

# 背包问题

背包问题及其特点：

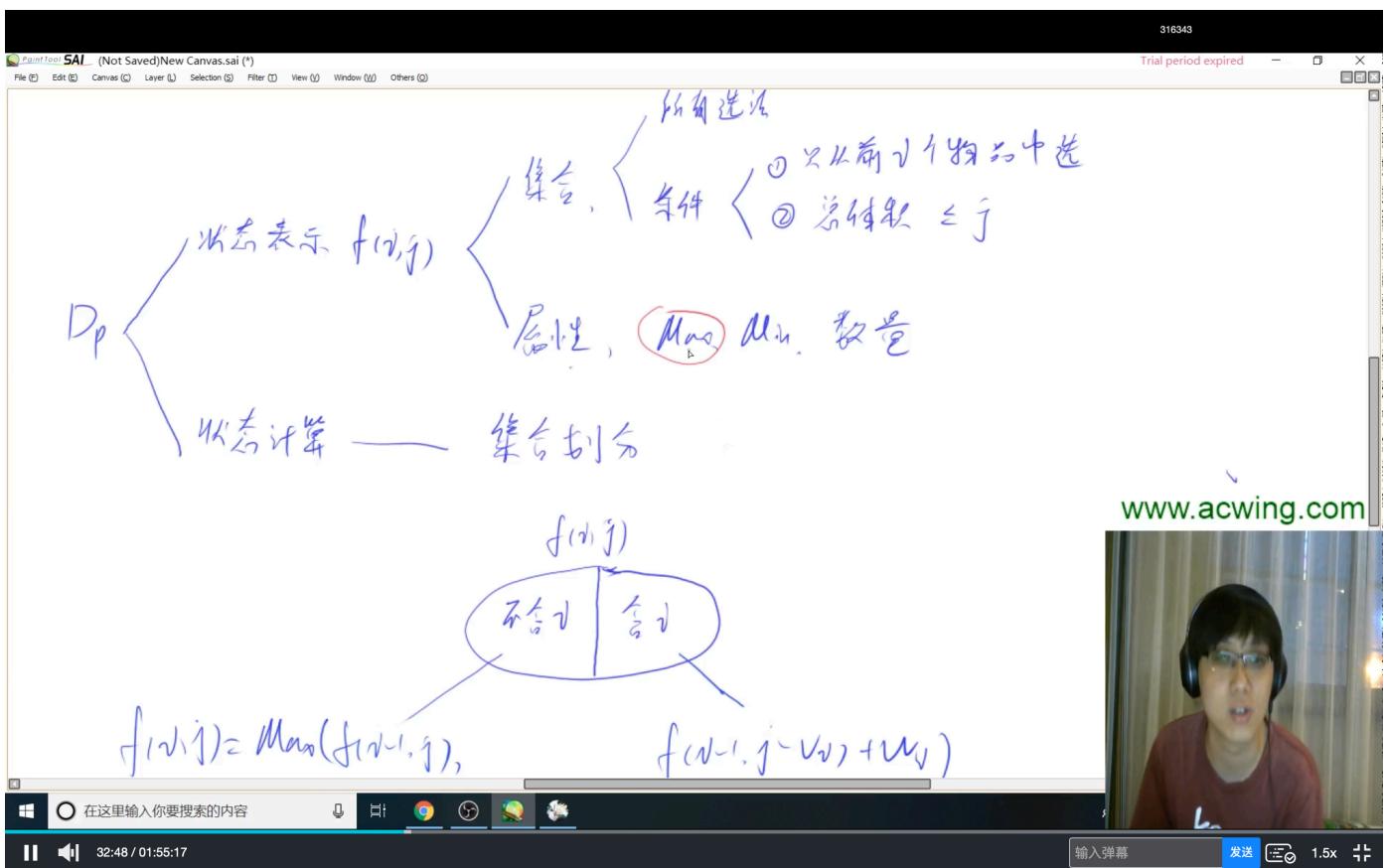
0-1背包：每种物品只能用一次

完全背包：每种物品有无数个

多重背包：每个物品的个数不一样

分组背包：物品有n组，每组物品有若干种，一组里面最多选一种

## 0-1背包问题



N个物品，容量是V的背包，每个物品有体积 $v_i$ ，价值 $w_i$ 。每一个物品只能用一次。求背包能装下的价值之和最大是多少。

二维：

```
1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4
5 const int N = 1010;
6 int n,m;//物品的数量，背包容量
7 int v[N],w[N];//物品的体积和价值
8 int f[N][N];//物品状态
9 int main(){
```

```

10    cin>>n>>m;
11    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];//读入物品状态
12    // f[0][0~m]=0 一件物品也没有
13    //因为初始化的数组都是0, 所以这种情况可以直接不用写, for循环从1开始写
14    for(int i=1;i<=n;i++){
15        for(int j=0;j<=m;j++){
16            f[i][j] = f[i-1][j];//不含i的情况
17            if(j>=v[i]){
18                //背包起码要装得下第i件物品才能有右边的情况
19                f[i][j] = max(f[i][j],f[i-1][j-v[i]]+w[i]);
20            }
21        }
22    }
23    cout<<f[n][m]<<endl;
24    return 0;
25 }
```

一维:

把二维的删成一维的, 但  $f[i][j] = \max(f[i][j], f[i-1][j-v[i]]+w[i])$   
 这一步会出问题, 直接删掉的话会变成  $f[j] = \max(f[j], f[j-v[i]]+w[i])$   
 实际计算的是  $f[i][j] = \max(f[i][j], f[i][j-v[i]]+w[i])$   
 所以需要更改内层for循环的顺序, 使当前的 $f[j-v[i]]+w[i]$ 是未更新过的,  
 也就是原来的 $f[i-1][j-v[i]]+w[i]$

```

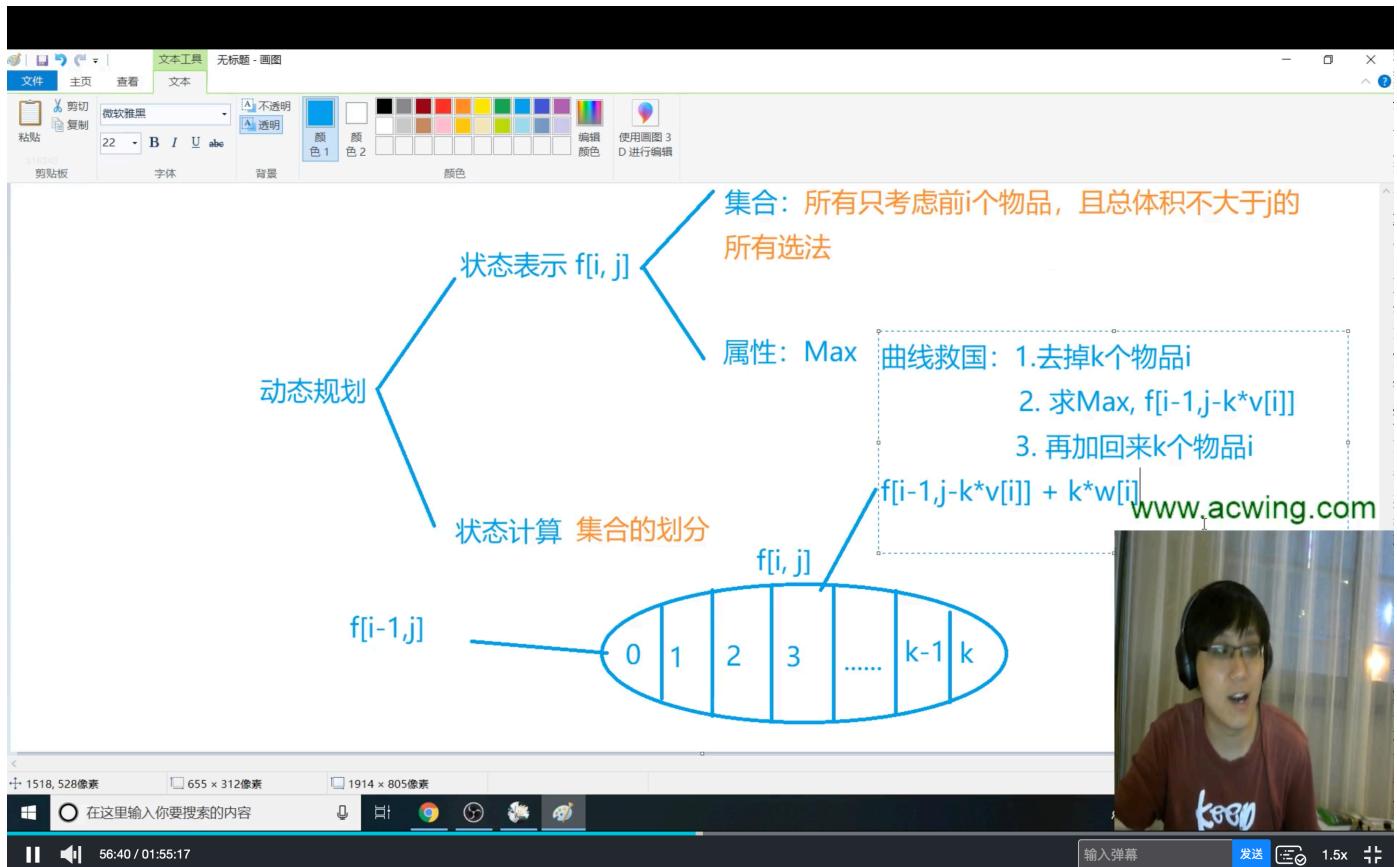
1 #include<iostream>
2 using namespace std;
3 /*
4 把二维的删成一维的, 但 f[i][j] = max(f[i][j], f[i-1][j-v[i]]+w[i])
5 这一步会出问题, 直接删掉的话会变成 f[j] = max(f[j], f[j-v[i]]+w[i])
6 实际计算的是 f[i][j] = max(f[i][j], f[i][j-v[i]]+w[i])
7 所以需要更改内层for循环的顺序, 使当前的f[j-v[i]]+w[i]是未更新过的, 也就是原来的f[i-1][j-
v[i]]+w[i]
8 */
9 const int N = 1010;
10 int n,m;//物品的数量, 背包容量
11 int v[N],w[N];//物品的体积和价值
12 int f[N];//物品状态
13
14 int main(){
15     cin>>n>>m;
16     for(int i=1;i<=n;i++){
17         cin>>v[i]>>w[i];
18     }
19     for(int i=1;i<=n;i++){
20         for(int j=m;j>=v[i];j--) {
21             //简化了原来的if条件判断
22             f[j] = max(f[j],f[j-v[i]]+w[i]);
23         }
24     }
25     cout<<f[m]<<endl;
26     return 0;
27 }
```

## 完全背包问题

有N件物品和一个最多能背重量为W的背包。第*i*件物品的重量是weight[i]，得到的价值是value[i]，每件物品都有无限个（也就是可以放入背包多次），求解将哪些物品装入背包里物品价值总和最大。

完全背包和01背包问题唯一不同的地方就是，每种物品有无限件。

//按每个物品选k个划分集合



```
1 | f[i][j] = f[i-1,j-v[i]*k] + w[i]*k
```

朴素算法代码实现：会超时的！三重循环花了太长时间了

```

1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4 const int N = 1010;
5 int n,m;//物品数，背包总容量
6 int v[N],w[N];//体积，价值
7 int f[N][N];//状态
8
9 int main(){
10     cin>>n>>m;
11     for(int i=1;i<=n;i++) cin>>v[i]>>w[i];

```

```

12
13     for(int i=1;i<=n;i++)
14         for(int j=0;j<=m;j++)
15             for(int k=0;k*v[i]<=j;k++){
16                 //k不能无限大, k倍的物品体积要小于背包容量
17                 f[i][j] = max(f[i][j],f[i-1][j-v[i]*k]+w[i]*k); //状态转移方程
18             }
19         cout<<f[n][m]<<endl;
20     return 0;
21 }
```

优化一下：

$$f[i,j] = f[i-1, j - v[i]*k] + w[i]*k$$

$$f[i, j] = \text{Max}(f[i-1, j], f[i-1, j-v] + w, f[i-1, j-2v] + 2w, f[i-1, j-3v] + 3w, \dots)$$

$$f[i, j-v] = \text{Max}(f[i-1, j-v], f[i-1, j-2v] + w, f[i-1, j-3v] + 2w, \dots)$$

$$f[i,j] = \text{Max}(f[i-1,j], f[i,j-v] + w)$$

观察一下发现 $f[i][j]$ 后面的情况和 $f[i][j-v]$ 是差不多的，只差了一个 $w$ ，所以只需要枚举两个状态就可以找到最大值，有点类似于0-1背包的状态转移方程

$$f[i,j] = \text{Max}(f[i-1,j], f[i-1,j-v] + w)$$

$$f[i,j] = \text{Max}(f[i-1,j], f[i,j-v] + w)$$

```

1 0-1:f[i][j] = max(f[i][j],f[i-1][j-v[i]]+w[i]);
2 //0-1背包的状态转移是从i-1转移过来的
```

修改代码后：

```

1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4 const int N = 1010;
```

```

5 int n,m;//物品数, 背包总容量
6 int v[N],w[N];//体积, 价值
7 int f[N][N];//状态
8
9 int main(){
10    cin>>n>>m;
11    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];
12
13    for(int i=1;i<=n;i++){
14        for(int j=0;j<=m;j++){
15            f[i][j] = f[i-1][j];
16            if(j>=v[i]) f[i][j] = max(f[i][j],f[i][j-v[i]]+w[i]);
17        }
18
19    cout<<f[n][m]<<endl;
20    return 0;
21}

```

变成一维：删掉i的这一维，完全背包问题的终极写法

```

1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4 const int N = 1010;
5 int n,m;//物品数, 背包总容量
6 int v[N],w[N];//体积, 价值
7 int f[N];//状态
8
9 int main(){
10    cin>>n>>m;
11    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];
12
13    for(int i=1;i<=n;i++)
14        for(int j=v[i];j<=m;j++){
15            //f[j] = f[j];删掉一维之后是恒等式, 直接删了
16            //if(j>=v[i]) 直接写在for条件里面了
17            f[j] = max(f[j],f[j-v[i]]+w[i]);
18            /*
19             不需要像0-1背包那样更改循环的顺序
20             因为j-v[i]<j, 所以这一步里面它已经更新过了, 是f[i][j-v[i]]+w[i]
21             和原来二维时候的状态转移方程是一样的
22            */
23        }
24
25    cout<<f[m]<<endl;
26    return 0;
27}

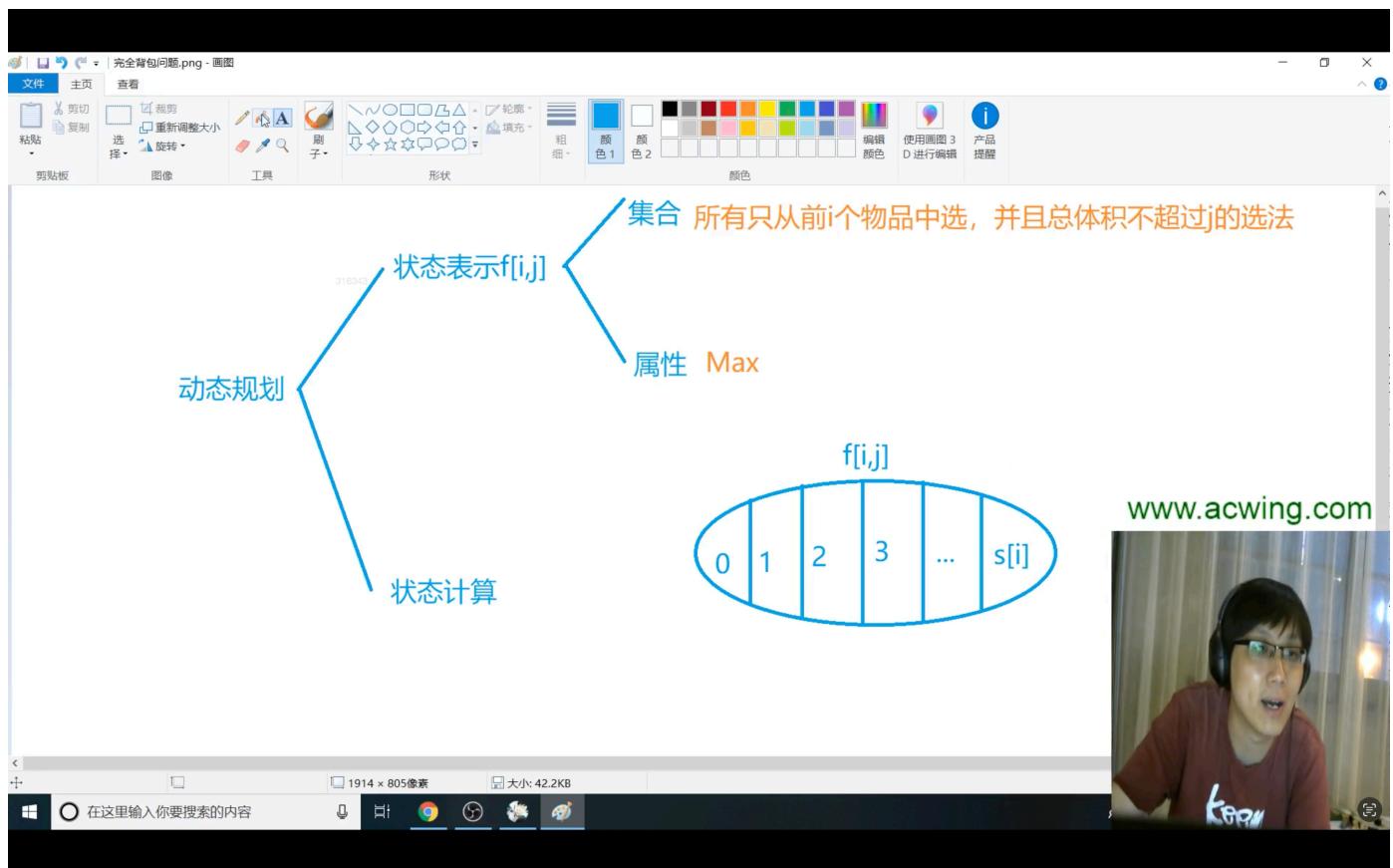
```

## 多重背包问题

有 N 种物品和一个容量是 V 的背包。

第  $i$  种物品最多有  $s_i$  件，每件体积是  $v_i$ ，价值是  $w_i$ 。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。



```
1 //状态转移方程
2 f[i][j] = max(f[i][j], f[i-1][j-v[i]*k]+w[i]*k);
3 k = 0,1,2,3.....s[i]
```

朴素写法代码：

```
1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4
5 const int N = 110;
6 int n,m;
7 int v[N],w[N],s[N];
8 int f[N][N];
9
10 int main(){
11     cin>>n>>m;
12     for(int i=1;i<=n;i++) cin>>v[i]>>w[i]>>s[i];
13
14     for(int i=1;i<=n;i++)
15         for(int j=0;j<=m;j++)
16             for(int k=0;k<=s[i]&&k*v[i]<=j;k++)
```

```

17     f[i][j] = max(f[i][j], f[i-1][j-v[i]*k]+w[i]*k);
18     cout<<f[n][m]<<endl;
19     return 0;
20 }
```

二进制优化：

假设  $s[i] = 1023$  0, 1, 2, 3, ..., 1023 现在把它们按照2的幂次进行分组打包，每组的数量为：1, 2, 4, 8, ..., 512 这里面的数可以枚举出来0~1023内的任意一种情况，比如1, 2可以枚举0~3 加上4, 可以枚举4~7, 也就是可以枚举0~7 加上8, 可以枚举8~15, 也就是可以枚举0~15 以此类推，可以得到所有的数

每一个打包起来的第*i*个物品，可以看成0-1背包里面的一个物品（因为只能选一次），相当于我们用10个新的物品选or 不选，替代了原来的第*i*个物品的所有方案则枚举1024次-->枚举10次( logn )

S

1, 2, 4, 8, ...,  $2^k$ , c

c <  $2^{k+1}$

316343

$k$ 是从1一直加到  $2^k$  （也就是 $2^{k+1}-1$ ）不超过s的最大的k, c是一个补的数，因为如果再来一个  $2^{k+1}$  就有可能会超出s的范围而当前又无法完全凑出来s, c就是s与 $2^{k+1}$ 次方再减一的差

代码：

```

1 #include<iostream>
2 using namespace std;
3 #include<algorithm>
4
5 const int N = 25000, M = 2010;
6 //一共1000件物品, si最大是2000
```

```

7 //所以一件物品最多打包成log2000, 所以问题规模开成25000>1000*log2000
8
9 int n,m;
10 int v[N],w[N];
11 int f[N];
12
13 int main(){
14     cin>>n>>m;
15
16     int cnt = 0;//表示所有新的物品编号,也就是打包后的
17     for(int i=1;i<=n;i++){
18         int a,b,s;//物品的体积、价值、个数
19         cin>>a>>b>>s;
20         int k=1;
21         while(k<=s){
22             //k<=s就可以分
23             //每次把k个第i个物品打包到一起
24             cnt++;
25             v[cnt] = a*k;//打包后体积等于k个物品体积和
26             w[cnt] = b*k;//价值为k个物品价值和
27             s -= k;//物品i下一轮能打包的数量的上限更新为s-k
28             k *= 2;//k每次更新为原来二倍
29         }
30         //如果需要补c
31         //此时的s是剩下的, 也就是刚才的c
32         if(s>0){
33             cnt++;
34             v[cnt] = a*s;
35             w[cnt] = b*s;
36         }
37     }
38     n = cnt;//将n更新成cnt, 二进制优化,n表示打包后物品的个数
39     //转成0-1背包问题
40     for(int i=1;i<=n;i++)
41         for(int j=m;j>=v[i];j--)
42             f[j] = max(f[j],f[j-v[i]]+w[i]);
43     cout<<f[m]<<endl;
44     return 0;
45 }
```

## 分组背包问题

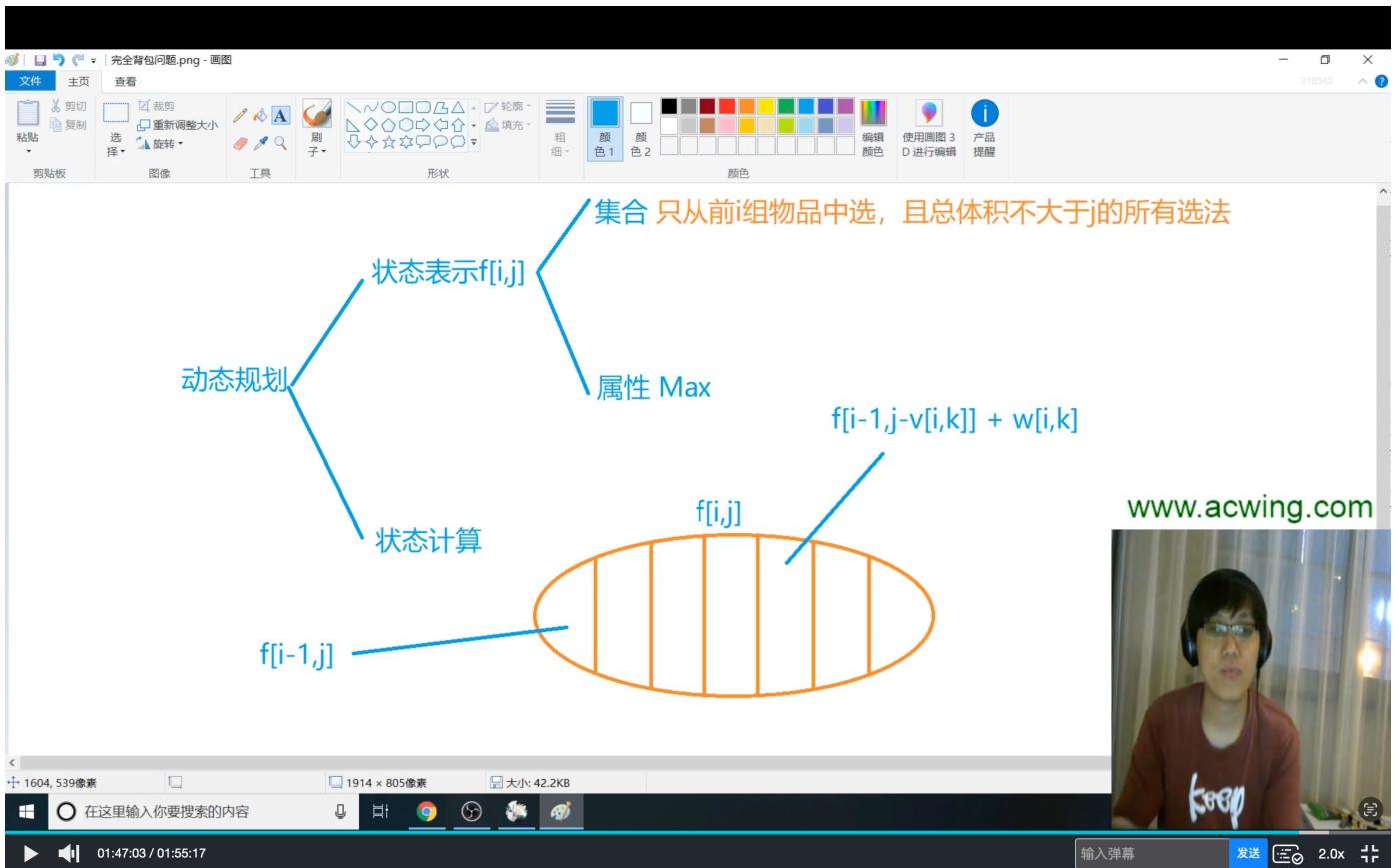
有 N 组物品和一个容量是 V 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。每件物品的体积是  $v[i][j]$ ，价值是  $w[i][j]$ ，其中  $i$  是组号， $j$  是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

注意：完全背包问题是枚举第*i*个物品选几个，而分组背包问题枚举的是第*i*组物品选哪个

第*i*组的物品如果不选则相当于  $f[i-1][j]$ ，如果选择第*k*个则相当于求解  $f[i-1,j-v[i,k]]+w[i,k]$



代码：

```

1 #include<iostream>
2 using namespace std;
3 #include<algorithm>
4
5 const int N = 110;
6 int n,m;
7 int v[N][N],w[N][N],s[N];//s存的是个数
8 int f[N];
9
10 int main(){
11     cin>>n>>m;
12
13     for(int i = 1;i <= n;i++){
14         cin>>s[i];
15         for(int j = 0;j < s[i]; j++)
16             cin>>v[i][j]>>w[i][j];
17     }
18     for(int i = 1;i <= n;i++)//枚举每一组
19         for(int j = m;j >= 0;j--)//从大到小枚举所有体积
20             for(int k = 0;k < s[i];k++)//枚举所有选择
21                 if(v[i][k] <= j)//需要v[i][k] <= j才更新
22                     f[j] = max(f[j],f[j-v[i][k]]+w[i][k]);
23
24     cout<<f[m]<<endl;
25     return 0;
26 }
```

# 线性DP

## 数字三角形

给定一个如下图所示的数字三角形，从顶部出发，在每一结点可以选择移动至其左下方的结点或移动至其右下方的结点，一直走到底层，要求找出一条路径，使路径上的数字的和最大。

|   |   |   |   |   |
|---|---|---|---|---|
| 7 |   |   |   |   |
| 3 | 8 |   |   |   |
| 8 | 1 | 0 |   |   |
| 2 | 7 | 4 | 4 |   |
| 4 | 5 | 2 | 6 | 5 |

### 输入格式

第一行包含整数  $n$ ，表示数字三角形的层数。

接下来  $n$  行，每行包含若干整数，其中第  $i$  行表示数字三角形第  $i$  层包含的整数。

### 输出格式

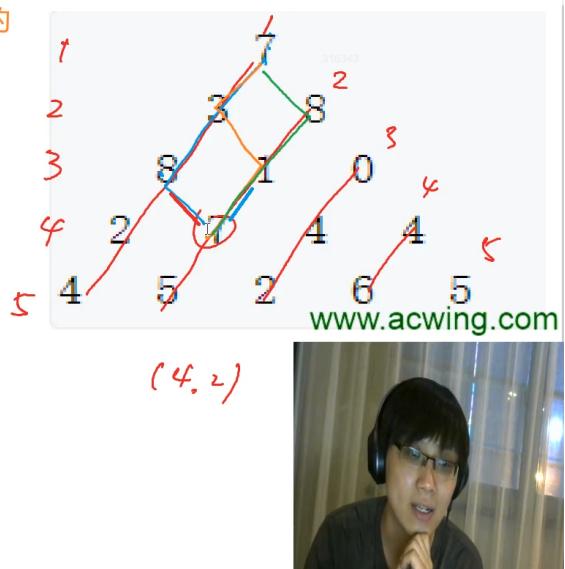
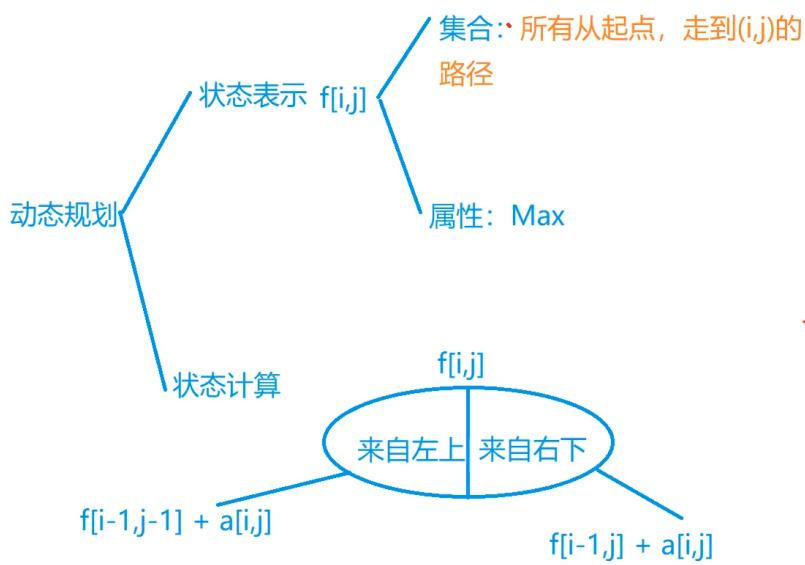
输出一个整数，表示最大的路径数字和。

### 数据范围

$1 \leq n \leq 500$ ,

$-10000 \leq$  三角形中的整数  $\leq 10000$

动态规划思路：



注：图中应为右上而不是右下

动态规划问题时间复杂度计算：状态数量\*转移计算量

```
1 #include<iostream>
2 #include<algorithm>
```

```

3
4 using namespace std;
5
6 const int N = 510, INF = 1e9;
7
8 int n;
9 int a[N][N]; //存储三角形中的元素
10 int f[N][N]; //存状态
11
12 int main()
13 {
14     cin >> n;
15     //读入元素
16     for(int i = 1; i <= n; i++)
17         for(int j = 1; j <= i; j++)
18             scanf("%d", &a[i][j]);
19     /*
20      初始化，注意这里要多初始化一列，即j<=i+1
21      由于三角形最右边的元素没有右上元素
22      如果没有初始化判断当前元素是从左上还是右上转移时会出现问题
23      此外由于会出现i-1的情况，所有存储时i从1开始初始化i从0开始
24     */
25     for(int i = 0; i <= n; i++)
26         for(int j = 0; j <= i+1; j++)
27             f[i][j] = -INF;
28
29     f[1][1] = a[1][1]; //第一个点最大值只能是它本身
30
31     //状态转移方程，从顶点下一层开始（第二层）
32     for(int i = 2; i <= n; i++)
33         for(int j = 1; j <= i; j++)
34             f[i][j] = max(f[i-1][j-1]+a[i][j], f[i-1][j]+a[i][j]);
35
36     int res = -INF;
37     //遍历最后一行找最大值
38     for(int j = 1; j <= n; j++) res = max(res, f[n][j]);
39
40     cout << res << endl;
41
42     return 0;
43 }

```

## 最长上升子序列

给定一个长度为  $N$  的数列，求数值严格单调递增的子序列的长度最长是多少。

### 输入格式

第一行包含整数  $N$ 。

第二行包含  $N$  个整数，表示完整序列。

### 输出格式

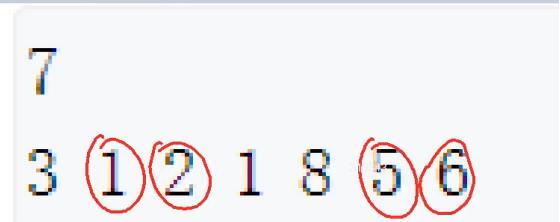
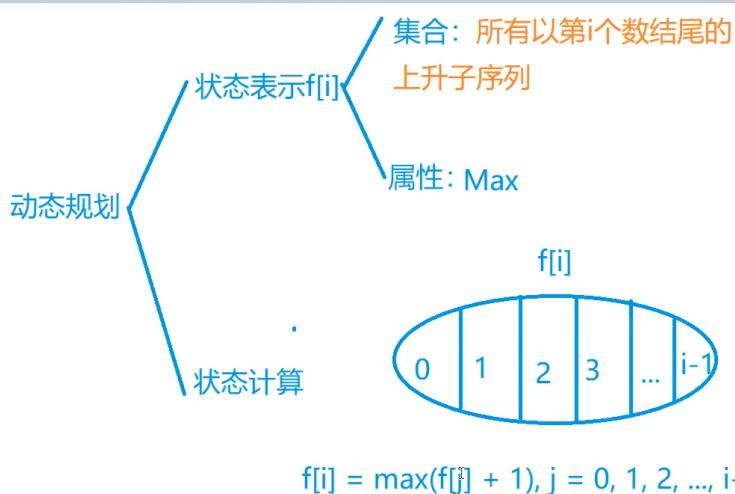
输出一个整数，表示最大长度。

### 数据范围

$1 \leq N \leq 1000$ ,  
 $-10^9 \leq$  数列中的数  $\leq 10^9$

从前到后挑数，保证数列严格递增

动态规划思路：



www.acwing.com



状态计算中按照以位置  $i$  上的数为结尾的序列中， $i$  的前一个数的位置划分。显然，位置  $i$  前的数并不一定都在序列中。

```
1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010;
7
8 int n;
9 int a[N], f[N];
10
11 int main()
12 {
13     scanf("%d", &n);
14     for(int i = 1; i <= n; i++) scanf("%d", &a[i]); // 读入序列
15 }
```

```

16     for(int i = 1;i <= n; i++)
17     {
18         f[i] = 1;//最长序列至少是1, 即仅有a[i]
19         for(int j = 1;j < i; j++)
20             if(a[j] < a[i])
21                 f[i] = max(f[i],f[j] + 1);
22             //看是否满足【上升】, 在满足的序列中取一个最长的加1
23     }
24
25     int res = 0;
26     for(int i = 1;i <= n;i++) res = max(res,f[i]);
27
28     printf("%d\n",res);
29
30 }
```

如果想要存下来这个序列，则可以额外设置一个g数组，记录下当前点是由哪个点转移过来的，输出序列正好是所求上升序列的逆序列

代码如下：

```

1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010;
7
8 int n;
9 int a[N],f[N],g[N];
10
11 int main()
12 {
13     scanf("%d",&n);
14     for(int i = 1;i <= n; i++) scanf("%d",&a[i]);//读入序列
15
16     for(int i = 1;i <= n; i++)
17     {
18         f[i] = 1;//最长序列至少是1, 即仅有a[i]
19         g[i] = 0;
20         for(int j = 1;j < i; j++)
21             if(a[j] < a[i])
22                 if(f[i] < f[j] + 1)
23                 {
24                     f[i] = f[j]+1;
25                     g[i] = j;
26                 }
27     }
28
29     int k = 1;//最长序列中结尾的下标
30     for(int i = 1;i <= n;i++)
31         if(f[k]<f[i])
```

```

32     k = i;
33
34     printf("%d\n", f[k]);
35
36     for(int i = 0, len = f[k]; i < len; i++)
37     {
38         printf(" %d ", a[k]);
39         k = g[k];
40     }
41
42     return 0;
43 }
```

输入输出如下：

输入

```

7
3 1 2 1 8 5 6
```

输出

```

4
6 5 2 1
```

## 优化版最长上升子序列

示例：3121856

观察3和第一个1，如果序列中有数字可以接到3的后面，由于 $3>1$ ，则这个数一定可以接到1的后面，但是能接到1的后面未必能接到3的后面，依次来看第一个3并没有继续计算的必要，直接删去并不会影响计算的结果

对子序列集合按照长度进行分类，长度为1的一类，长度为2的一类，依次类推，每一类保留一个结尾数字最小的即可。显然，存下来的序列长度越长，结尾的数字越大

对于当前第*i*个位置上的元素a，它一定可以接到上述序列所有比自己小的末尾上去，由于序列长度正比于序列末尾值大小，所以若想整体长度最长，a最好接到比它小且最大的序列后面

```

1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 100010;
7
8 int n;
9 int a[N];//存序列中的数
```

```
10 int q[N];//存依长度划分的序列的结尾元素
11
12 int main()
13 {
14     scanf("%d",&n);
15     for(int i = 0; i < n;i++) scanf("%d",&a[i]);//读入序列
16
17     int len = 0;//初始长度为0
18     for(int i = 0;i < n;i ++)//遍历序列每一个元素
19     {
20         /*
21             用二分法来查找那个最大的小于a[i]的数，然后用它来覆盖他之后的那个长度的最小值
22             举例，如二分查找到的最大的小于a[i]的值是4，而5一定是大于或等于他的，
23             此时又因为a[i]加到了长度为4子序列，让他的长度变成了5，
24             则此时最小的长度为5的子序列结尾应该就是a[i]，那么就得把5的结尾更新成a[i]
25         */
26         int l = 0,r = len;//r=len保证了二分不会查到当前点后面的小于a[i]的数
27         while(l < r)
28         {
29             int mid = l + r + 1 >> 1;
30             if(q[mid] < a[i]) l = mid;
31             else r = mid - 1;
32         }
33         len = max(len,r + 1);
34         q[r + 1] = a[i];//更新结尾最小值
35     }
36
37     printf("%d\n",len);
38
39     return 0;
40 }
```

## 最长公共子序列

给定两个长度分别为  $N$  和  $M$  的字符串  $A$  和  $B$ , 求既是  $A$  的子序列又是  $B$  的子序列的字符串长度最长是多少。

### 输入格式

第一行包含两个整数  $N$  和  $M$ 。

第二行包含一个长度为  $N$  的字符串, 表示字符串  $A$ 。

第三行包含一个长度为  $M$  的字符串, 表示字符串  $B$ 。

字符串均由小写字母构成。

### 输出格式

输出一个整数, 表示最大长度。

### 数据范围

$1 \leq N, M \leq 1000$

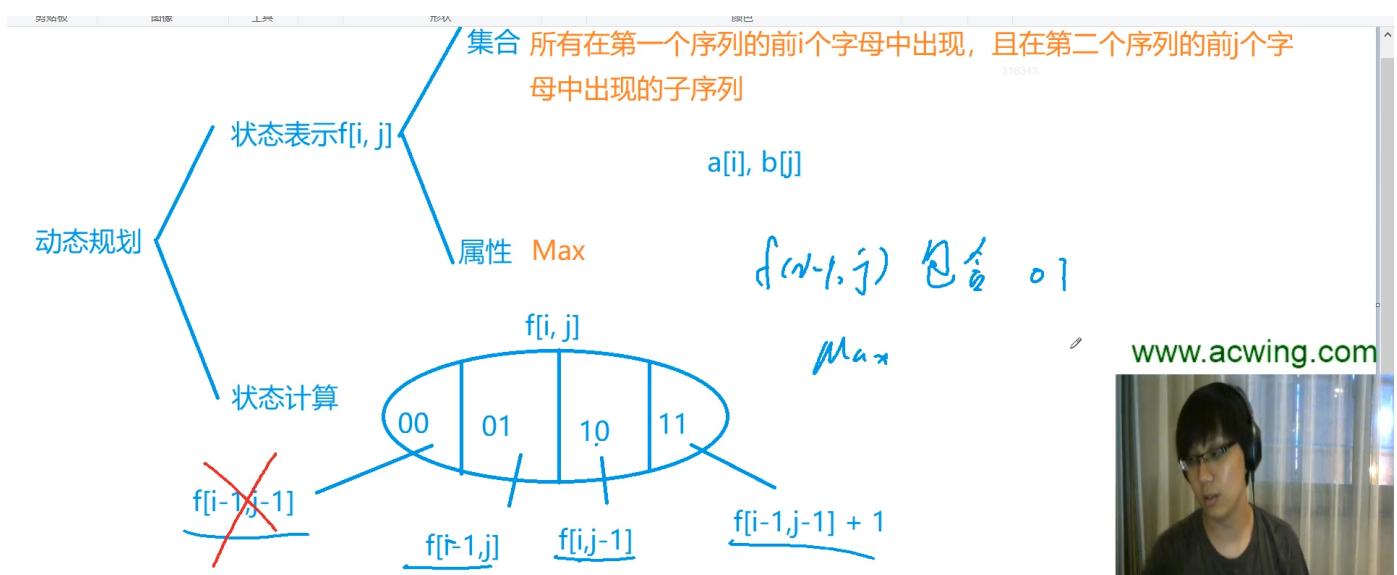
一个序列, 既是  $A$  的字串也是  $B$  的字串, 且长度最长

动态规划思路:

状态计算中按照  $a[i]$  和  $b[j]$  是否在序列中进行分类, 00 表示都不在, 11 表示都在

一般不写 00 的情况, 因为它会包含在后面几种情况之中

01 和 10 的情况是  $f[i-1][j]$  or  $f[i][j-1]$  两种情况集合的其中一个元素



代码:

```
1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010;
7
8 int n,m;
9 char a[N],b[N]; //存两个字符串
```

```

10 int f[N][N]; //状态
11
12 int main()
13 {
14     scanf("%d%d", &n, &m);
15     scanf("%s%s", a + 1, b + 1); //由于会出现i-1的情况，下标从1开始
16
17     for(int i = 1; i <= n; i++)
18         for(int j = 1; j <= m; j++)
19         {
20             f[i][j] = max(f[i-1][j], f[i][j-1]); //11情况不一定存在，先算01和10情况的最大值
21             if(a[i] == b[j]) f[i][j] = max(f[i][j], f[i-1][j-1] + 1);
22         }
23     printf("%d", f[n][m]);
24     return 0;
25 }
```

## 最短编辑距离

给定两个字符串  $A$  和  $B$ ，现在要将  $A$  经过若干操作变为  $B$ ，可进行的操作有：

1. 删除—将字符串  $A$  中的某个字符删除。
2. 插入—在字符串  $A$  的某个位置插入某个字符。
3. 替换—将字符串  $A$  中的某个字符替换为另一个字符。

现在请你求出，将  $A$  变为  $B$  至少需要进行多少次操作。

### 输入格式

第一行包含整数  $n$ ，表示字符串  $A$  的长度。

第二行包含一个长度为  $n$  的字符串  $A$ 。

第三行包含整数  $m$ ，表示字符串  $B$  的长度。

第四行包含一个长度为  $m$  的字符串  $B$ 。

字符串中均只包含大小写字母。

### 输出格式

输出一个整数，表示最少操作次数。

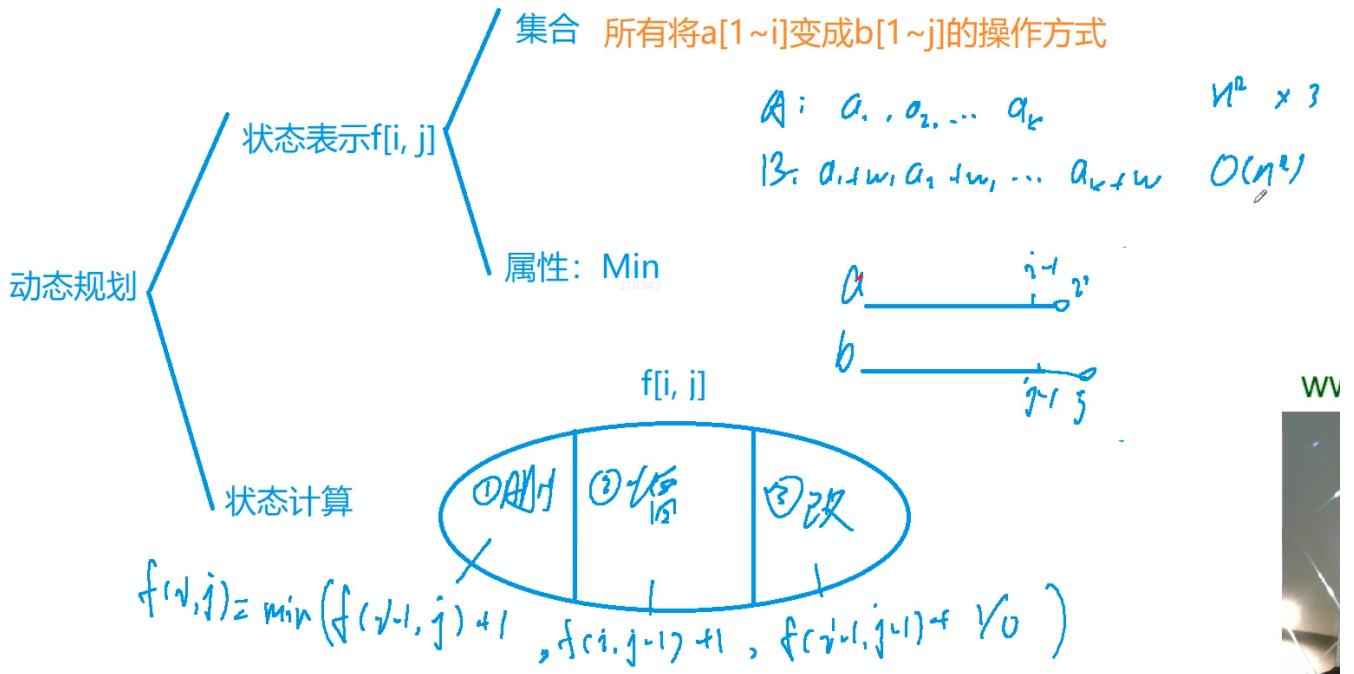
### 数据范围

$$1 \leq n, m \leq 1000$$

动态规划思路：

状态计算根据最后一步进行的操作进行分类

修改操作中，如果  $a[i] = b[j]$  则不需要加一



代码：

```
1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010;
7
8 int n, m;
9 char a[N], b[N];
10 int f[N][N];
11
12 int main()
13 {
14     scanf("%d%s", &n, a + 1);
15     scanf("%d%s", &m, b + 1);
16
17     // 初始化
18     for(int j = 0; j <= m; j++) f[0][j] = j; // A串中没有元素，B串中j个元素，想要匹配只能j次增加
19     for(int i = 0; i <= n; i++) f[i][0] = i; // A中i个元素，B中没有元素，想要匹配只能i次删除
20
21     for(int i = 1; i <= n; i++)
22         for(int j = 1; j <= m; j++)
23         {
24             f[i][j] = min(f[i-1][j] + 1, f[i][j-1] + 1);
25             if(a[i] == b[j]) f[i][j] = min(f[i][j], f[i-1][j-1]);
26             else f[i][j] = min(f[i][j], f[i-1][j-1] + 1);
27         }
28     printf("%d\n", f[n][m]);
```

```
29
30     return 0;
31 }
```

## 编辑距离

给定  $n$  个长度不超过 10 的字符串以及  $m$  次询问，每次询问给出一个字符串和一个操作次数上限。

对于每次询问，请你求出给定的  $n$  个字符串中有多少个字符串可以在上限操作次数内经过操作变成询问给出的字符串。

每个对字符串进行的单个字符的插入、删除或替换算作一次操作。

### 输入格式

第一行包含两个整数  $n$  和  $m$ 。

接下来  $n$  行，每行包含一个字符串，表示给定的字符串。

再接下来  $m$  行，每行包含一个字符串和一个整数，表示一次询问。

字符串中只包含小写字母，且长度均不超过 10。

### 输出格式

输出共  $m$  行，每行输出一个整数作为结果，表示一次询问中满足条件的字符串个数。

### 数据范围

$1 \leq n, m \leq 1000$ ,

代码：

```
1 #include<iostream>
2 #include<algorithm>
3 #include<string.h>
4
5 using namespace std;
6
7 const int N = 15, M = 1010;
8
9 int n,m;
10 char a[N],b[N];
11 int f[N][N];
12 char str[M][N];
13
14 //最短编辑距离
15 int edit_distance(char a[],char b[])
16 {
17     int la = strlen(a + 1),lb = strlen(b + 1);
18     for(int i = 0;i <= lb;i++) f[0][i] = i;
19     for(int i = 0;i <= la;i++) f[i][0] = i;
20
21     for(int i = 1;i <= la;i++)
22         for(int j = 1;j <= lb;j++)
23             if(a[i] == b[j]) f[i][j] = f[i - 1][j - 1];
24             else f[i][j] = min(f[i - 1][j],f[i][j - 1]) + 1;
25 }
```

```

22     for(int j = 1;j <= lb;j++)
23     {
24         f[i][j] = min(f[i-1][j] + 1,f[i][j-1] + 1);
25         if(a[i] == b[j]) f[i][j] = min(f[i][j],f[i-1][j-1]);
26         else f[i][j] = min(f[i][j],f[i-1][j-1] + 1);
27     }
28     return f[la][lb];
29 }
30
31 int main()
32 {
33     scanf("%d%d",&n,&m);
34     for(int i = 0;i < n;i++) scanf("%s",str[i] + 1); //读入字符串
35
36     while(m--)
37     {
38         int limit;
39         char s[N];
40         scanf("%s%d",s + 1,&limit);
41         int res = 0;
42         for(int i = 0;i < n;i++)
43             if(edit_distance(str[i],s) <= limit)
44                 res++;
45
46         printf("%d\n",res);
47     }
48
49     return 0;
50 }
```

## 区间DP

区间dp问题状态表示的时候一般是某一个区间

## 石子合并

设有  $N$  堆石子排成一排，其编号为  $1, 2, 3, \dots, N$ 。

每堆石子有一定的质量，可以用一个整数来描述，现在要将这  $N$  堆石子合并成为一堆。

每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。

例如有 4 堆石子分别为  $1\ 3\ 5\ 2$ ，我们可以先合并 1、2 堆，代价为 4，得到  $4\ 5\ 2$ ，又合并 1、2 堆，代价为 9，得到  $9\ 2$ ，再合并得到 11，总代价为  $4 + 9 + 11 = 24$ ；

如果第二步是先合并 2、3 堆，则代价为 7，得到  $4\ 7$ ，最后一次合并代价为 11，总代价为  $4 + 7 + 11 = 22$ 。

问题是：找出一种合理的方法，使总的代价最小，输出最小代价。

### 输入格式

第一行一个数  $N$  表示石子的堆数  $N$ 。

第二行  $N$  个数，表示每堆石子的质量(均不超过 1000)。

### 输出格式

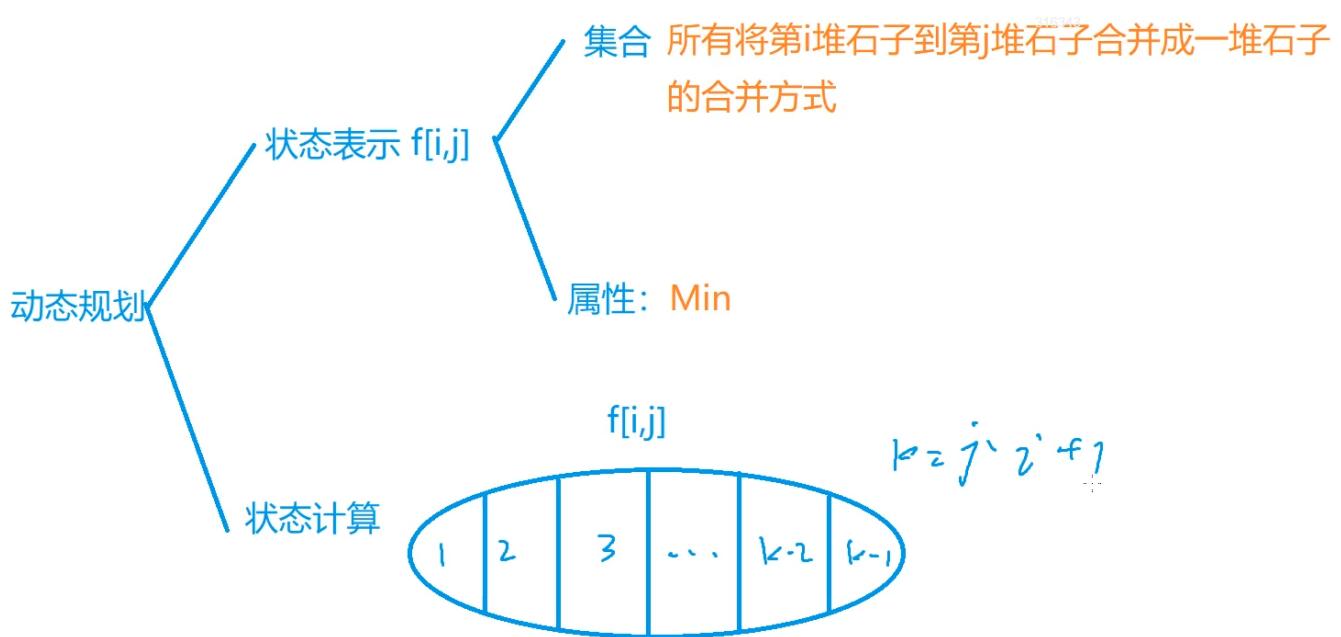
输出一个整数，表示最小代价。

### 数据范围

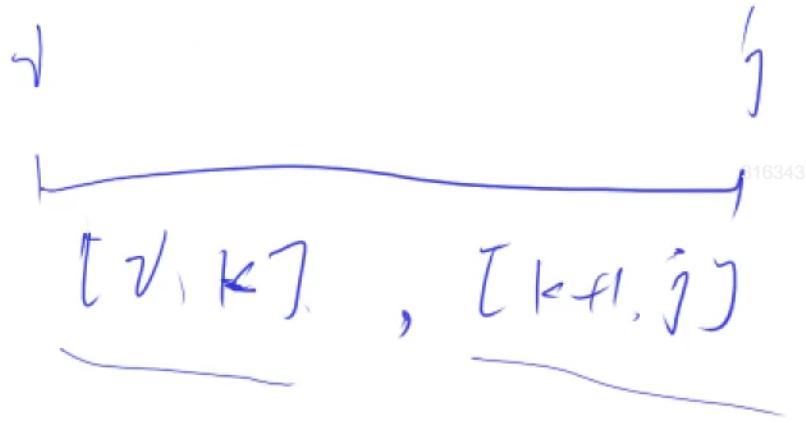
$1 \leq N \leq 300$

动态规划思路：

状态计算以最后一次分界线的位置来分类



从  $k$  处分割代价：



$$f[l, k] + f[l+1, j] + s[j] - s[l-1]$$

代码：

```

1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 310, INF = 1e8;
7
8 int n;
9 int s[N];
10 int f[N][N];
11
12 int main()
13 {
14     scanf("%d", &n);
15     for(int i = 1; i <= n; i++) scanf("%d", &s[i]); // 读入每堆石子质量
16
17     for(int i = 1; i <= n; i++) s[i] += s[i-1]; // 前缀和
18
19     for(int len = 2; len <= n; len++) // 从小到大枚举区间长度 (len=1为边界情况, 合并不需要体力)
20         for(int i = 1; i + len - 1 <= n; i++) // 枚举区间的起点
21         {
22             int l = i, r = i + len - 1; // 区间左右端点
23             f[l][r] = INF; // 初始化
24             for(int k = l; k < r; k++)
25                 f[l][r] = min(f[l][r], f[l][k] + f[k+1][r] + s[r] - s[l-1]);
26         }
27
28     printf("%d\n", f[1][n]);
29
30     return 0;
31 }
```

# 计数类DP

## 整数划分

一个正整数  $n$  可以表示成若干个正整数之和，形如： $n = n_1 + n_2 + \dots + n_k$ , 其中  $n_1 \geq n_2 \geq \dots \geq n_k, k \geq 1$ 。

我们将这样的一种表示称为正整数  $n$  的一种划分。

现在给定一个正整数  $n$ , 请你求出  $n$  共有多少种不同的划分方法。

### 输入格式

共一行，包含一个整数  $n$ 。

### 输出格式

共一行，包含一个整数，表示总划分数量。

由于答案可能很大，输出结果请对  $10^9 + 7$  取模。

### 数据范围

$1 \leq n \leq 1000$

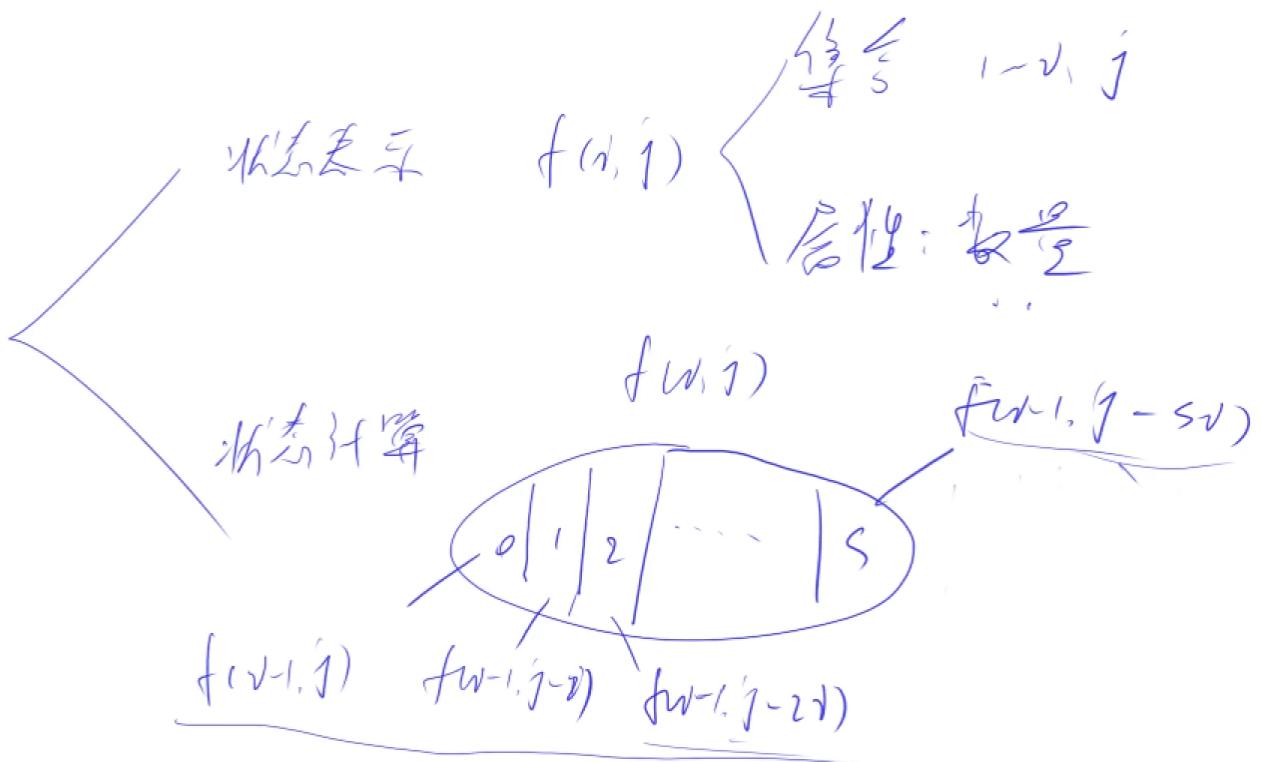
解法一：

转化成一个完全背包问题，背包容量是  $n$ , 物品的体积分别是 1 到  $n$ , 每种物品有无限个，问恰好装满背包的方案数

动态规划思路：

集合表示从 1-i 中选，体积恰好是 j 的选法的数量

状态计算按最后一个物品 i 选了多少个划分



优化一下：

```

1 f[i][j] = f[i - 1][j] + f[i - 1][j - i] + f[i - 1][j - i * 2] + ... + f[i - 1][j - i * s]
2 f[i][j - i] =           f[i - 1][j - i] + f[i - 1][j - i * 2] + ... + f[i - 1][j - i * s]

```

观察发现 $f[i][j-i]$ 和 $f[i][j]$ 后半部分完全相同，因此可以用它替换后半部分

再将二维变成一维，体积从小到大循环

```

1 f[i][j] = f[i-1][j] + f[i][j-1]
2 f[j] = f[j] + f[j-1]

```

完全背包方法代码：

$f[j]$ 表示选择物品体积和是 $j$ 的所有选法

```

1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010, mod = 1e9+7;
7
8 int n;
9 int f[N];
10
11 int main()
12 {
13     cin >> n;

```

```

14
15     f[0] = 1; // 初始化，如果 n = 0，则什么数都不选的时候为一种方案
16     // 否则对于其余数，什么数都不选不可能组成该数，初始值为0
17     for(int i = 1; i <= n; i++)
18         for(int j = i; j <= n; j++)
19             f[j] = (f[j] + f[j - i]) % mod;
20
21     cout << f[n] << endl;
22
23     return 0;
24 }
25

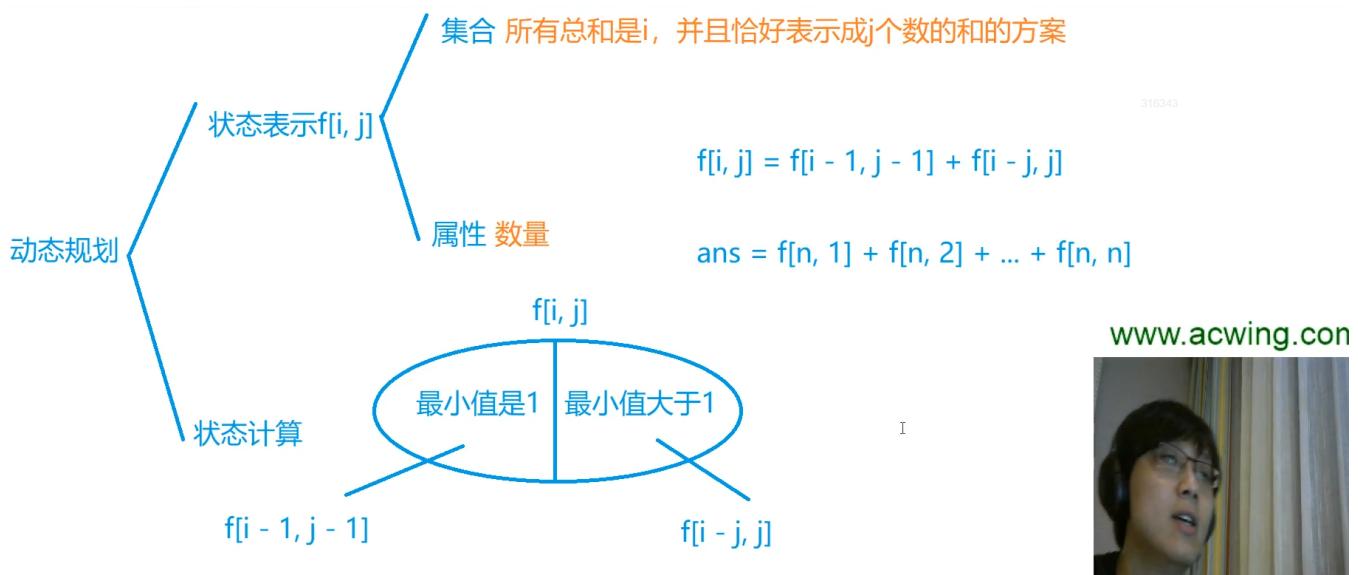
```

解法二：与解法一中的状态表示不相同

状态计算中按照  $j$  个数组成总和  $i$  的那  $j$  个数中的最小值是否为1进行划分

最小值是1的时候，先不考虑这个1，相当于求  $f[i-1][j-1]$  的个数，在此基础上加一就可以保证总和是  $i$ ，所以两者本质上方案数是一样的

最小值大于1的时候，将里面的每一个数减去一，此时每个数仍为正整数。相当于总和变为  $i-j$ ，仍由  $j$  个数组成



代码：

```

1 #include<iostream>
2 #include<algorithm>
3
4 using namespace std;
5
6 const int N = 1010, mod = 1e9+7;
7
8 int n;
9 int f[N][N];
10
11 int main()

```

```

12 {
13     cin >> n;
14
15     f[0][0] = 1; //总和为0, 由0个数组成, 方案数为1
16
17     for(int i = 1;i <= n;i++)
18         for(int j = 1;j <= i;j++) //和为i则最多由i个数组成(i个1)
19             f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % mod;
20
21     //枚举总和为n, 由1~n个数组成的方案数总和
22     int ans = 0;
23     for(int i = 1;i <= n; i++) ans = (ans+f[n][i]) % mod;
24
25     cout << ans << endl;
26
27 }
28

```

# 数位统计DP

## 计数问题

给定两个整数  $a$  和  $b$ , 求  $a$  和  $b$  之间的所有数字中  $0 \sim 9$  的出现次数。

例如,  $a = 1024$ ,  $b = 1032$ , 则  $a$  和  $b$  之间共有 9 个数如下:

1024 1025 1026 1027 1028 1029 1030 1031 1032

其中 0 出现 10 次, 1 出现 10 次, 2 出现 7 次, 3 出现 3 次等等...

### 输入格式

输入包含多组测试数据。

每组测试数据占一行, 包含两个整数  $a$  和  $b$ 。

当读入一行为 0 0 时, 表示输入终止, 且该行不作处理。

### 输出格式

每组数据输出一个结果, 每个结果占一行。

每个结果包含十个用空格隔开的数字, 第一个数字表示 0 出现的次数, 第二个数字表示 1 出现的次数, 以此类推。

### 数据范围

$0 < a, b < 1000000000$

题目需要分情况讨论,

求 $[a,b]$ 中出现0的次数，需要实现一个 $\text{count}(n, x)$ 函数，这个函数表示从1到n中出现x的个数，则题目所求转化成求 $\text{count}(b,x) - \text{count}(a-1,x)$ ，思想同前缀和

举例：

$1 \sim n, x = 1$

318343

$n = abcdefg$

分别求出1在每一位上出现的次数

求1在第4位上出现的次数

$1 \leq xxx1yyy \leq abcdefg$

(1)  $xxx = 000 \sim abc - 1$ ,  $yyy = 000 \sim 999$ ,  $abc * 1000$

(2)  $xxx = abc$

(2.1)  $d < 1$ ,  $abc1yyy > abc0efg, 0$

(2.2)  $d = 1$ ,  $yyy = 000 \sim efg, efg + 1$

(2.3)  $d > 1$ ,  $yyy = 000 \sim 999, 1000$

当枚举情况处于最高位时，情况 (1) 不存在

对于情况 (1)，考虑一种边界情况，现在求解0在第四位出现的次数，如果 $xxx = 000$ ，此时前四位都是0，组成一个数的前导0，实际上并不会被写出来，则 $xxx$ 应该从001开始计数，此时情况如下：

$n = \overbrace{abc}^{\text{d} < x}, \overbrace{defg}^{\text{d} \geq x}$        $x > 0$   
 $x = 0$

①  $000 \sim abc - 1, x, 000 \sim 999.$        $abc \times 1000$

②  $abc, x$ 

- (2.1)  $d < x, 0$
- (2.2)  $d = x, 000 \sim efg, efg + 1$
- (2.3)  $d > x, 000 \sim 999$        $1000$

代码：

```

1 #include<iostream>
2 #include<algorithm>
3 #include<cstring>
4 #include<vector>
5
6 using namespace std;
7
8 //求情况(1) 当前位(d) 前所有位构成数的值(abc的值)
9 int get(vector<int> num,int l,int r)
10 {
11     int res = 0;
12     for(int i = l;i >= r;i--) res = res *10 + num[i];
13     return res;
14 }
15
16 //求10的x次方
17 int power10(int x)
18 {
19     int res = 1;
20     while(x--) res *= 10;
21     return res;
22 }
23
24 //这个函数表示从1~n中出现x的个数
25 int count(int n,int x)
26 {
27     if(!n) return 0;//n = 0, 1~0之间没有数
28 }
```

```

29     //存每一位
30     vector<int> num;
31     while(n)
32     {
33         num.push_back(n % 10);
34         n /= 10;
35     }
36
37     n = num.size(); //n = 位数
38
39     int res = 0; //总共出现的次数
40     for(int i = n - 1 - !x;i >= 0;i --) //从第一位开始枚举,x = 0时从第二位开始枚举
41     {
42         if(i < n - 1) //此时情况1才存在
43         {
44             res += get(num,n - 1,i + 1) * power10(i);
45             if(!x) res -= power10(i);
46         }
47         if(num[i] == x) res += get(num, i - 1, 0) + 1; //情况2.2
48         else if(num[i] > x) res += power10(i); //情况2.3
49     }
50     return res;
51 }
52
53 int main()
54 {
55     int a,b;
56     while(cin >> a >> b,a)
57     {
58         if(a > b) swap(a,b); //给数的顺序不一定是前小后大的
59
60         for(int i = 0;i < 10;i++)
61             cout << count(b,i) - count(a - 1,i) << ' ';
62         cout << endl;
63     }
64
65     return 0;
66 }
67

```

参考网站上某同学的代码；

```

1 #include <iostream>
2 #include <cstring>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6 /*
7 */
8 */
9

```

```

10 int get(vector<int> num,int l,int r)//因为我们举的分类中，有需要求出一串数字中某个区间的数字，  

11 //例如abcdefg有一个分类需要求出efg+1  

12 {  

13     int res=0;  

14     for(int i=l;i>=r;i--) res=res*10+num[i];//这里从小到大枚举是因为下面count的时候读入数据  

15 //是从最低位读到最高位，那么此时在num里，最高位存的就是数字的最低位，那么假如我们要求efg，那就是从2算到  

16 0  

17     return res;  

18 }  

19  

20 int power10(int i)//这里有power10是因为有一个分类需要求得十次方的值，例如abc*10^3  

21 {  

22     int res=1;  

23     while(i--) res*=10;  

24     return res;  

25 }  

26  

27 int count(int n,int x)  

28 {  

29     vector<int> num;//num用来存储数中每一位的数字  

30     while(n)  

31     {  

32         num.push_back(n%10);//get里有解释  

33         n/=10;  

34     }  

35     n=num.size();//得出他的长度  

36     int res=0;  

37     for(int i=n-1-x;i>=0;i--)//这里需要注意，我们的长度需要减一，是因为num是从0开始存储，而长  

38 度是元素的个数，因此需要减1才能读到正确的数值，而! x出现的原因是因为我们不能让前导零出现，如果此时需要  

39 我们列举的是0出现的次数，那么我们自然不能让他出现在第一位，而是从第二位开始枚举  

40     {  

41         if(i<n-1)//其实这里可以不用if判断，因为for里面实际上就已经达成了if的判断，但为了方便理解  

42         还是加上if来理解，这里i要小于n-1的原因是因为我们不能越界只有7位数就最高从七位数开始读起  

43         {  

44             res+=get(num,n-1,i+1)*power10(i);//这里就是第一个分类，000~abc-1，那么此时情况个  

45             数就会是abc*10^3，这里的3取决于后面efg的长度，假如他是efgh，那么就是4  

46             //这里的n-1，i-1，自己将数组列出来然后根据分类标准就可以得出为什么1是n-1，r是i-1  

47             if(!x) res-=power10(i);//假如此时我们要列举的是0出现的次数，因为不能出现前导零，这  

48             样是不合法也不符合我们的分类情况，例如abcdefg我们列举d，那么他就得从001~abc-1，这样就不会直接到  

49             efg，而是会到0efg，因为前面不是前导零，自然就可以列举这个时候0出现的次数，所以要减掉1个power10  

50             }  

51             //剩下的这两个就直接根据分类标准来就好了  

52             if(num[i]==x) res+=get(num,i-1,0)+1;  

53             else if(num[i]>x) res+=power10(i);  

54         }  

55         return res;//返回res，即出现次数  

56     }  

57  

58 int main()  

59 {  

60     int a,b;  

61     while(cin>>a>>b,a)//读入数据，无论a，b谁是0，都是终止输入，因为不会有数字从零开始 (a, b>0)

```

```
53     {
54         if(a>b) swap(a,b); //因为我们需要从小到大，因此如果a大于b，那么就得交换，使得a小于b
55         for(int i=0;i<=9;i++) //列举a和b之间的所有数字中 0~9的出现次数
56             cout<<count(b,i)-count(a-1,i)<<' ';//这里有点类似前缀和，要求a和b之间，那么就先求0到
57             a i出现的次数，再求0到b i出现的次数，最后再相减就可以得出a和b之间i出现的次数
58             cout<<endl;
59     }
60     return 0;
61 }
62 作者: yxc
63 链接: https://www.acwing.com/activity/content/code/content/64211/
64 来源: AcWing
65 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

# 状态压缩DP

## 蒙德里安的梦想

求把  $N \times M$  的棋盘分割成若干个  $1 \times 2$  的长方形，有多少种方案。

例如当  $N = 2, M = 4$  时，共有 5 种方案。当  $N = 2, M = 3$  时，共有 3 种方案。

如下图所示：



### 输入格式

输入包含多组测试用例。

每组测试用例占一行，包含两个整数  $N$  和  $M$ 。

当输入用例  $N = 0, M = 0$  时，表示输入终止，且该用例无需处理。

### 输出格式

每个测试用例输出一个结果，每个结果占一行。

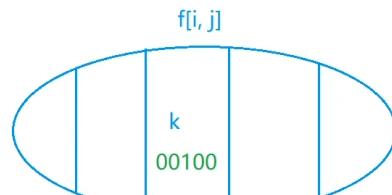
### 数据范围

$1 \leq N, M \leq 11$

动态规划思路：



状态计算:



$f[i-1, k]$

总方案数, 等于只放横着的小方块的合法方案数。

如何判断, 当前方案是否合法? 所有剩余位置, 能否填充充满竖着的小方块。可以按列来看, 每一列内部所有连续的空着的小方块, 需要是偶数个。

$f[m, 0]$

(1)  $(j \& k) == 0$

(2) 所有连续空着的位置的长度必须是偶数

www.acwing.com



状态表示中j表示, 前*i*-1列已经摆好从*i*-1列伸到第*i*列的所有状态, 它是一个二进制数, 伸出来则表示成1, 没有伸出来则表示成0

状态计算一般依据最后一步操作来分,

1. ( $i-1 \sim i$ ) 是已经固定的;
2. ( $i-2 \sim i-1$ ) 是可变的

所以划分集合的时候是根据( $i-2$ )伸到( $i-1$ )的不同状态(也就是k)划分的。 (k表示的是 $i-2$ 伸到 $i-1$ 列的所有状态)

一共有 $2^n$ 次方种情况, 每种情况都是一个二进制数, 如绿色方块表示的状态就是00100

k, j能合法拼在一起的条件是:

1. j和k不能在同一行重叠
2. 第*i*-1列空着的位置必须能被 $2 \times 1$ 的方格填满

将所有不产生冲突的累加到一块, 最后求的是 $f[m, 0]$  (列的计数是从0开始的), 也就是从第0列开始到第m-1列依据摆好, 没有从第m-1列伸到m列的方块。

```

1 #include <cstring>
2 #include <iostream>
3 #include <algorithm>
4 #include <vector>
5
6 using namespace std;
7
8 typedef long long LL;
9
10 const int N = 12, M = 1 << N;
11
12 int n, m;
13 LL f[N][M];
14 vector<int> state[M]; //所有合法状态
15 bool st[M]; //判断状态是否合法, 当前连续空格是否是偶数个

```

```

16
17 int main()
18 {
19     while (cin >> n >> m, n || m)
20     {
21         //预处理st数组
22         for (int i = 0; i < 1 << n; i++)
23         {
24             int cnt = 0;//cnt表示0的个数
25             bool is_valid = true;
26             for (int j = 0; j < n; j++)
27                 if (i >> j & 1)//当前位为1
28                 {
29                     if (cnt & 1)//奇数个连续空格
30                     {
31                         is_valid = false;
32                         break;
33                     }
34                     cnt = 0;
35                 }
36                 else cnt++;//当前位为0
37             if (cnt & 1) is_valid = false;//最后一段0为奇数个
38             st[i] = is_valid;
39         }
40         //枚举每种合法状态
41         for (int i = 0; i < 1 << n; i++)
42         {
43             state[i].clear();
44             for (int j = 0; j < 1 << n; j++)
45                 if ((i & j) == 0 && st[i | j])
46                     state[i].push_back(j);
47         }
48
49         memset(f, 0, sizeof f);//清空状态
50         f[0][0] = 1;
51         for (int i = 1; i <= m; i++)
52             for (int j = 0; j < 1 << n; j++)
53                 for (auto k : state[j])
54                     f[i][j] += f[i - 1][k];
55
56         cout << f[m][0] << endl;
57     }
58
59     return 0;
60 }
```

## 最短Hamilton路径

给定一张  $n$  个点的带权无向图，点从  $0 \sim n - 1$  标号，求起点  $0$  到终点  $n - 1$  的最短 Hamilton 路径。

Hamilton 路径的定义是从  $0$  到  $n - 1$  不重不漏地经过每个点恰好一次。

### 输入格式

第一行输入整数  $n$ 。

接下来  $n$  行每行  $n$  个整数，其中第  $i$  行第  $j$  个整数表示点  $i$  到  $j$  的距离（记为  $a[i, j]$ ）。

对于任意的  $x, y, z$ ，数据保证  $a[x, x] = 0$ ， $a[x, y] = a[y, x]$  并且  $a[x, y] + a[y, z] \geq a[x, z]$ 。

### 输出格式

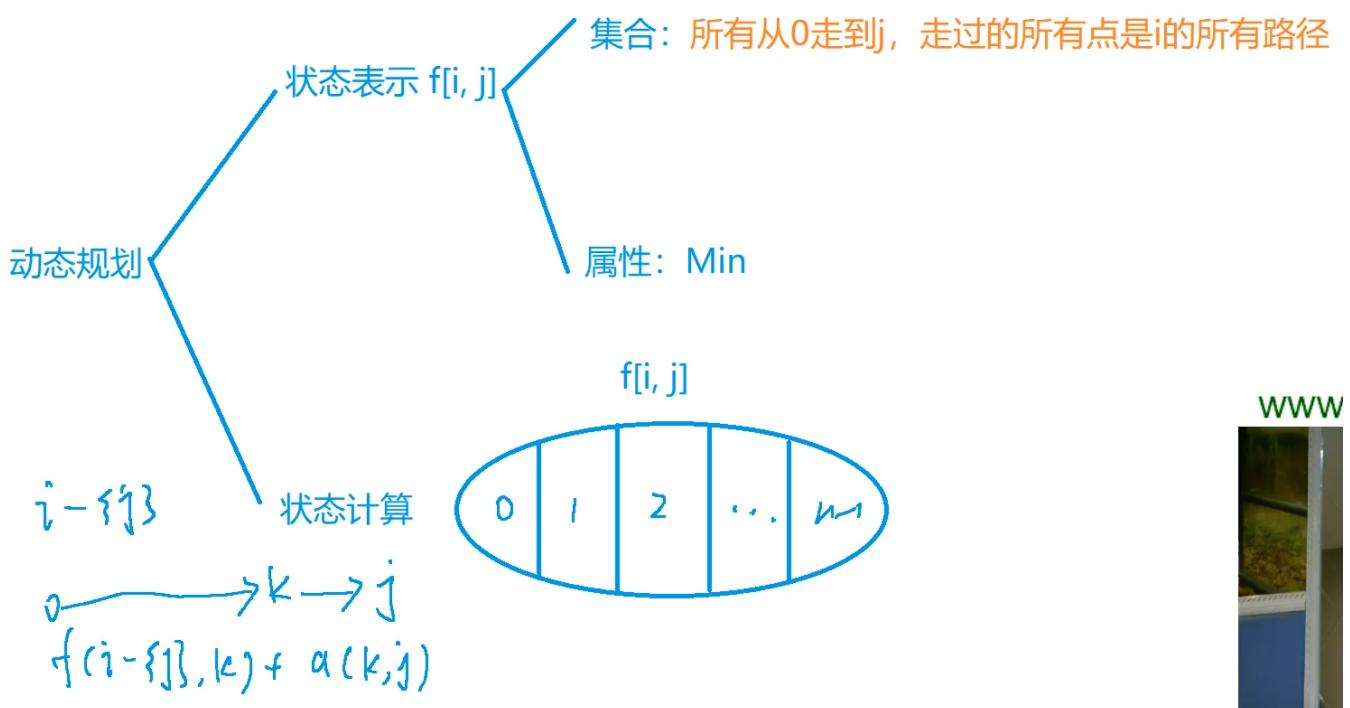
输出一个整数，表示最短 Hamilton 路径的长度。

### 数据范围

$$1 \leq n \leq 20$$

$$0 \leq a[i, j] \leq 10^7$$

动态规划思路：



$i$  是一个压缩的状态，用二进制表示，这个二进制数中的每一位表示某个点是否已经走过了

状态计算分类按照倒数第二个点是哪一个点来分类

假设倒数第二个点为  $k$ ，则路径为  $0 \dots k - j$ ， $k - j$  走的是边  $kj$ ，要想最短则需要  $0 - k$  最短，也就是需要计算从  $0$  走到  $k$ ，并且经过了  $i$  除去  $j$  的点，的最小值

```
1 #include<iostream>
2 #include<algorithm>
3 #include<cstring>
4
5 using namespace std;
```

```

6
7 const int N = 20, M = 1 << N;
8
9 int n;
10 int w[N][N]; //存图每一条边的权重
11 int f[M][N];
12
13 int main()
14 {
15     //读数据
16     cin >> n;
17     for(int i = 0; i < n; i++)
18         for(int j = 0; j < n; j++)
19             scanf("%d", &w[i][j]);
20
21     //初始化
22     memset(f, 0x3f, sizeof f); //初始化所有状态值为正无穷
23     f[1][0] = 0; //从0这个点走到0， 经过一个点， 路径长度为0
24
25     //遍历每一种情况
26     for(int i = 0; i < 1 << n; i++)
27         for(int j = 0; j < n; j++)
28             if(i >> j & 1) //从0走到j， 经过的点存在i里，则i里一定有j
29                 for(int k = 0; k < n; k++) //遍历， 从倒数第二个点转移到j
30                     if((i - (1 << j) >> k & 1)) //倒数第二个点从k转移过来，则i除去j后要包含k
31                         f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j]);
32
33
34     cout << f[(1 << n) - 1][n - 1] << endl; //每一位都走过 (i每一位都是1)， 并且落到了n-1 (终点)
35
36
37     return 0;
38
39 }

```

# 树形DP

## 没有上司的舞会

Ural 大学有  $N$  名职员，编号为  $1 \sim N$ 。

他们的关系就像一棵以校长为根的树，父节点就是子节点的直接上司。

每个职员有一个快乐指数，用整数  $H_i$  给出，其中  $1 \leq i \leq N$ 。

现在要召开一场周年庆宴会，不过，没有职员愿意和直接上司一起参会。

在满足这个条件的前提下，主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个最大值。

### 输入格式

第一行一个整数  $N$ 。

接下来  $N$  行，第  $i$  行表示  $i$  号职员的快乐指数  $H_i$ 。

接下来  $N - 1$  行，每行输入一对整数  $L, K$ ，表示  $K$  是  $L$  的直接上司。（注意一下，后一个数是前一个数的父节点，不要搞反）。

### 输出格式

输出最大的快乐指数。

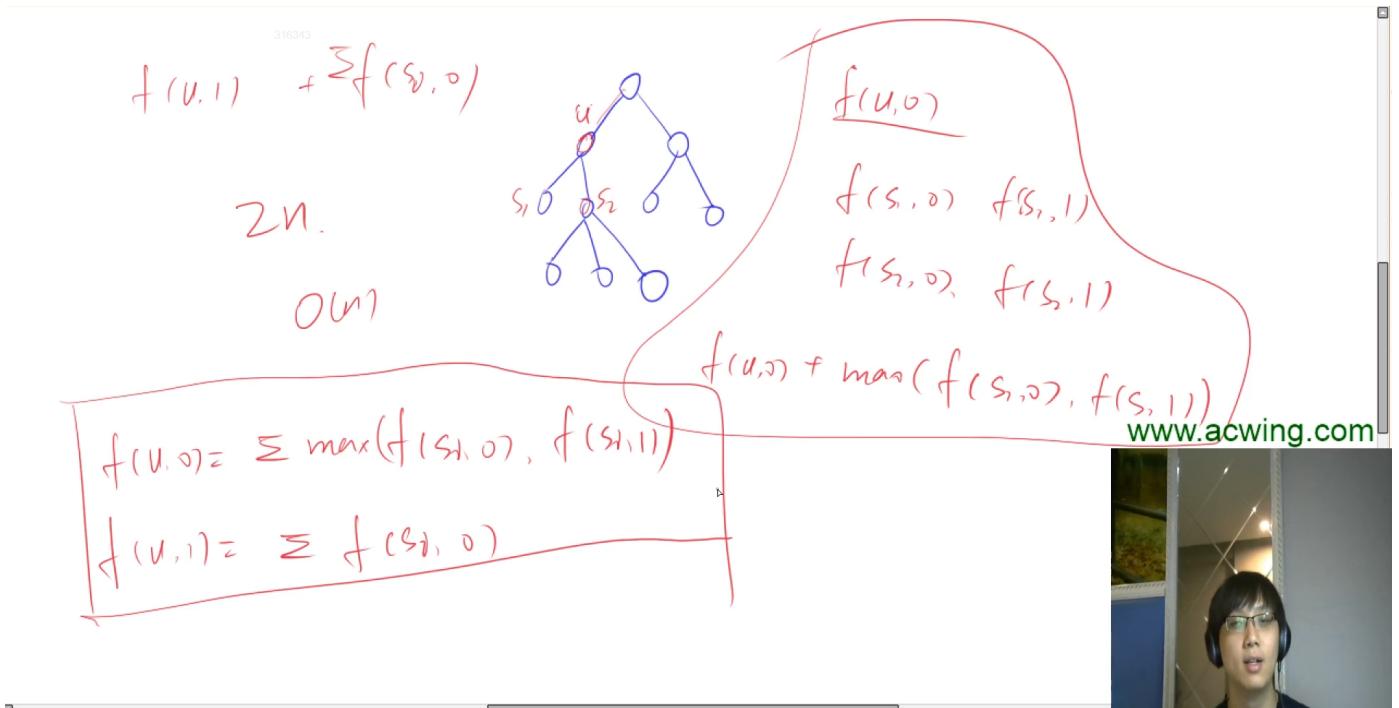
### 数据范围

$1 \leq N \leq 6000$ ,

$-128 \leq H_i \leq 127$

题意要求邀请一些人，其中不存在某人是另一个人的直接上司，但可以是间接上司（上司的上司）

动态规划思路：



状态表示中  $f[u, 0]$  表示不选根节点， $f[u, 1]$  表示选根节点

对于  $f[u, 0]$  情况：由于这棵树并没有选择根节点，所以在子节点为根节点的树中，根节点可以选也可以不选，取一个最大值即可

对于 $f[u,1]$ 情况：由于当前树的根节点已经被选择了，所以它的子节点不能选，将所有子树 $s[i]$ 的 $f[s,0]$ 进行求和即可得出 $f[u,1]$ 的值

代码：

```
1 #include <cstring>
2 #include <iostream>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int N = 6010;
8
9 int n;
10 int h[N], e[N], ne[N], idx;
11 int happy[N]; //每个人的高兴度
12 int f[N][2]; //所有状态，0表示不选，1表示选
13 bool has_fa[N]; //看是否有父节点
14 //临接表插入边，从a到b的边
15 void add(int a, int b)
16 {
17     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
18 }
19
20 void dfs(int u)
21 {
22     f[u][1] = happy[u]; //选这个点，加上高兴度
23     //枚举一下u的所有儿子
24     for (int i = h[u]; i != -1; i = ne[i])
25     {
26         int j = e[i];
27         dfs(j); //先算出来每个儿子节点选与不选的总高兴值
28         //状态计算
29         f[u][1] += f[j][0];
30         f[u][0] += max(f[j][0], f[j][1]);
31     }
32 }
33
34 int main()
35 {
36     scanf("%d", &n);
37
38     for (int i = 1; i <= n; i++) scanf("%d", &happy[i]);
39
40     memset(h, -1, sizeof h);
41     for (int i = 0; i < n - 1; i++)
42     {
43         int a, b;
44         scanf("%d%d", &a, &b); //b是a的父节点
45         add(b, a); //加入这条边
46     }
47 }
```

```
46     has_fa[a] = true; //a有父结点，为b
47 }
48
49 //从节点1开始找，没有父节点的就是根节点
50 int root = 1;
51 while (has_fa[root]) root++;
52
53 dfs(root);
54
55 printf("%d\n", max(f[root][0], f[root][1])); //选or不选根节点的最大值
56
57 return 0;
58 }
```

## 记忆化搜索

### 滑雪

给定一个  $R$  行  $C$  列的矩阵，表示一个矩形网格滑雪场。

矩阵中第  $i$  行第  $j$  列的点表示滑雪场的第  $i$  行第  $j$  列区域的高度。

一个人从滑雪场中的某个区域内出发，每次可以向上下左右任意一个方向滑动一个单位距离。

当然，一个人能够滑动到某相邻区域的前提是该区域的高度低于自己目前所在区域的高度。

下面给出一个矩阵作为例子：

```
1  2  3  4  5  
16 17 18 19 6  
15 24 25 20 7  
14 23 22 21 8  
13 12 11 10 9
```

在给定矩阵中，一条可行的滑行轨迹为  $24 - 17 - 2 - 1$ 。

在给定矩阵中，最长的滑行轨迹为  $25 - 24 - 23 - \dots - 3 - 2 - 1$ ，沿途共经过 25 个区域。

现在给定你一个二维矩阵表示滑雪场各区域的高度，请你找出在该滑雪场中能够完成的最长滑雪轨迹，并输出其长度(可经过最大区域数)。

#### 输入格式

第一行包含两个整数  $R$  和  $C$ 。

接下来  $R$  行，每行包含  $C$  个整数，表示完整的二维矩阵。

#### 输出格式

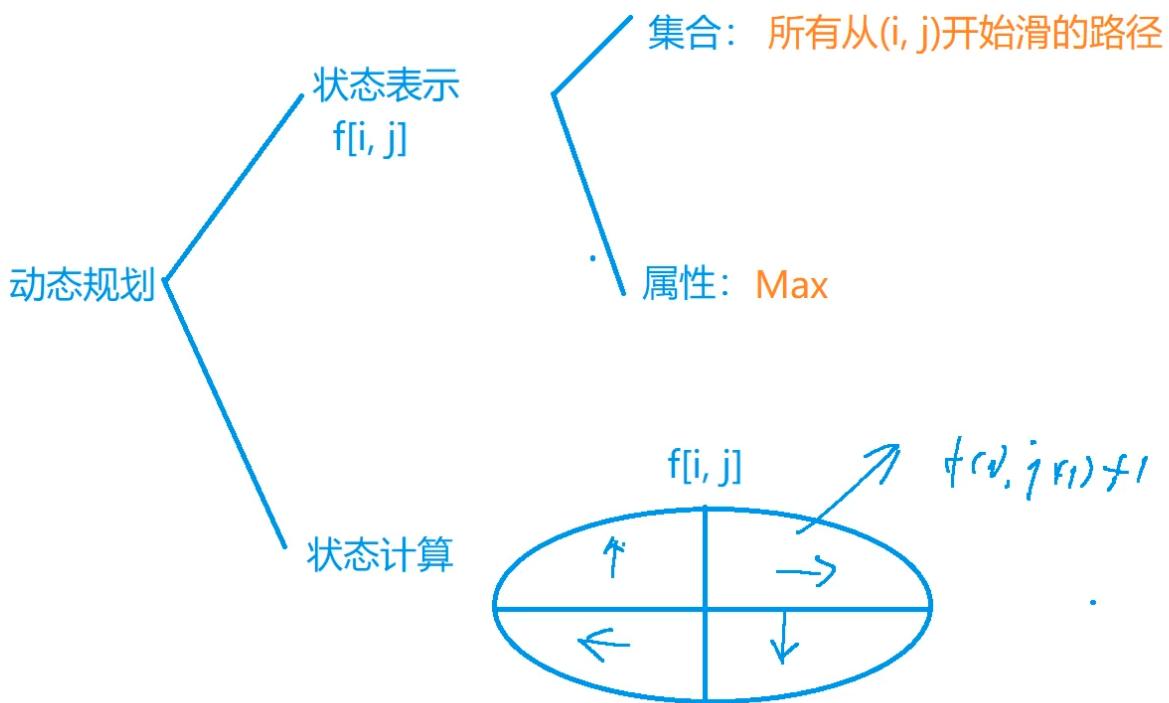
输出一个整数，表示可完成的最长滑雪长度。

#### 数据范围

$1 \leq R, C \leq 300$ ,

$0 \leq$  矩阵中整数  $\leq 10000$

动态规划思路：



状态计算按照往上下左右滑划分，但这四类并不一定全部存在，因为这四个方向的点不一定都小于当前值

当前点的最大值相当于下一步的最大值加一

代码：

```

1 #include<iostream>
2 #include<algorithm>
3 #include<cstring>
4
5 using namespace std;
6
7 const int N = 310;
8
9 int n,m;
10 int f[N][N];
11 int h[N][N]; //当前点高度
12
13 //上右下左向量
14 int dx[4] = {-1,0,1,0},dy[4] = {0,1,0,-1};
15
16 int dp(int x,int y)
17 {
18     int &v = f[x][y]; //引用， v等价于f[x][y]
19
20     if(v != -1) return v; //当前点已经被计算过了
21
22     v = 1; //从 (x, y) 点出发，至少可以走当前这个点
23     //遍历周围四个点，能走的条件是周围的点并未越界并且高度小于当前点
24     for(int i = 0;i < 4;i++)
25     {
26         int a = x + dx[i],b = y + dy[i];

```

```
27         if(a >= 0 && a < n && b >= 0 && b < m && h[a][b] < h[x][y])
28             v = max(v,dp(a,b) + 1);
29     }
30     return v;
31 }
32 int main()
33 {
34     scanf("%d%d",&n,&m);
35     for(int i = 0;i < n;i++)
36         for(int j = 0;j < m;j++)
37             scanf("%d",&h[i][j]);
38
39     memset(f,-1,sizeof f); //初始化f为-1, 表示当前点尚未被计算过
40
41     int res = 0;
42     for(int i = 0;i < n;i++)
43         for(int j = 0;j < m;j++)
44             res = max(res,dp(i,j));
45
46     printf("%d\n",res);
47
48     return 0;
49 }
```