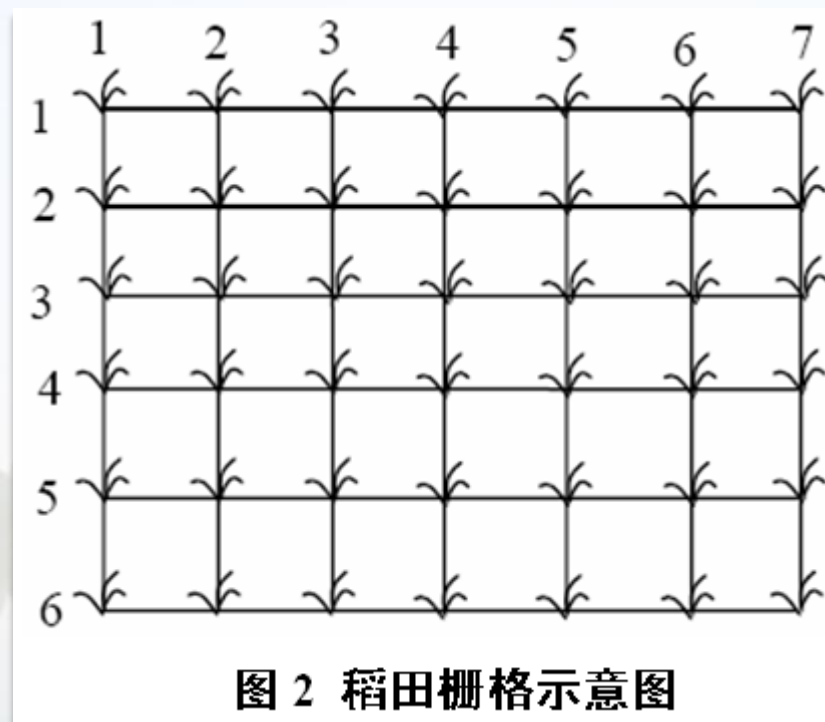


恼人的青蛙

在韩国, 有一种小青蛙. 每到晚上, 这种青蛙会跳越稻田, 从而踩踏稻子. 农民在早上看到被踩踏的稻子, 希望找到造成最大损害的那只青蛙经过的路径. 每只青蛙总是沿着一条直线跳越稻田, 而且每次跳跃的距离都相同, 如图1所示. 稻田里的稻子组成一个栅格, 如图2所示. 而青蛙总是从稻田的一侧跳进稻田, 然后沿着某条直线穿越稻田, 从另一侧跳出去, 如图3所示.



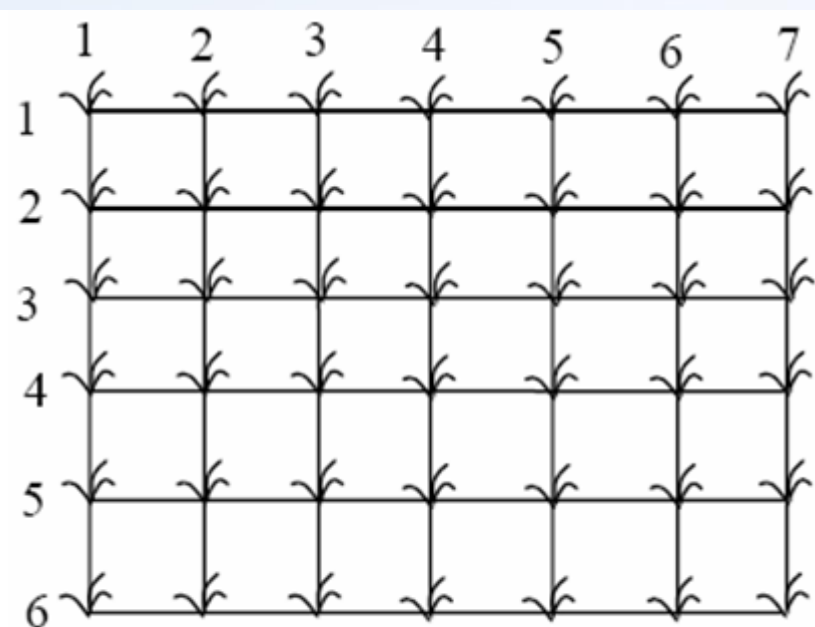
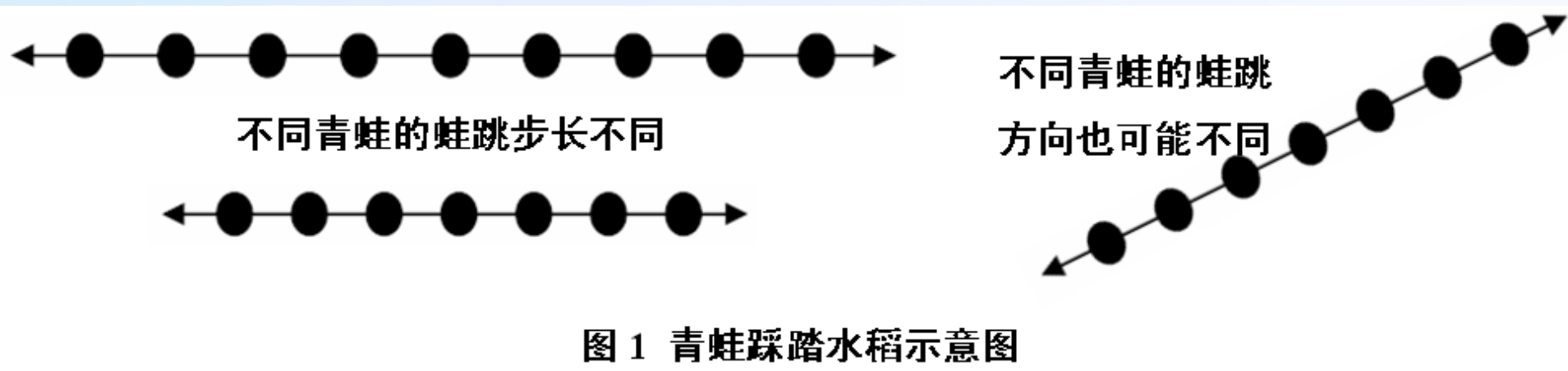


图 2 稻田栅格示意图

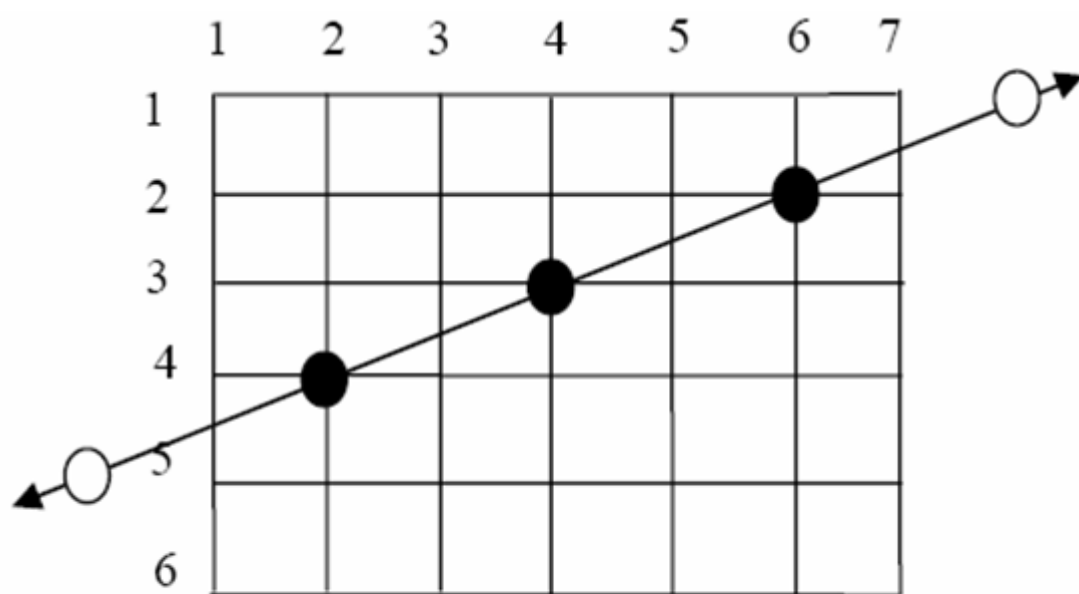
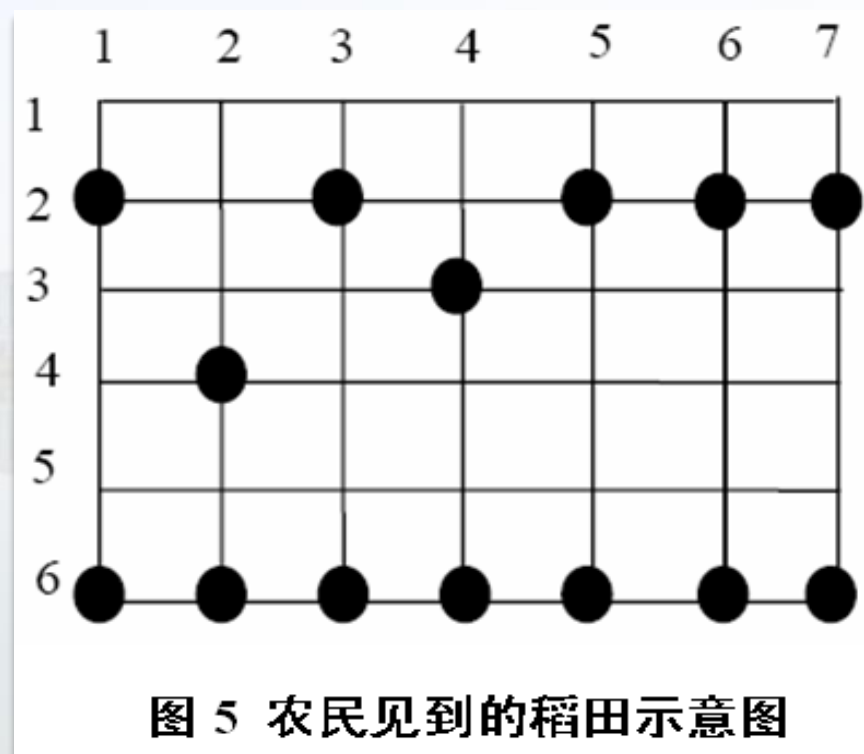
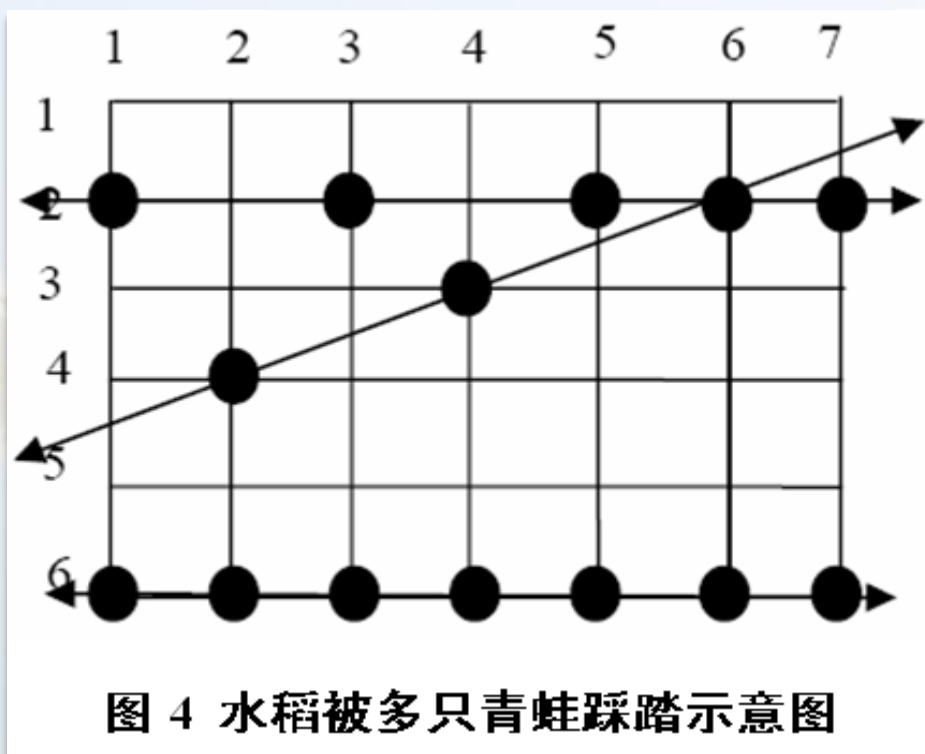


图 3 青蛙穿越稻田示意图

青蛙的每一跳都恰好踩在一棵水稻上, 将这棵水稻拍倒. 可能会有多只青蛙从稻田穿越, 有些水稻被多只青蛙踩踏, 如图4所示. 当然, 农民所见到的是图5中的情形, 看不到图4中的直线.



根据图5, 农民能够构造出青蛙穿越稻田时的行走路径, 并且只关心那些在穿越稻田时**至少踩踏了3 棵水稻的青蛙**. 因此, 每条青蛙行走路径上至少包括3 棵被踩踏的水稻. 而在一条青蛙行走路径的直线上, 也可能会有些被踩踏的水稻不属于该行走路径. 在图5中, 格点(2, 1)、(6, 1)上的水稻可能是同一只青蛙踩踏的, 但这条线上只有两棵被踩踏的水稻, 因此不能作为一条青蛙行走路径; 格点(2, 3)、(3, 4)、(6, 6)在同一条直线上, 但它们的间距不等, 因此不能作为一条青蛙行走路径; 格点(2, 1)、(2, 3)、(2, 5)、(2, 7)是一条青蛙行走路径, 该路径不包括格点(2, 6).

请你写一个程序, 确定在所有的青蛙行路径中, 踩踏水稻棵数最多的路径上有多少棵水稻被踩踏. 例如, 图5的答案是7, 因为第6行上全部水稻恰好构成一条青蛙行走路径.

● Standard Input

从标准输入设备上读入数据. 第一行上两个整数 R 、 C , 分别表示稻田中水稻的行数和列数, $1 \leq R, C \leq 5000$. 第二行是一个整数 N , 表示被踩踏的水稻数量, $3 \leq N \leq 5000$. 在剩下的 N 行中, 每行有两个整数, 分别是一颗被踩踏水稻的行号($1 \sim R$)和列号($1 \sim C$), 两个整数用一个空格隔开. 而且, 每棵被踩踏水稻只被列出一次.

● Standard Output

从标准输出设备上输出一个整数. 如果在稻田中存在青蛙行走路径, 则输出包含最多水稻的青蛙行走路径中的水稻数量, 否则输出0.

● Samples

Input	Output
6 7	7
14	
2 1	
6 6	
4 2	
2 5	
2 6	
2 7	
3 4	
6 1	
6 2	
2 3	
6 3	
6 4	
6 5	
6 7	

题目分析

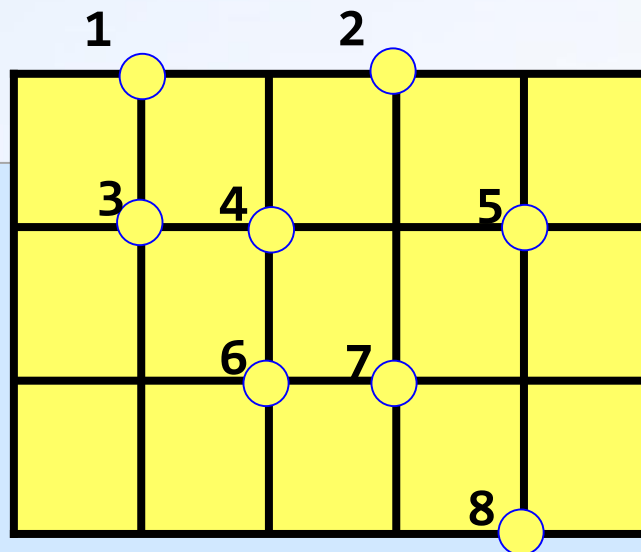
- 本题是帮助农民找到为害最大的青蛙. 也就是要找到一条穿越稻田的青蛙路径, 这个路径上被踩踏的水稻不少于其他任何青蛙路径上被踩踏的水稻数. 当然, 整个稻田中也可能根本就不存在青蛙路径.
- 问题的关键是: 找到穿越稻田的全部青蛙路径. 任何一条穿越稻田的青蛙路径 L , 至少包括3 棵被踩踏的水稻. 假设其中前两棵被踩踏的水稻分别是 (x_1, y_1) 、 (x_2, y_2) , 那么:
 - ➔ 令 $dx = x_2 - x_1$, $dy = y_2 - y_1$; $x_0 = x_1 - dx$, $y_0 = y_1 - dy$; $x_3 = x_2 + dx$, $y_3 = y_2 + dy$.
 - ➔ (x_0, y_0) 位于稻田之外, 青蛙从该位置经一跳后进入稻田、踩踏位置 (x_1, y_1) 上的水稻. (x_3, y_3) 是第三棵被踩踏的水稻.
 - ➔ $x_i = x_0 + i \times dx$, $y_i = y_1 + i \times dy$ ($i > 3$), 如果 (x_i, y_i) 位于稻田之内, 则 (x_i, y_i) 上的水稻必被青蛙踩踏.

- 根据上述规则，只要知道一条青蛙路径上的前两棵被踩踏的水稻，就可以找到该路径上其他的水稻。为了找到全部的青蛙路径，只要从被踩踏的水稻中，任取两棵水稻 $(x_1, y_1), (x_2, y_2)$ ，判断 $(x_1, y_1), (x_2, y_2)$ 是否能够作为一条青蛙路径上最先被踩踏的两颗水稻。
- 为方便，可以定义 `struct PLANT{ int x, y; }` 表示水稻，其中 x 是行， y 是列。
- 本题主要计算在于：从被踩踏的水稻中选择两棵 $(x_1, y_1), (x_2, y_2)$ ，确定蛙跳的方向和步长，判断它们能否能够作为一条青蛙路径上最先被踩踏的两颗水稻。接着从 (x_2, y_2) 开始，沿着这个方向和步长在稻田内走。每走一步，判断所到达位置上 (x, y) 的水稻是否被踩踏，直到走出稻田为止。如果在某一步上， (x, y) 没有被踩踏，则表明 $(x_1, y_1), (x_2, y_2)$ 是一条青蛙路径上最先被踩踏的两颗水稻的假设不成立。

- 用一个PLANT 型的数组plants[5001]表示全部被踩踏的水稻
- 将plants 中的元素按照行/列序号的升序(或者降序)排列
- 采用二分法查找plants中是否有值为(x,y)的元素, 将(x,y)与plants 中间的元素比较: (1)相等, 表明找到了元素; (2)比plants中间元素的小, 继续在plants 的前半部寻找; (3)比plants中间元素的大, 继续在plants 的后半部寻找.
- 采用上述方法判断每走一步所到达位置上(x,y)的水稻是否被踩踏, 最多只要比较 $\log_2 n$, 其中n是稻田中被踩踏水稻的总量.

参考程序

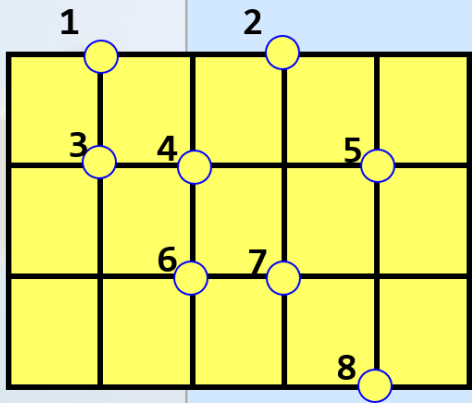
```
#include <cstdio>
#include <algorithm>
using namespace std;
int r,c,n;
struct PLANT //描述水稻
{
    int x, y; //行坐标及列坐标, 注意与平面坐标的区别
};
PLANT plants[5001];
PLANT plant;
bool myCompare(const PLANT& ele1,const PLANT& ele2)
{
    //排序及二分查找所用的比较函数
    if(ele1.x==ele2.x) //如果行坐标相同
        return (ele1.y < ele2.y); //则从左向右排列
    return (ele1.x < ele2.x); //否则从上到下排列
} //被踩水稻排序: 从左到右, 再从上到下
int searchPath(PLANT secPlant, int dX, int dY); //搜索路径
```



```

int main(void)
{
    int i,j,dX,dY,pX,pY,steps,max=2;
    scanf("%d%d",&r,&c);
    scanf("%d",&n);
    for(i=0;i<n;i++) scanf("%d %d",&plants[i].x,&plants[i].y);
    sort(plants,plants+n,myCompare); //被踩水稻排序
    for(i=0;i<n-2;i++) //双重循环查找所有可能路径
        for(j=i+1;j<n-1;j++) //路径方向: 朝右、左下、下或右下
        {
            dX=plants[j].x-plants[i].x;
            dY=plants[j].y-plants[i].y;
            pX=plants[i].x-dX; //用于判断plant[i]是否是起点
            pY=plants[i].y-dY; //但换第二点方向变了可能是
            if(pX<=r&& pX>=1&& pY<=c&& pY>=1) continue;
            if(plants[i].x+max*dX>r) break; //dX只会增大
            pY=plants[i].y+max*dY;
            if(pY>c||pY<1) continue; //换第二, 可能有更长路径
            steps=searchPath(plants[j],dX,dY); //判断和求长度
            if(steps>max) max=steps;
        }
}

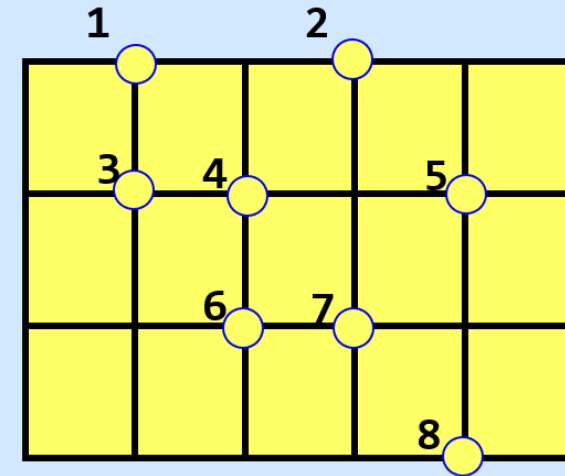
```



```

if(max==2) max=0;
printf("%d\n",max);
return 0;
}
int searchPath(PLANT secPlant,int dX,int dY )
{
    PLANT plant;
    int steps;
    plant.x=secPlant.x+dX;
    plant.y=secPlant.y+dY;
    steps=2;
    while(plant.x<=r&&plant.x>=1&&plant.y<=c&&plant.y>=1)
    {
        //没有出界
        if(!binary_search(plants,plants+n,plant,myCompare))
        {
            //应该有的但不在，所以不能成为踩踏路径
            steps=0; break;
        }
        plant.x+=dX; plant.y+=dY;    steps++;
    }
    return steps;
}

```



二、枚举排列

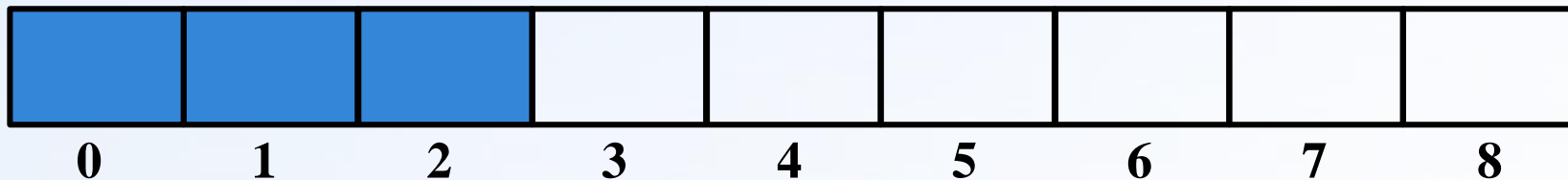
问题：枚举自然数1、2、3、4全排列。

```
选择D:\Projects\C_C+
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int i, j, k, l, m, a[4];
    for(i = 1; i <= 4; i++) {
        a[0] = i;
        for(j = 1; j <= 4; j++){
            if( j == i) continue;
            a[1] = j;
            for(k = 1; k <= 4; k++) {
                if(k == j || k == i) continue;
                a[2] = k;
                for(l = 1; l <= 4; l++) {
                    if(l==k || l==j || l==i) continue;
                    a[3] = l;
                    for(m = 0; m < 4; m++)
                        printf("%d ", a[m]);
                    printf("\n");
                }
            }
        }
    }
    return 0;
}
```


生成1~n的排列

由于不知n的值, 无法用多重循环实现. 考虑递归, 用n控制递归深度.



```
void create_permutation(int n, int* a, int cur)
{
    if(当前深度cur == 总深度n)
        从a中打印已生成的排列
    else
    {
        根据当前深度cur试探向a中放元素 (用循环)
        递归调用, 深度增1, 即create_permutation(n, a, cur+1)
    }
}
```

选择D:\Projects\C_C+

1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
2	1	4	3
2	3	1	4
2	3	4	1
2	4	1	3
2	4	3	1
3	1	2	4

参考程序

```
#include <stdio.h>
#include <string.h>
#define N 10
void create(int n, int* a, int cur);
int main()
{
    int n, a[N];
    while(scanf("%d", &n) == 1)
        create(n, a, 0);
    return 0;
}
```

0	1	2	3	4	5	6	7	8	

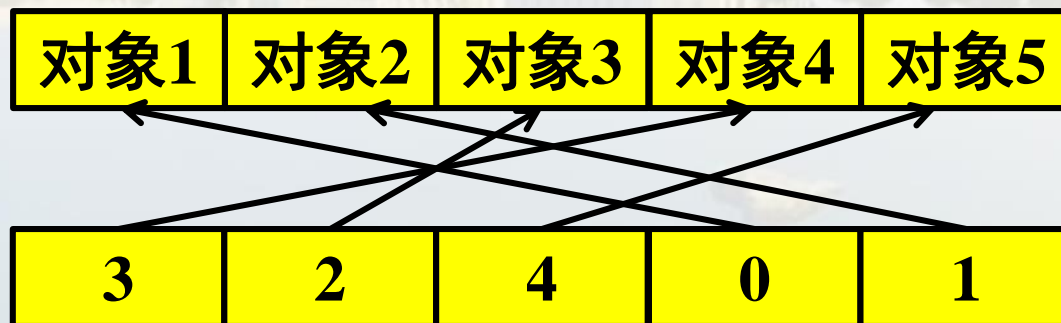
```
void create(int n, int* a, int cur)
{
    int i, j;
    if(cur == n)
    {
        for(i = 0; i < n; i++)
            printf("%d ", a[i]);
        printf("\n");
    }
    else
    {
        for(i = 1; i <= n; i++)
        {
            for(j = 0; j < cur; j++)
                if(a[j] == i) break;
            if(j == cur)
            {
                a[cur] = i;
                create(n, a, cur+1);
            }
        }
    }
}
```

问题:

- 1、如果有n个不相同的元素，如何修改上面的程序得到它们所有的排列？
- 2、如何从一个排列，得到字典序大于它的最小排列？

最简单的方法，使用C++中STL的库函数：

```
template <class BidirectionalIterator, class Compare>  
bool next_permutation (BidirectionalIterator first,  
                        BidirectionalIterator last, Compare comp);
```



参考程序

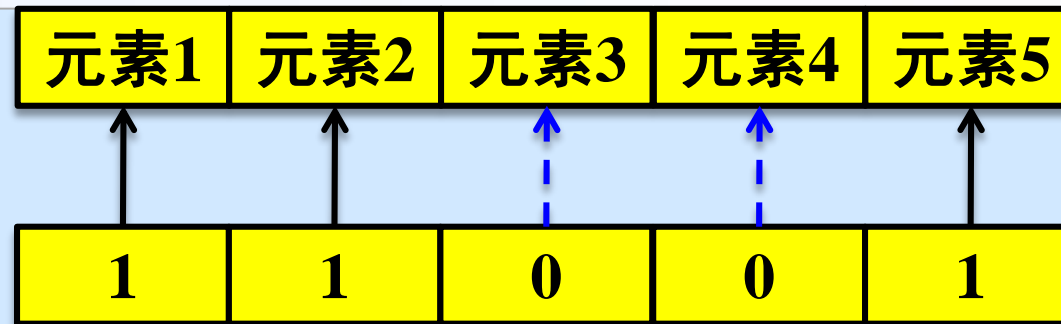
```
#include <bits/stdc++.h>
using namespace std;
#define N 10
int main()
{
    int i, n, a[N];
    while(scanf("%d", &n) == 1)
    {
        for(i = 0; i < n; i++)
            scanf("%d", &a[i]);
        sort(a, a+n);
        do
        {
            for(i = 0; i < n; i++)
                printf("%d ", a[i]);
            printf("\n");
        }while(next_permutation(a, a+n));
        printf("\n");
    }
    return 0;
}
```

输入	输出
3	1 2 3
1 2 3	1 3 2
3	2 1 3
1 2 1	2 3 1
	3 1 2
	3 2 1
	1 1 2
	1 2 1
	2 1 1

三、子集生成

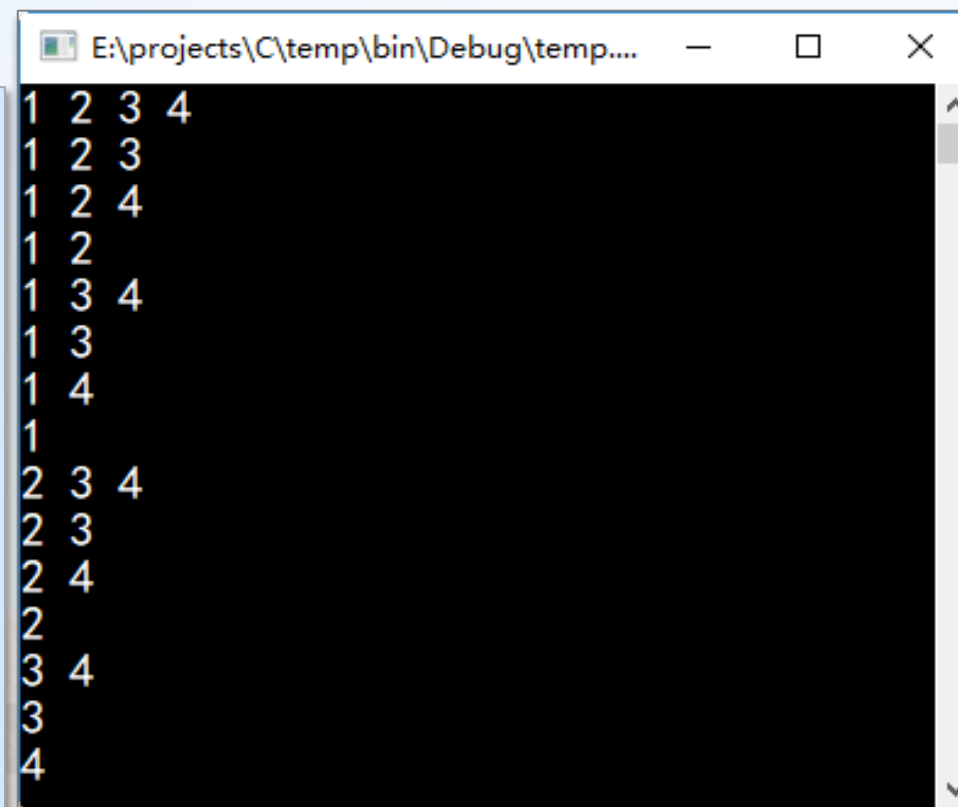
位向量法：假设有 n 个不相同元素

```
void create_subset(int n, int* a, int cur)
{
    if(当前深度cur == 总深度n)
        由a中元素值(0或1)打印一个子集
    else
    {
        选择第cur个元素(a[cur]=1), create_subset(n, a, cur+1);
        不选择第cur个元素(a[cur]=0), create_subset(n, a, cur+1);
    }
}
```



参考程序

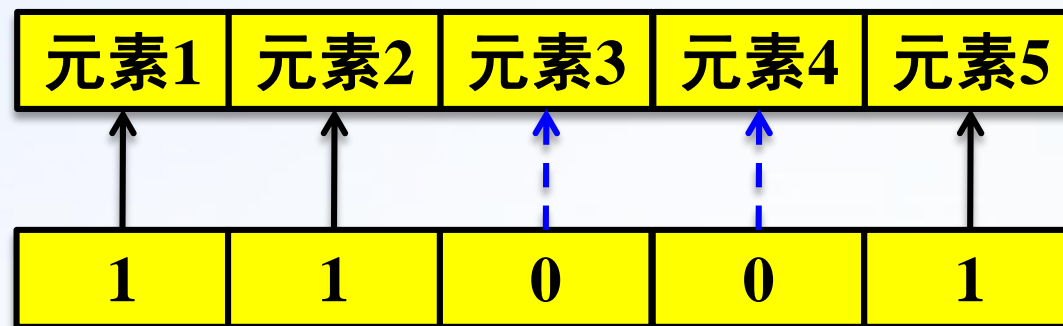
```
#include <stdio.h>
int b[4] = {1,2,3,4};
void create_subset(int n, int* a,int cur)
{
    int i, j;
    if(cur == n)
    {
        for(i = 0; i < n; i++)
            if(a[i] == 1)
                printf("%d ", b[i]);
        printf("\n");
    }
    else
    {
        a[cur] = 1;
        create_subset(n, a, cur+1);
        a[cur] = 0;
        create_subset(n, a, cur+1);
    }
}
```



```
int main()
{
    int a[4];
    create_subset(4, a, 0);
    return 0;
}
```

二进制法

```
#include <stdio.h>
int b[10];
void create_subset(int n, int cur);
int main()
{
    int i, n;
    while(scanf("%d", &n) == 1)
    {
        for(i = 0; i < n; i++)
            scanf("%d", &b[i]);
        for(i = 0; i < (1<<n); i++)
            create_subset(n, i);
    }
    return 0;
}
```



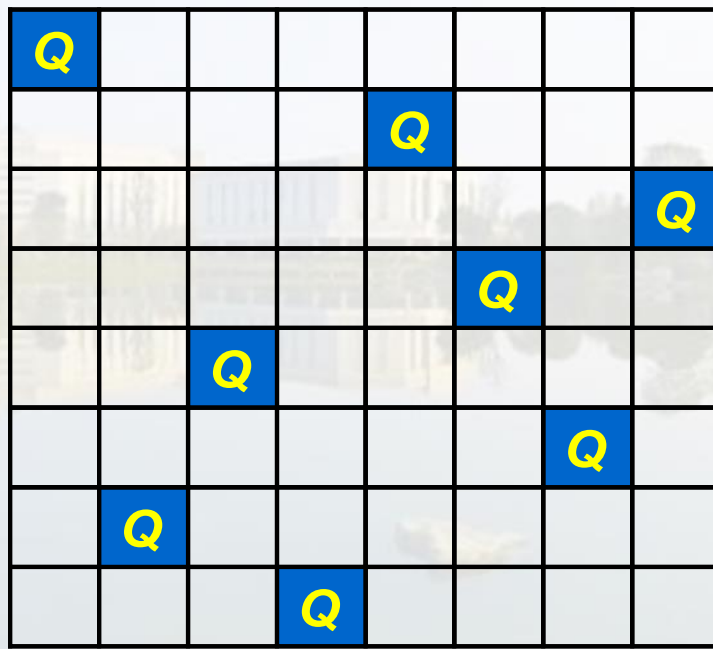
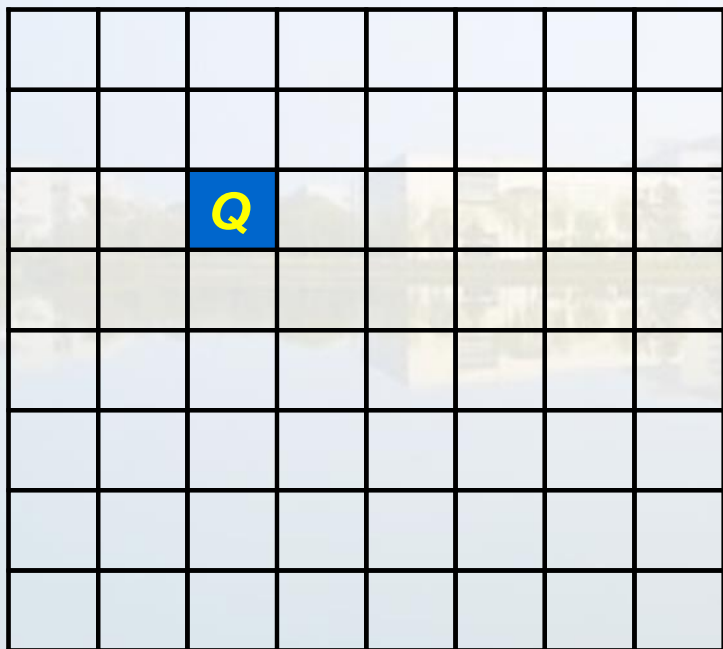
```
void create_subset(int n, int cur)
{
    int i;
    for(i = 0; i < n; i++)
        if(cur & (1<<i))
            printf("%d ", b[i]);
    printf("\n");
}
```

四、回溯法

- 回溯法(探索与回溯法)是一种选优搜索法, 又称为**试探法**, 按选优条件向前搜索, 以达到目标. 但当探索到某一步时, 发现原先选择并不优或达不到目标, 就退回一步重新选择, 这种走不通就退回再走的技术为回溯法.
- 在包含问题的所有解的解空间树中, 按照深度优先搜索的策略, 从根结点出发深度探索解空间树. 当探索到某一结点时, 要先判断该结点是否包含问题的解, 如果包含, 就从该结点出发继续探索下去, 如果该结点不包含问题的解, 则逐层向其祖先结点回溯. (其实**回溯法就是对隐式图的深度优先搜索算法**).
- 若用回溯法求问题的所有解时, 要回溯到根, 且根结点的所有可行的子树都要已被搜索遍才结束. 而若使用回溯法求任一个解时, 只要搜索到问题的一个解就可以结束.

八皇后问题

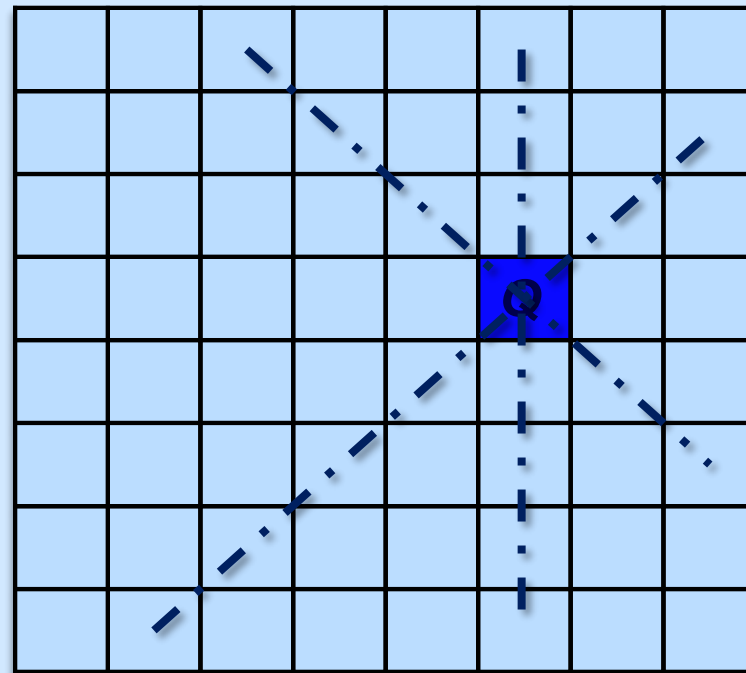
在棋盘上放置8个皇后，使得她们互不攻击，此时每个皇后的攻击范围为同行同列和同对角线，要求找出所有解。



Input	Output
2	No solution!
4	2 4 1 3
5	3 1 4 2
	1 3 5 2 4
	1 4 2 5 3
	2 4 1 3 5
	2 5 3 1 4
	3 1 4 2 5
	3 5 2 4 1
	4 1 3 5 2
	4 2 5 3 1
	5 2 4 1 3
	5 3 1 4 2

参考程序

```
void search(int cur)
{
    int i, j;
    if (cur == n)
    {
        tot++;
        for(i = 0; i < n; i++)
            printf("%d ", s[i]+1);
        printf("\n");
    }
    else
    {
        for(i = 0; i < n; i++)
        {
            s[cur] = i; //尝试cur行放在第i列
            for(j = 0; j < cur; j++)
                if((s[j]==s[cur]) || (cur-j == s[cur]-s[j]) || (cur-j == s[j]-s[cur]))
                    break;
            if(j == cur) search(cur+1); //如果合法继续递归，否则重新试探
        }
    }
}
```



马的走法

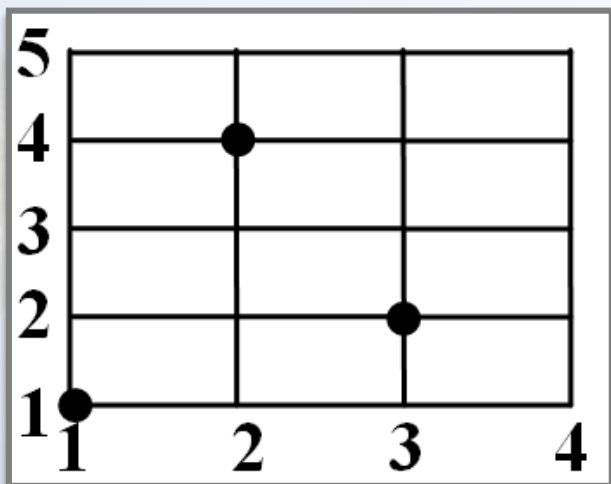
在一个 5×4 的棋盘上，马的起始位置坐标(横，纵)位置由键盘输入，求马能返回初始位置的所有不同走法的总数(马走过的位置不能重复，马走“日”字)。

- Samples

Input	Output
2 2	4596
1 1	1508
1 3	4772
2 3	5460

题目分析

- 由于 4×5 的棋盘的问题规模比较小, 用回溯法可以较好的解决问题.
- 如下图所示, 如果从(1,1)出发, 那么马首先从(1,1)点走向(3,2)点, 再到(2,4)点, 这样过程可以一步一步递推下去, 最后要么找到一条路径, 要么进入“死胡同”, 这时函数dfs(p1,p2)就不能递归调用自己了, 因此实现了回溯.



- 为了避免走重复的位置, 可以定义一个数组 `visited[5][4]` 表示棋盘, 若 `visited[i][j]=1`, 则表示马已经走过, 在递推时就不能走此位置, 应该换一个方向.
- 对于马走的方向, 可以定义数组 `d[2][8]`, 每列表示一个马可以走的方向.

参考程序

```
#include <stdio.h>
#include <string.h>
#define M 5
#define N 4
int sr, sc, counter, flag, visited[M+1][N+1];
int d[2][8]={{-1,-1,-2,-2,2,2,1,1},
              {2,-2,1,-1,1,-1,2,-2}};
void dfs(int row, int col);
int main()
{
    while(scanf("%d%d", &sr, &sc) == 2)
    {
        counter = 0;  flag = 0; //标志dfs是不是第一次调用，避免误判
        memset(visited, 0, sizeof(visited));
        dfs(sr, sc);
        printf("%d\n", counter);
    }
    return 0;
}
```

```
void dfs(int row, int col)
{
    if(flag != 0 && row == sr && col == sc) counter++;
    else
    {
        int i,pr,pc;
        flag = 1;
        for(i = 0; i < 8; i++)
        {
            pr = row + d[0][i]; pc = col + d[1][i];
            if((pr >= 1) && (pr <= M) && (pc >= 1) && (pc <= N) && (visited[pr][pc] == 0))
            {
                visited[pr][pc] = 1;
                dfs(pr, pc);
                visited[pr][pc] = 0;
            }
        }
    }
}
```

素数环

输入正整数 n , 把整数 $1, 2, 3, \dots, n$ 组成一个环, 使得相邻两个整数之和均为素数. 输出时从整数1开始逆时针排列. 同一个环应恰好输出一次. $n \leq 16$.

- Samples

Input	Output
6	1 4 3 2 5 6 1 6 5 2 3 4

参考程序

```
#define N 20
int n, tot, s[N+1], vis[N+1], prime[2*N];
void dfs(int cur);
int main()
{
    int i, j, k;
    memset(prime, 0, sizeof(prime));
    for(i = 2; i < 2*N; i++) //生成素数表
    {
        k = (int)sqrt(i);
        for(j = 2; j <= k; j++) if(i%j == 0) break;
        if(j > k) prime[i] = 1;
    }
    while(scanf("%d", &n) == 1)
    {
        tot = 0; s[0] = 1;
        memset(vis, 0, sizeof(vis));
        dfs(1);
        if(tot == 0) printf("No solution!\n");
    }
    return 0;
}
```

```
void dfs(int cur)
{
    int i;
    if((cur == n) && (prime[s[0]+s[n-1]]))
    {
        tot++;
        for(i = 0; i < n; i++) printf("%d ", s[i]);
        printf("\n");
    }
    else
    {
        for(i = 2; i <= n; i++)
        { //试探
            if(!vis[i] && prime[i+s[cur-1]])
            {
                s[cur] = i; vis[i] = 1;
                dfs(cur+1);
                vis[i] = 0;
            }
        }
    }
}
```