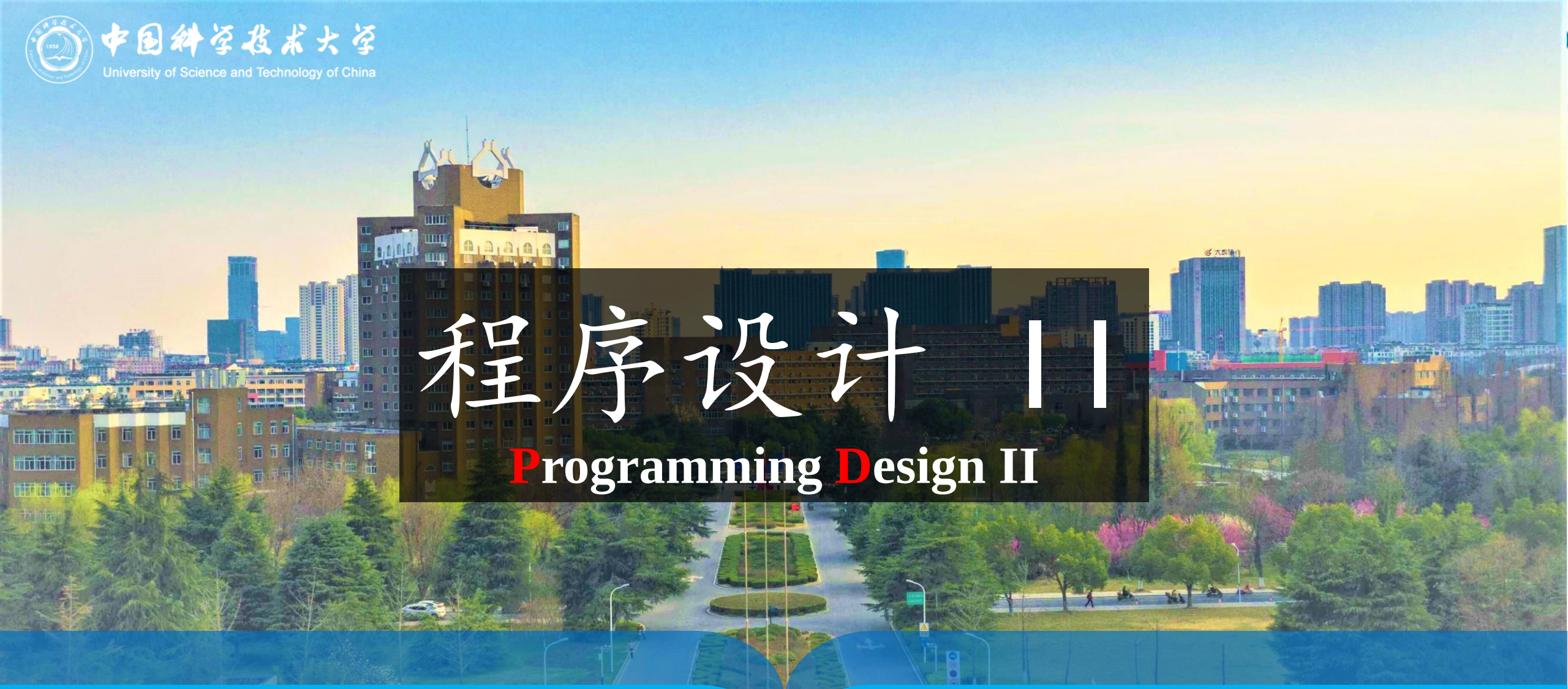

中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II

枚举

主讲：吴锋

目录
CONTENTS

◎ 枚举的思想

- 逐个排除
  - 例如：求小于N的最大素数
  - 虽然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，但可以采取枚举的方法。
  - 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，这也是一种递归的思想。
  - **解题思路**：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PRIMk+1。如果PRIMk+1大于N，则PRIMk是我们需要找的素数，否则继续寻找。

枚举的思想

- 依次猜测
  - 根据所知道的知识，给一个猜测的答案。
    - 如：2是素数。
  - 判断猜测的答案是否正确。
    - 如：2是小于N的最大素数吗？
  - 不断进行新的猜测，直到找到正确的答案。每次猜测的结果必须是前面的猜测中没有出现过的。
    - 如：每次猜测是素数一定比已经找到的素数大。
  - 猜测的过程中要尽早排除错误的答案。
    - 如：除2之外，只有奇数才可能是素数。

◎ 例题1：称硬币

- 问题描述（P159）
  - 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：1）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。2）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。

例题1：称硬币

- 输入
  - 输入有三行，每行表示一次称量的结果。赛利事先将12枚银币分别标号为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1：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1：称硬币

- 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。
  - 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试探法。
- 解题思想：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。如果符合，问题即解决。
  - 把所有硬币都试一遍，一定能找到特殊硬币。

◎ 例题1：称硬币

- 具体实现
  - 定义变量存储称量结果

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

- 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（总共12枚硬币），多出一个字符位置是为了放 ‘\0’，以便字符数组使用字符串函数。
- 例如：
  - left[0] = “ABCD”
  - right[0] = “EFGH”
  - result[0] = “even”

例题1：称硬币

- 具体实现：逐一枚举硬币的代码

```
for (char c = 'A'; c <= 'L'; c++) {
    if (isLight(c)) {
        printf("%c is the counterfeit coin and it is light.\n", c);
        break;
    } else if (isHeavy(c)) {
        printf("%c is the counterfeit coin and it is heavy.\n", c);
        break;
    }
}
```

例题1：称硬币

```
// 判断硬币x是否为轻的代码
bool isLight(char x)
{
  int i;
  // 判断是否与三次称量结果矛盾
  for (i = 0; i < 3; i++) {
     switch (result[i][0]) {
       case 'u':
         if (!inRight(i, x))
           return false;
         break;
       case 'e':
         if (inRight(i,x) || inLeft(i,x))
           return false;
         break;
       case 'd':
         if (!inLeft(i,x))
           return false;
           break;
     }
  }
  return true;
}
```

```
// 判断硬币x是否为重的代码
bool isHeavy(char x)
{
  int i;
  // 判断是否与三次称量结果矛盾
  for (i = 0; i < 3; i++) {
     switch (result[i][0]) {
       case 'u':
         if (!inLeft(i,x))
           return false;
         break;
       case 'e':
         if (inRight(i,x) || inLeft(i,x))
           return false;
         break;
       case 'd':
         if (!inRight(i,x))
           return false;
         break;
     }
  }
  return true;
}
```

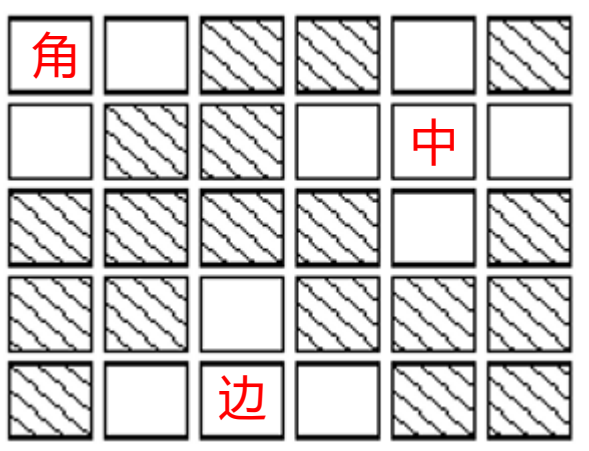
◎ 例题1：称硬币

```
// 判断硬币x 是否在第i次称量左侧
bool inLeft(int i, char x)
{
      return strchr(left[i], x);
}

// 判断硬币x 是否在第i次称量右侧
bool inRight(int i, char x)
{
     return strchr(right[i], x);
}
```

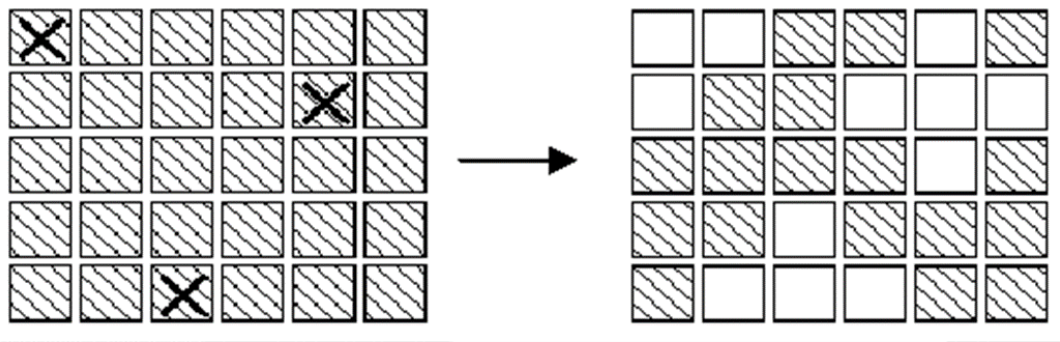
◎ 例题2：熄灯问题

- 问题描述（P163）
  - 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。如果灯原来是亮的，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
    - 其他矩阵中的按钮改变5盏灯的状态
  - 请写一个程序，确定需要按下哪些按钮，恰好使得所有的灯都熄灭。

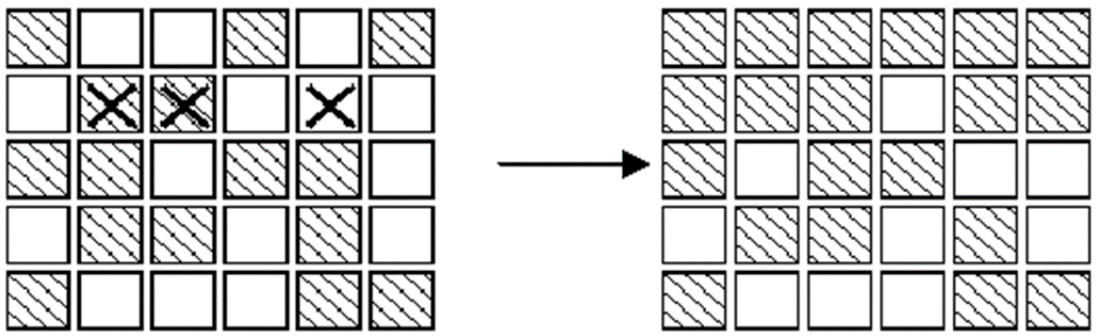


例题2：熄灯问题

- 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。



- 对矩阵中的每盏灯设置一个初始状态。请你按按钮，直至每一盏等都熄灭。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。



例题2：熄灯问题

- 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。

◎ 例题2：熄灯问题

- 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

◎ 例题2：熄灯问题

- 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。
  - **思路一**：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}，太多了，运行会超时。如何减少枚举的状态数呢？

例题2：熄灯问题

- 解题思路
  - 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。
  - 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。
  - 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。

例题2：熄灯问题

- 解题思路
  - 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。
  - 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 $2^6 = 64$
  - 有没有状态数更少的做法？
  - 枚举第一列，状态数是 $2^5 = 32$

例题2：熄灯问题

- 具体实现
  - 用一个矩阵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
  - 用一个矩阵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)

◎ 例题2：熄灯问题

- 具体实现
  - anSwitch[0] 里放着第1行开关的状态，可用六重循环进行枚举：

```
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++) {
                            anSwitch[0][0] = a0; anSwitch[0][1] = a1;
                            anSwitch[0][2] = a2;   ……
                        }
```

  - 如果每行开关数目是可变数N那怎么办?

例题2：熄灯问题

- 具体实现：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（即：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，例如：
    - 0 的二进制表示形式是 00 0000，即代表所有开关都不按下
    - 63 的二进制表示形式是 11 1111，即代表所有开关都按下
    - 5 的二进制表示形式是 00 00101，即代表右数第1，3个开关按下

◎ 例题2：熄灯问题

- 具体实现：适用于一行有N个开关的办法
  - 要写一个从二进制数到状态的转换函数：

    void SwitchStatus(int n, int * pSwitchLine);

  - 该函数将整数n(0≤n<64)的二进制表示形式对应到数组pSwitchLine里去（anSwitch[0][i] 对应第i位)

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

◎ 例题2：熄灯问题

- 具体实现：适用于一行有N个开关的办法
  - 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）。

例题2：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++) {
        if (pSwitchs[i]) {
            // 开关左边的灯改变状态
            if (i > 0)pLights[i-1] = 1 - pLights[i-1];
            // 开关所在位置的灯改变状态
            pLights[i] = 1 - pLights[i];
            // 开关右边的灯改变状态
            if(i < 5) pLights[i+1] = 1 - pLights[i+1];
            // 开关下边的灯改变状态
            pNextLights[i] = 1 - pNextLights[i];
        }
    }
}
```

例题2：熄灯问题

```
#include <memory.h>
#include <string.h>
#include <stdio.h>

int T;
int anPuzzle[6][6];
int anOriPuzzle[6][6];
int anSwitch[6][6]; //开关状态

//输出结果
void OutputResult(int t) {
     printf("PUZZLE #%d\n", t);
     for (int i = 0;i < 5; i++) {
          for (int j = 0; j < 6; j++) {
               printf("%d", anSwitch[i][j]);
               if( j < 5 ) printf(" ");
          }
          printf("\n");
     }
}
```

例题2：熄灯问题

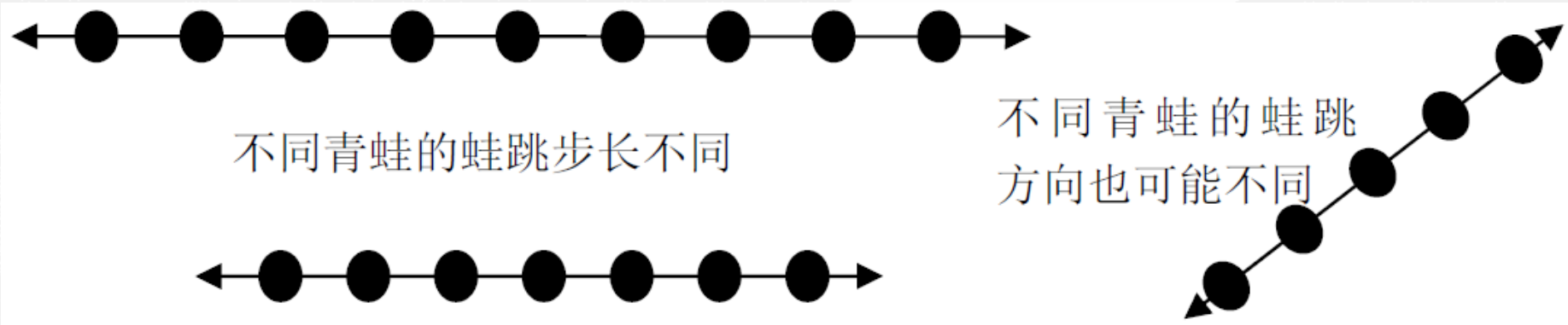
```
int main(int argc, char ** argv)   {
    scanf("%d", &T);
     for (int t = 0; t < T; t++) {
          for (int i = 0; i < 5; i++)
               for (int j = 0; j < 6; j++)
                    scanf("%d", &anOriPuzzle[i][j]);
         //遍历首行开关的64种状态
          for(int n = 0; n < 64; n++) {
               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，再让它们起作用……
               for (int 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]));
               }
```

例题2：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 (int 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
                }
            } // for (int n = 0; n < 64; n++)
        }
    return 0;
}
```

例题3：讨厌的青蛙

- 问题描述（P167）
  - 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。



例题3：讨厌的青蛙

- 问题描述（P167）
  - 如下图所示，稻田里的稻子组成一个栅格，每株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。

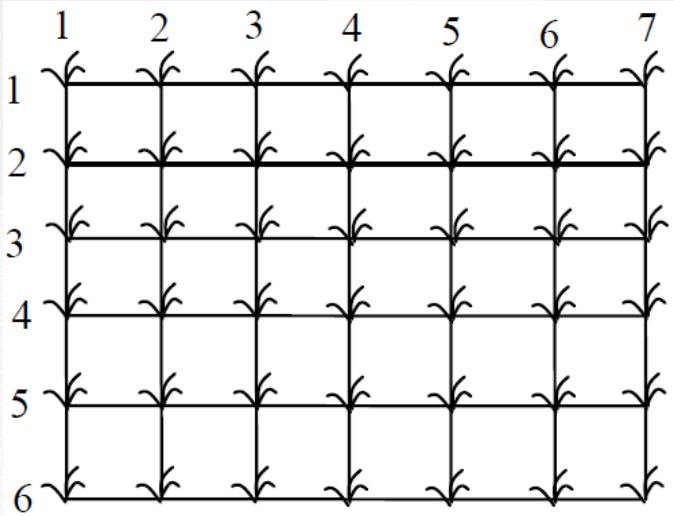


图 8-5 稻田栅格示意图

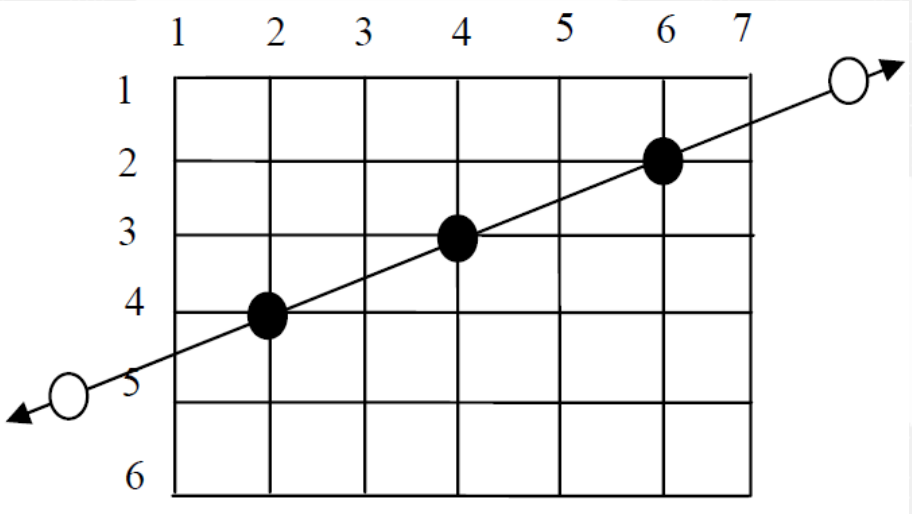


图 8-6 青蛙穿越稻田示意图

例题3：讨厌的青蛙

- 问题描述
  - 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株水稻上，将这株水稻拍倒。有些水稻可能被多只青蛙踩踏。当然，农民所见到的是图8-8中的情形，并看不到图8-7中的直线，也见不到别人家田里被踩踏的水稻。

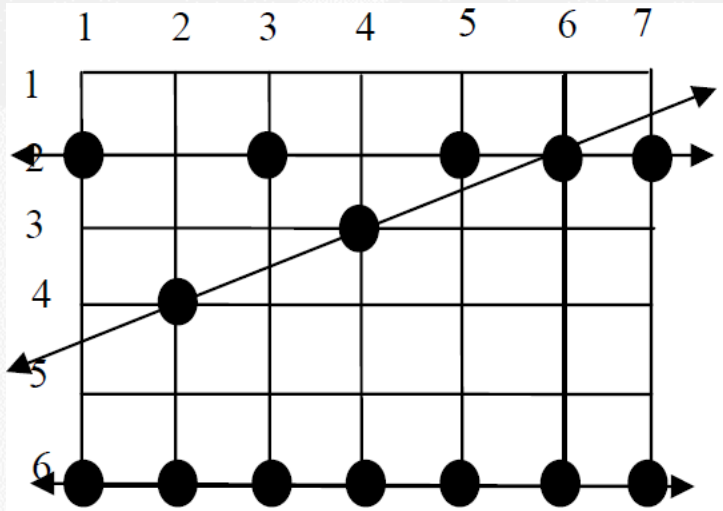


图 8-7 水稻被多只青蛙踩踏示意图

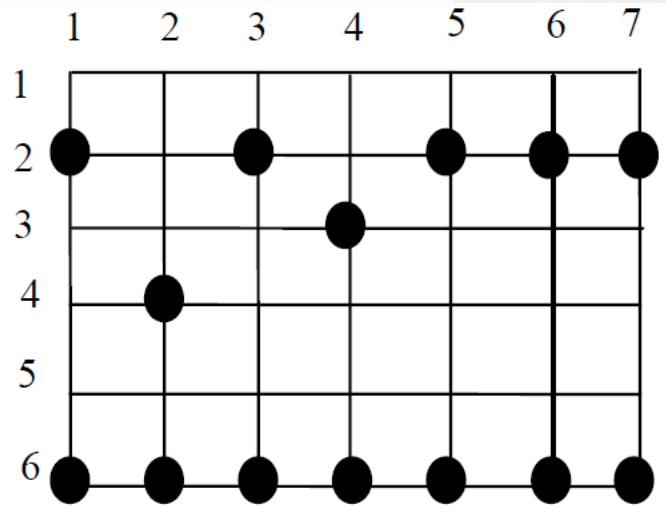


图 8-8 农民见到的稻田示意图

◎ 例题3：讨厌的青蛙

- 根据图示，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。

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）上的水道；

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。

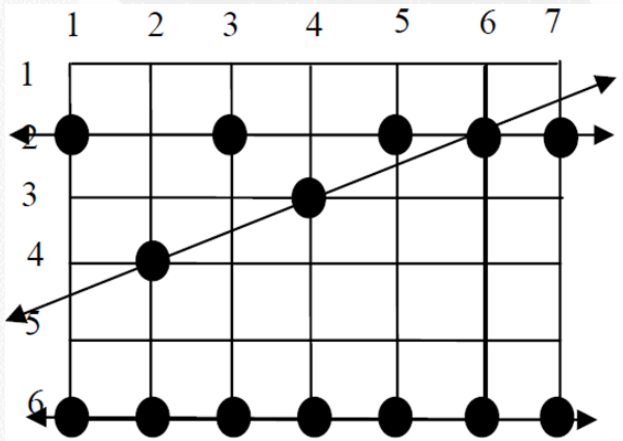


图 8-7 水稻被多只青蛙踩踏示意图

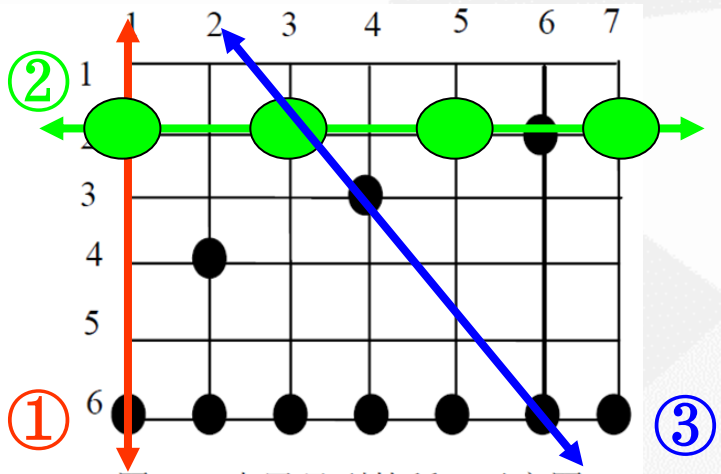


图 8-8 农民见到的稻田示意图

例题3：讨厌的青蛙

- **要求**：请写一个程序来确定，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，上图的答案是7，因为第6行上全部水稻（7株）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≤R、C≤5000。第二行是一个整数N，表示被踩踏的水稻数量，3≤N≤5000。在剩下的N行中，每行有两个整数，分别是一株被踩踏水稻的行号(1~R)和列号(1~C)，两个整数用一个空格隔开。而且，每株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。

例题3：讨厌的青蛙

- 样例输入

```
6 7   // R, C稻田中水稻的行数和列数
14  //N表示被踩踏的水稻数量
2 1 //被踩踏水稻的行号和列号
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 //包含最多水稻的青蛙行走路径中的水稻数量
```

例题3：讨厌的青蛙

- 解题思路：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（起点）
    - 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（终点）

例题3：讨厌的青蛙

- 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。
  - 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(MAXSTEPS - 1)步，就会跳到稻田之外。 MAXSTEPS是当前已经找到的最好答案。

例题3：讨厌的青蛙

- 具体实现：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
    - 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

例题3：讨厌的青蛙

- 具体实现：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。

◎ 例题3：讨厌的青蛙

- 具体实现
  - 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，查到返回地址，查不到返回空指针：

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

  - 数组的内容应根据 fcmp 所对应的比较函数升序排序。

例题3：讨厌的青蛙

```
#include <stdio.h>
#include <stdlib.h>

int r, c, n;
struct PLANT {
     int x, y;
};
PLANT plants[5001];
PLANT plant;

int myCompare(const void *ele1, const void *ele2);
int searchPath(PLANT secPlant, int dX, int dY);

int main(int argc, char** argv){
     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);
     qsort(plants, n, sizeof(PLANT), myCompare);
```

例题3：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;
            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
                            //二点导致的步长不合理，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都是不成立的，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);
     return 0;
}
```

例题3：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(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 (!bsearch(&plant, plants, n, sizeof(PLANT), myCompare))  {
               // 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;
}
```

例题3：讨厌的青蛙

```
int myCompare(const void *ele1, const void *ele2){
     PLANT *p1, *p2;
     p1 = (PLANT*) ele1;
     p2 = (PLANT*) ele2;
     if (p1->x == p2->x)
       return (p1->y - p2->y);
     return (p1->x - p2->x);
}
```

作业

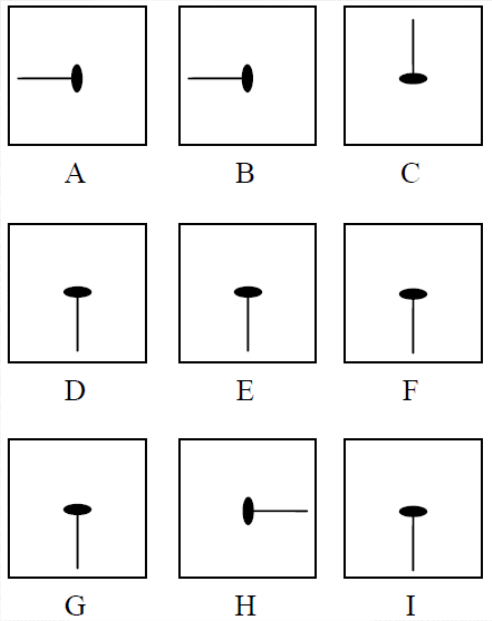
- 1. 计算对数(P171)
  - 给定两个正整数a 和b。可以知道一定存在整数x，使得 x <= $\log_a b$ < x + 1 求出x。输入数据保证x 不大于20
- 2. 数字方格(P171)
  - 任给一个整数n (0 n 100)，找到三个满足下列条件的正数a1、a2、a3，使得a1 + a2 + a3最大：
    - 0 ≤ a1、a2、a3 ≤ n;
    - a1 + a2 是2 的倍数;
    - a2 + a3 是3 的倍数;
    - a1 + a2 + a3 是5 个倍数。

作业

- 3. 画家问题(P171)
  - 有一个正方形的墙，由N*N 个正方形的砖组成，其中一些砖是白色的，另外一些砖是黄色的。Bob 是个画家，想把全部的砖都涂成黄色。但他的画笔不好使。当他用画笔涂画第(i, j)个位置的砖时， 位置(i-1, j)、 (i+1, j)、 (i, j-1)、 (i, j+1)上的砖都会改变颜色。请你帮助Bob 判断能否将所有的砖都涂成黄色，并且在能将所有的砖都涂成黄色时计算出最少需要涂画多少块砖。

作业

- 5. 拨钟问题(P172)
  - 有9 个时钟，排成一个3*3 的矩阵，各时钟指针的起始位置可以是12 点、3 点、6 点、9 点，如图8-9 所示。共允许有9 种不同的移动。如图8-10 所示，每个移动会将若干个时钟的指针沿顺时针方向拨动90 度。给定这9 个时钟指针的其始位置，请计算最少需要用最少个移动才能将9 个时钟的指针都拨到12 点的位置，并输出你采用的移动序列。



移动	影响的时钟
1	ABDE
2	ABC
3	BCEF
4	ADG
5	BDEFH
6	CFI
7	DEGH
8	GHI
9	EFHI