c b b a b d b b

X =		0	1	2	3	4	5	6	7	8	9
		0	0	0	0	0	0	0	0	0	0
a	1	0	1								
b	2	0									
c	3	0									
b	4	0									
d	5	0									
b	6	0									

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

$$L[1][3] = \max\{L[1][2], L[0][3]\} = 1 \quad (x_1 \neq y_3)$$

Y=		a	c	b	b	a	b	d	b	b	
X		0	1	2	3	4	5	6	7	8	9
=		0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1						
b	2	0									
c	3	0									
b	4	0									
d	5	0									
b	6	0									



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

$$L[2][1] = \max\{L[2][0], L[1][1]\} = 1 \quad (x_2 \neq y_1)$$

Y= **a** c b b a b d b b

X		0	1	2	3	4	5	6	7	8	9
=		0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1	1	1	1	1
b	2	0	1								
c	3	0									
b	4	0									
d	5	0									
b	6	0									

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

$$L[2][2] = \max\{L[2][1], L[1][2]\} = 1 \quad (x_2 \neq y_2)$$

Y= a **c** b b a b d b b

X=		0	1	2	3	4	5	6	7	8	9
		0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	1	1	1	1	1	1
b	2	0	1	1							
c	3	0									
b	4	0									
d	5	0									
b	6	0									

例题： hdu 1159

求2个序列的最长公共子序列。

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int dp[1005][1005];
```

```
string str1, str2;
```

```
int LCS(){
```

```
    memset(dp, 0, sizeof(dp));
```

```
    for(int i=1; i<=str1.length(); i++)
```

```
        for(int j=1; j<=str2.length(); j++){
```

```
            if(str1[i-1] == str2[j-1])
```

```
                dp[i][j] = dp[i-1][j-1] + 1;
```

```
            else
```

```
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

```
        }
```

```
    return dp[str1.length()][str2.length()];
```

```
}
```

```
int main(){  
    while(cin >> str1 >> str2)  
        cout<< LCS() << endl;  
    return 0;  
}
```

● 最长递增子序列

- 最长递增子序列 (Longest Increasing Subsequence, **LIS**) 问题：给定一个长度为N的数组，找出一个最长的单调递增子序列。
- 例：一个长度为6的序列 $A = \{5, 6, 7, 4, 2, 8, 3\}$ ，它最长的单调递增子序列为 $\{5, 6, 7, 8\}$ ，长度为4。

例题：hdu 1257 最少拦截系统

某国有一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能超过前一发的高度。某天，雷达捕捉到敌国的导弹来袭，请帮助计算一下最少需要多少套拦截系统。

输入：导弹总个数，导弹依此飞来的高度。

输出：最少要配备多少套这种导弹拦截系统。

Sample Input

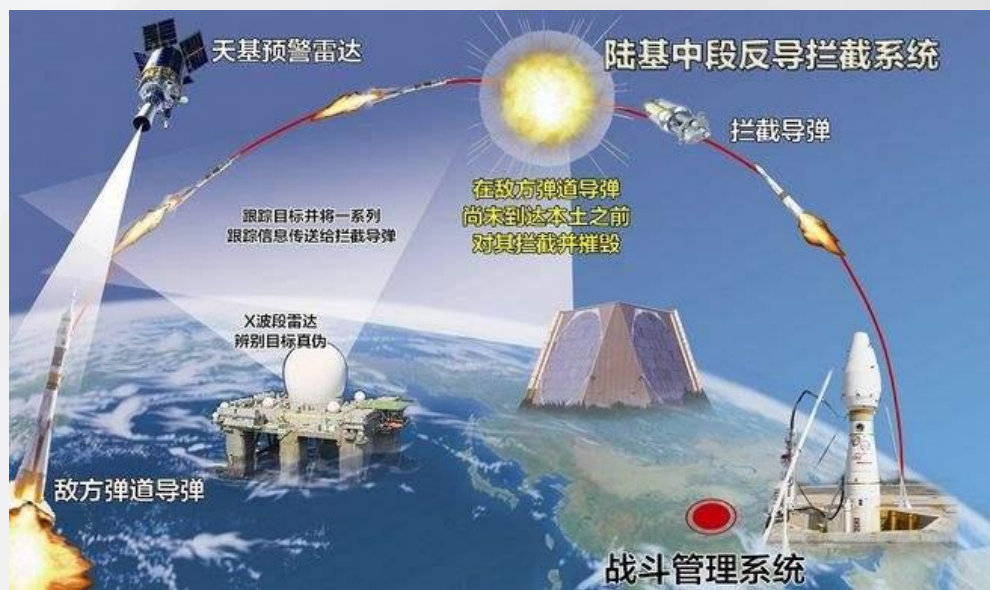
8 389 207 155 300 299 170 158 65

Sample Output

2

解法1：贪心

假设发射了很多高度无穷大的导弹。
读入第一个炮弹时，一个导弹下降来拦截。
以后每读入一个新的炮弹，就由能拦截它的最低的那个导弹来拦截。
最后，统计用到了几个导弹，就是最少的拦截系统。



解法2: DP

- 题目要统计一个序列中的单调递减子序列，最少有多少个。和最长递增子序列LIS有什么关系呢？
- 假设已经有了求LIS的算法，读者可能想这么做：把序列反过来，就变成了求反序列的递增子序列；先求反序列的第1个LIS，然后从原序列中去掉这个LIS，再对剩下的求第2个LIS，一直到序列为空；这些LIS的数量就是题目的解。
- 但是，其实不用这么麻烦，这个题目实际上等价于求原序列的LIS，这是一道求LIS的**裸题**。
- 为什么？

原因解释:

- 模拟计算过程:①从第1个数开始, 找一个最长的递减子序列, 即第1个拦截系统X, 在样例中是{389, 300, 299, 170, 158, 65}, 去掉这些数, 序列中还剩下(207, 155);②在剩下的序列中再找一个最长递减子序列, 即第2个拦截系统Y, 是{207, 155}。
- 在Y中, 至少有一个数a大于X中的某个数, 否则a比X的所有数都小, 应该在X中。所以, 从每个拦截系统中拿出一个数能构成个递增子序列. 即拦截系统的数量等于这个递增子序列的长度。如果这个递增子序列不是最长的, 那么可以从某个拦截系统中拿出两个数c, d, 在拦截系统中 $c > d$, c和d不是递增的, 这与递增序列的要求矛盾。

总结：求解LIS的几种方法

(1) 方法1：借助最长公共子序列LCS。首先对序列排序，得到 $A'=\{2, 3, 4, 5, 6, 7, 8\}$ ，那么A的LIS就是A和A'的LCS。

复杂度是 $O(n^2)$ 。

(2) 方法2：直接用DP求解LIS。

定义状态 $dp[i]$ ，表示以第 i 个数为结尾的最长递增子序列的长度，那么：

$$dp[i] = \max\{0, dp[j]\} + 1, 0 < j < i, A_j < A_i$$

最后答案是 $\max\{dp(i)\}$ 。

方法2的复杂度也是 $O(n^2)$ ，和方法1一样。

(3) 方法3。有一种更快的、复杂度 $O(n\log n)$ 的方法。这个方法不是DP算法，它巧妙地利用了序列本身的特征，通过一个辅助数组 $d[]$ ，统计最长递增子序列的长度。

递推与记忆化搜索

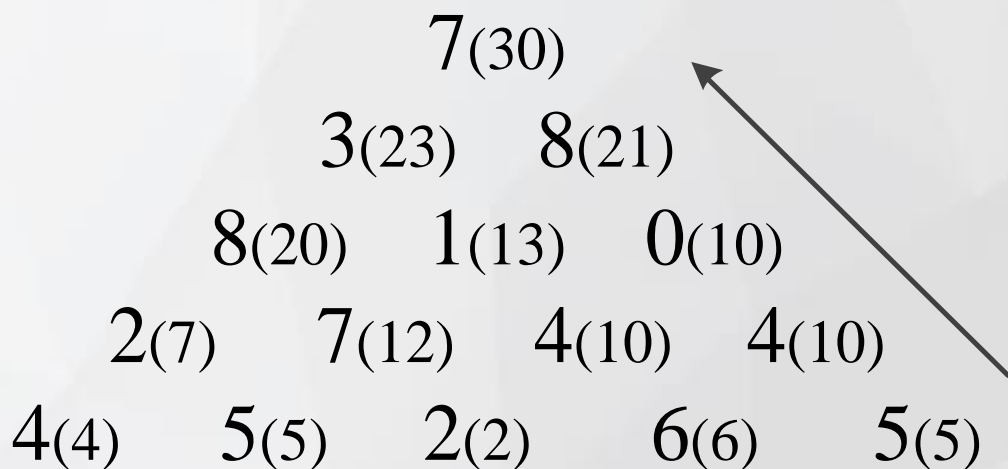
- 前面DP的状态转移，都是用递推的方法。
- 有另一种方法，逻辑上的理解更加直接，这就是用“递归+记忆化搜索”来实现DP。

经典题：poj 1163 The Triangle

- 给定一个n层的三角形数塔，从顶部第一个数往下走，每层经过一个数字，直到最底层。只能走斜下方的左边一个数或右边一个数。问所有可能走到的路径，最大的数字和是多少？
- 此题如果按“从顶往下”的计算方法，有 2^n 个路径，导致TLE。

递推：“从底往上”计算

- 对数塔上的每个点记录状态， $dp[i][j]$ 记录从第 i 层第 j 个数开始往下走的数字和，每个结点算一次，
- 一共有 $O(n^2)$ 个结点，所以复杂度是 $O(n^2)$ 。



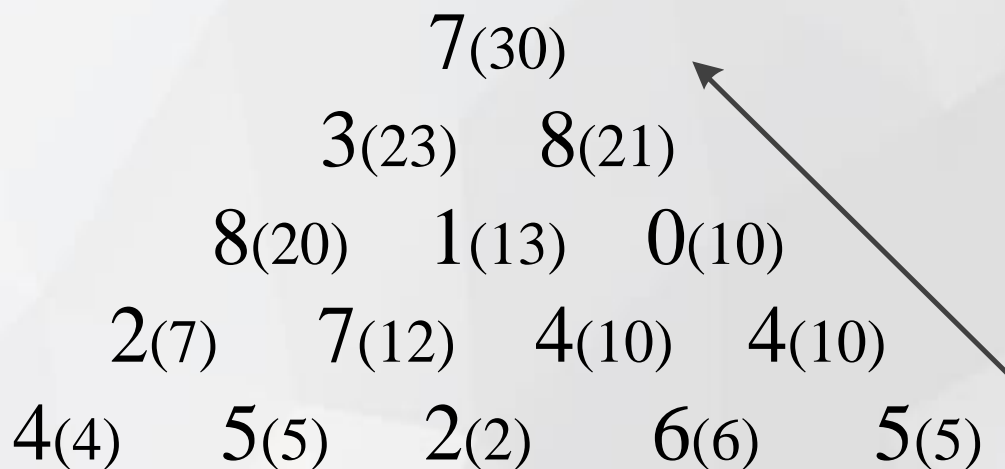
DP的另一种写法：“递归+记忆化搜索”

- 首先，写递归程序，暴力搜索所有 2^n 个路径。

```
int dfs(int i, int j) {  
    if(i == n)  
        return a[i][j]; //递归边界：到达最后一行，返回  
    return dp[i][j] = max(dfs(i+1, j), dfs(i+1, j+1)) +  
        a[i][j];  
    //从左边走上来，或者从右边走上来，取其中大的  
}
```

递归的优化：记忆化搜索

- 递归时，有大量重复计算，其实能避免。
- 观察第3层的中间数“1”，从第2层的“3”往下走会经过“1”，计算一次从“1”出发的递归；从第2层的“8”往下走也会经过“1”，又重新计算了从“1”出发的递归。所以，只要避免这些重复计算，就能优化。



```
int dfs(int i, int j) {  
    if(i == n)  
        return a[i][j];  
    if(dp[i][j] >= 0) return dp[i][j]; //记忆!  
    return dp[i][j] = max(dfs(i+1,j), dfs(i+1,j+1))+a[i][j];  
}
```

- **红色**代码：如果发现dp[i][j]已经计算过，就不再重算。
- 由于数塔的结点有 $O(n^2)$ 个，每个点只需要计算一次dp[i][j]，所以dfs()的运行次数只有 $O(n^2)$ 次，和递推程序的复杂度一样。

记忆化搜索

- 用递归实现DP时，在递归程序中记录计算过的状态，并在后续的计算中跳过已经算过的重复的状态，从而大大减少递归的计算次数，这就是“记忆化搜索”的思路。

习题：矩阵矩阵连乘问题

给定 n 个矩阵 $\{A_1, A_2, \dots, A_n\}$, 其中 A_i 与 A_{i+1} 是可乘的, $i=1, 2, \dots, n-1$ 。现在要计算这 n 个矩阵的连乘积。由于矩阵的乘法满足结合律, 所以通过加括号可以使得计算矩阵的连乘积有许多不同的计算次序。然而采用不同的加括号方式, 所需要的总计算量是不一样的。

分析

- 对于这个问题，穷举法虽然易于入手，但是经过计算，它所需要的计算次数是 n 的指数函数，因此在效率上显得过于低下。现在我们按照动态规划的基本步骤来分析解决这个问题，并比较它与穷举法在时间消耗上的差异。

(1) 分析最优解的结构

- 现在，将矩阵连乘积 $A_i A_{i+1} \dots A_j$ 简记为 $A[i:j]$ 。对于 $A[1:n]$ 的一个最优次序，设这个计算次序在矩阵 A_k 和 A_{k+1} 之间将矩阵链断开 ($1 \leq k < n$)，那么完全加括号的方式为 $((A_1 \dots A_k)(A_{k+1} \dots A_n))$ 。依此次序，我们应该先分别计算 $A[1:k]$ 和 $A[k+1:n]$ ，然后将计算结果相乘得到 $A[1:n]$ ，总计算量为 $A[1:k]$ 的计算量加上 $A[k+1:n]$ 的计算量，再加上 $A[1:k]$ 和 $A[k+1:n]$ 相乘的计算量。
- 通过反证法可以证明，问题的关键特征在于，计算 $A[1:n]$ 的一个最优次序所包含的计算矩阵子链 $A[1:k]$ 和 $A[k+1:n]$ 的次序也是最优的。因此，矩阵连乘积计算次序问题的最优解包含着其子问题的最优解。这种最优子结构性质是该问题可以用动态规划解决的重要特征。

(2) 建立递归关系定义最优值

- 设计算 $A[i:j]$ ($1 \leq i \leq j \leq n$)所需的最少数乘次数为 $m[i][j]$, 则原问题的最优值为 $m[1][n]$ 。而且易见, 当 $i=j$ 时, $m[i][j]=0$ 。
- 根据上述最优子结构性质, 当 $i < j$ 时, 若计算 $A[i:j]$ 的最优次序在 A_k 和 A_{k+1} 之间断开, 可以定义 $m[i][j] = m[i][k] + m[k+1][j] + p_{i-1} * p_k * p_j$ (其中, A_i 的维数为 $p_{i-1} * p_i$)。从而有:
- 当 $i=j$ 时, $m[i][j]=0$ 。当 $i < j$ 时,
 $m[i][j] = \min\{m[i][k] + m[k+1][j] + p_{i-1} * p_k * p_j\}$ ($i \leq k < j$)。除此之外, 若将对应于 $m[i][j]$ 的断开位置记为 $s[i][j]$, 在计算出最优值 $m[i][j]$ 后, 可以递归地由 $s[i][j]$ 构造出相应的最优解。

(3) 计算最优值

- 如果直接套用 $m[i][j]$ 的计算公式，进行简单的递归计算需要耗费指数计算时间。然而，实际上不同的子问题的个数只是 n 的平方项级（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）。用动态规划解决此问题，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

代码

```
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++) //链长度控制
        for ( int i=1;i<=n-r+1;i++) //链起始位置控制
            {int j=i+r-1; //链终止位置
                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;
                        }
                    }
            }
    }
}
```

(4) 构造最优解

- 算法首先设定 $m[i][i] = 0 (i = 1, 2, \dots, n)$ 。然后再根据递归式按矩阵链长的递增方式依次计算出各个 $m[i][j]$ ，在计算某个固定的 $m[i][j]$ 时，只用到已计算出的 $m[i][k]$ 和 $m[k+1][j]$ 。
- 稍加分析就可以得出，这个算法以 $O(n^2)$ 的空间消耗大大降低了时间复杂度，计算时间的上界为 $O(n^3)$ 。
- 通过以上算法的计算，我们知道了要计算所给矩阵连乘积所需的最少数乘次数，但是还不知道具体应该按照什么顺序来做矩阵乘法才能达到这个次数。然而， $s[i][j]$ 已经存储了构造最优解所需要的足够的信息。从 $s[1][n]$ 记录的信息可知计算 $A[1:n]$ 的最优加括号方式为 $(A[1:s[1][n]])(A[s[1][n]+1:n])$ 。同理，每个部分的最优加括号方式又可以根据数组 s 的相应元素得出。照此递推下去，最终可以确定 $A[1:n]$ 的最优完全加括号方式，即构造出问题的一个最优解。

习题：花插花瓶

花瓶粘在架子上，并从左到右从1到V连续编号，其中V是花瓶的数量。花是可移动的，并且由1到F之间的整数唯一地标识。这些id号具有重要意义：它们确定花瓶行中花束的出现顺序，因此，束i必须在花瓶中每当 $i < j$ 时，花瓶的左侧装有束j。现在，必须将所有这些束放入花瓶中，并保持其编号一致。如果花瓶多于束鲜花，多余的花瓶将被留空。一个花瓶只能容纳一束鲜花。将一束鲜花放在花瓶中会产生一定的美学价值，用整数表示。美学价值如下表所示。将花瓶留空的审美价值为0。求在保持所需顺序的同时最大化布置的美学价值总和。 $1 \leq F \leq V \leq 100$, $-50 \leq A_{ij} \leq 50$ 其中 A_{ij} 是将花束i放入花瓶j中获得的美学价值。

样本输入

3 5

7 23 -5 -24 16

5 21 -4 10 23

-21 5 -4 -20 20

样本输出

53

		V A S E S				
		1	2	3	4	5
Bunches	1 (azaleas)	7	23	-5	-24	16
	2 (begonias)	5	21	-4	10	23
	3 (carnations)	-21	5	-4	-20	20

分析

- 可以花（也可以花瓶）的数目来划分阶段，第*i*个阶段决定花*k*是否插花瓶；状态变量*j*表示前*i*束花插了多少个花瓶；而对于任意一个状态*j*，决策就是第*j*个花瓶是否放第*i*束花，决策*path[i][j]*用1或0来表示。最优指标函数*dp[i][j]*表示前*i*个花插在了*j*个花瓶中，所能取得的最大美学值。注意，这里仍然是倒过来考虑。

决策：*path[i][j]*取0或1

- 动态规划方程为

$$dp[i][j] = \max(dp[i-1][j-1] + a[i][j], dp[i][j-1])$$

- 边界条件为

$$dp[i][0] = 0, 0 \leq i \leq F$$

```
#include <stdio.h>
#include <string.h>
#define N 102
#define MIN -0x3fffffff
#define max(a,b) a>b?a:b
int f,v;
int dp[N][N],a[N][N];

int main(){
    //freopen("a.txt","r",stdin);
    scanf("%d %d",&f,&v);
    int i,j;
    memset(dp,0,sizeof(dp));
    for(i = 1;i<=f;i++){
        for(j = 1;j<=v;j++){
            scanf("%d",&a[i][j]);
        }
        for(i = 1;i<=f;i++){
            for(j = 1;j<=v;j++){
                dp[i][j] = max(dp[i-1][j-1]+a[i][j],dp[i][j-1]);
            }
        }
        printf("%d\n",dp[f][v]);
    }
    return 0;
}
```

扩展：求出插花的方案

```
#include <stdio.h>
#include <string.h>
#define N 105
int v[N][N], dp[N][N], path[N][N];
int n, m;
void print(int x, int y){
    int i;
    if(!x)
        return ;
    for(i = y; i >= 1; i--){
        if(path[x][i]){ //表示第x朵花就插在第i个花盆中 否则表示第i个花盆空着
            print(x-1, i-1);
            if(x != n)
                printf("%d ", i);
            else
                printf("%d\n", i);
            return ;
        }
    }
}
int main(){
    int i, j;
    //freopen("a.txt", "r", stdin);
    memset(dp, 0, sizeof(dp));
    scanf("%d %d", &n, &m);
    for(i = 1; i <= n; i++){
        for(j = 1; j <= m; j++){
            scanf("%d", &v[i][j]);
            //v[i][j] += 50;
        }
    }
    for(i = 1; i <= n; i++){
        for(j = 1; j <= m; j++){
            if(dp[i-1][j-1] + v[i][j] >= dp[i][j-1]){
                dp[i][j] = dp[i-1][j-1] + v[i][j];
                path[i][j] = 1;
            }else{
                dp[i][j] = dp[i][j-1];
                path[i][j] = 0;
            }
        }
    }
    printf("%d\n", dp[n][m]);
    print(n, m);
    return 0;
}
```

习题

(1) 简单题

hdu 2018, 2041, 2044, 2050, 2182, 4489。

滚动数组：

hdu 1024 "Max Sum Plus Plus";

hdu 4576 "Robot";

hdu 5119 "Happy Matt Friends"。

(2) 背包

有0-1背包、完全背包、分组背包、多重背包等。

hdu 1864 最大报销额，0/1背包。

hdu 2159 FATE，完全背包。

hdu 2844 Coins，多重背包。

hdu 2955 Robberies，0/1背包。

hdu 3092 Least common multiple，完全背包+数论。

poj 1015 Jury Compromise。

poj 1170 Shopping Offers，状态压缩背包。

(3) LIS

hdu 1003 Max Sum，最大连续子序列。

hdu 1087 Super Jumping!

hdu 4352 XHXJ' s LIS，数位DP+LIS。

poj 1239 Increasing Sequence，两次dp。

(4) LCS

hdu 1503 Advanced Fruits，LCS变形。

poj 1080 Human Gene Functions，LCS变形。