



中国科学技术大学

University of Science and Technology of China

程序设计II

第6讲 枚举

计算机学院 黄章进

zhuang@ustc.edu.cn

- 例题：称硬币 2692
- 例题：完美立方 2810
- 例题：熄灯问题 2811
- 例题：讨厌的青蛙 2812

- 枚举是一种解决问题的方法。
- 例如：求小于 N 的最大素数
 - 找不到一个数学公式，使得我们根据 N 就可以计算出这个素数。怎么办？
 - 逐一实验： $N-1$ 是素数吗？ $N-2$ 是素数吗？
 - $N-K$ 是素数的充分必要条件是： $N-K$ 不能被任何一个大于1、小于 $N-K$ 的素数整除。
 - 判断 $N-K$ 是否是素数的问题又成了求小于 $N-K$ 的全部素数。

- 解决方法：
 - 2是素数，记为 PRIM_0
 - 根据 PRIM_0 、 PRIM_1 、...、 PRIM_k ，寻找比 PRIM_k 大的最小素数 PRIM_{k+1} 。如果 PRIM_{k+1} 大于N，则 PRIM_k 是我们需要找的素数，否则继续寻找

- 根据所知道的**知识**，给一个猜测的答案：2是素数
- 判断猜测的答案是否正确：2是小于N的最大素数吗？
- 进行**新**的猜测：有两个关键因素要注意
 - 猜测的结果必须是前面的猜测中没有出现过的
 - 每次猜测时，素数一定比已经找到的素数大
 - 猜测的过程中要及早排除错误的答案
 - 除2之外，只有奇数才可能是素数

- 列出所有可能的情况，逐一检查是否是问题的解
- 两个关键：
 - 有序地枚举解空间，不漏掉情况
 - 尽早发现不是解的情况

例题：称硬币



- 问题描述

- 赛利有12枚银币。其中有11枚真币和1枚假币。假币看起来和真币没有区别，但是重量不同。但赛利不知道假币比真币轻还是重。
- 于是他向朋友借了一架天平。朋友希望赛利称三次就能找出假币并且确定假币是轻是重。例如：如果赛利用天平称两枚硬币，发现天平平衡，说明两枚都是真的。如果赛利用一枚真币与另一枚银币比较，发现它比真币轻或重，说明它是假币。
- 经过精心安排每次的称量，赛利保证在称三次后确定假币。

- 输入
 - 第一行有一个数字 n ，表示有 n 组测试用例。
 - 对于每组测试用例：输入有三行，每行表示一次称量的结果。赛利事先将银币标号为A-L。每次称量的结果用三个以空格隔开的字符串表示：
天平左边放置的硬币 天平右边放置的硬币 平衡状态
 - 其中平衡状态用"up", "down", 或 "even"表示, 分别为右端高、右端低和平衡。天平左右的硬币数总是相等的
- 输出
 - 输出哪一个标号的银币是假币，并说明它比真币轻还是重(heavy or light)。

- 输入样例

1

ABCD EFGH even

ABCI EFJK up

ABIJ EFGH even

- 输出样例

K is the counterfeit coin and it is light.

- 问题分析

- 此题并非要求你给出如何称量的方案，而是数据已经保证三组称量后答案唯一。不是那种传统的智商测验题。
- 此题可以有多种解法。这里只介绍一种比较容易想到和理解的 – 逐一枚举法。

- 答案可以用两个变量表示： x - 假币的标号、 w - 假币是比真币轻还是比真币重。
 - x 共有 12 种猜测； w 有 2 种猜测。
- 根据赛利设计的称量方案， (x, w) 的 24 种猜测中，只有唯一的猜测与三组称量数据都不矛盾
- 因此，如果猜测 (x, w) 满足下列条件，这个猜测就是要找的答案：
 - 在称量结果为"even" 的天平两边，没有出现 x
 - 如果 w 表示假币比真币轻，则在称量结果为"up"的天平右边一定出现 x 、在称量结果为"down"的天平左边一定出现 x
 - 如果 w 表示假币比真币重，则在称量结果为"up"的天平左边一定出现 x 、在称量结果为"down"的天平右边一定出现 x

- 总体构想 – 逐一试探法
 - 对于每一枚硬币
 - 先假设它是轻的，看这样是否符合称量结果。如果符合，问题即解决。
 - 如果不符合，就假设它是重的，看是否符合称量结果。
 - 把所有硬币都试一遍，一定能找到特殊硬币。

- 定义变量存储称量结果

```
char left[3][7], right[3][7], result[3][5];
```

- 数组下标 3 代表3次称量;
- 数组下标 7 代表每次左右至多6枚硬币，多出一个字符位置是为了放'\0'，以便使用字符串函数

2692 称硬币



```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
char left[3][7], right[3][7], result[3][5];
bool isHeavy( char ); // 判断假币 x 是否为重的代码
bool isLight( char ); // 判断假币 x 是否为轻的代码
int main() {
    int n, i;
    char c;
    scanf("%d", &n);
    while ( n > 0 ) {
        for ( i = 0; i < 3; i++ )
            scanf("%s %s %s", left[i], right[i], result[i]);
        for ( c = 'A'; c <= 'L'; c++ ) {
            if ( isLight(c) ) {
                printf("%c is the counterfeit coin and it is light.\n", c);
                break;
            }
            if ( isHeavy(c) ) {
                printf("%c is the counterfeit coin and it is heavy.\n", c);
                break;
            }
        }
        n--;
    }
}
```

称硬币



```
bool isLight( char x )    // 判断假币 x 是否为轻的代码
```

```
{
```

```
    // 判断是否与三次称量结果矛盾
```

```
    for ( int i = 0; i < 3; i++ ) { // 请补全循环体代码
```

```
char left[3][7], right[3][7], result[3][5];  
// result[i]: "up", "even" or "down"
```

```
}
```

```
return true;
```

```
}
```

例题：完美立方



- 问题描述

- $a^3 = b^3 + c^3 + d^3$ 为完美立方等式。例如 $12^3 = 6^3 + 8^3 + 10^3$
- 编写一个程序，对任给的正整数 N ($N \leq 100$)，寻找所有的四元组 (a, b, c, d) ，使得 $a^3 = b^3 + c^3 + d^3$ ，其中 $1 < a, b, c, d \leq N$ ，且各不相等。

- 输入
 - 正整数 N ($N \leq 100$)
- 输出
 - 每行输出一个完美立方，按照 a 的值，从小到大依次输出。当两个完美立方等式中 a 的值相同，则依次按照 b 、 c 、 d 进行非降升序排列输出，即 b 值小的先输出、然后 c 值小的先输出、然后 d 值小的先输出。

- 输入样例

24

- 输出样例

Cube = 6, Triple = (3,4,5)

Cube = 12, Triple = (6,8,10)

Cube = 18, Triple = (2,12,16)

Cube = 18, Triple = (9,12,15)

Cube = 19, Triple = (3,10,18)

Cube = 20, Triple = (7,14,17)

Cube = 24, Triple = (12,16,20)

- 解题思路

- 给定 4 个整数的四元组(a, b, c, d)，判断它们是否满足完美立方等式 $a^3 = b^3 + c^3 + d^3$
- 对全部的四元组进行排序，依次进行判断。如果一个四元组满足完美立方等式，则按照要求输出
- 先判断 a 值小的四元组；两个四元组的 a 值相同，则先判断b 值小的；两个四元组的 a 值和 b 值分别相同，则先判断 c 值小的

- 关键问题

- 确定全部需要判断的四元组(a, b, c, d)，并对它们进行排序
 - (1) $a \geq 6$ ，因为 a 最小必须是 5，才能使得 b、c、d 分别是 3 个大于 1 的不同整数，但(5, 2, 3, 4)不满足完美立方等式的要求；
 - (2) $1 < b < c < d$ ，否则该四元组在序列中的位置就要向前移；
 - (3) 如果(a, b, c, d)满足完美立方等式，则 b、c、d 都要比 a 小。
- 避免对一个整数的立方的重复计算。
 - 在开始完美立方等式的判断之前，先用一个数组保存[2 N]中的每个整数的立方值。

2810 完美立方



中国科学技术大学
University of Science and Technology of China

```
#include <stdio.h>
int main( ) {
    int n, i, a, b, c, d;
    long int cube[101];
    scanf("%d", &n);
    for ( i = 1; i <= n; i++ )
        cube[i] = i * i * i;
    for ( a = 6; a <= n; a++ ) // 请补全循环体代码
```

四元组输出格式:
Cube = 6, Triple = (3,4,5)

```
}
```

例题：熄灯问题



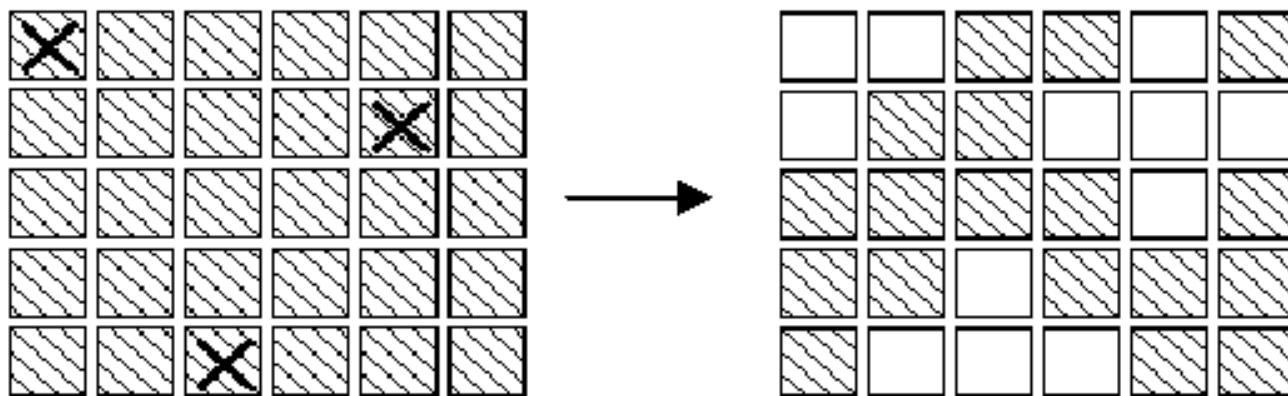
- 问题描述

- 有一个由按钮组成的矩阵，其中每行有6个按钮，共5行。
- 每个按钮的位置上有一盏灯。当按下一个按钮后，该按钮以及周围位置(上边、下边、左边、右边)的灯都会改变一次。即，如果灯原来是点亮的，就会被熄灭；如果灯原来是熄灭的，则会被点亮。
 - 在矩阵角上的按钮改变3盏灯的状态
 - 在矩阵边上的按钮改变4盏灯的状态
 - 其他的按钮改变5盏灯的状态

熄灯问题



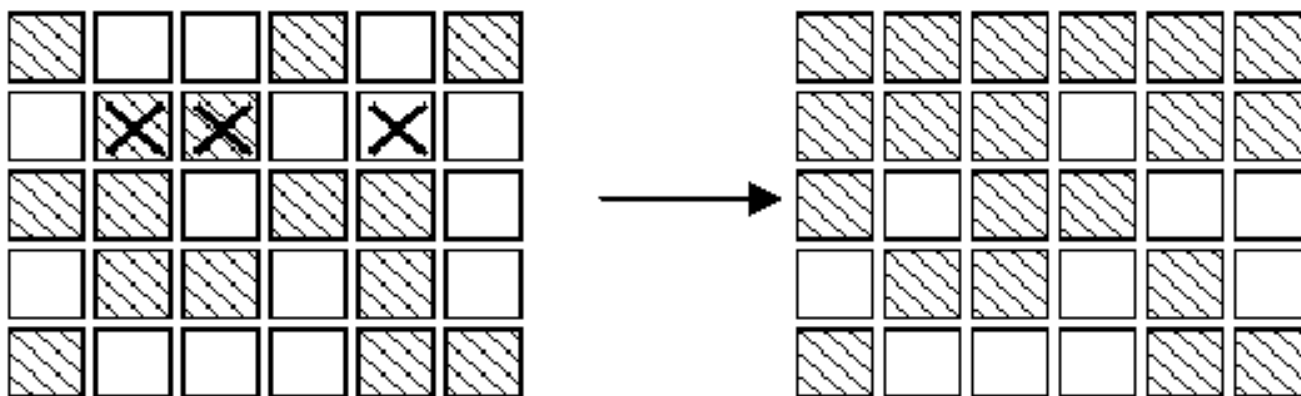
- 在下图中，左边矩阵中用X标记的按钮表示被按下，右边的矩阵表示灯状态的改变。



熄灯问题



- 与一盏灯毗邻的多个按钮被按下时，一个操作会抵消另一次操作的结果。在下图中，第2行第3、5列的按钮都被按下，因此第2行、第4列的灯的状态就不改变。



- 对矩阵中的每盏灯设置一个初始状态。请你写一个程序，确定需要按下哪些按钮，恰好使得所有的灯都熄灭。

- 输入
 - 第一行是一个正整数N，表示需要解决的案例数。每个案例由5行组成，每一行包括6个数字。这些数字以空格隔开，可以是0或1。0表示灯的初始状态是熄灭的，1表示灯的初始状态是点亮的。
- 输出
 - 对每个案例，首先输出一行，输出字符串“PUZZLE #m”，其中m是该案例的序号。接着按照该案例的输入格式输出5行，其中的1表示需要把对应的按钮按下，0则表示不需要按对应的按钮。每个数字以一个空格隔开。

熄灯问题



中国科学技术大学
University of Science and Technology of China

- 输入样例

2

0 1 1 0 1 0

1 0 0 1 1 1

0 0 1 0 0 1

1 0 0 1 0 1

0 1 1 1 0 0

0 0 1 0 1 0

1 0 1 0 1 1

0 0 1 0 1 1

1 0 1 1 0 0

0 1 0 1 0 0

- 输出样例

PUZZLE #1

1 0 1 0 0 1

1 1 0 1 0 1

0 0 1 0 1 1

1 0 0 1 0 0

0 1 0 0 0 0

PUZZLE #2

1 0 0 1 1 1

1 1 0 0 0 0

0 0 0 1 0 0

1 1 0 1 0 1

1 0 1 1 0 1

- 解题思路

- 第2次按下同一个按钮时，将抵消第1次按下时所产生的结果。因此，每个按钮最多只需要按下一次。
- 各个按钮被按下的顺序对最终的结果没有影响
- 对第1行中每盏点亮的灯，按下第2行对应的按钮，就可以熄灭第1行的全部灯。如此重复下去，可以熄灭第1、2、3、4行的全部灯。

- 解题思路

- 第一种想法：枚举所有可能的按钮（开关）状态，对每个状态计算一下最后灯的情况，看是否都熄灭。每个按钮有两种状态(按下或不按下)，一共有30个开关，那么状态数是 2^{30} ，太多，会超时。
- 如何减少枚举的状态数目呢？一个基本思路是，如果存在某个局部，一旦这个局部的状态被确定，那么剩余其他部分的状态只能是确定的一种，或者不多的n种，那么就只需枚举这个局部的状态就行了。

- 本题是否存在这样的“局部”呢？
- 经过观察，发现第1行就是这样的“局部”。因为第1行的各开关状态确定的情况下，这些开关作用过后，将导致第1行某些灯是亮的，某些灯是灭的。此时要熄灭第1行某个亮着的灯（假设位于第 i 列），那么唯一的办法就是按下第2行第 i 列的开关（因为第一行的开关已经用过了，而第3行及其后的开关不会影响到第1行）。因此，为了使第1行的灯全部熄灭，第2行的合理开关状态就是唯一的。

- 第2行的开关起作用后，为了熄灭第二行的灯，第3行的合理开关状态就也是唯一的，以此类推，最后一行的开关状态也是唯一的。
- 总之，只要第1行的状态定下来，比如叫A，那么剩余行的情况就是确定唯一的了。推算出最后一行的开关状态，然后看看最后一行的开关起作用后，最后一行的所有灯是否都熄灭，如果是，那么A就是一个解的状态。如果不是，那么A不是解的状态，第1行换个状态重新试试。
- 因此，只需枚举第一行的状态，状态数是 $2^6 = 64$

熄灯问题



中国科学技术大学
University of Science and Technology of China

有没有状态数更少的做法？

- 枚举第一列，状态数是 $2^5 = 32$

- 用矩阵`anPuzzle [5][6]`表示灯的初始状态
 - `anPuzzle[i][j]=1`: 灯(i, j)初始时是被点亮的
 - `anPuzzle[i][j]=0`: 灯(i, j)初始时是熄灭的
- 用矩阵`anSwitch [5][6]`表示要计算的结果
 - `anSwitch[i][j]=1`: 需要按下按钮(i, j)
 - `anSwitch[i][j]=0`: 不需要按下按钮(i, j)

- anSwitch[0]里放着第1行开关的状态，如何进行枚举呢？
- 可以使用六重循环：

```
for( int a0 = 0; a0 < 2; a0 ++ )  
    for( int a1 = 0; a1 < 2; a0 ++ )  
        for( int a2 = 0; a2 < 2; a0 ++ )  
            for( int a3 = 0; a3 < 2; a0 ++ )  
                for( int a4 = 0; a4 < 2; a0 ++ )  
                    for( int a5 = 0; a5 < 2; a0 ++ )  
                        {  
                            anSwitch[0][0] = a0;                anSwitch[0][1] = a1;  
                            anSwitch[0][2] = a2;                .....  
                        }
```

- 如果每行灯很多，或每行开关数目是可变数N那怎么办？

适用于一行有N个开关的办法

- 一个6位二进制数的所有取值正好是64种。让该数的每一位对应于anSwitch[0]里的一个元素：anSwitch[0][5] 对应最高位，anSwitch[0][4]对应次高位.....，那么这个二进制数的每个取值正好表示了第一行开关的一种状态。
- 如果一行有N个开关，那么就用一个N位二进制数。
- 比如：
 - 0 的二进制表示是 000000，即代表所有开关都不按下
 - 63的二进制表示是 111111，即代表所有开关都按下
 - 5 的二进制表示是 000101，即代表右数第1，3个开关按下

- 要写一个从二进制数到状态的转换函数：
void SwitchStatus(int k, int * pSwitch);
- 该函数将整数 k ($0 \leq k < 64$) 的二进制表示形式对应到数组pSwitch里去
 - anSwitch[0][i] 对应第i位（从右数）

```
void SwitchStatus( int k, int * pSwitch)
{
    int i;
    for( i = 0; i < 6 ; i ++ )
        pSwitch[i] = (k >> i) & 1;
}
```

- 要写一个让开关起作用的函数

`void ApplySwitch(int *pLights, int *pNextLights,
int *pSwitchs);`

- pSwitchs 表示一行开关的状态
- pLights 表示与开关同一行的灯的状态
- pNextLights表示开关下一行的灯的状态
- 本函数根据 pSwitchs 所代表的开关状态，计算这行开关起作用后， pLights行和pNextLights行的灯的状态
- 不考虑开关的上一行的灯，是因为设定 pSwitchs的值的时候，已经确保会使得上一行的灯变成全灭（或没有上一行）

熄灯问题



```
void ApplySwitch( int *pLights, int *pNextLights, int *pSwitchs)
{ // 依次让每个开关起作用
    for ( int i = 0; i < 6; i++ ) { // 请补全循环体代码
```

pLights: 当前行灯
pNextLights: 下一行灯
pSwitchs: 当前行开关
都是int[6]型数组

```
    }
}
```

2811 熄灯问题



```
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include <stdbool.h>
int anPuzzle[5][6]; // 灯状态
int anOriPuzzle[5][6]; // 初始灯状态
int anSwitch[5][6]; // 开关状态
void OutputResult(int t) //输出结果
{
    printf("PUZZLE #%d\n", t);
    int i, j;
    for ( i = 0; i < 5; i++ ) {
        for ( j = 0; j < 6; j++ ) {
            printf("%d", anSwitch[i][j]);
            if ( j < 5 ) printf(" ");
        }
        printf("\n");
    }
}
```

熄灯问题



```
int main() {
int T, t, i, j, k, n;
    scanf("%d", &T);
    for ( t = 0; t < T; t++ ) {
        for ( i = 0; i < 5; i++ )
            for ( j = 0; j < 6; j++ )
                scanf("%d", &anOriPuzzle[i][j]);
        //遍历首行开关的64种状态
        for ( n = 0; n < 64; n++ ) {
            memcpy( anPuzzle, anOriPuzzle, sizeof(anPuzzle));
            //算出n所代表的开关状态，放到anSwitch[0]
            SwitchStatus( n, anSwitch[0]);
            //下面逐行让开关起作用，并算出下一行开关应该是什么状态
            for( k = 0; k < 5; k++ ) {
                // 算出第k行开关起作用后的结果
                ApplySwitch( anPuzzle[k], anPuzzle[k+1], anSwitch[k]);
                // 第k+1行的开关状态应和第k行的灯状态一致
                memcpy( anSwitch[k+1], anPuzzle[k], sizeof(anPuzzle[k]));
            }
        }
    }
}
```

熄灯问题



```
bool bOk = true; //记录最后一行灯是不是全灭  
//看最后一行灯是不是全灭
```

```
for ( k = 0; k < 6; k ++ ) {  
    if ( anPuzzle[4][k] ) {
```

```
        bOk = false;
```

```
        break;
```

```
    }
```

```
}
```

```
if ( bOk ) {
```

```
    OutputResult(t+1); //输出解
```

```
    break; //找到解， 就不用再试下一种状态了
```

```
}
```

```
}
```

```
}
```

```
}
```


例题：讨厌的青蛙



中国科学技术大学
University of Science and Technology of China

- 问题描述

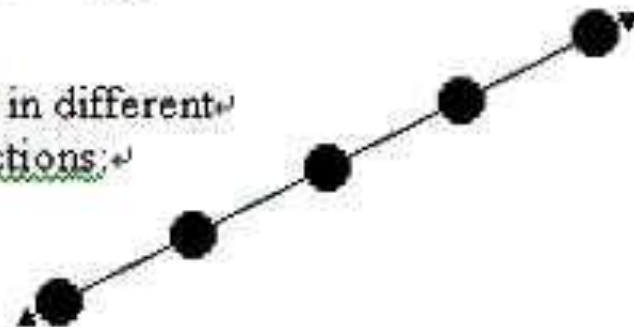
- 在韩国，有一种小的青蛙。每到晚上，这种青蛙会跳越稻田，从而踩踏稻子。农民在早上看到被踩踏的稻子，希望找到造成最大损害的那只青蛙经过的路径。
- 每只青蛙总是沿着一条直线跳越稻田，而且每次跳跃的距离都相同。



Different frogs can jump with
different hop lengths:



And in different
directions:



讨厌的青蛙



- 如下图所示，稻田里的稻子组成一个栅格，每棵稻子位于一个格点上。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。

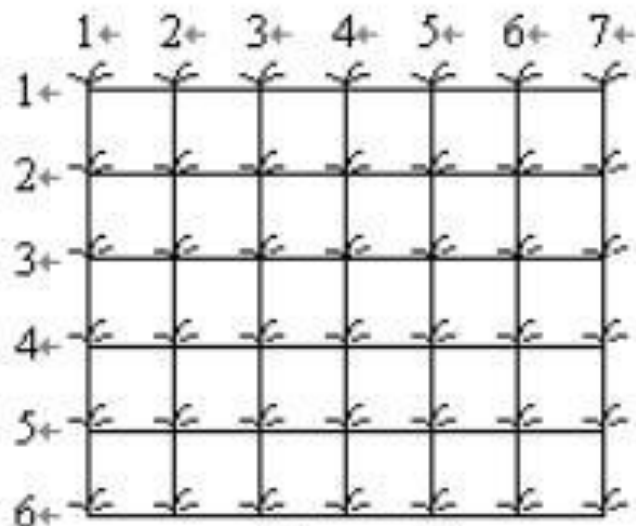


Figure-1

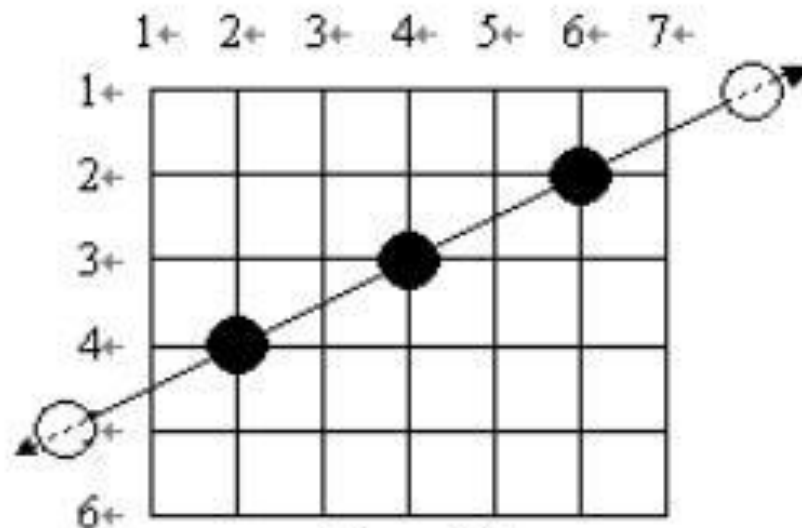


Figure-2

讨厌的青蛙



- 如下图所示，可能会有多只青蛙从稻田穿越。青蛙的每一跳都恰好踩在一棵水稻上，将这棵水稻拍倒。有些水稻可能被多只青蛙踩踏。

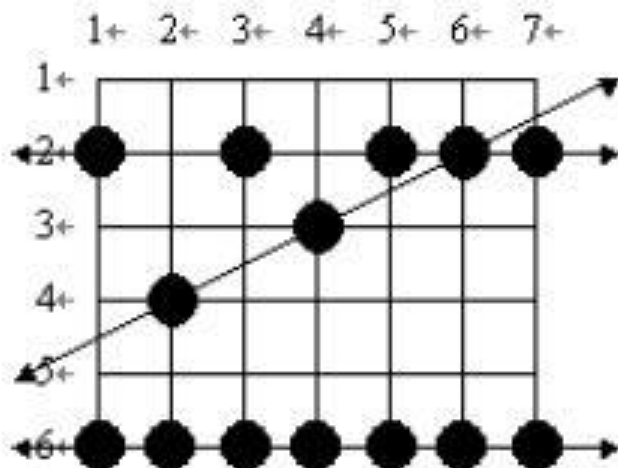


Figure-3

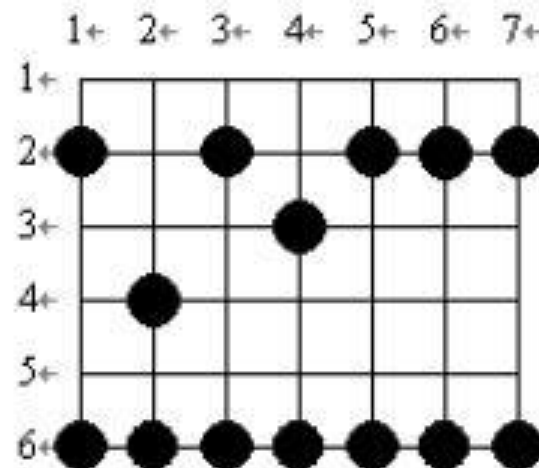


Figure-4

- 当然，农民所见到的是图4中的情形，并看不到图3中的直线，也见不到别人家田里被踩踏的水稻。

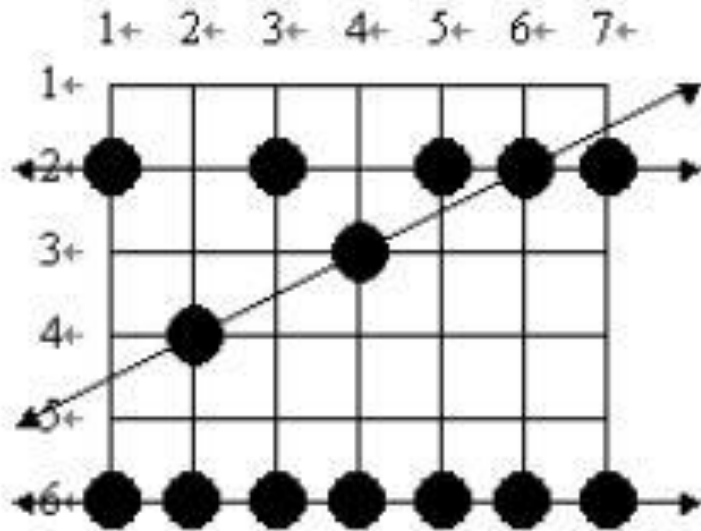


Figure-3+

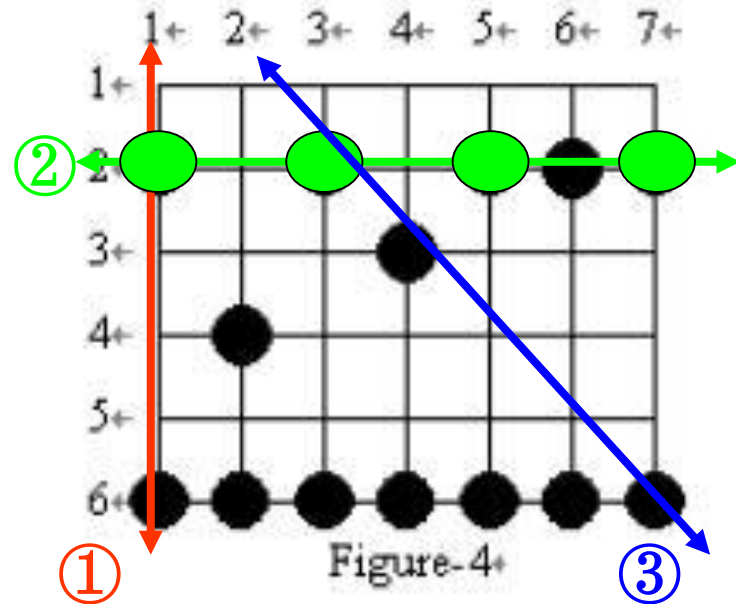


Figure-4+

- 根据图4，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。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
 - ①不是一条行走路径：只有两棵被踩踏的水稻
 - ②是一条行走路径，但不包括（2, 6）上的水稻
 - ③不是一条行走路径：虽然有3棵被踩踏的水稻，但这三棵水稻之间的距离间隔不相等



- 请你写一个程序，确定：在一条青蛙行走路径中，最多有多少颗水稻被踩踏。例如，图4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。

- 输入

- 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**。

- 输出

- 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。

讨厌的青蛙



中国科学技术大学
University of Science and Technology of China

- 输入样例

6 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

- 输出样例

7

- 枚举什么？
 - 枚举路径上的开始两点
- 每条青蛙行走路径中至少有3棵水稻
- 假设一只青蛙进入稻田后踩踏的前两棵水稻分别是 (X_1, Y_1) 、 (X_2, Y_2) 。那么：
 - 青蛙每一跳在X方向上的步长 $dX = X_2 - X_1$ 、在Y方向上的步长 $dY = Y_2 - Y_1$
 - $(X_1 - dX, Y_1 - dY)$ 需要落在稻田之外；
 - 当青蛙踩在水稻 (X, Y) 上时，下一跳踩踏的水稻是 $(X + dX, Y + dY)$ ；
 - 将路径上的最后一棵水稻记作 (X_K, Y_K) ， $(X_K + dX, Y_K + dY)$ 需要落在稻田之外；

猜测一条路径

- 猜测的办法需要保证：每条可能的路径都能够被猜测到。
- 从输入的水稻中任取两棵，作为一只青蛙进入稻田后踩踏的前两棵水稻，看能否形成一条穿越稻田的行走路径。

- 猜测的过程需要**尽快排除错误的答案**：猜测 (X_1, Y_1) 、 (X_2, Y_2) 就是所要寻找的行走路径上的前两棵水稻。当下列条件之一满足时，这个猜测就不成立：
 - 青蛙不能经过一跳从稻田外跳到 (X_1, Y_1) 上
 - 按照 (X_1, Y_1) 、 (X_2, Y_2) 确定的步长，从 (X_1, Y_1) 出发，青蛙最多经过 $(\text{MAXSTEPS} - 1)$ 步，就会跳到稻田之外。**MAXSTEPS**是当前已经找到的最好答案。

- 选择合适的数据结构：采用的数据结构需要与问题描述中的概念对应。
- 关于被踩踏的水稻的坐标，该如何定义？
- 方案1：

```
struct {  
    int x, y;  
} plants[5000];
```
- 方案2：

```
int plantsRow[5000], plantsCol[5000];
```
- 显然方案1更符合问题本身的描述

- 设计的算法要简洁，尽量使用C提供的函数完成任务
- 猜测一条行走路径时，需要从当前位置(X, Y)出发时，看($X + dX, Y + dY$)位置的水稻是否被踩踏
 - 方案1：自己写一段代码，看看($X + dX, Y + dY$) 是否在数组plants中；
 - 方案2：先用qsort对plants中的元素排序，然后用bsearch从中查找元素($X + dX, Y + dY$)；
- 显然基于方案2设计的算法更简洁、更容易实现、更不容易出错误；
 - 通常，所选用的数据结构对算法的设计有很大影响。

- 一个有 n 个元素的数组，每次取两个元素，遍历所有取法的代码写法：

```
for( int i = 0; i < n - 1; i ++ )  
    for( int j = i + 1; j < n; j ++ )  
    {  
        a[i] = ...;  
        a[j] = ...;  
    }
```

- 二分查找函数：

```
void *bsearch(const void *key, const void *base, size_t nelem,  
size_t width, int (*fcmp)(const void *, const void *));
```

- 查到返回地址，查不到返回空指针。

2812 讨厌的青蛙



```
#include <stdio.h>
#include <stdlib.h>
int r, c, n;
struct PLANT {
    int x, y;
};
struct PLANT plants[5001], plant;

int myCompare( const void *ele1, const void *ele2 )
{ // 按x的升序排序
    struct PLANT *p1, *p2;
    p1 = (struct PLANT*) ele1;
    p2 = (struct PLANT*) ele2;
    if ( p1->x == p2->x ) {
        return(p1->y - p2->y);
    }
    return ( p1->x - p2->x );
}
```

讨厌的青蛙



//判断从 secPlant点开始，步长为dx, dy，那么最多能走几步

```
int searchPath(struct PLANT secPlant, int dX, int dY)
{
    struct PLANT plant; // 下一步
    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) {
        //每一步都必须踩倒水稻才算合理，否则这不是一条行走路径
        if (!bsearch(&plant, plants, n, sizeof(struct PLANT), myCompare)) {
            steps = 0;
            break;
        }
        plant.x += dX;
        plant.y += dY;
        steps++;
    }
    return(steps);
}
```

讨厌的青蛙



```
int main()
{
    int i,j, dX, dY, pX, pY, steps, max = 2;
    struct PLANT plant;
    scanf("%d%d", &r, &c);
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d%d", &plants[i].x, &plants[i].y);
    }
    // 按x坐标升序排序
    qsort(plants, n, sizeof(struct PLANT), myCompare);
```


讨厌的青蛙



```
for (i = 0; i < n - 2; i++) { //plants[i]是第一个点
    for (j = i + 1; j < n - 1; j++) { // plants[j]是第二个点
        dX = plants[j].x - plants[i].x; // x方向步长, >=0
        dY = plants[j].y - plants[i].y; // y方向步长
        pX = plants[i].x - dX; // 第一点的前一点x坐标
        pY = plants[i].y - dY; // 第一点的前一点y坐标

        if (pX <= r && pX >= 1 && pY <= c && pY >= 1) {
            // 第一点的前一点在稻田里, 说明本次选的第
            // 二点导致的步长不合理, 取下一个点作为第二点
            continue;
        }
        if (plants[i].x + (max - 1) * dX > r) {
            //x方向过早越界了。说明本次选的第二点不成立。
            //如果换下一个点作为第二点, x方向步长只会更大, 更不成立
            //所以应该认为本次选的第一点都是不成立的,
            //那么取下一个点作为第一点再试
            break;
        }
    }
}
```

讨厌的青蛙



```
pY = plants[i].y + (max - 1) * dY;  
if ( pY > c || pY < 1) {  
    continue; // y方向过早越界了，应换一个点作为第二点再试  
}  
  
// 看看从这两点出发，一共能走几步  
steps = searchPath(plants[j], dX, dY);  
if (steps > max) {  
    max = steps;  
}  
}  
}  
if ( max == 2 ) max = 0;  
printf("%d\n", max);  
}
```