

2019年春

程序设计实习(II): 算法设计

第十三讲 枚举

刘家瑛

liujiaying@pku.edu.cn





枚举

- 基于已有知识 → 答案猜测的一种问题求解策略
- 例如: 求小于 N 的最大素数
 - 找不到一个数学公式, 使得根据 N 就可以计算出这个素数
 - $N-1$ 是素数吗? $N-2$ 是素数吗?
 - $N-K$ 是素数的充分必要条件:
 $N-K$ 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
→ 转化为求小于 $N-K$ 的全部素数



枚举

■ 解决办法

- 2是素数, 记为 PRIM_0
- 根据 $\text{PRIM}_0, \text{PRIM}_1, \dots, \text{PRIM}_k$, 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



枚举的思想:猜测

■ 枚举

■ 从可能的集合中一一列举各元素

- 根据所知道的**知识**, 给一个猜测的答案
- 2是素数

■ 枚举算法

- 对问题可能解集合的每一项
- 根据问题给定的检验条件判定哪些是成立的
- 使条件成立的即是问题的解



枚举的思想:猜测

枚举过程

- 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**
每次猜测是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



枚举中三个关键问题

问题一

给出解空间, 建立简洁的数学模型

可能的情况是什么

→ 模型中变量数尽可能少, 它们之间相互独立

- “求小于 N 的最大素数”中的条件是“ n 不能被 $[2, n)$ 中任意一个素数整除”
- 而不是“ n 不能被 $[2, n)$ 中任意一个整数整除”



枚举中三个关键问题

问题二

减少搜索的空间

- 利用知识缩小模型中各变量的取值范围, 避免不必要的计算

→ 减少代码中循环体执行次数

- 除2之外, 只有奇数才可能是素数, $\{2, 2*i+1 | 1 \leq i, 2*i+1 < n\}$



枚举中三个关键问题

问题三

采用合适的搜索顺序

搜索空间的遍历顺序要与模型中条件表达式一致

- 对 $\{2, 2*i+1 | 1 \leq i, 2*i+1 < n\}$ 按照从小到大的顺序



例题1: 完美立方

- 形如 $a^3 = b^3 + c^3 + d^3$ 的等式被称为完美立方等式。

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 a, b, c, d 大于 1, 小于等于 N , 且 $b \leq c \leq d$

- 输入

一个正整数 N ($N \leq 100$)

- 输出

每行输出一个完美立方

输出格式为:

Cube = a, Triple = (b, c, d)

其中 a, b, c, d 所在位置分别用实际求出四元组值代入



例题1: 完美立方

请按照 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 b 值小的优先输出, 仍相同则 c 值小的优先输出,

再相同则 d 值小的先输出

● 样例输入

24



例题1: 完美立方

● 样例输出

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)



例题1: 完美立方

● 解题思路

四重循环枚举a, b, c, d,

a在最外层, d在最里层, 每一层都是从小到大枚举,

a枚举范围[2, N]



例题1: 完美立方

● 解题思路

四重循环枚举 a, b, c, d ,

a 在最外层, d 在最里层, 每一层都是从小到大枚举,

a 枚举范围 $[2, N]$

b 范围 $[2, a-1]$



例题1: 完美立方

● 解题思路

四重循环枚举 a, b, c, d , a 在最外层, d 在最里层,
每一层都是从小到大枚举,

a 枚举范围 $[2, N]$

b 范围 $[2, a-1]$

c 范围 $[b, a-1]$

d 范围 $[c, a-1]$



例题1: 完美立方

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    int N;
    scanf("%d", &N);
    for(int a = 2; a <= N; ++a)
        for(int b = 2; b < a; ++b)
            for(int c = b; c < a; ++c)
                for(int d = c; d < a; ++d)
                    if( a*a*a == b*b*b + c*c*c + d*d*d)
                        printf("Cube = %d, Triple = (%d,%d,%d)\n", a, b, c, d);
    return 0;
}
```



例题2: 生理周期

人有体力, 情商, 智商的高峰日子,

它们分别每隔23天, 28天和33天出现一次

对于每个人, 我们想知道**何时三个高峰落在同一天**

给定三个高峰出现的日子p, e和i (不一定是第一次高峰出现的日子),

再给定另一个指定的日子d

你的任务是输出日子d之后, 下一次三个高峰落在同一天的日子

(用距离d的天数表示)

例如 :

给定日子为10, 下次出现三个高峰同一天的日子是12, 则输出2



例题2: 生理周期

● 输入

输入四个整数： p , e , i 和 d

p , e , i 分别表示体力、情感和智力高峰出现的日子

d 是给定的日子, 可能小于 p , e 或 i

所有给定日子是非负的并且小于或等于365, 所求的日子小于或等于21252

● 输出

从给定日子起, 下一次三个高峰同一天的日子 (距离给定日子的天数)



例题2: 生理周期

● 输入样例

0 0 0 0

0 0 0 100

5 20 34 325

4 5 6 7

283 102 23 320

203 301 203 40

-1 -1 -1 -1



例题2: 生理周期

● 输出样例

Case 1: the next triple peak occurs in 21252 days.

Case 2: the next triple peak occurs in 21152 days.

Case 3: the next triple peak occurs in 19575 days.

Case 4: the next triple peak occurs in 16994 days.

Case 5: the next triple peak occurs in 8910 days.

Case 6: the next triple peak occurs in 10789 days.



例题2: 生理周期

● 解题思路

- 从d+1天开始, 一直试到第21252 天,
- 对其中每个日期k, 看是否满足
$$(k - p) \% 23 == 0 \ \&\& \ (k - e) \% 28 == 0 \ \&\& \ (k - i) \% 33 == 0$$
- 如何试得更快?



例题2: 生理周期

● 解题思路

- 从d+1天开始, 一直试到第21252 天,
- 对其中每个日期k, 看是否满足

$$(k - p)\%23 == 0 \ \&\& \ (k - e)\%28 == 0 \ \&\& \ (k - i)\%33 == 0$$

- 如何试得更快?

跳着试!



```
#include <iostream>
#include <cstdio>
using namespace std;
#define N 21252
int main(){
    int p, e, i, d, caseNo = 0;
    while( cin >> p >> e >>i >>d && p!= -1 ) {
        ++ caseNo;
        int k;
        for(k = d+1; (k-p)%23; ++k);
        for(; (k-e)%28; k+= 23);
        for(; (k-i)%33; k+= 23*28);
        cout << "Case " << caseNo <<
            ": the next triple peak occurs in "<< k-d <<" days."<< endl;
    }
    return 0;
}
```



例题: POJ1013 称硬币

有12枚硬币。其中有11枚真币和1枚假币
假币和真币重量不同，但不知道假币比真币轻还是重。
现在，用一架天平称了这些币三次，告诉你称的结果，
请你找出假币并且确定假币是轻是重
(数据保证一定能找出来)



例题: POJ1013 称硬币

● 输入

第一行是测试数据组数

每组数据有三行, 每行表示一次称量的结果

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



例题: POJ1013 称硬币

- 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.




例题: POJ1013 称硬币

● 解题思路

- 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



```
#include <iostream>
#include <cstring>
using namespace std;
char Left[3][7];    //天平左边硬币
char Right[3][7];   //天平右边硬币
char result[3][7];  //结果
bool IsFake(char c, bool light) ;
//light 为真表示假设假币为轻，否则表示假设假币为重
```



```
int main() {
    int t;
    cin >> t;
    while(t--) {
        for(int i = 0; i < 3; ++i)
            cin >> Left[i] >> Right[i] >> result[i];
        for(char c='A'; c<='L'; c++) {
            if( IsFake(c,true) ){
                cout << c << " is the counterfeit coin and it is light.\n";
                break;
            }
            else if( IsFake(c,false) ){
                cout << c << " is the counterfeit coin and it is heavy.\n";
                break;
            }
        }
    }
    return 0; }
```



```
bool IsFake(char c, bool light)
//light 为真表示假设假币为轻，否则表示假设假币为重
{
    for(int i = 0;i < 3; ++i) {
        char * pLeft,*pRight; //指向天平两边的字符串
        if(light) {
            pLeft = Left[i];
            pRight = Right[i];
        }
        else {
            pLeft = Right[i];
            pRight = Left[i];
        }
    }
}
```



```
switch(result[i][0]) {
    case 'u':
        if ( strchr(pRight,c) == NULL)
            return false;
        break;
    case 'e':
        if( strchr(pLeft,c) || strchr(pRight,c))
            return false;
        break;
    case 'd':
        if ( strchr(pLeft,c) == NULL)
            return false;
        break;
}
}
return true;
}
```

枚举 — 熄灯问题

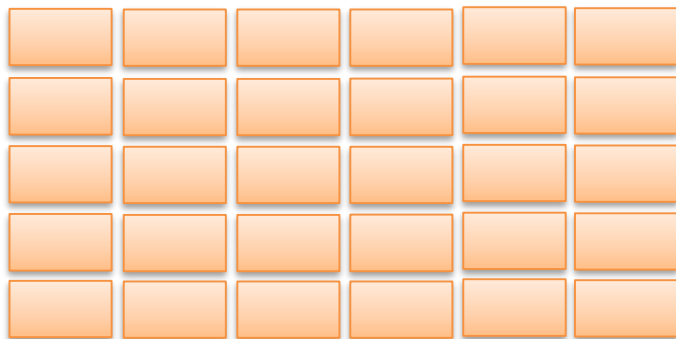




熄灯问题

■ 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

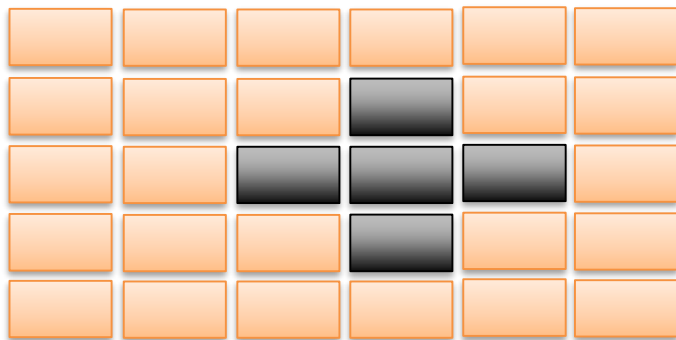




熄灯问题

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

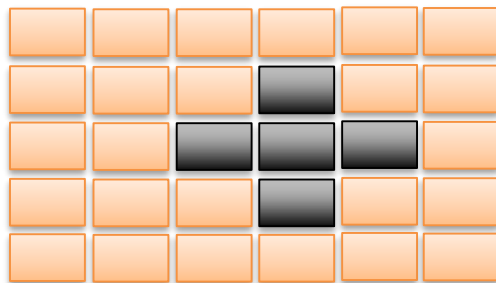
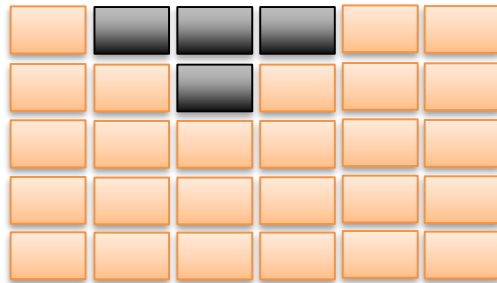
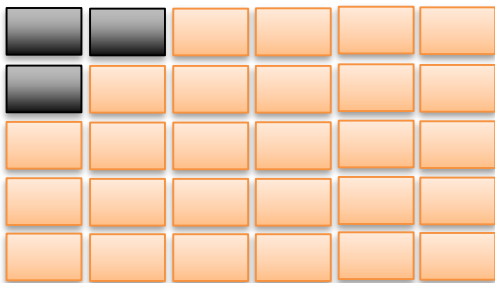




熄灯问题

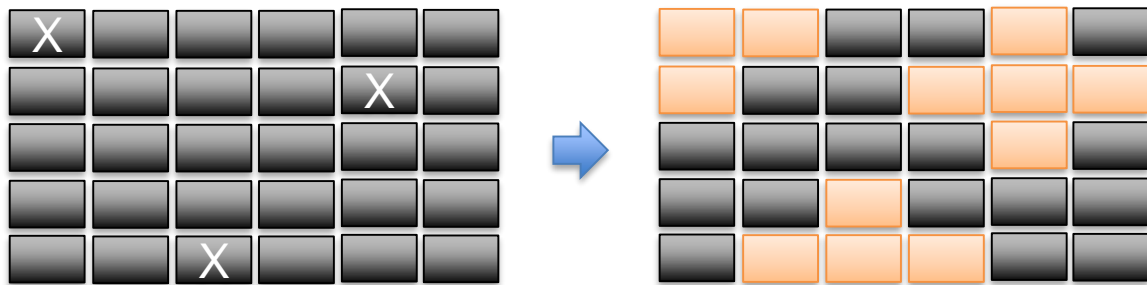
问题描述

- 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





- 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

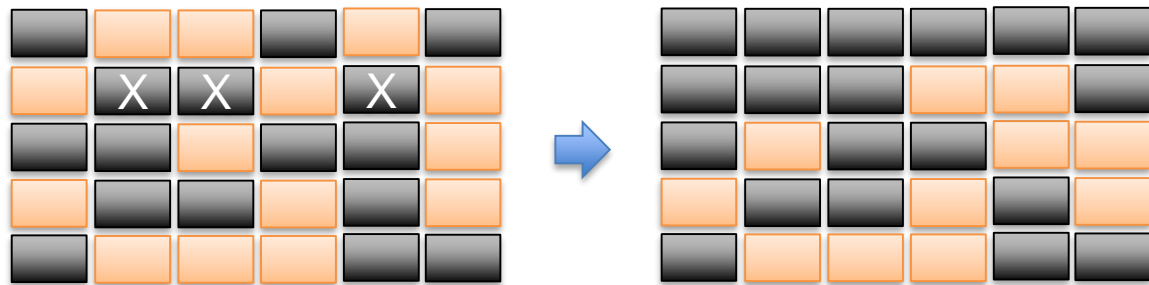




- 与一盏灯毗邻的多个按钮被按下时，
一个操作会抵消另一次操作的结果

- 第2行第3, 5列的按钮都被按下

→ 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
 - **1** 表示需要把对应的按钮按下
 - **0** 表示不需要按对应的按钮
 - 每个数字以一个空格隔开



▲ 样例输入

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

▲ 样例输出

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



解题分析

- 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
- 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



解题分析

■ 首先的想法:

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 / 不按下
- 一共有30个开关, 那么状态数是 2^{30} , 太多, 会超时



解题分析

如何减少枚举的状态数目呢？

基本思路：

如果存在某个局部，

- 一旦这个局部的状态被确定

→ 剩余其他部分的状态只能是确定的一种，或不多的 n 种

- [问题的关键]：只需枚举这个局部的状态即可



解题分析

- 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—一个“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
 - 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1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)
 - 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



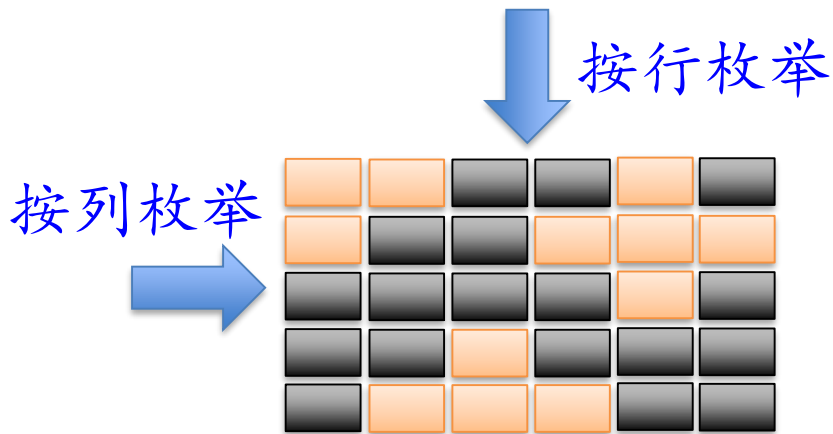
解题分析

- 第2行的开关起作用后,
 - 为了熄灭第2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
- 只要第1行的状态定下来, 记作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
- 只需枚举第1行的状态, 状态数是 $2^6 = 64$



有没有状态数更少的做法

- 枚举第一列, 状态数是 $2^5 = 32$





具体实现

- 用一个矩阵 `puzzle[5][6]` 表示灯的初始状态
 - `puzzle[i][j]=1`: 灯(i, j)初始时是被点亮的
 - `puzzle[i][j]=0`: 灯(i, j)初始时是熄灭的
- 用一个矩阵 `press[5][6]` 表示要计算的结果
 - `press[i][j]=1`: 需要按下按钮(i, j)
 - `press[i][j]=0`: 不需要按下按钮(i, j)



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



实现方案

- 根据上面的分析, 按钮矩阵的第1行元素的各种取值进行枚举, 依次考虑如下情况:

0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	0	0
⋮					
1	1	1	1	1	1

- 枚举方式

- 将按钮矩阵的第1行看作一个二进制数
- 通过实现++操作实现



实现方案

- 用一个 6×8 的数组来表示 按钮矩阵:

简化计算数组下一行的值的计算公式

- 第0行, 第0列和第7列不属于PRESS矩阵范围, 可全置0

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)

- 给定PRESS的第1行取值, 计算出PRESS的其他行的值
 - $\text{press}[r+1][c] = (\text{puzzle}[r][c] + \text{press}[r][c-1] + \text{press}[r][c] + \text{press}[r][c+1] + \text{press}[r-1][c]) \% 2$
 - $0 < r < 5, 0 < c < 7$



程序实现

```
#include <stdio.h>
int puzzle[6][8], press[6][8];
bool guess(){
    int c, r;
    for(r=1; r<5; r++)
        for(c=1; c<7; c++)
            press[r+1][c] = (puzzle[r][c]+press[r][c]+
                press[r-1][c]+ press[r][c-1]+press[r][c+1]) %2;
    for(c=1; c<7; c++)
        if ((press[5][c-1] + press[5][c] + press[5][c+1] +
            press[4][c]) %2 != puzzle[5][c] )
            return(false);
    return(true);
}
```

根据press第1行和
puzzle数组, 计算
press其他行的值

判断所计算的
press数组能否熄
灭第5行的所有灯



```
void enumerate (){
    int c;
    bool success;
    for (c=1; c<7; c++)
        press[1][c] = 0;
    while(guess()==false){
        press[1][1]++;
        c = 1;
        while (press[1][c] > 1) {
            press[1][c] = 0;
            c++;
            press[1][c]++;
        }
    }
    return;
}
```

对press第1行的元素
press[1][1]~press[1][6]的
各种取值情况进行枚举，
依次考虑：

0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	0	0
⋮					
1	1	1	1	1	1



主程序

```
int main() {  
    int cases, i, r, c;  
    scanf("%d", &cases);  
    for ( r=0; r<6; r++ )  
        press[r][0] = press[r][7] = 0;  
    for ( c=1; r<7; r++ )  
        press[0][c] = 0;  
    for (i=0; i<cases; i++){  
        for(r=1; r<6; r++)  
            for(c=1; c<7; c++)  
                scanf("%d", &puzzle[r][c]);
```

//读入输入数据



```
enumerate (); //枚举第1行元素这个“局部情况”
printf("PUZZLE #%%d\n", i+1);
for(r=1; r<6; r++){
    for(c=1; c<7; c++)
        printf("%d ", press[r][c]);
    printf("\n");
}
return 0;
}
```



总结

枚举过程 `enumerate()`

- `press[1][]`中每一个元素表示一个二进制数0/1, 通过模拟二进制加法方式实现枚举
- 需要处理进位

推测验证过程 `guess()`

- 用 6×8 按钮矩阵来简化下一行按钮值的计算公式
- 根据`press[1][]`和`puzzle`数组
用公式来计算使得1-4行所有灯熄灭的`press`其他行的值,
再判断所计算的`press`数组能否熄灭矩阵第5行的所有灯

枚 举 — 讨厌的青蛙



北京大學



讨厌的青蛙



问题描述

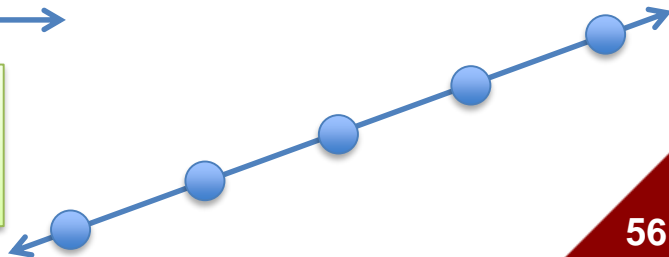
- 在韩国, 有一种小青蛙
- 每到晚上, 这种青蛙会跳跃稻田, 从而踩踏稻子
- 农民早上看到被踩踏的稻子, 希望找到造成最大损害的那只青蛙经过的路径
- 每只青蛙总是沿着一条直线跳跃稻田
- 且每次跳跃的距离都相同



不同青蛙的蛙跳步长不同

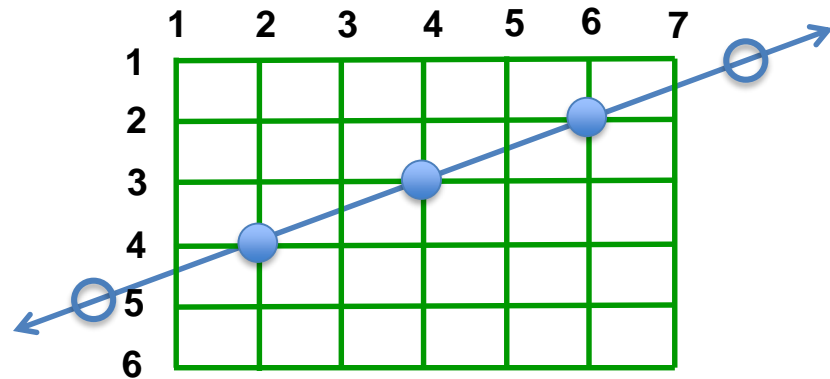
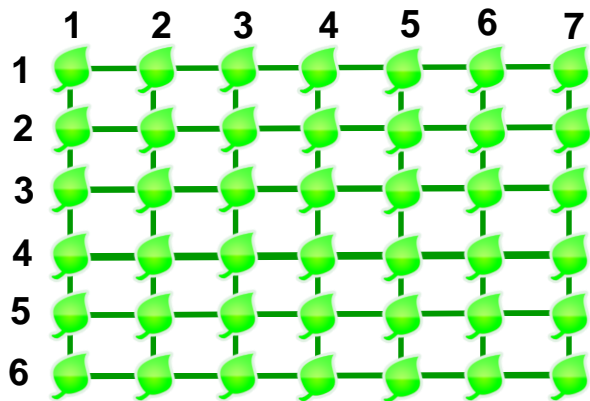


不同青蛙的蛙跳
方向可能不同





- 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





- 可能会有多只青蛙从稻田穿越
- 青蛙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
- 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

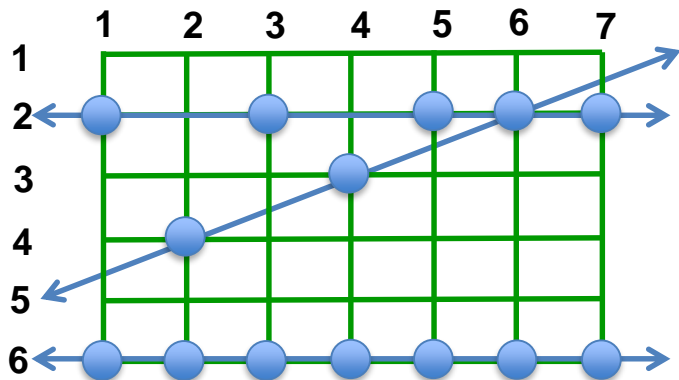


图 3

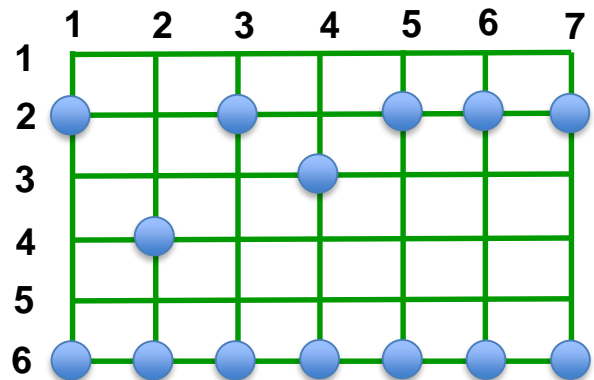


图 4

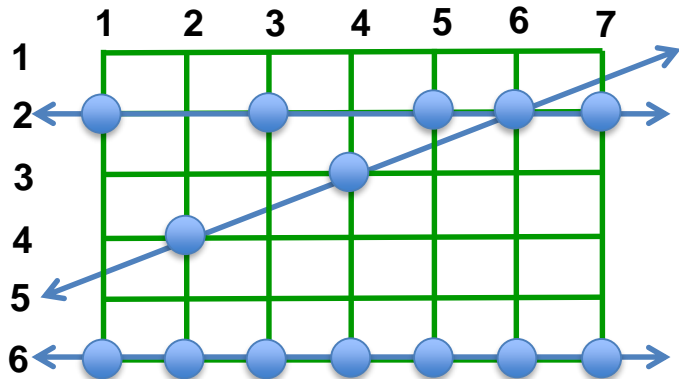


图 3

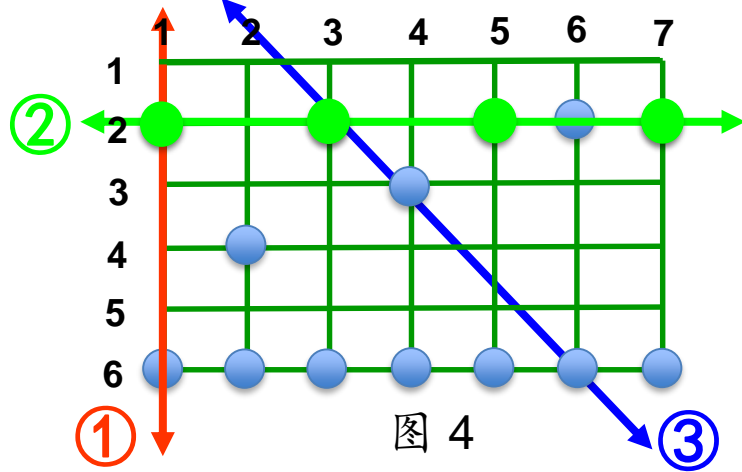


图 4

■ 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2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3棵水稻之间的距离间隔不相等



题目要求

■ 请写一个程序, 确定:

- 在各条青蛙行走路径中, 踩踏水稻最多的那一条上, 有多少颗水稻被踩踏
- 例如, 图4中答案是7, 因为第6行上全部水稻恰好构成一条青蛙行走路径

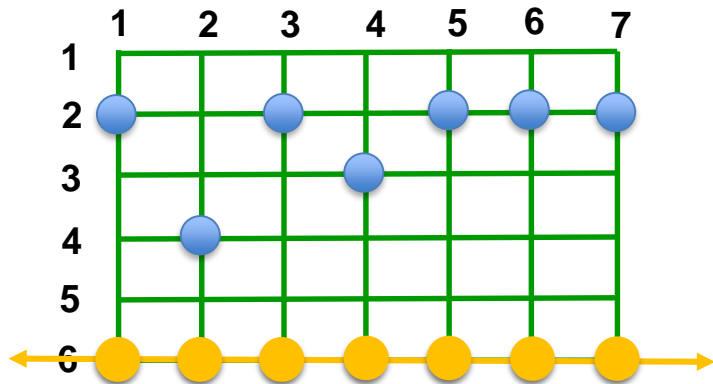


图 4



程序输入

- 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
- 且每棵被踩踏水稻只被列出一次



■ 程序输出

- 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



▲ 样例输入

6 7 //6行7列

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

▲ 样例输出

7



解题思路——枚举

枚举什么

- 枚举每个被踩的稻子作为行走路径起点 (5000个)
- 对每个起点, 枚举行走方向 (5000种)
- 对每个方向枚举步长 (5000种)
- 枚举步长后还要判断是否每步都踩到水稻

时间: $5000 * 5000 * 5000$, 不行!



解题思路——枚举

- 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) ,
则 $(X_K + dX, Y_K + dY)$ 需要落在稻田之外



解题思路: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



解题思路:猜测一条路径

- 猜测的过程需要**尽快排除错误的答案**
 - 猜测 $(X1, Y1), (X2, Y2)$ -- 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 $(X1, Y1)$ 上
 - 按照 $(X1, Y1), (X2, Y2)$ 确定的步长, 从 $(X1, Y1)$ 出发, 青蛙最多经过 $(MAXSTEPS - 1)$ 步, 就会跳到稻田之外
 - $MAXSTEPS$ 是当前已经找到的最佳答案



选择合适的数据结构

■ 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++提供的函数完成计算的任务
- 猜测一条行走路径时, 需要从当前位置(X, Y)出发上时, 看看($X + dX, Y + dY$)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 `sort()` 对 `plants` 中的元素排序, 再用 `binary_search ()` 从中查找元素($X + dX, Y + dY$)
- 基于 方案2 设计的算法更简洁, 易实现, 更不容易出错误



注意:

- 一个有 n 个元素的数组, 每次取两个元素, 遍历所有取法
- 代码写法:

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



参考程序

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
using namespace std;
int r, c, n;
struct PLANT {
    int x, y;
};
PLANT plants[5001];
PLANT plant;
int searchPath(PLANT secPlant, int dX, int dY) ;
```



```
int main()
```

```
{
```

```
    int i, j, dX, dY, pX, pY, steps, max = 2;
```

```
    scanf("%d %d", &r, &c);
```

```
    //行数 and 列数
```

```
    scanf("%d", &n);
```

```
    for (i = 0; i < n; i++)
```

```
        scanf("%d %d", &plants[i].x, &plants[i].y);
```

```
    //将水稻按x坐标从小到大排序, x坐标相同按y从小到大排序
```

```
    sort(plants, plants + n);
```




```
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;
        dY = plants[ j ].y - plants[i].y;
        pX = plants[ i ].x - dX;
        pY = plants[ i ].y - dY;
        if (pX <= r && pX >= 1 && pY <= c && pY >= 1)
            continue;
        //第一点的前一点在稻田里,
        //说明本次选的第二点导致的x方向步长不合理(太小)
        //取下一个点作为第二点
        if (plants[ i ].x + (max - 1) * dX > r)
            break;
```

//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必然是不成立的,那么取下一个点作为第一点再试



```
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
steps = searchPath(plants[ j ], dX, dY);
//看看从这两点出发, 一共能走几步
if (steps > max)      max = steps;
}
if ( max == 2 ) max = 0;
printf("%d\n", max);
}
bool operator < ( const PLANT & p1, const PLANT & p2)
{
    if ( p1.x == p2.x )
        return p1.y < p2.y;
    return p1.x < p2.x ;
}
```



//判断从secPlant点开始, 步长为dx, dy, 那么最多能走几步

```
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) {
        if (! binary_search(plants, plants + n, plant)) {
            //每一步都必须踩倒水稻才算合理, 否则这就不是一条行走路径
            steps = 0;
            break;
        }
        plant.x += dX;
        plant.y += dY;
        steps++;
    }
    return(steps);
}
```



小结

- 枚举顺序十分重要, 好的枚举顺序, 能够及早排除不可能的情况, 减少枚举次数
- 本题将踩倒的水稻按照位置进行排序
- 枚举的时候先枚举序号低的, 就能够有效减少枚举次数:

```
if (plants[ i ].x + (max - 1) * dX > r) break;
```

//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必然是不成立的,

//那么取下一个点作为第一点再试