



北京大学
PEKING UNIVERSITY

北京大学暑期课 《ACM/ICPC竞赛训练》

北京大学信息学院 郭炜

guo_wei@PKU.EDU.CN

<http://weibo.com/guoweiofpku>

课程网页: http://acm.pku.edu.cn/summerschool/pku_acm_train.htm



北京大学
PEKING UNIVERSITY

动态规划

北京大学信息学院 郭炜

例题一、数字三角形(POJ1163)

```
      7
     3 8
    8 1 0
   2 7 4 4
  4 5 2 6 5
```

在上面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大。路径上的每一步都只能往左下或右下走。只需要求出这个最大和即可，不必给出具体路径。

三角形的行数大于1小于等于100，数字为 0 - 99

输入格式：

5 //三角形行数。下面是三角形

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

要求输出最大和

解题思路：

用二维数组存放数字三角形。

$D(r, j)$: 第 r 行第 j 个数字(r, j 从1开始算)

$MaxSum(r, j)$: 从 $D(r, j)$ 到底边的各条路径中,
最佳路径的数字之和。

问题：求 $MaxSum(1, 1)$

典型的递归问题。

$D(r, j)$ 出发，下一步只能走 $D(r+1, j)$ 或者 $D(r+1, j+1)$ 。故对于 N 行的三角形：

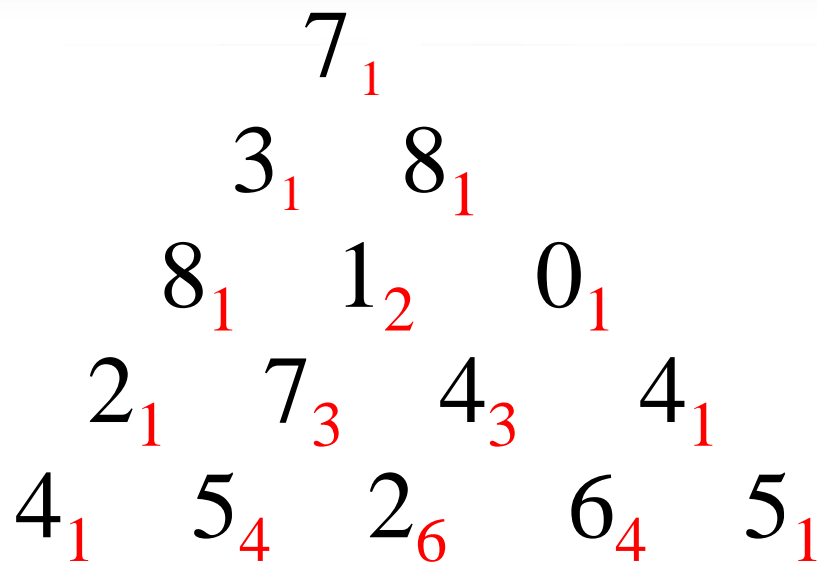
```
if ( r == N)
    MaxSum(r, j) = D(r, j)
else
    MaxSum( r, j) = Max{ MaxSum(r+1, j), MaxSum(r+1, j+1) }
                      + D(r, j)
```

数字三角形的递归程序:

```
#include <iostream>
#include <algorithm>
#define MAX 101
using namespace std;
int D[MAX][MAX];
int n;
int MaxSum(int i, int j){
    if(i==n)
        return D[i][j];
    int x = MaxSum(i+1,j);
    int y = MaxSum(i+1,j+1);
    return max(x,y)+D[i][j];
}

int main(){
    int i,j;
    cin >> n;
    for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
            cin >> D[i][j];
    cout << MaxSum(1,1) << endl;
}
```

为什么超时？



- 回答：重复计算

如果采用递归的方法，深度遍历每条路径，存在大量重复计算。则时间复杂度为 2^n ，对于 $n = 100$ 行，肯定超时。

改进

如果每算出一个 $\text{MaxSum}(r,j)$ 就保存起来，下次用到其值的时候直接取用，则可免去重复计算。那么可以用 $O(n^2)$ 时间完成计算。因为三角形的数字总数是 $n(n+1)/2$

数字三角形的记忆递归型动归程序：

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101

int D[MAX][MAX];    int n;
int maxSum[MAX][MAX];

int MaxSum(int i, int j){
    if( maxSum[i][j] != -1 )
        return maxSum[i][j];
    if(i==n)    maxSum[i][j] =
D[i][j];
    else {
        int x = MaxSum(i+1,j);
        int y = MaxSum(i+1,j+1);
        maxSum[i][j] = max(x,y)+
D[i][j];
    }
    return maxSum[i][j];
}
```

```
int main(){
    int i,j;
    cin >> n;
    for(i=1;i<=n;i++)
        for(j=1;j<=i;j++) {
            cin >> D[i][j];
            maxSum[i][j] = -1;
        }
    cout << MaxSum(1,1) << endl;
}
```

递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

4	5	2	6	5

递归转成递推

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

7				
4	5	2	6	5

递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

7	12			
4	5	2	6	5

递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

7	12	10		
4	5	2	6	5

递归转成递推

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

7	12	10	10	
4	5	2	6	5

递归转成递推

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

20				
7	12	10	10	
4	5	2	6	5

递归转成递推

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

20	13			
7	12	10	10	
4	5	2	6	5

递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

20	13	10		
7	12	10	10	
4	5	2	6	5

递归转成递推

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

30				
23	21			
20	13	10		
7	12	10	10	
4	5	2	6	5

“人人为我” 递推型动归程序

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];    int n;
int maxSum[MAX][MAX];
int main()    {
    int i,j;
    cin >> n;
    for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
            cin >> D[i][j];
    for( int i = 1;i <= n; ++ i )
        maxSum[n][i] = D[n][i];
    for( int i = n-1; i>= 1;  --i )
        for( int j = 1; j <= i; ++j )
            maxSum[i][j] =
                max(maxSum[i+1][j],maxSum[i+1][j+1]) + D[i][j]
    cout << maxSum[1][1] << endl;
}
```

空间优化

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

4	5	2	6	5
---	---	---	---	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

7	5	2	6	5
---	---	---	---	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

7	12	2	6	5
---	----	---	---	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

7	12	10	6	5
---	----	----	---	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

7	12	10	10	5
---	----	----	----	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

20	12	10	10	5
----	----	----	----	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

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

20	13	10	10	5
----	----	----	----	---

没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推,那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

空间优化

进一步考虑，连maxSum数组都可以不要，直接用D的第n行替代maxSum即可。

节省空间，时间复杂度不变

空间优化

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];
int n; int * maxSum;
int main(){
    int i,j;
    cin >> n;
    for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
            cin >> D[i][j];
    maxSum = D[n]; //maxSum指向第n行
    for( int i = n-1; i>= 1; --i )
        for( int j = 1; j <= i; ++j )
            maxSum[j] = max(maxSum[j],maxSum[j+1]) + D[i][j];
    cout << maxSum[1] << endl;
}
```

递归到动规的一般转化方法

- 递归函数有 n 个参数，就定义一个 n 维的数组，数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值，这样就可以从边界值开始，逐步填充数组，相当于计算递归函数值的逆过程。

动规解题的一般思路

1. 将原问题分解为子问题

- 把原问题分解为若干个子问题，子问题和原问题形式相同或类似，只不过规模变小了。子问题都解决，原问题即解决(数字三角形例)。
- 子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。

动规解题的一般思路

2. 确定状态

- 在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题，所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。

动规解题的一般思路

2. 确定状态

所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。在数字三角形的例子里，一共有 $N \times (N+1)/2$ 个数字，所以这个问题的状态空间里一共就有 $N \times (N+1)/2$ 个状态。

整个问题的时间复杂度是状态数目乘以计算每个状态所需时间。

在数字三角形里每个“状态”只需要经过一次，且在每个状态上作计算所花的时间都是和 N 无关的常数。

动规解题的一般思路

2. 确定状态

用动态规划解题，经常碰到的情况是， K 个整型变量能构成一个状态（如数字三角形中的行号和列号这两个变量构成“状态”）。如果这 K 个整型变量的取值范围分别是 N_1, N_2, \dots, N_k ，那么，我们就可以用一个 K 维的数组 `array[N1][N2].....[Nk]` 来存储各个状态的“值”。这个“值”未必就是一个整数或浮点数，可能是需要一个结构才能表示的，那么 `array` 就可以是一个结构数组。一个“状态”下的“值”通常会是一个或多个子问题的解。

动规解题的一般思路

3. 确定一些初始状态（边界状态）的值

以“数字三角形”为例，初始状态就是底边数字，值就是底边数字值。

动规解题的一般思路

4. 确定状态转移方程

定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移——即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”（“人人为我”递推型）。状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

数字三角形的状态转移方程: 

$$\text{MaxSum}[r][j] = \begin{cases} D[r][j] & r = N \\ \text{Max}\{ \text{MaxSum}[r+1][j], \text{MaxSum}[r+1][j+1] \} + D[r][j] & \text{其他情况} \end{cases}$$

能用动规解决的问题的特点

- 1) **问题具有最优子结构性质**。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。
- 2) **无后效性**。当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。



例题二：最长上升子序列(百练2757)

问题描述

一个数的序列 a_i ，当 $a_1 < a_2 < \dots < a_s$ 的时候，我们称这个序列是上升的。对于给定的一个序列 (a_1, a_2, \dots, a_N) ，我们可以得到一些上升的子序列 $(a_{i_1}, a_{i_2}, \dots, a_{i_K})$ ，这里 $1 \leq i_1 < i_2 < \dots < i_K \leq N$ 。比如，对于序列 $(1, 7, 3, 5, 9, 4, 8)$ ，有它的一些上升子序列，如 $(1, 7)$ ， $(3, 4, 8)$ 等等。这些子序列中最长的长度是4，比如子序列 $(1, 3, 5, 8)$ 。

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

输入数据

输入的第一行是序列的长度 N ($1 \leq N \leq 1000$)。第二行给出序列中的 N 个整数，这些整数的取值范围都在0到10000。

输出要求

最长上升子序列的长度。

输入样例

7

1 7 3 5 9 4 8


输出样例

4

解题思路

1. 找子问题

“求序列的前 n 个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”

假设 $F(n) = x$ ，但可能有多个序列满足 $F(n) = x$ 。有的序列的最后一个元素比 a_{n+1} 小，则加上 a_{n+1} 就能形成更长上升子序列；有的序列最后一个元素不比 a_{n+1} 小……以后的事情受如何达到状态 n 的影响，不符合“无后效性”

解题思路

1. 找子问题

“求以 a_k ($k=1, 2, 3\cdots N$) 为终点的最长上升子序列的长度”

一个上升子序列中最右边的那个数，称为该子序列的“终点”。

虽然这个子问题和原问题形式上并不完全一样，但是只要这 N 个子问题都解决了，那么这 N 个子问题的解中，最大的那个就是整个问题的解。

2. 确定状态：

子问题只和一个变量——数字的位置相关。因此序列中数的位置 k 就是“状态”，而状态 k 对应的“值”，就是以 a_k 做为“终点”的最长上升子序列的长度。

状态一共有 N 个。

3. 找出状态转移方程:

$\text{maxLen}(k)$ 表示以 a_k 做为“终点”的最长上升子序列的长度那么:

初始状态: $\text{maxLen}(1) = 1$

$$\text{maxLen}(k) = \max \{ \text{maxLen}(i) : 1 \leq i < k \text{ 且 } a_i < a_k \text{ 且 } k \neq 1 \} + 1$$

若找不到这样的 i , 则 $\text{maxLen}(k) = 1$

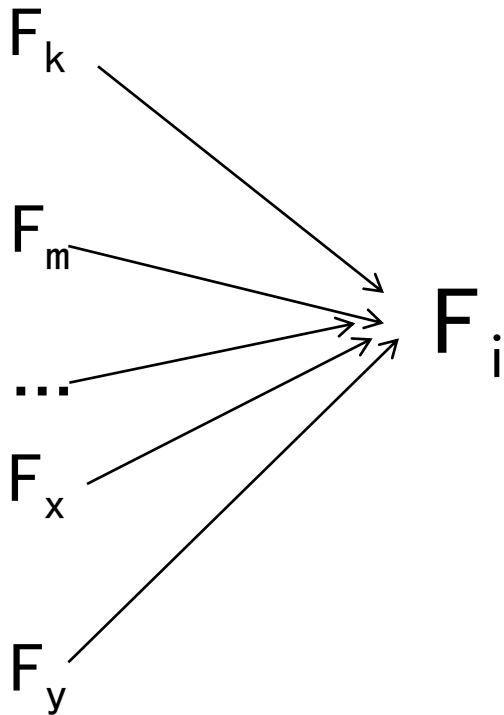
$\text{maxLen}(k)$ 的值, 就是在 a_k 左边, “终点”数值小于 a_k , 且长度最大的那个上升子序列的长度再加1。因为 a_k 左边任何“终点”小于 a_k 的子序列, 加上 a_k 后就能形成一个更长的上升子序列。

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
```

“人人为我” 递推型动归程序

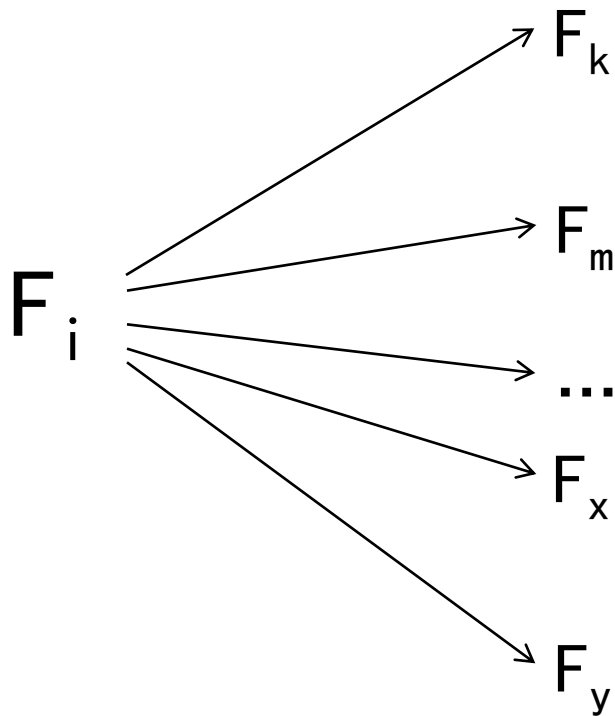
```
const int MAXN = 1010;
int a[MAXN];    int maxLen[MAXN];
int main() {
    int N;          cin >> N;
    for( int i = 1; i <= N; ++i) {
        cin >> a[i];    maxLen[i] = 1;
    }
    for( int i = 2; i <= N; ++i) {
        //每次求以第i个数为终点的最长上升子序列的长度
        for( int j = 1; j < i; ++j)
            //察看以第j个数为终点的最长上升子序列
            if( a[i] > a[j] )
                maxLen[i] = max(maxLen[i], maxLen[j] + 1);
    }
    cout << * max_element(maxLen + 1, maxLen + N + 1 );
    return 0;
} //时间复杂度O(N²)
```

“人人为我” 递推型动归



状态 i 的值 F_i 由若干个值
已知的状态值 F_k, F_m, \dots, F_y
推出，如求和，取最大值
.....

“我为人人” 递推型动归



状态 i 的值 F_i 在被更新（不一定是最终求出）的时候，依据 F_i 去更新（不一定是最终求出）和状态 i 相关的其他一些状态的值

F_k, F_m, \dots, F_y

“我为人人” 递推型动归程序

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
```

```
const int MAXN = 1010;
```

```
int a[MAXN];
```

```
int maxLen[MAXN];
```

```
int main() {
```

```
    int N;
```

```
    cin >> N;
```

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

```
        cin >> a[i];
```

```
        maxLen[i] = 1;
```

```
    }
```

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

```
        for( int j = i + 1; j <= N; ++j ) //看看能更新哪些状态的值
```

```
            if( a[j] > a[i] )
```

```
                maxLen[j] = max(maxLen[j], maxLen[i] + 1);
```

```
    cout << * max_element(maxLen + 1, maxLen + N + 1 );
```

```
    return 0;
```

```
} //时间复杂度 $O(N^2)$ 
```

人人为我:

```
for( int i = 2; i <= N; ++i)
```

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

```
        if( a[i] > a[j] )
```

```
            maxLen[i] =
```

```
            max(maxLen[i], maxLen[j] + 1);
```

动归的三种形式

1) 记忆递归型

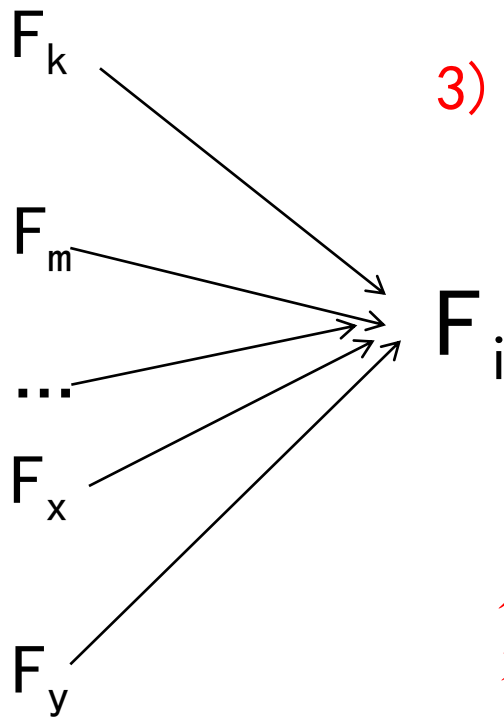
优点：只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费

缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。无法使用滚动数组节省空间。总体来说，比递推型慢。 📄

2) “我为人人”递推型

没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比“人人为我”型节省空间。

“人人为我”递推型中的优化



3) “人人为我”递推型

状态 i 的值 F_i 由若干个值
已知的状态值 F_k, F_m, \dots, F_y
推出，如求和，取最大值
.....

在选取最优备选状态的值 F_m, F_n, \dots, F_y 时，
有可能有好的算法或数据结构可以用来显
著降低时间复杂度。📖

例三、最长公共子序列 (POJ1458)

给出两个字符串，求出这样的一个最长的公共子序列的长度：子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

最长公共子序列

Sample Input

abcfbc abfcab
programming contest
abcd mnp

Sample Output

4
2
0

最长公共子序列

输入两个串s1,s2,

设MaxLen(i,j)表示:

s1的左边i个字符形成的子串, 与s2左边的j个字符形成的子串的最长公共子序列的长度(i,j从0开始算)

MaxLen(i,j) 就是本题的“状态”

假定 $\text{len1} = \text{strlen}(s1), \text{len2} = \text{strlen}(s2)$

那么题目就是要求 $\text{MaxLen}(\text{len1}, \text{len2})$

最长公共子序列

显然：

$$\text{MaxLen}(n,0) = 0 \quad (n = 0 \dots \text{len1})$$

$$\text{MaxLen}(0,n) = 0 \quad (n = 0 \dots \text{len2})$$

递推公式：

if ($s1[i-1] == s2[j-1]$) //s1的最左边字符是s1[0]

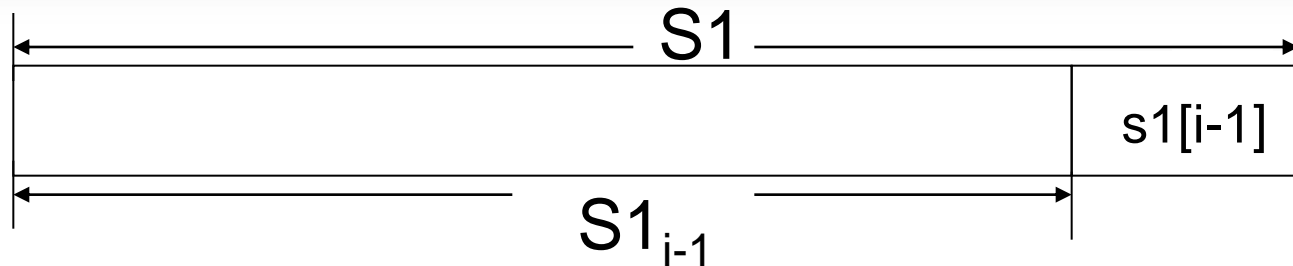
$$\text{MaxLen}(i,j) = \text{MaxLen}(i-1,j-1) + 1;$$

else

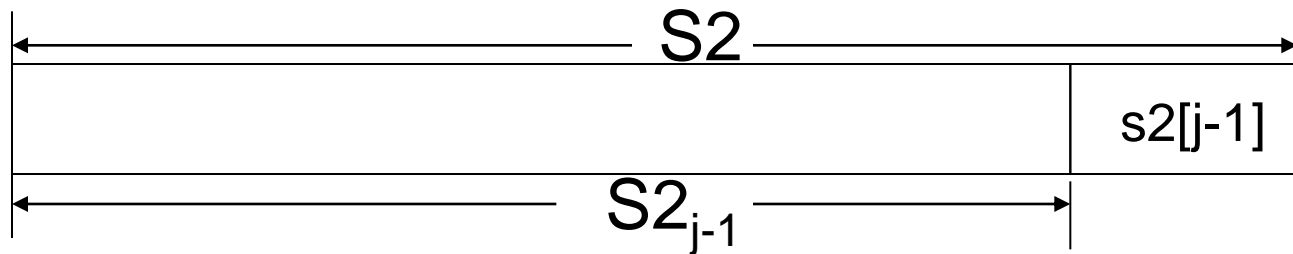
$$\text{MaxLen}(i,j) = \text{Max}(\text{MaxLen}(i,j-1), \text{MaxLen}(i-1,j));$$

时间复杂度 $O(mn)$ m,n是两个字符串长度

S1长度为 i



S2长度为 j



$s1[i-1] \neq s2[j-1]$ 时, $\text{MaxLen}(S1, S2)$ 不会比 $\text{MaxLen}(S1, S2_{j-1})$ 和 $\text{MaxLen}(S1_{i-1}, S2)$ 两者之中任何一个小, 也不会比两者都大。

```
#include <iostream>
#include <cstring>
using namespace std;
char sz1[1000];
char sz2[1000];
int maxLen[1000][1000];
int main() {
    while( cin >> sz1 >> sz2 ) {
        int length1 = strlen( sz1 );
        int length2 = strlen( sz2 );
        int nTmp;
        int i,j;
        for( i = 0; i <= length1; i ++ )
            maxLen[i][0] = 0;
        for( j = 0; j <= length2; j ++ )
            maxLen[0][j] = 0;
```

```
for( i = 1; i <= length1; i ++ ) {  
    for( j = 1; j <= length2; j ++ ) {  
        if( sz1[i-1] == sz2[j-1] )  
            maxLen[i][j] = maxLen[i-1][j-1] + 1;  
        else  
            maxLen[i][j] = max(maxLen[i][j-1],  
                               maxLen[i-1][j]);  
    }  
}  
cout << maxLen[length1][length2] << endl;  
}  
return 0;  
}
```

活学活用

- 掌握递归和动态规划的思想，解决问题时灵活应用

例四、最佳加法表达式

有一个由1..9组成的数字串.问如果将 m 个加号插入到这个数字串中,在各种可能形成的表达式中,值最小的那个表达式的值是多少

解题思路

假定数字串长度是 n ，添完加号后，表达式的最后一个加号添加在第 i 个数字后面，那么整个表达式的最小值，就等于在前 i 个数字中插入 $m-1$ 个加号所能形成的最小值，加上第 $i+1$ 到第 n 个数字所组成的数的值（ i 从1开始算）。

解题思路

设 $V(m,n)$ 表示在 n 个数字中插入 m 个加号所能形成的表达式最小值，那么：

if $m = 0$

$V(m,n) = n$ 个数字构成的整数

else if $n < m + 1$

$V(m,n) = \infty$

else

$V(m,n) = \text{Min}\{ V(m-1,i) + \text{Num}(i+1,n) \} \ (i = m \dots n-1)$

$\text{Num}(i,j)$ 表示从第 i 个数字到第 j 个数字所组成的数。数字编号从1开始算。此操作复杂度是 $O(j-i+1)$ ，可以预处理后存起来。

总时间复杂度： $O(mn^2)$ 。

例五、神奇的口袋(百练2755)

- 有一个神奇的口袋，总的容积是40，用这个口袋可以变出一些物品，这些物品的总体积必须是40。
- John现在有 n ($1 \leq n \leq 20$) 个想要得到的物品，每个物品的体积分别是 a_1, a_2, \dots, a_n 。John可以从这些物品中选择一些，如果选出的物体的总体积是40，那么利用这个神奇的口袋，John就可以得到这些物品。现在的问题是，John有多少种不同的选择物品的方式。

- 输入

输入的第一行是正整数 n ($1 \leq n \leq 20$), 表示不同的物品的数目。接下来的 n 行, 每行有一个1到40之间的正整数, 分别给出 a_1, a_2, \dots, a_n 的值。

- 输出

输出不同的选择物品的方式的数目。

- 输入样例

3

20

20

20

- 输出样例

3

枚举的解法：

枚举每个物品是选还是不选，共 2^{20} 种情况

递归解法

```
#include <iostream>
using namespace std;
int a[30];  int N;
int Ways(int w ,int k )  { // 从前k种物品中选择一些，凑成体积w的做法数目
    if( w == 0 )    return 1;
    if( k <= 0 )    return 0;
    return Ways(w, k -1 ) + Ways(w - a[k], k -1 );
}
int main()    {
    cin >> N;
    for( int i = 1;i <= N; ++ i )
        cin >> a[i];
    cout << Ways(40,N);
    return 0;
}
```

动 规 解 法

```
#include <iostream>
using namespace std;
int a[40];   int N;
int Ways[50][40]; //Ways[i][j] 表示从前j种物品里凑出体积i的方法数
int main()   {
    cin >> N;
    memset(Ways, 0, sizeof(Ways));
    for( int i = 1; i <= N; ++ i ) {
        cin >> a[i];   Ways[0][i] = 1;
    }
    Ways[0][0] = 1;
    for( int w = 1 ; w <= 40; ++ w ) {
        for( int k = 1; k <= N; ++ k ) {
            Ways[w][k] = Ways[w][k-1];
            if( w-a[k] >= 0 )
                Ways[w][k] += Ways[w-a[k]][k-1];
        }
    }
    cout << Ways[40][N];
    return 0;
}
```


例六、0-1背包问题 (POJ3624)

有 N 件物品和一个容积为 M 的背包。第 i 件物品的体积 $w[i]$ ，价值是 $d[i]$ 。求解将哪些物品装入背包可使价值总和最大。每种物品只有一件，可以选择放或者不放 ($N \leq 3500, M \leq 13000$)。

0-1背包问题 (POJ3624)

用 $F[i][j]$ 表示取前 i 种物品，使它们总体积不超过 j 的最优取法取得的价值总和。要求 $F[N][M]$

边界: $\text{if } (w[1] \leq j)$
 $F[1][j] = d[1];$
 else
 $F[1][j] = 0;$

0-1背包问题 (POJ3624)

用 $F[i][j]$ 表示取前 i 种物品，使它们总体积不超过 j 的最优取法取得的价值总和

递推: $F[i][j] = \max(F[i-1][j], F[i-1][j-w[i]]+d[i])$

取或不取第 i 种物品，两者选优
($j-w[i] \geq 0$ 才有第二项)

0-1背包问题 (POJ3624)

$$F[i][j] = \max(F[i-1][j], F[i-1][j-w[i]]+d[i])$$

本题如用记忆型递归，需要一个很大的二维数组，会超内存。注意到这个二维数组的下一行的值，只用到了上一行的正上方及左边的值，因此可用滚动数组的思想，只要一行即可。即可以用一维数组，用“人人为我”递推型动归实现。

例七、滑雪(百练1088)

Michael喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。

可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。

Michael想知道载一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

```
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
```

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。输入输入的第一行表示区域的行数R和列数C($1 \leq R, C \leq 100$)。下面是R行，每行有C个整数，代表高度h， $0 \leq h \leq 10000$ 。输出输出最长区域的长度。

输入

输入的第一行表示区域的行数 R 和列数 C
($1 \leq R, C \leq 100$)。下面是 R 行，每行有 C 个整数，
代表高度 h ， $0 \leq h \leq 10000$ 。

输出

输出最长区域的长度。

样例输入

```
5 5
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
```

样例输出

```
25
```

解题思路

$L(i,j)$ 表示从点 (i,j) 出发的最长滑行长度。

一个点 (i,j) , 如果周围没有比它低的点, $L(i,j) = 1$

否则

递推公式: $L(i,j)$ 等于 (i,j) 周围四个点中,比 (i,j) 低, 且 L 值最大的那个点的 L 值, 再加1

复杂度: $O(n^2)$

解题思路

解法1) “人人为我”式递推

$L(i,j)$ 表示从点 (i,j) 出发的最长滑行长度。

一个点 (i,j) , 如果周围没有比它低的点, $L(i,j) = 1$

将所有点按高度从小到大排序。每个点的 L 值都初始化为1

从小到大遍历所有的点。经过一个点 (i,j) 时, 用递推公式求 $L(i,j)$

解题思路

解法2) “我为人人”式递推

$L(i,j)$ 表示从点 (i,j) 出发的最长滑行长度。

一个点 (i,j) , 如果周围没有比它低的点, $L(i,j) = 1$

将所有点按高度从小到大排序。每个点的 L 值都初始化为1

从小到大遍历所有的点。经过一个点 (i,j) 时, 要更新他周围的, 比它高的点的 L 值。例如:

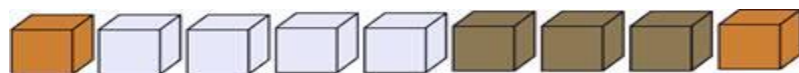
if $H(i+1,j) > H(i,j)$ // H 代表高度

$L(i+1,j) = \max(L(i+1,j), L(i,j)+1)$

例八、POJ1390 方盒游戏

问题描述

N个方盒(box)摆成一排，每个方盒有自己的颜色。连续摆放的同颜色方盒构成一个方盒片段(box segment)。下图中共有四个方盒片段，每个方盒片段分别有1、4、3、1个方盒

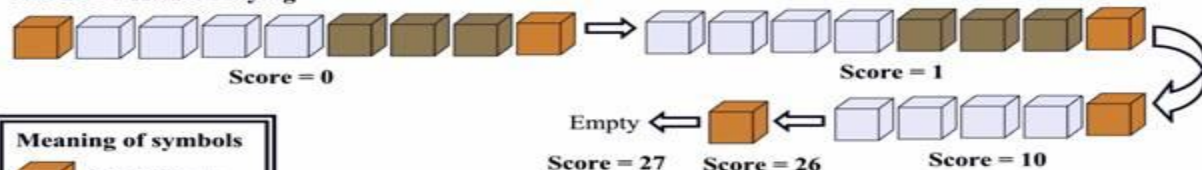


玩家每次点击一个方盒，则该方盒所在方盒片段就会消失。若消失的方盒片段中共有k个方盒，则玩家获得 $k*k$ 个积分。

One Possible Playing



Another Possible Playing



Meaning of symbols



- 请问：给定游戏开始时的状态，玩家可获得的最高积分是多少？
- 输入：第一行是一个整数 $t(1 \leq t \leq 15)$ ，表示共有多少组测试数据。每组测试数据包括两行
 - 第一行是一个整数 $n(1 \leq n \leq 200)$ ，表示共有多少个方盒
 - 第二行包括 n 个整数，表示每个方盒的颜色。这些整数的取值范围是 $[1, n]$
- 输出：对每组测试数据，分别输出该组测试数据的序号、以及玩家可以获得的最高积分

■ 样例输入

2

9

1 2 2 2 2 3 3 3 1

1

1

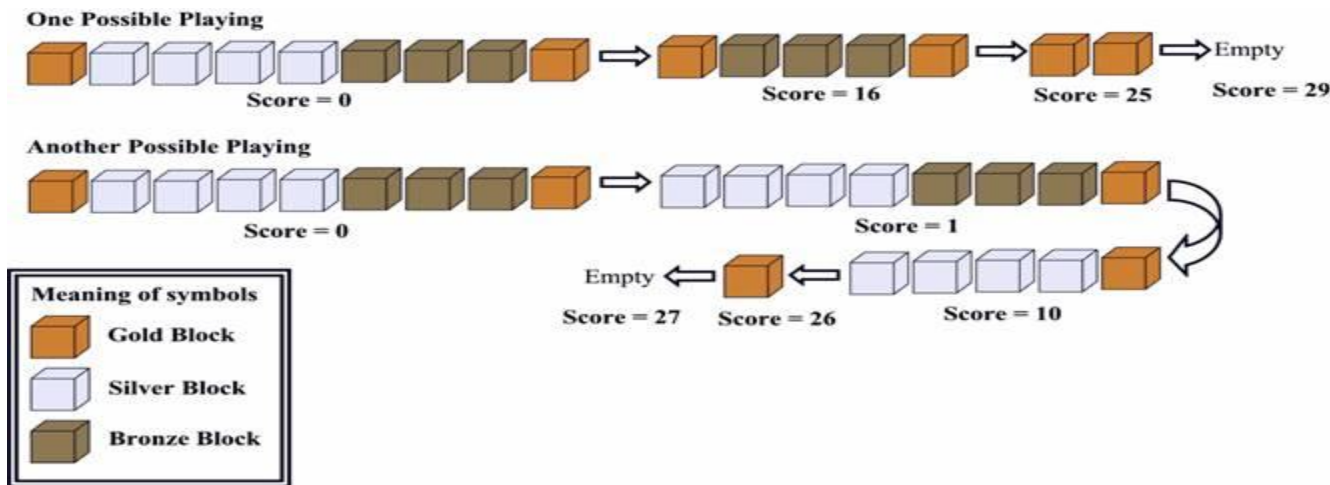
■ 样例输出

Case 1: 29

Case 2: 1

问题分析

- 当同颜色的方盒摆放在不连续的位置时，方盒的点击顺序影响玩家获得的积分



- 同种颜色的方盒被点击的次数越少，玩家获得的积分越高
- 明显的递归问题：每次点击之后，剩下的方盒构成一个新的方盒队列，新队列中方盒的数量减少了。然后计算玩家从新队列中可获得的最高积分

问题分析

- 点击下图中黑色方盒之前，先点击绿色方盒可提高玩家的积分：同颜色方盒A和B被其他颜色的方盒隔开时，先点击其他颜色方盒，使得A和B消失前能够在同一个方盒片段中
- 点击下图中红色和蓝色方盒可获得的积分
 - 所有红色方盒合并到同一个片段： $49+1+36=86$
 - 所有蓝色方盒合并到同一个片段： $49+16+9=74$



问题分析

■ 思路:

将连续的若干个方块作为一个“大块” (box_segment)

考虑, 假设开始一共有 n 个“大块”, 编号0到 $n-1$

第 i 个大块的颜色是 $color[i]$, 包含的方块数目, 即长度, 是 $len[i]$

用 $click_box(i,j)$ 表示从大块 i 到大块 j 这一段消除后所能得到的最高分

则整个问题就是: $click_box(0,n-1)$

问题分析

要求click_box(i,j)时，考虑最右边的大块j，对它有兩種处理方式，要取其优者：

1) 直接消除它，此时能得到最高分就是：

$$\text{click_box}(i, j-1) + \text{len}[j] * \text{len}[j]$$

2) 期待以后它能和左边的某个同色大块合并

考虑和左边的某个同色大块合并:

左边的同色大块可能有很多个，到底和哪个合并最好，不知道，只能枚举。假设大块 j 和左边的大块 $k(i \leq k < j-1)$ 合并，此时能得到的最高分是多少呢？

考虑和左边的某个同色大块合并:

左边的同色大块可能有很多个，到底和哪个合并最好，不知道，只能枚举。假设大块j和左边的大块 $k(i \leq k < j-1)$ 合并，此时能得到的最高分是多少呢？

是不是：

$$\text{click_box}(i, k-1) + \text{click_box}(k+1, j-1) + (\text{len}[k] + \text{len}[j])^2$$

问题分析

$$\text{click_box}(i, k-1) + \text{click_box}(k+1, j-1) + (\text{len}[k] + \text{len}[j])^2$$

不对！

因为将大块 k 和大块 j 合并后，形成的新大块会在最右边。将该新大块直接将其消去的做法，才符合上述式子，但直接将其消去，未必是最好的，也许它还应该和左边的同色大块合并，才更好

递推关系无法形成，怎么办？

问题分析

需要改变问题的形式。

`click_box(i,j)` 这个形式不可取，因为无法形成递推关系

考虑新的形式：

`click_box(i,j,ex_len)`

表示：

大块j的右边已经有一个长度为`ex_len`的大块(该大块可能是在合并过程中形成的，不妨就称其为`ex_len`)，且j的颜色和`ex_len`相同，在此情况下将i到j以及`ex_len`都消除所能得到的最高分。

于是整个问题就是求：`click_box(0,n-1,0)`

问题分析

求 $\text{click_box}(i,j,\text{ex_len})$ 时，有两种处理方法，取最优者

假设 j 和 ex_len 合并后的大块称作 Q

1) 将 Q 直接消除，这种做法能得到的最高分就是：

$$\text{click_box}(i,j-1,0) + (\text{len}[j] + \text{ex_len})^2$$

2) 期待 Q 以后能和左边的某个同色大块合并。需要枚举可能和 Q 合并的大块。假设让大块 k 和 Q 合并，则此时能得到的最大分数是：

$$\text{click_box}(i,k,\text{len}[j] + \text{ex_len}) + \text{click_box}(k+1,j-1,0)$$

递归的终止条件是什么？

click_box(i,j,ex_len) 递归的终止条件:

$$i == j$$

```
#include <iostream>
#include <cstring>
using namespace std;
const int M = 210;
struct Segment {
    int color;
    int len;
};
Segment segments[M];
int score[M][M][M];
```

```
int ClickBox(int i,int j,int len)  {
    if( score[i][j][len] != -1)
        return score[i][j][len];
    int result = (segments[j].len + len) *
                (segments[j].len + len);
    if( i == j )
        return result;
    result += ClickBox(i,j-1,0);
    for(int k = i;k <= j-1; ++k ) {
        if( segments[k].color != segments[j].color )
            continue;
        int r = ClickBox(k+1,j-1,0);
        r += ClickBox(i,k,segments[j].len + len);
        result = max(result,r);
    }
}
```

```
score[i][j][len] = result;  
return result;
```

```
}
```

```
int main()
```

```
{
```

```
    int T;
```

```
    cin >> T;
```

```
    for(int t = 1; t <= T; ++ t) {
```

```
        int n;
```

```
        memset(score,0xff,sizeof(score)) ;
```

```
        cin >> n;
```

```
        int lastC = 0;
```

```
        int segNum = -1;
```



```
for(int i = 0;i < n; ++ i) {  
    int c;  
    cin >> c;  
    if( c != lastC ) {  
        segNum ++;  
        segments[segNum].len = 1;  
        segments[segNum].color = c;  
        lastC = c;  
    }  
    else segments[segNum].len ++;  
}
```

```
cout << "Case " << t << ": " << ClickBox(0,segNum,0) << endl;  
}  
return 0;  
}
```

例九、灌溉草场 (POJ2373)

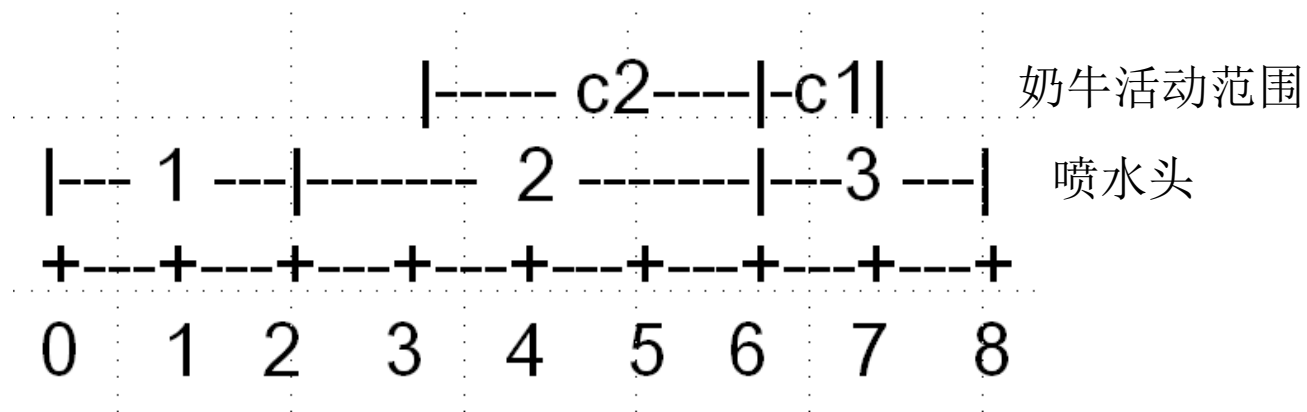
在一片草场上:有一条长度为 L ($1 \leq L \leq 1,000,000$, L 为偶数)的线段。John的 N ($1 \leq N \leq 1000$) 头奶牛都沿着草场上这条线段吃草, 每头牛的活动范围是一个开区间 (S, E) , S, E 都是整数。不同奶牛的活动范围可以有重叠。

John要在这条线段上安装喷水头灌溉草场。每个喷水头的喷洒半径可以随意调节, 调节范围是 $[A, B]$ ($1 \leq A \leq B \leq 1000$), A, B 都是整数。要求

- 线段上的每个整点恰好位于一个喷水头的喷洒范围内
- 每头奶牛的活动范围要位于一个喷水头的喷洒范围内
- 任何喷水头的喷洒范围不可越过线段的两端(左端是0, 右端是 L)

请问, John 最少需要安装多少个喷水头。

灌溉草场 (POJ2373)



在位置2和6，喷水头的喷洒范围不算重叠

- 输入

第1行：整数N、L。

第2行：整数A、B。

第3到N+2行：每行两个整数S、E ($0 \leq S < E \leq L$)，表示某头牛活动范围的起点和终点在线段上的坐标(即到线段起点的距离)。

- 输出：最少需要安装的多少个喷水头；若没有符合要求的喷水头安装方案，则输出-1。

- 输入样例

2 8

1 2

6 7

3 6

- 输出样例

3

问题分析

- 从线段的起点向终点安装喷水头，令 $f(X)$ 表示：所安装喷水头的喷洒范围恰好覆盖直线上的区间 $[0, X]$ 时，最少需要多少个喷水头
- 显然， X 应满足下列条件
 - X 为偶数
 - X 所在位置不会出现奶牛，即 X 不属于任何一个 (S, E) 📄
 - $X \geq 2A$
 - 当 $X > 2B$ 时，存在 $Y \in [X-2B, X-2A]$ 且 Y 满足上述三个条件，使得 $f(X) = f(Y) + 1$

解题思路

- 递推计算 $f(X)$
 - $f(X) = \infty$: X 是奇数
 - $f(X) = \infty$: $X < 2A$
 - $f(X) = \infty$: X 处可能有奶牛出没
 - $f(X)=1$: $2A \leq X \leq 2B$ 、且 X 位于任何奶牛的活动范围之外
 - $f(X)=1+\min\{f(Y): Y \in [X-2B, X-2A]$ 、 Y 位于任何奶牛的活动范围之外}: $X > 2B$

解题思路

$f(X) = 1 + \min\{f(Y) : Y \in [X-2B, X-2A], Y \text{ 位于任何奶牛的活动范围之外}\} : X > 2B$

- 对每个 X 求 $f(X)$ ，都要遍历区间 $[X-2B, X-2A]$ 去寻找其中最小的 $f(Y)$ ，则时间复杂度为： $L * B = 1000000 * 1000$ ，太慢
- 快速找到 $[X-2B, X-2A]$ 中使得 $f(Y)$ 最小的元素是问题求解速度的关键。

解题思路

- 可以使用优先队列`priority_queue`! (`multiset`也可以, 比`priority_queue`慢一点) !
- 求 $F(X)$ 时, 若坐标属于 $[X-2B, X-2A]$ 的二元组 $(i, F(i))$ 都保存在一个`priority_queue`中, 并根据 $F(i)$ 值排序, 则队头的元素就能确保是 $F(i)$ 值最小的。

解题思路

- 在求 X 点的 $F(x)$ 时，必须确保队列中包含所有属于 $[X-2B, X-2A]$ 的点。而且，不允许出现坐标大于 $X-2A$ 的点，因为这样的点对求 $F(X)$ 无用，如果这样的点出现在队头，因其对求后续点的 F 值有用，故不能抛弃之，于是算法就无法继续了。
- 队列中可以出现坐标小于 $X-2B$ 的点。这样的点若出现在队头，则直接将其抛弃。
- 求出 X 点的 F 值后，将 $(X-2A+2, F(X-2A+2))$ 放入队列，为求 $F(X+2)$ 作准备
- 队列里只要存坐标为偶数的点即可

```
#include <iostream>
#include <cstring>
#include <queue>
using namespace std;
const int INFINITE = 1<<30;
const int MAXL = 1000010;
const int MAXN = 1010;
int F[MAXL]; // F[L]就是答案
int cowThere[MAXL]; //cowThere[i]为1表示点i有奶牛
int N,L,A,B;
struct Fx {
    int x;          int f;
    bool operator<(const Fx & a) const
    { return f > a.f; }
    Fx(int xx=0,int ff=0):x(xx),f(ff) { }
}; // 在优先队列里，f值越小的越优先
priority_queue<Fx> qFx;
```

```
int main()
{
    cin >> N >> L;
    cin >> A >> B;
    A <<= 1; B <<= 1; //A,B的定义变为覆盖的直径
    memset(cowThere,0,sizeof(cowThere));
    for( int i = 0;i < N; ++i ) {
        int s,e;
        cin >> s >> e;
        ++cowThere[s+1]; //从s+1起进入一个奶牛区
        --cowThere[e]; //从e起退出一个奶牛区
    }
    int inCows = 0; //表示当前点位于多少头奶牛的活动范围之内
    for( int i = 0;i <= L ; i ++ ) { //算出每个点是否有奶牛
        F[i] = INFINITE;
        inCows += cowThere[i];
        cowThere[i] = inCows > 0;
    }
```

```

for( int i = A; i <= B ; i += 2 ) //初始化队列
    if( ! cowThere[i] ) {
        F[i] = 1;
        if( i <= B + 2 - A )
            //在求F[i]的时候, 要确保队列里的点x,  $x \leq i - A$ 
            qFx.push( Fx(i, 1) );
    }
for( int i = B + 2 ; i <= L; i += 2 ) {
    if( !cowThere[i] ) {
        Fx fx;
        while( !qFx.empty() ) {
            fx = qFx.top();
            if( fx.x < i - B )
                qFx.pop();
            else
                break;
        }
        if ( ! qFx.empty() )
            F[i] = fx.f + 1;
    }
}

```

```
        if( F[i- A + 2] != INFINITE) {  
            //队列中增加一个+1可达下个点的点  
            qFx.push(Fx(i- A + 2, F[i- A + 2]));  
        }  
    }  
    if( F[L] == INFINITE )  
        cout << -1 << endl;  
    else  
        cout << F[L] << endl;  
    return 0;  
} // 复杂度:  $O(n \log n)$ 
```

手工实现优先队列的方法

- 如果一个队列满足以下条件：
 - 1) 开始为空
 - 2) 每在队尾加入一个元素 a 之前，都从现有队尾往前删除元素，一直删到碰到小于 a 的元素为止，然后再加入 a
- 那么队列就是递增的，当然队头的元素，一定是队列中最小的



北京大学
PEKING UNIVERSITY

状态压缩动态规划

状态压缩动态规划

- 有时，状态相当复杂，看上去需要很多空间，比如一个数组才能表示一个状态，那么就需要对状态进行某种编码，进行压缩表示。
- 比如：状态和某个集合有关，集合里可以有一些元素，没有另一些元素，那么就可以用一个整数表示该集合，每个元素对应于一个bit，有该元素，则该bit就是1。

例十 百练4124:海贼王之伟大航路

N 个城市，编号1到 N 。起点是1，终点是 N ($N \leq 16$)。

任意两个城市间都有路， $A \rightarrow B$ 和 $B \rightarrow A$ 的路可能不一样长。

已知所有路的长度，问经每个城市恰好一次的最短路径的长度

- 用 $dp[s][j]$ 表示经过集合 s 中的每个点恰好一次，且最后走的点是 j ($j \in s$) 的最佳路径的长度。
- 最终就是要求：
$$\min([dp[all][j]]) \quad (0 \leq j < N)$$

 all 是所有点的集合

- 状态方程: $dp[s][j] = \min\{ dp[s'] [k] + w[k][j] \}$
($j \in s$, $s' = s - j$, $k \in s'$, 枚举每个 k , $w[k][j]$ 是 k 到 j 的边权值)
- 边界条件: $dp[\{i\}][i] = 0$

- 问题：如何表示点集s？

由于只有16个点，可以用一个short变量表示点集。每个点对应一个bit。例如：

$$5 = 000000000000000101_2$$

5代表的点集是{0, 2}

全部n个点的点集，对应的整数是： $(1 \ll n) - 1$

最终要求： $\min(dp[(1 \ll n) - 1][j]) \quad (0 \leq j < n)$

- 问题：如何进行集合操作？

位运算。例：从集合*i*中去掉点*j*，得到新集合*s'*：

$$s' = s \& (\sim(1 \ll j))$$

或

$$s' = s - (1 \ll j)$$

- 问题：最终时间复杂度：

状态数目： $dp[s][j]$ $s: 0 - 2^n-1$ $j: 0 - (n-1)$

状态转移： $O(n)$

总时间： $O(n^2 2^n)$

例十一 课程大作业(百练 4149)

小明有多个课程大作业要交。每个课程大作业都有截止时间。如果提交时间超过截止时间 X 天，那么他将会被扣掉 X 分。对于每个大作业，小明要花费一天或者若干天来完成。他不能同时做多个大作业，只有他完成了当前的大作业，才可以开始一个新的大作业。小明希望你可以帮助他规划出一个最好的办法(完成大作业的顺序)来减少扣分。

例十一 课程大作业(百练 4149)

小明有多个课程大作业要交。每个课程大作业都有截止时间。如果提交时间超过截止时间 X 天，那么他将会被扣掉 X 分。对于每个大作业，小明要花费一天或者若干天来完成。他不能同时做多个大作业，只有他完成了当前的大作业，才可以开始一个新的大作业。小明希望你可以帮助他规划出一个最好的办法(完成大作业的顺序)来减少扣分。

例十一 课程大作业(百练 4149)

输入

输入的第一行是一个正整数 T ，代表测试样例数目。

对于每组测试样例，第一行为正整数 N ($1 \leq N \leq 15$) 代表课程数目。

接下来 N 行，每行包含一个字符串 S (不多于50个字符) 代表课程名称和两个整数 D (代表大作业截止时间) 和 C (完成该大作业需要的时间)。

注意所有的课程在输入中出现的顺序按照字典序排列。

输出

对于每组测试样例，请输出最小的扣分以及相应的课程完成的顺序。

如果最优方案有多个，请输出字典序靠前的方案。

例十一 课程大作业(百练 4149)

样例输入

2
3
Computer 3 3
English 20 1
Math 3 2
3
Computer 3 3
English 6 3
Math 6 3

样例输出

2
Computer
Math
English
3
Computer
English Math

例十一 课程大作业(百练 4149)

思路:

➤作业最多15个，可以状态压缩到一个short

例十一 课程大作业(百练 4149)

思路:

➤作业最多15个，可以状态压缩到一个short

➤ $dp[i]$ 表示已完成的作业为 i 时，最低的扣分数 (i 的第 k bit为1就表示第 k 个作业已经完成)

例十一 课程大作业(百练 4149)

思路:

➤作业最多15个，可以状态压缩到一个short

➤ $dp[i]$ 表示已完成的作业为 i 时，最低的扣分数 (i 的第 k bit为1就表示第 k 个作业已经完成)

➤ $dp[0] = 0$; 要求 $dp[2^{15}-1]$

例十一 课程大作业(百练 4149)

➤状态转移:

$$dp[i] = \min \{ dp[i - (1 \ll k)] + \text{score}(i, k) \mid i \text{ 的第 } k \text{ 位为 } 1 \}$$

$i - (1 \ll k)$: 完成的作业比 i 少了1个, 即第 k 个

$\text{score}(i, k)$: 完成第 k 个作业后增加的扣分

考虑 $s = D[k] - (\text{FinishDay}(i - (1 \ll k)) + C[k])$

$D[k]$ 第 k 个作业的截止日期, $C[k]$ 第 k 个作业的耗时
 $\text{FinishDay}(n)$ n 所代表的作业集合被完成时的日期

if ($s \leq 0$) $\text{score}(i, k) = 0$
else $\text{score}(i, k) = s$

例十一 课程大作业(百练 4149)

```
#include <iostream>
#include <cstring>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;
int t;
int n;
struct Homework
{
    string name;
    int d;    //截止时间
    int c;    //要花的时间
};
Homework hw[20];
```

```
struct Node
{
    int pre;           //上一个状态（比当前状态完成的作业少了1个）
    int minScore;      //当前状态的分数
    int last;          //当前状态下，最后完成的作业的编号
    int finishDay;     //作业last完成的时间
};

Node dp[(1 << 16) + 20];
vector<int> GetPath( int status)
{ //从状态 status 出发找出作业完成的顺序
    vector<int> path;
    while( status ) {
        path.push_back(dp[status].last);
        status = dp[status].pre;
    }
    reverse(path.begin(), path.end());
    return path;
}
```



```
int main()
{
    cin >> t;
    while(t--> 0) {
        cin >> n;
        char name[60];
        int d,c;
        for(int i = 0; i < n; ++i)
            cin >> hw[i].name >> hw[i].d >> hw[i].c;
        dp[0].finishDay = 0;
        dp[0].minScore = 0;
        dp[0].pre = -1;
        int m = 1 << n;
```

```

for(int i = 1; i < m; ++i) {//i代表已完成作业的集合
    dp[i].minScore = 1 << 30;
    for(int j = 0; j < n; ++j) {
        if( i & ( 1 << j )) {//i包含第j个作业
            int pre = i - ( 1 << j);
//要求 dp[i]时, 任何 dp[pre] 肯定已经算出了,因为 pre < i
            int finishDay =
                dp[pre].finishDay + hw[j].c;
            int tmpScore =
                finishDay - hw[j].d;
            if( tmpScore < 0)
                tmpScore = 0;
            if( dp[i].minScore >
                dp[pre].minScore + tmpScore ) {
                dp[i].minScore =
                    dp[pre].minScore + tmpScore;
                dp[i].pre = pre;
                dp[i].finishDay = finishDay;
                dp[i].last = j;
            }
        }
    }
}

```

```

        if( dp[i].minScore ==
            dp[pre].minScore + tmpScore){
            vector<int> p1 =
                GetPath(dp[i].pre);
            vector<int> p2 =
                GetPath(pre);
            if ( p2 < p1) {
                dp[i].pre = pre;
                dp[i].finishDay =
                    finishDay;
                dp[i].last = j;
            }
        }
    }
}

cout << dp[m-1].minScore << endl;

```

```
    int status = m-1;
    vector<int> path = GetPath(status);
    for( int i = 0; i < path.size(); ++ i)
        cout << hw[path[i]].name << endl;
}
return 0;
}
```

例十二 炮兵阵地 (POJ1185)

- 司令部的将军们打算在 $N \times M$ 的网格地图上部署他们的炮兵部队。一个 $N \times M$ 的地图由 N 行 M 列组成，地图的每一格可能是山地（用"H"表示），也可能是平原（用"P"表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

例十二 炮兵阵地 (POJ1185)

- 如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少炮兵部队。

- 数据范围： $1 \leq n \leq 100$, $1 \leq m \leq 10$

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

例十二 炮兵阵地 (POJ1185)

- 思路：如果用 $dp[i]$ 表示前 i 行所能放的最多炮兵数目，能否形成递推关系？显然不能。因为不满足无后效性

例十二 炮兵阵地 (POJ1185)

- 思路：如果用 $dp[i]$ 表示前 i 行所能放的最多炮兵数目，能否形成递推关系？显然不能。因为不满足无后效性
- 按照加限制条件加维度的思想，加个限制条件：
 $dp[i][j]$ 表示第 i 行的炮兵布局为 j 的前提下，前 i 行所能放的最多炮兵数目

布局为 j 体现了状态压缩。 j 是个10位二进制数，表示一行炮兵的一种布局。有炮兵的位置，对应位为1，没有炮兵的位置，对应位为0

例十二 炮兵阵地 (POJ1185)

- 思路：如果用 $dp[i]$ 表示前 i 行所能放的最多炮兵数目，能否形成递推关系？显然不能。因为不满足无后效性
- 按照加限制条件加维度的思想，加个限制条件：
 $dp[i][j]$ 表示第 i 行的炮兵布局为 j 的前提下，前 i 行所能放的最多炮兵数目

布局为 j 体现了状态压缩。 j 是个10位二进制数，表示一行炮兵的一种布局。有炮兵的位置，对应位为1，没有炮兵的位置，对应位为0

依然不满足无后效性。因仅从 $dp[i-1][k]$ ($k = 0 \cdots 1023$) 无法推出 $dp[i][j]$ 。达成 $dp[i-1][k]$ 可能有多种方案，有的方案允许第 i 行布局为 j ，有的方案不允许第 i 行布局为 j ，然而却没有信息可以用来进行分辨。

例十二 炮兵阵地 (POJ1185)

- 再加限制条件，再加一维：

$dp[i][j][k]$ 表示第 i 行布局为 j , 第 $i-1$ 行布局为 k 时, 前 i 行的最多炮兵数目。

- 1) j, k 这两种布局必须相容。否则 $dp[i][j][k] = 0$
- 2) $dp[i][j][k] = \max \{dp[i-1][k][m], m = 0 \dots 1023\} + \text{Num}(j)$,
 $\text{Num}(j)$ 为布局 j 中炮兵的数目, j 和 m 必须相容, k 和 m 必须相容。此时满足无后效性

例十二 炮兵阵地 (POJ1185)

- 再加限制条件，再加一维：

$dp[i][j][k]$ 表示第 i 行布局为 j ，第 $i-1$ 行布局为 k 时，前 i 行的最多炮兵数目。

- 1) j, k 这两种布局必须相容。否则 $dp[i][j][k] = 0$
- 2) $dp[i][j][k] = \max \{dp[i-1][k][m], m = 0 \dots 1023\} + \text{Num}(j)$,
 $\text{Num}(j)$ 为布局 j 中炮兵的数目， j 和 m 必须相容， k 和 m 必须相容。此时满足无后效性
- 3) 初始条件：
 $dp[0][j][0] = \text{Num}(j)$
 $dp[1][i][j] = \max \{dp[0][j][0]\} + \text{Num}(i)$

例十二 炮兵阵地 (POJ1185)

- 问题：dp数组为：
`int dp[100][1024][1024]`，太大，时间复杂度和空间复杂度都太高。

例十二 炮兵阵地 (POJ1185)

- 问题：dp数组为：

`int dp[100][1024][1024]`，太大，时间复杂度和空间复杂度都太高。

解决：

每一行里最多能放4个炮兵。就算全是平地，能放炮兵的方案数目也不超过60(用一遍dfs可以全部求出)

例十二 炮兵阵地 (POJ1185)

- 问题：dp数组为：

`int dp[100][1024][1024]`，太大，时间复杂度和空间复杂度都太高。

解决：

每一行里最多能放4个炮兵。就算全是平地，能放炮兵的方案数目也不超过60（用一遍dfs可以全部求出）

算出一行在全平地情况下所有炮兵的排列方案，存入数组 `state[70]`

`int dp[100][70][70]` 足矣

例十二 炮兵阵地 (POJ1185)

- 问题：dp数组为：

`int dp[100][1024][1024]`，太大，时间复杂度和空间复杂度都太高。

解决：

每一行里最多能放4个炮兵。就算全是平地，能放炮兵的方案数目也不超过60（用一遍dfs可以全部求出）

算出一行在全平地情况下所有炮兵的排列方案，存入数组 `state[70]`

`int dp[100][70][70]` 足矣

`dp[i][j][k]`表示第*i*行布局为`state[j]`，第*i-1*行布局为`state[k]`时，前*i*行的最多炮兵数目。