第 1 章

2004 ACM-ICPC 世界总决赛试题解析

试题 1–1　蚂蚁 Carl（Carl the Ant）

【问题描述】

蚂蚁在地面爬过的时候会留下微量的化学痕迹，以便于其他蚂蚁跟着自己的路径走。通常这些痕迹是直线形状的。但在一群蚂蚁中，有一只名叫 Carl 的不同寻常的蚂蚁。Carl 经常走“之”字形，有时候会无数次穿越自己所走过的路。而当其他蚂蚁到达一个交叉点时，它们总是沿着有着最强气味的路径，也就是沿着离开交叉点的、最新被走过的路径走。

蚂蚁长 1 厘米，行走或钻洞的速度是每秒 1 厘米，并且严格地沿着它们的路径走（在走到角落的时候转 90 度）。蚂蚁彼此间不能交叉地同时走过一个点，也不能一只蚂蚁叠在另一只蚂蚁的身体上。如果两只蚂蚁在同一时间到达同一交叉点，则在 Carl 的路径上走了较长距离的蚂蚁将优先走；否则，在交叉点等待时间较长的蚂蚁将优先走。

Carl 先挖洞爬到地面上，在原点于 0 时刻出发。它将沿着它的路径走到终点，再钻洞回到地下。其他的蚂蚁以特定的时间间隔跟着 Carl 出发。给出 Carl 所走的路径的描述以及其他蚂蚁的出发时间，请计算所有蚂蚁走完给定的路径，并钻洞返回地下所需要的时间，保证所有的蚂蚁都可以走完。注意：这里是由出题人确保，而不是要求解题人保证。

输入：

输入包含多组测试用例。输入的第一行是一个整数，表示测试用例的数目。

每组测试用例的第一行是 3 个正整数 n（$1 \leqslant n \leqslant 50$）、$m$（$1 \leqslant m \leqslant 100$）和 d（$1 \leqslant d \leqslant 100$），其中 n 表示 Carl 所走的路径中的线段个数，m 表示要走这条路径的蚂蚁的总数（包括 Carl），d 表示每两只连续出发的蚂蚁出发的时间间隔。Carl（编号为 0）在 0 时刻出发。下一只蚂蚁（编号为 1）在 d 时刻出发，第三只蚂蚁在 $2d$ 时刻出发，以此类推。如果蚂蚁爬出来的洞穴被堵住了，蚂蚁会按照正确的顺序尽可能快地从洞中爬出。

在每组测试用例的接下来的 n 行每行包括一对不重复的整数对 x，y（$-100 \leqslant x$，$y \leqslant 100$），这是按照 Carl 所走的路径顺序而给出的在路径上的线段的端点。第一条线段从原点（0,0）出发，后一条线段的起点就是前一条线段的终点。

为了方便起见，Carl 总是在平行于坐标轴的线段上行走，线段的端点不会与其他线段相交。

输出：

每组测试用例的输出如下：

```
Case C:
Carl finished the path at time t1
The ants finished in the following order:
a1 a2 a3 ... am
```

```
The last ant finished the path at time t2
```

这里 C 表示测试数据组编号（从 1 开始），$a_1, a_2, a_3, \cdots, a_m$ 表示按顺序到达目的地的蚂蚁，t_1 和 t_2 表示 Carl 和最后一只蚂蚁到达目的地的时刻（以秒表示）。每组测试用例之间输出一个空行。

样例输入	样例输出
2	Case 1:
4 7 4	Carl finished the path at time 13
0 4	The ants finished in the following order:
2 4	0 2 1 3 4 5 6
2 2	The last ant finished the path at time 29
−2 2	
4 7 2	Case 2:
0 4	Carl finished the path at time 13
2 4	The ants finished in the following order:
2 2	0 4 1 5 2 6 3
−2 2	The last ant finished the path at time 19

试题解析

看完试题后很自然地得出判断：这是一道模拟题，直接按照题意模拟每一时刻。

首先判断是否有新蚂蚁出现，然后判断每一只蚂蚁是否可以向前走。这里不能走的可能情况有两个：

1）在交叉口的蚂蚁优先级不够高。

2）前面的蚂蚁动不了。

把所有能向前走的蚂蚁都向前移一步，在此基础上判断是否所有蚂蚁都到达终点。但这仅是一个思维的粗浅轮廓。具体实现时，需要详细记录下面几点：

- 每一点最新的方向。
- 头在点（x,y）并朝向方向 k 的蚂蚁编号。
- 尾在点（x,y）并朝向方向 k 的蚂蚁编号。

在此基础上展开模拟运算：

若有新蚂蚁出现，就加入一只头和尾都在点（0,0）的蚂蚁 1。由于 Carl 的优先级最高且线段之间不能重合（每条线段的端点不与其他线段相交），Carl 一定不会被堵住，因此先假设蚂蚁 Carl 已经走过了（即更新某一点的最新方向），然后把它作为普通的蚂蚁处理。

对于每一只蚂蚁，如果它的头与多个蚂蚁的头在同一个交叉点，那么判断哪只蚂蚁的优先级较高，优先级较低的蚂蚁必须停留。对于剩下的可能前进的蚂蚁，还要判断在它前进的方向是否有蚂蚁。如果前面没有蚂蚁或前面的蚂蚁可以移动，那么可以向前走；否则必须停留。

当所有蚂蚁能否移动都确定了，就把其中可移动的蚂蚁向前移一步，并记录是否有蚂蚁到达终点。

若所有蚂蚁都到达终点了，则模拟结束并输出答案；否则继续下一时刻。

程序清单

```
#include <stdio.h>
#include <string>
struct info                    //蚂蚁的结构类型
```

```
{
    int x,y,len,wait,dir;        //坐标点（x,y），走过的距离为 len，等待的时间 wait，朝向 dir
};
const int MaxN=50;               //线段数上限
const int MaxM=100;              //蚂蚁数上限
const int MaxL=250;              //每条线段的最大长度
const int Add[4][2]={-1,0,  0,1,  1,0,  0,-1};    //上、右、下、左四个方向的单位向量
int N,cases,n,m,D,Rs,fin,sta,T,ex,ey;   //Rs 记录路线的长度,fin 记录到达终点的蚂蚁个数,sta
                                        //记录已经出发的蚂蚁个数，T 记录当前时刻
int route[MaxN * MaxL];                 //第 i 条单位线段的方向为 route[i]
int map[MaxL+1][MaxL+1];                //路线上（x,y）的最新方向
int ants[MaxL+1][MaxL+1][4];            //记录地图上头在点（x,y）并朝向方向 k 的蚂蚁编号为
                                        //ants[x][y][k]
int list[MaxM];                         //第 i 个到达目的地的蚂蚁编号为 list[i]
info ant[MaxM];                         //蚂蚁序列
void init()                             //输入当前测试组信息并计算 Carl 所走的路径
{
    int i,j,k,t,x1,y1,x2,y2;
    scanf("%d %d %d",&n,&m,&D);         //读入 Carl 所走的路径中的线段个数，要走这条路径的
                                        //蚂蚁的总数，蚂蚁相继出发的时间间隔
    x1=0; y1=0; Rs=0;
    for(i=0;i<n;i++)                    //计算 Carl 所走的路径
    {
        scanf("%d %d",&x2,&y2);         //读入当前线段的终点
        if(x1>x2) t=0;                  //判断线段的走向
        if(y1<y2) t=1;
        if(x1<x2) t=2;
        if(y1>y2) t=3;
        for(;x1!=x2 || y1!=y2;)         //计算当前线段的终点(x1,y1)，即下一条线段的起点
        {
           route[Rs]=t; Rs++;
           x1+=Add[t][0]; y1+=Add[t][1];
        }
    }
    fin=0; sta=0;                       //到达终点的蚂蚁数和已经出发的蚂蚁数初始化为零
    for(i=0;i<=MaxL;i++)                //清空地图上所有蚂蚁
        for(j=0;j<=MaxL;j++)
            for(k=0;k<4;k++) ants[i][j][k]= -1;
}
void work()                             //对当前时刻进行模拟计算
{
       bool ok;
    int i,j,x,y,d,p,tmp;
    int go[MaxM];                       //蚂蚁停留的标志
    memset(go,0,sizeof(go));            //1 表示不移动，0 表示可能可以移动
    ok=0;                               //累计到达终点的蚂蚁数
    for(i=0;i<sta;i++) if(ant[i].x<0) ok++;
    for(i=0;i<sta;i++) if(ant[i].x>=0)  //根据优先级计算每个蚂蚁能否移动的标志
    {
        x=ant[i].x; y=ant[i].y;         //（x,y）表示当前坐标
        for (j=0;j<4;j++)
        if(ants[x][y][j]>=0)
            {
               tmp=ants[x][y][j];       //计算头在（x,y）且方向为 j 的蚂蚁序号
```

```
                if (ant[tmp].wait>ant[i].wait || (ant[tmp].wait==ant[i].wait && ant[tmp].
len>ant[i].len))          //若蚂蚁 tmp 的优先级较高，则蚂蚁 i 就不能动了，累计不能移动的蚂蚁数
                {
                    p=1; go[i]=1; ok++;
                    break;
                    }
                }
        }
    do{                                                    //若前方有蚂蚁，则设停留标志
        ok=0;
        for(i=0;i<sta;i++)
        if (!go[i]&&ant[i].x>=0)
        {
            d=map[ant[i].x][ant[i].y];                     //计算蚂蚁 i 的前进方向
            x=ant[i].x + Add[d][0]; y=ant[i].y+Add[d][1];//计算移动后的新坐标
            if(ants[x][y][d]>=0&&go[ants[x][y][d]]==1)   //若新坐标上有停留的蚂蚁，则蚂
                                                           //蚁 i 不能动
            {
                go[i]=1; ok=1;
                continue;
            }
        }
    } while (ok);       //如果有新的蚂蚁被堵住，那么需要继续判断会不会导致更多的蚂蚁被堵住
    for(i=0;i<sta;i++)
        if(!go[i]&&ant[i].x>=0)
            {           //若蚂蚁 i 可移动，则移动距离+1，撤去等待时间和移前位置的蚂蚁编号
            ant[i].len++; ant[i].wait=0;
            ants[ant[i].x][ant[i].y][ant[i].dir]=-1;
            }else if(ant[i].x>=0) ant[i].wait++;          //否则等待时间+1
            for(i=0;i<sta;i++)
            if (!go[i]&&ant[i].x>=0)                      //加新标记
            {
                x=ant[i].x; y=ant[i].y;
            if(x==100&&y==100&&i!=sta-1) { ants[100][100][map[100][100]]=i+1; }
                                                           //若洞口被移开了，新出现一只蚂蚁
            d=map[x][y];                                   //蚂蚁 i 的前进方向
            ant[i].x+=Add[d][0]; ant[i].y+=Add[d][1]; ant[i].dir=d; //求出新坐标
            if(ant[i].x==ex&&ant[i].y==ey)                 //蚂蚁 i 到达目的地
            {
            list[fin]=i;
            fin++;
            ant[i].x=-1;
            }else ants[ant[i].x][ant[i].y][d]=i;          //记录蚂蚁 i 进入新位置
        }
    }
    void write()                                           //输出当前测试数据组的解
    {
        int i;
        printf("Case %d:\n",cases);                        //输出测试数据组编号
        printf("Carl finished the path at time %d\n",ant[0].len+1);
                        //由于蚂蚁 Carl 的优先级最高，因此其移动时间即为移动长度
        printf("The ants finished in the following order:\n");
        printf("%d",list[0]);                              //输出按顺序到达目的地的蚂蚁
        for(i=1;i<m;i++) printf(" %d",list[i]);
        printf("\n");
```

```
    printf("The last ant finished the path at time %d\n",T+1);
                                                    //输出最后一只蚂蚁到达目的地的时间
    if(cases<N) printf("\n");                       //若非最后一组测试组，则换行
}
int main()
{
    int X,Y;                                        //记录蚂蚁 Carl 在某时刻的坐标
    scanf("%d",&N);                                 //输入测试用例数
    for(cases=1;cases<=N;cases++)                   //依次处理每组测试用例
    {
        init();
        X=100; Y=100; ex=-1; ey=-1;                 //输入当前测试组信息并初始化（由于负数处理起来
                                                    //比较麻烦，因此将所有坐标统一加上 100）
        for(T=0;fin<m;T++)                          //顺序模拟每一时刻，直至所有蚂蚁到达终点
        {
            if(T<Rs)
            {
                map[X][Y]=route[T];                 //根据 Carl 的路线更新地图上的标记
                X+=Add[route[T]][0]; Y+=Add[route[T]][1];
                if(T==Rs-1){ ex=X; ey=Y;}           //记录 Carl 所走路线的终点
            }
            if(T%D==0 && sta<m)                     //若有新出发的蚂蚁
            {
                if(ants[100][100][map[100][100]]<0) ants[100][100][map[100][100]]=sta;
                ant[sta].x=100; ant[sta].y=100;
                ant[sta].len=0; ant[sta].wait=0; ant[sta].dir=map[100][100];
                sta++;
            }
            work();                                 //对当前时刻进行模拟计算
            }
            write();                                //输出当前测试数据组的解
        }
}
```

试题 1-2　直升机机场（Heliport）

【问题描述】

当前，工作节奏很快，某公司投资建造了直升机机场，来减少其繁忙的管理人员的旅行时间。直升机机场通常是圆形的，建造在公司总部的楼顶上（见图 1.2-1）。

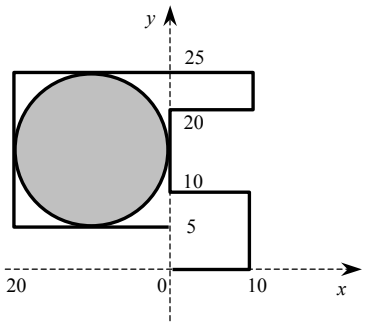

图　1.2-1

请编写一个程序，找到能建造在一个建筑物的平坦屋顶上的圆形直升机机场的最大半径。

屋顶的形状是一个简单多边形。由于这仅仅是施工的设计阶段的工作，因此该程序只需给出直升机机场的半径。图 1.2-1 中显示的直升机机场的最大半径是 10。

输入：

输入文件包含若干组测试用例，每组测试用例只有两行：第一行包含一个偶数 n（$4 \leqslant n \leqslant 20$），表述建筑物的边数；第二行包含 n 组形式为（m,d）的数据，其中 m 是一个整数（$1 \leqslant m \leqslant 50$），$d$ 是 U、R、D 或 L 中的一个字母。假定屋顶画在平面直角坐标系中，m 是屋顶的一条边的长度，d 是当你逆时针沿着屋顶边缘走时，经过这条边的方向。U、R、D 和 L 分别表示“上”、“右”、“下”和“左”。屋顶的边缘都是与 x 轴或 y 轴平行的，按逆时针顺序给出。起始位置是原点（0,0）。

最后一个测试用例后，给出一行，是一个数字 0。

输出：

对于每组测试用例，输出一行，包含测试用例编号（从 1 开始）和一个实数（四舍五入到小数点后两位），表示直升机机场的最大半径。在两个测试用例之间输出一个空行，如样例输出所示。

样例输入	样例输出
4 2 R 2 U 2 L 2 D 10 10 R 10 U 10 L 10 U 10 R 5 U 30 L 20 D 20 R 5 D 0	Case Number 1 radius is: 1.00 Case Number 2 radius is: 10.00

试题解析

1. 计算多边形的顶点坐标

建筑物是一个多边形，试题仅给出每条边的长度 m 和方向 d。我们从（$x[0]$,$y[0]$）=（0,0）出发，依据这些信息计算递推多边形 n 个顶点的坐标（$x[i]$,$y[i]$）（$0 \leqslant i \leqslant n-1$）：

当 d='U'时，（$x[i+1], y[i+1]$）=（$x[i], y[i]+m$）；

当 d='R'时，（$x[i+1], y[i+1]$）=（$x[i]+m, y[i]$）；

当 d='D'时，（$x[i+1], y[i+1]$）=（$x[i], y[i]-m$）；

当 d='L'时，（$x[i+1], y[i+1]$）=（$x[i]-m, y[i]$）。

2. 计算直升机机场的最大半径

采用二分的方法求直升机机场的最大半径。假设半径的可能区间为[ra,rb]，设 r=（ra+rb）/2，判断半径为 r 的直升机机场能否建造：如果能，则在右区间[r,rb]搜索最大半径；否则在左区间[ra,r]搜索最大半径。区间的初值设为[0,999]，当 $rb-ra<10^{-6}$ 即可停止二分过程，下面将展开对这个问题的分析。

3. 判断半径为 r 的直升机机场能否建造

假设半径为 r 的直升机机场可以建造在多边形内，我们可以移动这个直升机机场，使它满足以下 3 个条件中的一个，然后求出候选的圆心坐标，再逐一判断所求出的坐标是否合法。

1）直升机机场贴着两条互相垂直的边。

2）直升机机场贴着一条边和一个顶点。

3）直升机机场贴着两个顶点。

下面依次讨论这 3 种情况：

1）直升机机场贴着两条互相垂直的边（如图 1.2-2 所示）。

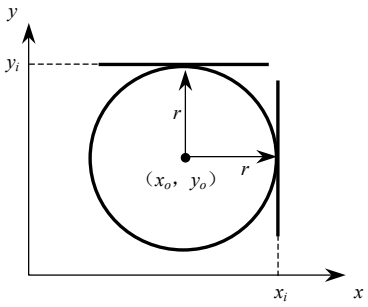

图　1.2-2

假设与 y 轴平行的边的 x 坐标为 x_i，与 x 轴平行的边的 y 坐标为 y_i，则候选的圆心坐标有 4 个：(x_i+r, y_i+r)、(x_i+r, y_i-r)、(x_i-r, y_i+r) 和 (x_i-r, y_i-r)。

2）直升机机场贴着一条边和一个顶点（如图 1.2-3 所示）。

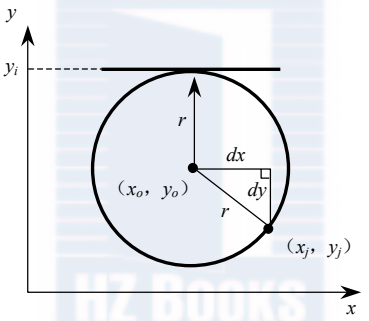

图　1.2-3

以与 x 轴平行的边为例，与 y 轴平行的边可以类似地处理。假设与 x 轴平行的边的 y 坐标为 y_i，顶点坐标为 (x_j,y_j)，则候选圆心的 y 坐标为 $y_o=y_i-r$ 或 $y_o=y_i+r$。如果 $y_j<y_i$，则取 $y_o=y_i-r$，否则取 $y_o=y_i+r$，求得 $dy=|y_o-y_j|$。如果 $dy>r$，则没有候选的方案；否则可以求得 $dx=\sqrt{r^2-dy^2}$，候选的圆心坐标为 (x_j+dx,y_o) 和 (x_j-dx,y_o)。

3）直升机机场贴着两个顶点（如图 1.2-4 所示）。

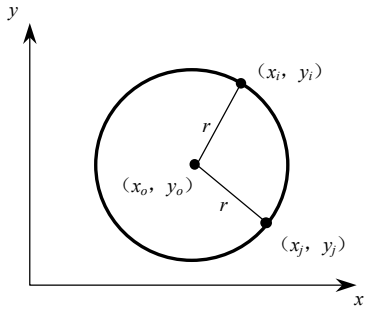

图　1.2-4

假设顶点坐标为 (x_i,y_i) 和 (x_j,y_j)，通过解方程组

$$(x_i-x_o)\times(x_i-x_o)+(y_i-y_o)\times(y_i-y_o)=r^2$$
$$(x_j-x_o)\times(x_j-x_o)+(y_j-y_o)\times(y_j-y_o)=r^2$$

即可求得圆心坐标（x_o,y_o）。

剩下的问题就是当圆心坐标和半径都确定之后，判断此方案是否合法。显然，只要满足以下两个条件，该方案就是合法的：

1）圆心在多边形内。

2）多边形的边不与圆相交。

于是，我们将候选的圆心坐标逐一判断，就可以知道半径为 r 的直升机机场能否建造，最终二分得到最大的可以建造的直升机机场半径。

其他细节见程序清单。

程序清单

```
#include <cstdio>
#include <math.h>
const double eps=1e-6;                  //精度

int x[25],y[25],n,i,len,px,py,cases;//第 i 条边的终点坐标为（x[i],y[i]）；当前边的长度为 len
                                        //终点坐标为（px,py）；测试用例编号为 cases
double ra,rb,r;                         //直升机机场半径的可能区间范围为[ra,rb]，实际半径值为 r
char dr;                                //边的方向字符
bool check(double ox,double oy)         //判断圆心为(ox,oy)、半径为 r 的方案是否可行
{
   int i,s;
   s=0;                                 //射线与多边形边相交的次数初始化
   for(i=0;i<n;i++)                     //从(ox,oy)往 x 轴正方向射出一条射线
      if(x[i]>ox&&((y[i]>oy)^(y[i+1]>oy))) s++;//累计射线与多边形边相交的次数
      if(s%2==0)return false;           //如果是偶数，则说明圆心在多边形外，方案不可行
      for(i=0;i<n;i++)                  //逐一判断每条边和每个顶点
      {
         if((x[i]-ox)*(x[i]-ox)+(y[i]-oy)*(y[i]-oy)<(r-eps)*(r-eps))
                                        //若顶点在圆内，方案不可行
            return false;
         if(x[i]==x[i+1]&&((y[i]>oy)^(y[i+1]>oy))&&fabs(x[i]-ox)<r-eps) return false;
                                        //若与 y 轴平行的边在圆内，方案不可行
        if(y[i]==y[i+1]&&((x[i]>ox)^(x[i+1]>ox))&&fabs(y[i]-oy)<r-eps)return false;
                                        //若与 x 轴平行的边在圆内，方案不可行
   }
   return true;                         //否则方案可行
}
bool ok()                               //判断半径为 r 的机场能否建造
{
   int i,j;
   double di,dd,mx,my,dx,dy;
   for(i=0;i<n;i++)                     //机场贴着两条互相垂直的边，求圆心坐标
      if(x[i]==x[i+1])
         for(j=0;j<n;j++)
            if(y[j]==y[j+1])
            {
               if(check(x[i]+r,y[j]+r)) return true;
               if(check(x[i]+r,y[j]-r)) return true;
```

```
            if(check(x[i]-r,y[j]+r)) return true;
            if(check(x[i]-r,y[j]-r)) return true;
        }
        for(i=0;i<n;i++)                    //机场贴着一条边和一个顶点，求圆心坐标
        for(j=0;j<n;j++)
            if(x[i]==x[i+1])
                {
                di=fabs(x[j]-(x[i]+r));
                if(di<r)
                {
                    dd=sqrt(r*r-di*di);
                    if(check(x[i]+r,y[j]+dd)) return true;
                    if(check(x[i]+r,y[j]-dd)) return true;
                }
                di=fabs(x[j]-(x[i]-r));
                if(di<r)
                {
                    dd=sqrt(r*r-di*di);
                    if(check(x[i]-r,y[j]+dd)) return true;
                    if(check(x[i]-r,y[j]-dd)) return true;
                }
            }
            else
            {
                di=fabs(y[j]-(y[i]+r));
                if(di<r)
                {
                    dd=sqrt(r*r-di*di);
                    if(check(x[j]+dd,y[i]+r)) return true;
                    if(check(x[j]-dd,y[i]+r)) return true;
                }
                di=fabs(y[j]-(y[i]-r));
                if(di<r)
                {
                    dd=sqrt(r*r-di*di);
                    if(check(x[j]+dd,y[i]-r)) return true;
                    if(check(x[j]-dd,y[i]-r)) return true;
                }
        }
    for(i=0;i<n-1;i++)                      //机场贴着两个顶点，求圆心坐标
        for(j=i+1;j<n;j++)
        {
            mx=(x[i]+x[j])/2.0;
            my=(y[i]+y[j])/2.0;
            di=sqrt((x[i]-mx)*(x[i]-mx)+(y[i]-my)*(y[i]-my));
            if(di>0&&di<r)
            {
                dd=sqrt(r*r-di*di);
                dx=(my-y[i])/di*dd;
                dy=(x[i]-mx)/di*dd;
                if(check(mx+dx,my+dy)) return true;
                if(check(mx-dx,my-dy)) return true;
            }
        }
        return false;
```

```
    }
    int main()
    {
        while(scanf("%d",&n)&&n)          //读入建筑物的边数 n
        {
            if(cases) printf("\n");
            px=0;py=0;                    //(px,py)赋初值(0,0)
            for(i=1;i<=n;i++)
            {
                scanf("%d %c",&len,&dr); //读入第 i 条边的长度和方向
                if(dr=='R') px+=len;
                if(dr=='L') px-=len;
                if(dr=='U') py+=len;
                if(dr=='D') py-=len;      //根据长度和方向决定下一个点的坐标
                x[i]=px;
                y[i]=py;                  //将顶点坐标记录到 x 数组和 y 数组中
            }
            ra=0;rb=999;                  //可能半径的初始区间为[ra,rb]=[0,999]
            while(rb-ra>eps)              //若区间长度小于 eps，则结束二分
            {
                r=(ra+rb)/2;              //取区间中点 r
                if(ok())ra=r;             //如果半径 r 可行，则在右区间[r,rb]搜索最大半
                                          //径；否则在左区间[ra,r] 搜索最大半径
                else rb=r;
            }
            printf("Case Number %d radius is: %.2lf\n",++cases,r);//输出结果
        }
        return 0;
    }
```

试题 1-3 六面视图（Image Is Everything）

【问题描述】

你的新公司正在制造一个可以举起轻重量小体积物体的机器人。这个机器人必须有足够的智力来判断一个物体是否轻得可以举起来。它在物体的 6 个方向对物体拍照，然后基于这些图片来求出物体重量的上限，以此判断这一物体是否可以被举起。请为机器人编写一个程序来判断物体是否可以被举起。

本题设定每个物体都被放置在一个 $N\times N\times N$ 的立方体中，在这个立方体中有些 1×1×1 的立方空间没有被该物体占据。物体的每个 1×1×1 的立方块重量为 1 克，每个这样的方块涂一种颜色。这个物体可能是不连通的。

输入：

输入包含多组测试用例，每组用例描述不同的物体。每组测试用例的第一行是一个整数 N，表示物体的大小（$1\leqslant N\leqslant 10$）。后面的 N 行给出按前、左、后、右、上、下的顺序，给出物体在 6 个方向上看到的不同的 $N\times N$ 的视图。每个视图之间以一个空格隔开。物体的俯视图的底边与前视图的顶边相对应。同样，底视图的顶边与前视图的底边相对应。在每个视图中，颜色以一个唯一的大写字母来表示，一个点（.）表示物体在那个位置是可以看透的。

在最后的测试用例输入后，给出一行，是整数 0。

输出：

对于每组测试用例，输出一行，给出物体的最大可能的重量，具体格式见样例输出。

样例输入	样例输出
3 .R. YYR .Y. RYY .Y. .R. GRB YGR BYG RBY GYB GRB .R. YRR .Y. RRY .R. .Y. 2 ZZ ZZ ZZ ZZ ZZ ZZ ZZ ZZ ZZ ZZ ZZ ZZ 0	Maximum weight: 11 gram(s) Maximum weight: 8 gram(s)

试题解析

考虑到本题中 N 的上限至多为 10，不妨采用枚举的方法：

首先，假设物体是“实的”，也就是说没有空的小方块。此时可能会产生一些矛盾：例如，某一块有两个颜色，或者某个可以看透的地方出现了小方块，这说明那个位置上不可能有小方块存在，因此应该把那个小方块删去。

这样一步一步地把所有矛盾的小方块都删去，剩下的就是合法的方案了。由于所有删去的小方块都是“不得不”删除的，因此这个合法方案也是重量最大的可能方案。

具体实现时，不妨设计一个 $N\times N\times N$ 的表，来记录物体中的每一个小方块的情况（空的或颜色还不确定），同时对每个面设计一个 $N\times N$ 的表，记录该面上对应的位置挖去了几个小方块。

此时矛盾与否是很容易判断的：若有两个颜色不同的面对应了同一个方块，此时第一次对应给这个方块颜色 A，第二次对应发现该方块已有的颜色与应有的颜色 B 不同，则说明产生了该方块有两种颜色的矛盾；同理，若某一个面对应的方块是“.”，但对应的小方块有颜色，则说明产生了可看透的地方出现了小方块的矛盾。一旦出现矛盾，只要在记录物体实际信息的表中把该小方块标记为空即可。

算法步骤大致为：

1）初始化所有的点都是不确定的，没有删去任何小方块。

2）判断把矛盾的小方块删去，直到不存在矛盾为止。

3）输出答案。

程序清单

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
const int MaxN=10+5;
char g1[MaxN][MaxN], g2[MaxN][MaxN], g3[MaxN][MaxN], g4[MaxN][MaxN], g5[MaxN][MaxN],
g6[MaxN][MaxN];                          //记录 6 个方向上的视图
int p1[MaxN][MaxN], p2[MaxN][MaxN], p3[MaxN][MaxN], p4[MaxN][MaxN], p5[MaxN][MaxN],
p6[MaxN][MaxN];                          //记录 6 个方向上的视图位置被删去的小方块数
int g[MaxN][MaxN][MaxN];                 //记录物体的实际信息
int n;
```

```
void init();                                    //读入各个视图
void work();                                    //求最大的重量
bool update_front();                            //判别前方向视图的矛盾情况
bool update_left();                             //判别左方向视图的矛盾情况
bool update_back();                             //判别后方向视图的矛盾情况
bool update_right();                            //判别右方向视图的矛盾情况
bool update_top();                              //判别上方向视图的矛盾情况
bool update_bottom();                           //判别下方向视图的矛盾情况
int main()
{
   for (; ;)
   {
      scanf("%d", &n);                          //输入物体大小
      if(n==0) break;                           //若输入 0，则结束
      init();                                   //读入各个视图
      work();                                   //求最大的重量
   }
   return 0;
   }
   void init()                                  //读入当前视图
   {
      for (int i=0; i<n; ++i)                   //读入第 i 行 6 个方向的视图
         scanf(" %s %s %s %s %s %s", g1[i], g2[i], g3[i], g4[i], g5[i], g6[i]);
   }
   void work()                                  //求最大的重量
   {
      fill(p1[0], p1[n], 0); fill(p2[0], p2[n], 0); fill(p3[0], p3[n], 0);
         fill(p4[0], p4[n], 0); fill(p5[0], p5[n], 0); fill(p6[0], p6[n], 0);
         fill(g[0][0], g[n][0], -1);//初始化，所有小方块的情况未知，目前一个方块都没有删除
      for(; ;)
      {
         bool quit=true;                        //quit 为可以跳出，即没有矛盾的标志
         if(update_front()) quit=false;         //判断前方向视图中是否发现矛盾
         if(update_left()) quit=false;          //判断左方向视图中是否发现矛盾
         if(update_back()) quit=false;          //判断后方向视图中是否发现矛盾
         if(update_right()) quit=false;         //判断右方向视图中是否发现矛盾
         if(update_top()) quit=false;           //判断上方向视图中是否发现矛盾
         if(update_bottom()) quit=false;        //判断下方向视图中是否发现矛盾
         if(quit) break;                        //若 6 个视图未发现矛盾，则转而计算输出最大重量
      }
      int ans=n*n*n;                            //最大重量初始化
      for(int i= 0; i<n; ++i)                   //减去一定是空的小方块
         for(int j=0; j<n; ++j)
            for(int k=0; k<n; ++k)
            if(g[i][j][k]==0) --ans;
            printf("Maximum weight: %d gram(s)\n", ans);//输出最大重量
   }
   bool update_front()                                          //判别前方向视图的矛盾情况
{
   bool upd=false;
   for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
   {
      int &t=p1[i][j];              //t 等价于 p1[i][j]，记录(i,j)处挖去了几个小方块
      char ch=g1[i][j];             //ch 等价于 g1[i][j]，记录(i,j)对应什么字母
```

```
        if(t==n) continue;                  //如果全都挖掉了，没有判断的必要
        int &val=g[n-t-1][j][i];            //val 等价于物体中对应的小方块的信息
        if (val==0){ ++t; upd=true;}        //如果是空的，说明还要挖去一个小方块
        else if(ch=='.'){ val=0; ++t; upd=true; }
                                            //如果这一列是空的，说明整个都要挖去，一个一个挖走
        else if(val==-1){ val=ch-'A'+1;upd=true;}
                                            //如果无法判断，则说明只有这个面“到达”过对应的小
                                            //方块，赋予其相应的颜色
        else if(val!=ch-'A'+1){ val=0; ++t; upd=true; }
                                            //如果与另外某个面上的颜色有矛盾，则说明小方块是空
                                            //的，挖去
        }
        return upd;
    }
    bool update_left()                      //判别左方向视图的矛盾情况，与 update_front 类似
    {
        bool upd = false;
        for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j)
    {
           int &t=p2[i][j];
              char ch=g2[i][j];
              if(t==n) continue;
              int &val=g[j][t][i];
              if(val==0){ ++t; upd=true;}
                  else if(ch=='.'){ val=0; ++t; upd=true;}
                     else if(val==-1) { val=ch-'A'+1; upd=true;}
                         else if(val!=ch-'A'+1) { val=0; ++t; upd=true;}
    }
    return upd;
}
bool update_back()                          //判别后方向视图的矛盾情况，与 update_front 类似
{
    bool upd = false;
    for(int i=0; i<n; ++i)
    for(int j=0; j<n; ++j)
    {
        int &t = p3[i][j];
        char ch = g3[i][j];
        if(t==n) continue;
        int &val=g[t][n-j-1][i];
        if(val==0){ ++t; upd=true; }
        else if(ch='.'){ val = 0; ++t; upd=true; }
        else if(val==-1){ val=ch-'A'+1; upd=true; }
        else if(val!=ch-'A'+1) { val=0; ++t; upd=true; }
    }
    return upd;
}
bool update_right()                         //判别右方向视图的矛盾情况，与 update_front 类似
{
    bool upd=false;
    for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
    {
        int &t=p4[i][j];
```

```
            char ch=g4[i][j];
            if(t==n)continue;
            int &val=g[n-j-1][n-t-1][i];
            if (val== 0){ ++t; upd=true; }
            else if (ch=='.'){ val=0; ++t; upd=true; }
            else if (val==-1){ val=ch-'A'+1; upd=true; }
            else if (val!=ch-'A'+1) { val=0; ++t; upd=true; }
        }
        return upd;
    }

    bool update_top()          //判别上方向视图的矛盾情况，与 update_front 类似
    {
        bool upd=false;
        for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j)
        {
            int &t=p5[i][j];
            char ch=g5[i][j];
            if(t==n) continue;
            int &val=g[i][j][t];
                if(val==0){ ++t; upd=true; }
                else if(ch=='.'){ val=0; ++t; upd=true; }
                else if(val==-1){ val=ch-'A'+1; upd=true; }
                else if(val!=ch-'A'+1){ val=0; ++t; upd=true; }
        }
        return upd;
    }

    bool update_bottom()       //判别下方向视图的矛盾情况，与 update_front 类似
    {
        bool upd = false;
        for (int i=0; i<n; ++i) for (int j=0; j<n; ++j)
        {
            int &t=p6[i][j];
            char ch = g6[i][j];
            if(t==n) continue;
            int &val=g[n-i-1][j][n-t-1];
            if(val==0){ ++t; upd=true; }
            else if(ch=='.'){ val=0; ++t; upd=true; }
            else if(val==-1) { val=ch - 'A' + 1; upd=true; }
            else if(val!=ch-'A'+1) { val = 0; ++t; upd = true; }
    }
    return upd;
    }
```

试题 1–4　危险的布拉格城（Insecure in Prague）

【问题描述】

Prague（布拉格）对于密码系统的开发者来说是一个危险的城市。2001 年，在 Prague，两位研究人员公布了著名的 PGP 加密协议中的一个安全漏洞。2003 年，也在 Prague，SSL/TLS（加密套接字协议层（Secure Sockets Layer，SSL）和传输层安全（Transport Layer Security，TLS））协议中的一个漏洞也被发现了。尽管这样，Prague 在密码协议领域中的名声并不能阻止业余的密码爱好者和全职的“疯子”，Immanuel Kant-DeWitt（被他的朋友们称为 I. Kant-DeWitt）把他

的最新的密码方案带到 Prague。以下就是其编码的原理。

假设需要传输一个长度为 n 的纯文本消息 p。发送方挑选一个数字 $m \geqslant 2n$，以及整数 s、t、i 和 j，其中 $0 \leqslant s$、t、i、$j < m$ 以及 $i < j$。编码的方式是这样的：m 为需要传输的加密文本字符串 c 的长度，在初始的时候，c 有 m 个空槽。首先将 p 的第一个字符置于字符串 c 中的位置 s 中；p 的第 k 个字符（$k \geqslant 2$）被放在 c 中的第 k–1 个字符位置后再跳过 i 个空槽以后的第一个空槽中，如果已经到字符串 c 的结尾，则转回 c 的开头处。已经包含了字母的槽就不再是空槽。例如，对于发送的消息 PRAGUE，如果令 s=1，i=6，m=15，那么在字符串 c 中放置的字母如下：

A	P	_	U	_	_	_	_	R	G	_	_	E	_	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

然后，从字符串 c 中的第 t 个位置上或第 t 个位置之后的第一个空槽开始，再次将纯文本消息 p 输入一次，但这次在两个字母之间要跳过 j 个空槽。例如，如果 t=0，j=8，那么 p 的第二次输入情况如下（从 t=0 开始，第一个空槽的位置为 2）：

A	P	P	U	R	_	A	U	R	G	E	G	E	_	_
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

最后，在 c 中所有未被填充的空槽内随机地填入被选的字母。

A	P	P	U	R	A	A	U	R	G	E	G	E	W	E
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Kant-DeWitt 相信，包含了随机字母的重复的消息，会使得基于分析字母出现频率等来破译密码的做法变得困难，并且在不知道 s 和 i 的情况下，没人可以推导出原来的信息究竟是什么。请尝试证明他的观点是错误的。给出一些加密文本的字符串（没有其他的信息），应用 Kant-DeWitt 的方法给出被编码的最长的可能的消息。

输入：

输入若干加密文本的字符串，每个字符串一行。每个字符串仅包括大写字母，前后没有空格字符，长度在 2～40 之间。

输入结束数据为一行，仅包括一个字母 X。

输出：

对于每个加密文本字符串，输出最长的可以被加密为该字符串的消息。如果具有最长长度的消息多于一条，则输出“Codeword not unique”。具体格式见样例输出。

样例输入	样例输出
APPURAAURGEGEWE ABABABAB THEACMPROGRAMMINGCONTEST X	Code 1: PRAGUE Code 2: Codeword not unique Code 3: Codeword not unique

试题解析

本题可采用枚举法。需要枚举的量一共有 5 个：

1）消息长度 n，n 的范围是 1～20。

2）开始位置 s，s 的范围是 0～39。

3）步长 i，i 的范围是 0～38。

4）第二遍的开始位置 t，t 的范围是 0～39。

5）第二遍的步长值 j，j 的范围是（i+1）～39。

对于规模最大的数据，枚举量为 2.5×10^7。把 n、s、i、t、j 确定下来后，检查是否符合要求所需的运算量为 $n\times j$。显然，直接枚举的运算量过大，不能在题目要求的时间限制内求出答案。

上述枚举方案实际上有一些不必要的重复计算。为了避免重复计算，可进行相应的预处理：用数组 pos[*len*][*j*][*k*]来表示在长度为 *len* 的字符串中，从第 0 个字符开始，以步长为 *j* 遍历字符串 *k* 次之后到达的位置。若要从第 *t* 个字符开始，以步长为 *j* 遍历字符串，而不是从第 0 个字符开始，则遍历 *k* 次之后到达的位置为（pos[*len*][*j*][*k*]+*t*）%*len*。预处理这个数组所需的运算量为 $40^4/2=1.3\times10^6$。有了数组 pos[][][]，检查当前方案是否符合要求的时候就不需要每次都以步长 *j* 来定位，直接访问这个数组即可定位，这样可使得检查一个方案所需的运算量降为 n。

另外，如果找到长度为 n 的解，则后面只需对不小于 n 的长度进行枚举。一旦找到两个解，则可立即停止对长度为 n 的解进行枚举，直接把最小长度设为（n+1）。

实现过程和具体细节见程序清单。

程序清单

```
#include <stdio.h>
#include <memory>
char a[41],count[26],current[21],mark[41], remain[81],ans[21],pos[41][41][41];
//输入的字串为 a[]，字符的频率为 count[]，第一遍遍历过的字符为 current[]，位置标记为 mark[]，第
一//遍未遍历过的字符为 remain[]，最优解方案为 ans[]，从第 0 个字符出发，以步长 j 遍历长度为 len 的
字//串 k 次后到达的位置为 pos[len][j][k]
int i,j,k,n,go,len,step,skips,maxl,remains,start,unique,cases,ok;
//当前走到的位置为 go，步长值为 step，一步跨过的字符数为 skips，符合要求的最大长度为 maxl，剩余字
//符数为 remains，第二遍的开始位置为 start，当前长度的解存在且唯一地被标记为 unique，测试用例编
//号为 cases，成功标志为 ok
int main()
{
    for(len=1;len<41;len++)                          //预处理
        for(step=1;step<41;step++)                   //对长度为 len 的字符串，以步长 step 遍历
        {
            memset(mark,0,sizeof(mark));             //某位置是否到达过的标记
            go=0;                                    //当前走到的位置
            mark[0]=1;                               //对第 0 个位置做标记
            pos[len][step][0]=0;
            for(i=1;i<len;i++)                       //遍历 len 遍
            {
                skips=(step-1)%(len-i)+1;            //这一步跨过的字符数
                while(skips)                         //跨过若干个字符
                {
                    go++;                            //更新当前走到的位置
                    if(go>=len) go=0;                //如果到达末尾，则回到首位
                    if(!mark[go]) skips--;           //如果当前位置未标记，则计一次数
                }
                mark[go]=1;                          //对当前走到的位置做标记
                pos[len][step][i]=go;                //记下第 i 次到达的位置
                }
}
while(1)
{
    gets(a);
    for(n=0;a[n];n++);                               //求读入的字符串的长度
```

```
    if(n<2) break;                          //如果长度小于 2，则结束程序
cases++;                                    //记录加密文本字串的序号
maxl=0;                                     //目前找到的符合要求的最大长度
unique=0;                                   //当前长度的解是否存在且唯一地被标记
memset(ans,0,sizeof(ans));
memset(count,0,sizeof(count));
for(i=0;i<n;i++)                            //统计每个字符出现的次数
    count[a[i]-'A']++;
for(i=0;i<n;i++)                            //第一遍的开始位置
    if(count[a[i]-'A']>1)                   //这个字符要出现 2 次或 2 次以上
        for(j=1;j<n;j++)                    //第一遍的步长
        {
            memset(current,0,sizeof(current));      //第一遍走过的字符
            memset(mark,0,sizeof(mark));            //第一遍到过的位置
            go=i;                                   //当前到达的位置
            for(len=0;len<n/2;len++)                //枚举解的长度
            {
                current[len]=a[go];                 //记录第一遍走过的字符
                mark[go]=1;                         //在第一遍到过的位置做标记
                if(len>=maxl&&strcmp(current,ans))  //如果长度不小于最优解且当前
                                                    //解与最优解不同
                {
                    remains=0;                      //统计剩余字符
                    ok=0;                           //是否找到符合要求的解的标记
                    for(k=0;k<n;k++)                //逐个检查每个字符
                        if(!mark[k]) remain[remains++]=a[k];
                                                    //记录第一遍未遍历过的字符
                    for(k=0;k<remains;k++)          //为方便后面的计算，把未遍历过
                                                    //的字符复制一份接在末尾
                        remain[remains+k]= remain[k];
                        for(start=0;start<remains;start++) //枚举第二遍的开始位置
                        if(remain[start]==current[0])//第二遍的开始位置要和第一遍的开始
                                                    //位置相同
                        {
                            for(step=j+1;step<=n;step++)        //枚举第二遍的步长
                            {
                                for(k=1;k<=len;k++)             //逐一检查后面的每个字符
                                  if(remain[pos[remains][step][k]+start]!=current[k])
                                    break;          //如果和第一遍不匹配，则退出循环
                                if(k>len)           //如果全部字符均匹配
                                {
                                   ok=1;
                                   if(len>maxl)     //如果比最优解好
                                   {
                                      maxl=len;     //更新最优解
                                      unique=1;     //当前的最优解存在且唯一
                                      memcpy(ans,current,sizeof(current));
                                                    //把最优解记录到 ans 数组
                                   }
                                   else if(unique)  //如果不比最优解好
                                   {                //如果当前长度已经找到解了
                                      maxl++;       //最优解的长度增加 1
                                      unique=0;     //当前长度尚未找到解
                                   }
                                   else             //如果当前长度尚未找到解
```

```
                        {
                            unique=1;       //当前长度已经找到解而且是唯一解
                            memcpy(ans,current,sizeof(current));
                                            //把当前长度的解记录到 ans 数组
                        }
                        break;              //退出循环
                        }
                    }
                    if(ok)break;            //如果找到符合要求的解就退出循环
                }
            }
            skips=j;                        //跨过 j 个字符
            while(skips)
            {
                go++;                       //更新当前位置
                if(go>=n)go=0;              //如果到达末尾，则回到首位
                if(!mark[go]) skips--;      //如果当前位置未标记，则计一次数
            }
        }
    }
    printf("Code %d: ",cases);                  //输出加密文本字符串的序号
    puts(unique?ans:"Codeword not unique");//输出加密结果
    }
    return 0;
}
```

试题 1–5　相交的时间段（Intersecting Dates）

【问题描述】

一个研究小组正在开发一个计算机程序，通过访问一台服务器来获取以前的股票市场的报价信息，而这台服务器要对每天传送来的报价收取固定的费用。这个小组通过调查以前访问服务器的要求报价的信息集合，发现其中有大量的重复信息，这也导致了金钱的浪费。因此，新的程序需要维护一个包含过去所有的报价信息的列表，当有新的报价要求的时候，只有以前没有被获取的报价信息才会从服务器被获取，这样可以使花费最小。

请编写一个程序，来确定什么时候有新的报价产生。程序的输入包括过去有报价请求的时间段信息，以及当前的报价请求的时间段信息。程序确定服务器要获取哪些时间段的报价。

输入：

输入包含多组测试用例，每组测试用例以两个非负整数 *NX* 和 *NR* 开头，（1≤*NX*，*NR*≤100）。*NX* 表示过去有多少个时间段存在报价请求，*NR* 表示将要有多少个时间段的报价请求。接下来给出 *NX*+*NR* 对数据。在每对数据中，第一个日期小于或等于第二个日期。前面的 *NX* 对给出了过去的服务器已经获得的报价请求的时间段，后面的 *NR* 对给出将要有的报价请求的时间段。

在最后一组测试用例后，用两个 0 表示结束。

每个日期以 YYYYMMDD 的形式给出。YYYY 表示年份（从 1700 到 2100），MM 表示月份（从 01 到 12），DD 表示日（在当前年月允许的日期范围内），04 月、06 月、09 月和 11 月有 30 天，01 月、03 月、05 月、07 月、08 月、10 月、12 月有 31 天，02 月在闰年有 29 天，平年有 28 天。如果某个年份可以被 4 整除且不是一个世纪年（100 的整数倍），或该年份可以被 400 整除，那么这个年份是闰年。

输出：

对于每组输入的测试用例，输出测试用例编号（1，2，…），然后输出一个日期范围的列表，表示服务器要获取的报价的日期范围，每个日期范围一行，如样例输出，以美式日期格式输出。如果没有新的报价被获取，则要如样例所示明确地指出。如果两个日期范围是连续的或重叠的，将它们合并为一个日期范围。如果一个日期范围内仅包含一个日期，那么把它作为单个的日期输出，而不是两个相同的日期组成的日期范围。从最小的日期开始按时间顺序输出。

样例输入	样例输出
1 1 19900101 19901231 19901201 20000131 0 3 19720101 19720131 19720201 19720228 19720301 19720301 1 1 20010101 20011231 20010515 20010901 0 0	Case 1: 1/1/1991 to 1/31/2000 Case 2: 1/1/1972 to 2/28/1972 3/1/1972 Case 3: No additional quotes are required.

试题解析

本题的数据范围是 1700 年到 2100 年。从 1700 年 1 月 1 日算起，到 2100 年 12 月 31 日，共计不超过 147 000 天，而且已知的区间和新的询问区间均不超过 100 个。所以对于所有的已知区间和新的询问区间，可以逐一在一个长度为 147 000 的数组里做标记。这样就完全可以在本题的时间限制内直接求得答案。

实现上述方法需要将原始日期对应到 1～147 000 的范围中，并反过来将 1～147 000 的数对应到日期并输出。使用两个函数就可以实现。

函数 1：传入一个日期，把 1700 年 1 月 1 日作为第 1 天，统计当前日期距离 1700 年 1 月 1 日的天数。

函数 2：传入一个天数，计算从 1700 年 1 月 1 日算起，这一天是哪年哪月哪日。

具体细节见程序清单。

程序清单

```
    #include <stdio.h>
    #include <memory>
    int  n,m,i,j,a,b,cases,no_quotes,st[101],ed[101],days[13]={0,31,28,31,30,31,30,31,
31,30,31,30,31};                          //过去报价的时间段数为 n，其中第 i 个时间段为[st[i],
                                          //ed[i]]；将来报价的时间段数为 m，当前时间段为[a，b]；
                                          //没有获取新报价的标志为 no_quotes，平年第 i 个月的天数
                                          //为 days[i]
    char c[147000];                       //第 i 天新老报价的标志为 c[i]
    int leap(int y)                       //判断年份 y 是否为闰年
    {
        return y%400==0||y%4==0&&y%100;
    }
```

```
void get_date(int day)                    //计算输出从 1700 年 1 月 1 日算起的第 day 天的美式日期
{
    int y,m;
    for(y=1700;day>365+leap(y);y++)       //计算年份 y 和 y 年的天数 day
        day-=365+leap(y);
    for(m=1;day>days[m]+(m==2&&leap(y));m++) //计算月份 m 和 m 月的天数 day
        day-=days[m]+(m==2&&leap(y));
    printf("%d/%d/%d",m,day,y);               //输出美式日期
}

int f(int x)                                  //统计当前日期 x 距离 1700 年 1 月 1 日的天数
{
    int i,tmp,sum;
    tmp=x/10000;                          //截出年份
    sum=0;
    for(i=1700;i<tmp;i++)                 //累计 1700 年至 tmp-1 年的天数
    {
        sum+=365;
        if(leap(i)) sum++;
    }
    tmp=x%10000/100;                      //截出月份
    for(i=1;i<tmp;i++)                    //累计至 tmp-1 月的天数
    {
        sum+=days[i];
        if(i==2&&leap(x/10000)) sum++;
    }
    return sum+x%100;                     //累加 tmp 月的天数
}
void fill(int a,int b,int x)              //将区间[日期 a,日期 b]标记为 x
{
    int fa,fb;
    fa=f(a);                              //把日期 a 换算成天数
    fb=f(b);                              //把日期 b 换算成天数
    memset(&c[fa],x,fb-fa+1);             //c[fa]…c[fb]标记为 x
}
int main()
{
    while(scanf("%d%d",&n,&m)&&n+m)       //反复输入过去报价和将来报价的时间段数
    {
        if(cases) puts("");
        memset(c,0,sizeof(c));            //初始化标记日期的数组
        for(i=0;i<n;i++)                  //输入过去报价的时间段
        scanf("%d%d",&st[i],&ed[i]);
        for(i=0;i<m;i++)                  //输入将来报价的时间段
        {
            scanf("%d%d",&a,&b);
            fill(a,b,2);                  //把将来报价的时间段标记成 2
        }
        for(i=0;i<n;i++)
            fill(st[i],ed[i],1);          //把过去报价的时间段标记成 1
        printf("Case %d:\n",++cases);
        no_quotes=1;
        for(i=0;i<147000;i++)             //枚举 1700 年 1 月 1 日至 2100 年 12 月 31 日的每一天
            if(c[i]>1)                    //若当天属于将来报价的时间，则转换成日期并输出
            {
```

```
        no_quotes=0;                //设获取新报价的标志
        get_date(i);
        if(c[i+1]<2) puts("");      //若新报价仅一天，则输出完毕
            else                    //否则输出新报价的日期区间
            {
                printf(" to ");
                for(j=i;c[j]>1;j++);//找到区间结束端
                get_date(j-1);      //把区间结束端转换成日期并输出
                puts("");
                i=j;                //循环变量 i 转到区间结束端
                    }
                }
                if(no_quotes) puts("No additional quotes are required.");
                                    //没有获取新报价
            }
            return 0;
        }
```

试题 1-6　拼接地图（Merging Maps）

【问题描述】

如果要将从飞机或卫星上对一个区域拍摄的照片用于绘制地图，那么就要求这些照片具有很高的分辨率，能很好地描述地形地貌。由于一张照片只能覆盖地球的一小部分，因此在绘制一个范围较大的地图时，就需要把覆盖了小区域的有重叠的照片拼接成一个大区域的地图。

本题给出一些矩形区域的地图，每张地图用一个字符串数组表示。如果一个字符表示一个特定的地形，则用一个从“A”到“Z”的大写字母来表示，不同的字母表示不同的地形，但相同的地形（如一条公路）用多个字符表示。如果字符是一个连字符号（“-”），表示这个字符对应的区域没有明显的地形。把两张地图合并，就是将两张地图适当地重叠在一起，来表示一个更大的区域。同一个地点可能在一张地图上以某个特定字符表示，而在另一张地图上以“-”表示，但不可能在两张地图上用两个不同的字母表示。

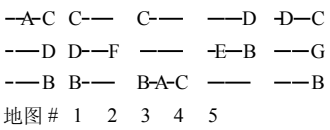

图　1.6-1

如图 1.6-1 所示，给出 5 张 3 行 5 列的地图。地图 1 的最右边的一列与地图 2 的最左边的一列可以完美地匹配，因此这两张地图可以合并产生一张 3 行 9 列的地图。但是，地图 1 同样可以和地图 3 合并，因为地图 1 最右边的一列中，地形 C 和地形 B 与地图 3 的最左边的一列相符，地形 D 与“-”并不能完美地覆盖，但两者没有冲突。同样，地图 1 的第一行也可以与地图 3 的最后一行重叠。

两张地图的“分数”表示这两张地图匹配的程度。两张要拼接的地图的分数就是两张地图彼此覆盖起来完美重合的字符的数目。两张地图的分数也就是所有可能的拼接中最高的分数。因此，两张要拼接的 3 行 5 列的地图的分数的范围在 0～15 之间。

“偏移量”以一对整数（*r*,*c*）表示，记录了两张地图 a 和 b 是如何拼接的。*r* 的值表示地图 b 中的行相对于地图 a 的偏移量；类似的，*c* 的值表示地图 b 中的列相对于地图 a 的偏移量。例如，覆盖上面的地图 1 和地图 2 的偏移量为（0,4），分数为 3。地图 1 和地图 3 的两个拼接的分数均为 2，偏移量为（0,4）与（–2,0）。

以下步骤描述了如何把一系列地图拼接起来：

1）把分数最高的两张地图（这个分数必须是正的）拼接起来（如果存在分数相同的多对地图，取标号较小的地图）。

2）把已经拼接的地图删去。

3）把拼接好的新地图加进来，地图标号为当前地图序列中标号最大的地图标号+1。

在图 1.6-1 所示的例子中，地图 1 和地图 2 应该合并为地图 6，地图 1 和地图 2 将从地图集合中删除。重复上述步骤，直到地图中只剩下一张地图，或任何一对地图都不能拼起来了（即每对地图的分数都为 0）。

如果两张地图可以以多种方式拼出相同的分数，那么用最小的行偏移量来拼接。如果还有多种方案，选择列偏移量最小的那个。

输入：

输入包含一组或多组测试用例，每组测试用例包含 2～10 张地图。每组测试用例先给出一个整数，表示测试用例中有多少张小地图；然后描述这些地图，每张地图的第一行为两个整数 *NR* 和 *NC*（1≤*NR*，*NC*≤10），表示地图的行数和列数；接下来紧跟着 *NR* 行，每行的前 *NC* 个字母描述了地图的数据，对于后面可能出现的多余的字符要忽略。

最后一组测试用例后为仅包含一个整数 0 的一行。

输出：

对于每组测试用例，输出测试用例的编号数（1，2，…），然后输出拼接而成的地图，每张地图用它的序列号编号，以边界线包围起来。具体格式见样例输出。每张拼接而成的地图不会多于 70 列。

样例输入	样例输出
5 3 5 --A-C ----D ----B 3 5 C---- D---F B---- 3 5 C---- ----- B-A-C 3 5 ----D -E--B ----- 3 5 -D--C ----G ----B 2 3 5 ----A	Case 1 MAP 9: +------------+ |-D--C--------| |----G--------| |----B-A-C----| |--------D---F| |-----E--B----| |------------| +------------+ Case 2 MAP 1: +-----+ |----A| |----B| |----C| +-----+ MAP 2: +-----+ |A----| |B----| |D----|

（续）

样例输入	样例输出
----B ----C 3 5 A---- B---- D---- 0	+-----+

试题解析

本题要求将至多 10 张地图按照某种拼接规则拼接为一张或多张大地图。由于题目范围很小，不必使用优化，仅按照题目的要求一步步模拟下去即可。整个过程如下：

1）计算任意两张地图之间的拼接分数以及相应的偏移量。

2）取拼接分数最大的一对地图，拼接为一张大地图。

3）删除原来的两张小地图，把新的大地图添加到地图列表中。

4）重复上述步骤，直至仅剩下一张地图或任意两张地图的拼接分数为 0。

为了提高程序效率，避免多余的计算，可以将计算拼接分数和偏移量的部分移到循环外部，整个过程为：

1）计算任意两张地图之间的拼接分数以及相应的偏移量。

2）判断是否还可以继续拼接：若是，则进行步骤 3），否则进行步骤 7）。

3）取拼接分数大的一对地图，拼接为一张大地图。

4）删除原来的两张小地图，把新的大地图添加到地图列表中。

5）计算大地图和地图列表中其他地图之间的拼接分数和偏移量。

6）返回步骤 2），循环执行。

7）终止程序并输出答案。

此方案和前一个计算方案的区别在于：仅计算新的地图的相关量，而不重复计算已有地图的拼接分数和偏移量，节约了一定的时间。具体实现方式参考程序清单。

程序清单

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

struct MAPS                          //保存地图的相关信息，包括行列数以及地形图
{
    int n,m;
    char feature[100][100];
};
struct Merge                         //保存一对地图拼接的信息，包括行列偏移量以及拼接分数
{
    int score,r,c;
};
char s[2000];                        //地图的当前行
int cases,n,i,j,k,sum,max,x,y;       //测试用例编号为 cases，地图数为 n，最大拼接分数为 max
MAPS map[20];                        //地图序列
```

```
Merge merge[20][20];        //地图 x 和地图 y 拼接成地图 merge[x][y]
bool sign[20];              //地图剩下的标记
int calc(char a, char b)    //若其中某个地形是空地，则分数为 0；若两个地形相匹配，则分数为 1；
                            //不匹配则不是合法方案，记其分数为-10000，以保证最终分数和为负
{
    if(a=='-') return 0;
    if(b=='-') return 0;
    if(a==b) return 1;
else return -10000;         //分数至多为 70×100=7000，一旦有不合法的分数就减 10000，保证了不
                            //合法的最后分数一定是负的
}
int Max(int a,int b)                          //返回 a 和 b 的大者
{
    if(a>b) return a;
    else return b;
}
int Min(int a,int b)                          //返回 a 和 b 的小者
{
    if(a<b) return a;
    else return b;
}
Merge Try(int x,int y)                        //拼接地图 x 和地图 y，返回分数和行列偏移量
{
    int r,c,i,j,k,score,x1,y1,x2,y2;
    Merge tmp;
    tmp.score=0;                              //分数初始化为 0
    for(r=-map[y].n+1;r<map[x].n;r++)         //枚举行列偏移量
        for(c=-map[y].m+1;c<map[x].m;c++)
        {
            x1=Max(0, r); y1=Max(0, c);       //计算重合区域的两个顶点(x1, y1)和(x2, y2)
            x2=Min(map[x].n, map[y].n+r); y2=Min(map[x].m, map[y].m+c);
            score=0;
            for (i=x1;i<x2;i++)                       //计算拼接分数
                for (j=y1;j<y2;j++) score+=calc(map[x].feature[i][j], map[y].
                    feature[i-r] [j-c]);
            if(score>tmp.score)                       //更新最优解
            { tmp.score=score; tmp.r=r; tmp.c=c; }
    }
    return tmp;                               //返回拼接地图的信息
}
void add_map(int x,int y,int z)               //将地图 x 和 y 拼接成地图 z
{
    int i,j,k,Dx,Dy;

    Dx=Max(-merge[x][y].r, 0); Dy=Max(-merge[x][y].c, 0);//计算地图 x 在拼接地图中的平移量
    map[z].n=Max(map[x].n, map[y].n+merge[x][y].r)+Dx;  //计算地图 z 的大小
    map[z].m=Max(map[x].m, map[y].m+merge[x][y].c)+Dy;
    for(i=0;i<map[z].n;i++)                           //将地图 z 填充为空地
        for(j=0;j<map[z].m;j++) map[z].feature[i][j] = '-';
    for(i=0;i<map[x].n;i++)                           //将地图 x 中的地形填入地图 z
        for(j=0;j<map[x].m;j++) map[z].feature[i+Dx][j+Dy]=map[x].feature[i][j];
    Dx+=merge[x][y].r; Dy+=merge[x][y].c;             //计算地图 y 在拼接地图中的平移量
        for(i=0;i<map[y].n;i++)                       //将地图 y 中的地形填入地图 z
            for(j=0;j<map[y].m;j++)
      if(map[y].feature[i][j]!='-') map[z].feature[i+Dx][j+Dy]=map[y].feature[i][j];
```

```
}
int main()                                                  //主算法
{
    for(cases=1;;cases++)                                   //依次处理每个测试用例
    {
        scanf("%d",&n);                                     //输入地图数
        if(n==0) break;                                     //若输入 0，则结束
        for(i=1;i<=n;i++)                                   //输入地图信息
        {
            scanf("%d %d ",&map[i].n,&map[i].m);            //输入第 i 张地图的规模
            for(j=0;j<map[i].n;j++)                         //逐行输入第 i 张地图的信息
            {
                gets(s);
                for (k=0;k<map[i].m;k++) map[i].feature[j][k] = s[k];
            }
        }
        for(i=1;i<=n;i++)                   //计算任意两张地图之间的拼接分数和偏移量
        for(j=i+1;j<=n;j++) merge[i][j]=Try(i,j);
        memset(sign,1,sizeof(sign));        //标记所有地图均可用
        while(1)
        {
            max=0;
            for(i=1;i<=n;i++) if(sign[i]) //从现存地图中找到拼接分数最大的一对地图
                for(j=i+1;j<=n;j++) if(sign[j])
                    if(merge[i][j].score>max)
                    {
                        max=merge[i][j].score;
                        x=i;y=j;
                    }
                    if(max==0) break;       //若只剩一张地图或所有地图的拼接分数为 0,则此时 max
                                            //为 0，拼接过程结束
                    sign[x]=sign[y]=0;      //标记拼接的两张原始地图被丢弃
                    n++;                    //计算拼接地图的编号
                    add_map(x,y,n);         //将地图 x 和地图 y 拼接成地图 n
                    for(i=1;i<n;i++)        //计算地图 n 与现存的其他地图之间的拼接分数和偏移量
                        if(sign[i]) merge[i][n]=Try(i,n);
        }
        if(cases>1) printf("\n");
        printf("Case %d\n",cases);          //输出测试用例编号
        x=0;
        for(i=1;i<=n;i++)
        if(sign[i])                         //按要求输出最终剩下的所有地图
        {
            if(x) printf("\n"); x++;
            printf("    MAP %d:\n", i);     //输出拼接而成的地图编号和地图信息
            printf("    +"); for (j=0;j<map[i].m;j++) printf("-"); printf("+\n");
            for(j=0;j<map[i].n;j++)
            {
                printf("    |");
                for(k=0;k<map[i].m;k++) printf("%c",map[i].feature[j][k]);
                printf("|\n");
            }
            printf("    +"); for(j=1;j<=map[i].m;j++) printf("-"); printf("+\n");
        }
    }
}
```

试题 1–7 导航（Navigation）

【问题描述】

全球定位系统（Global Positioning System，GPS）是基于一系列在地球大约 20 000 公里的高空按轨道运行的人造卫星的导航系统。每颗人造卫星按一个已知的轨道运行，并发送对当前时间编码的无线电信号。如果一辆装备了 GPS 的车上有一个很精确的钟，它可以把本地时间与从卫星接收到的信号中的时间进行比较。由于无线电信号以一个已知的速度传播，因此车辆可以计算出从它的当前位置到卫星发射信号时的位置之间的距离。通过测量一些在已知轨道上运转的卫星的距离，就可以十分精确地计算出这辆车所在的位置。

请编写一个简单的基于 GPS 导航的“自动导航”程序。为了简化问题，我们设定这个问题是二维的，也就是说，本题不需要考虑地面的曲率和人造卫星的高度。而且，本题使用的速度对飞行器和声波比对卫星和无线电波更适合。

给出来自移动信号源的信号集合，所编写的程序要计算出在笛卡尔坐标系中接收信号的位置；然后给出平面上的目的地，所编写的程序需要计算出从接收信号的位置到目的地的罗盘方向。所有的罗盘方向以角度制表示。罗盘方向 0（正北）对应 y 轴正方向，罗盘方向 90（正东）对应 x 轴正方向，如图 1.7-1(1)所示。

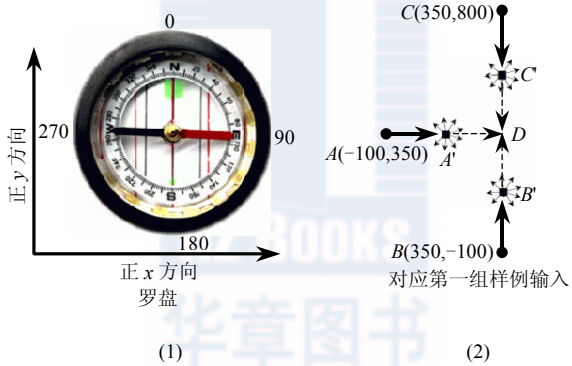

图 1.7-1

输入：

输入包含多组测试用例。每组测试用例的第一行第一个数是一个整数 N（$1 \leqslant N \leqslant 10$），表示测试用例中信号源的数量。接下来是 3 个浮点数：t，x 和 y，其中 t 表示收到所有信号时本地的精确时间，表示在标准时刻（时刻 0）以后的多少秒，x 和 y 表示在笛卡尔坐标系中目的地的坐标。接下来的 N 行，每行有 4 个浮点数，包含每个信号源的信息。前两个数字表示在标准时刻信号源在笛卡尔坐标系中的坐标。第三个数字以罗盘方向 D 的形式（$0 \leqslant D < 360$）表示信号源移动的方向。第四个数字是信号中所包含的时间，即信号发出的时刻，以在标准时刻以后多少秒的形式表示。输入中包含的所有数字均小于 10 000，且不存在小数点后有多于 5 位数字的浮点数。

最后一组测试用例以 4 个 0 表示输入结束。

在坐标系中的单位距离为 1 米。假设每个信号源均在笛卡尔坐标系中以 100 米/秒的速度移动，发出的信号速度为 350 米/秒。考虑到钟的同步误差，距离计算仅精确到 0.1 米即可。也就是说，如果两个点之间的距离不大于 0.1 米，就可以把它们视为同一个点。由于信号在传送过程

中出错的可能性是存在的，因此由多个信号源收到的信号可能存在不一致。

输出：

对于每组测试用例，输出测试用例编号，然后输出从收到信号的方向到目的地的罗盘方向，以角度制表示，四舍五入到最接近的整数。具体标号方式见样例输出。如果信号中没有包含足以计算出接收位置的信息（即有多于 1 个可能的位置），则输出“Inconclusive”；如果信号中存在不一致（即不存在与信号中的信息相符合的位置），则输出“Inconsistent”；如果接收到信号的点距离目标点不大于 0.1 米，则输出“Arrived”；如果情况是不确定或有矛盾的，就不需要考虑是否到达目标点。

图 1.7-1(2)对应第一组样例输入。在 $t=0$ 时刻，三个卫星位于 A（−100, 350），B（350, −100）和 C（350, 800）坐标处。GPS 接收到的信号在 $t=1.75$ 时刻发送，此时人造卫星位于点 A'、B' 和 C'（但是通常来说，GPS 接收到的信号是在不同的时刻发出的）。三个卫星发出的信号与时刻 $t=2.53571$ 在 D 位置汇合，这里 D 就是 GPS 的位置。从位置 D 到目的点（1050, 1050）的罗盘方向为 45 度。

样例输入	样例输出
3 2.53571 1050.0 1050.0 −100.0 350.0 90.0 1.75 350.0 −100.0 0.0 1.75 350.0 800.0 180.0 1.75 2 2.0 1050.0 1050.0 −100.0 350.0 90.0 1.0 350.0 −100.0 0.0 1.0 0 0 0 0	Trial 1: 45 degrees Trial 2: Inconclusive

试题解析

对于每个接收到的信号，根据信号的传输时间以及信号的传输速度可以确定信号的传输距离；再根据信号源的原位置和移动速度及方向就可以确定信号发出的位置。以信号发出的位置为圆心，信号的传输距离为半径作一个圆，则接收者的可能位置位于圆周上。根据每个信号的传输时间、传输速度和方向均可以确定一个圆。如果所有的圆周有且只有一个公共点，那么就可以完全确定接收者的位置。如果所有的圆周不止一个公共点，就不能完全确定接收者的位置。如果圆周没有公共点，就说明收到的信息有误。

设信号源的原位置为（px,py），信号源的移动方向为 degree，信号源的移动时间为 t_i，则信号源的移动距离 dis 等于 t_i*100。求信号发出的位置时，先将 degree 化为弧度，则信号发出的位置（ox,oy）=（px+dis*cos（degree），py+dis*sin（degree））。设接收信号的时间是 t，则信号的传输时间为（t–t_i），信号发出的位置离接收者的距离为 350*（t–t_i）。对于每个信号，以（ox,oy）为圆心，350*（t–t_i）为半径作一个圆，求所有圆的公共点。

实际上，只要从中任意选取两个不重叠的圆就可以大致地把解的情况确定下来：

1）若两个圆没有交点，则无解。

2）若两个圆相交，则只要检查两个交点是否在别的圆上即可。

3）若两个圆相切，则也需要检查一下切点是否在别的圆上。

如果找不到两个不重叠的圆，则说明所有的圆都重叠在一起了，有多解。

可以用如下方法求两圆的交点 M 和 N：

设两圆心距离为 d，圆 P 的半径为 a，圆 Q 的半径为 b。在三角形 PMQ 中，根据余弦定理，有 $\cos(P)=(a^2+d^2-b^2)/2ad$，然后求得 $x=a*\cos(P)=(a^2+d^2-b^2)/2d$，$y=\sqrt{a^2-x^2}$。最后，根据线段 PQ 的指向和 x、y 的值可以求得两个交点 M、N 的坐标。设 P、Q 的坐标分别为（px,py）、（qx,qy），则 M、N 的坐标分别为$(px+(qx-px)*x/d\pm(qy-py)*y/d,py+(qy-py)*x/d\pm(px-qx)*y/d)$（见图 1.7-2）。

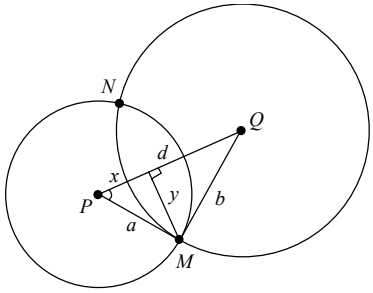

图 1.7-2

其他具体细节见程序清单。

程序清单

```
#include <cstdio>
#include <cmath>

double t,x,y,ox[12],oy[12],r[12],px,py,dx,dy,dr,degree,ti,pi,dis,lx,ly,xa,ya,xb,yb;
//信号发出位置的坐标序列为（ox[]、oy[]）、传输距离为 r[]，圆心横坐标的差距为 dx、纵坐标的差距 为
//dy、半径的差距为 dr，当前信号源的笛卡尔坐标为 px 和 py、方向为 degree、发出时间为 ti，pi 存储圆
//周率π，信号移动的距离为 dis，示意图中的 x 值和 y 值为 lx 和 ly，交点 1 的坐标为（xa,ya），交点 2 的
//坐标为（xb,yb）
int n,i,cases,c1,c2;                    //测试用例编号为 cases，交点 1 和交点 2 符合条件的标志为 c1 和 c2
int check(double x,double y)            //若(x,y)与所有圆的距离差在 0.1 的范围内，则符合条件，返回 1；
                                        //否则返回 0
{
    int i;
    double dx,dy;
    for(i=0;i<n;i++)                                    //顺序检查每个圆
    {
        dx=x-ox[i];                                     //计算(x,y)与第 i 个圆的距离差 dr
        dy=y-oy[i];
        dr=sqrt(dx*dx+dy*dy)-r[i];
        if(fabs(dr)>0.1) return 0;                      //差值在 0.1 范围内是可以接受的，返回 0 表
                                                        //示不符合条件
    }
    return 1;                                           //返回 1 表示符合条件
}

int main()                                              //主算法
{
    pi=acos(-1.0);                                      //圆周率π的精确表达式
    while(scanf("%d%lf%lf%lf",&n,&t,&x,&y)&&n)          //反复输入信号源数、收到所有信号时的精确
                                                        //时间和目的地坐标，直至输入 4 个 0 为止
    {
```

```
for(i=0;i<n;i++)                    //输入每个信号源的信息
{
    scanf("%lf%lf%lf%lf",&px,&py,&degree,&ti);
                                    //输入第 i 个信号源的笛卡尔坐标、方向和发出时间
    degree=(90-degree)/180*pi;      //把方向度数化为弧度
    dis=100*ti;                     //计算信号移动的距离
    ox[i]=px+dis*cos(degree);       //计算信号发出位置的坐标
    oy[i]=py+dis*sin(degree);
    r[i]=350*(t-ti);                //以信号的传输距离作为半径
}
printf("Trial %d: ",++cases);       //输出测试用例编号
for(i=1;i<n;i++)                    //以第一个圆为基准，逐一判断后面的圆是否与之重合
{
    dx=ox[i]-ox[0];                 //计算圆心横坐标的差距、纵坐标的差距和半径的差距
    dy=oy[i]-oy[0];
    dr=r[i]-r[0];
    if(dx*dx+dy*dy+dr*dr>0.01) break; //若 3 个差距的平方和大于 0.01，则说明两个圆不
                                    //重合，于是选择当前的圆，退出循环
}
if(i>=n)                            //如果所有的圆都重合，则输出多解
{
    puts("Inconclusive");
    continue;
}
dis=sqrt(dx*dx+dy*dy);              //求两个圆心的距离
if(dis<0.1)                         //在两圆不重合的情况下若圆心距离过小，则说明半
                                    //径有一定的差距，输出无解
{
    puts("Inconsistent");
    continue;
}
lx=(dis*dis+r[0]*r[0]-r[i]*r[i])/dis/2;   //lx 即为示意图中的 x
if(fabs(lx)>r[0]+0.1)                     //若 lx 的值比半径还大了不少，则说明两圆
                                          //无交点，输出无解
{
    puts("Inconsistent");
    continue;
}
if(lx>r[0]) lx=r[0];                      //若 lx 的值比半径略大，则把此值改为半径值
if(lx<-r[0]) lx=-r[0];                    //此值有可能是负数，同上处理
ly=sqrt(r[0]*r[0]-lx*lx);                 //计算示意图中的 y 值
dx/=dis;                                  //把示意图中的向量 PQ 单位化
dy/=dis;                                  //同上
xa=ox[0]+dx*lx-dy*ly;                     //计算交点 1 的 x 坐标和 y 坐标
ya=oy[0]+dy*lx+dx*ly;
xb=ox[0]+dx*lx+dy*ly;                     //计算交点 2 的 x 坐标和 y 坐标
yb=oy[0]+dy*lx-dx*ly;
if(sqrt((xa-xb)*(xa-xb)+(ya-yb)*(ya-yb))<0.1)
                                          //若两个交点距离很近，则废除其中一个交点
{
    xb=1e9;
    yb=1e9;
}
c1=check(xa,ya);                          //检查交点 1 是否符合条件
c2=check(xb,yb);                          //检查交点 2 是否符合条件
if(c1+c2==1)                            //如果只有一个交点符合条件
```

```
    {
        if(c2)                                  //如果是交点 2 符合条件
        {
            xa=xb;                              //统一记为(xa,ya)
            ya=yb;
        }
        dx=x-xa;                                //计算接收点与目的地的 x 坐标差、y 坐标差
        dy=y-ya;
        dis=sqrt(dx*dx+dy*dy);                  //计算接收点与目的地的距离
        if(dis<0.1) puts("Arrived");            //如果距离小于 0.1，则认为到达了
        else                          //否则计算从收到信号的方向到目的地的罗盘方向(角度制)
        {
            if(dy>0) degree=acos(dx/dis);       //把坐标差化为弧度
            else degree=pi*2-acos(dx/dis);
            degree=90-degree/pi*180;            //把弧度化为角度
            if(degree<0) degree+=360;           //把角度标准化
            if(degree>360) degree-=360;
            printf("%.0lf degrees\n",degree);
                                  //输出以角度制表示的罗盘方向，四舍五入到最接近的整数
        }
    }
    else                                        //在不是仅一个交点符合条件的情况下
        if(c1) puts("Inconclusive");            //若均符合条件，则输出多解
        else puts("Inconsistent");              //若均不符合条件，则输出无解
    }
    return 0;
}
```

试题 1-8　道路绿化（Tree-Lined Streets）

【问题描述】

最近 Greenville 市议会投票通过了改进市中心街道景观的提案。为了在景色中加入更多的绿色，市议会决定沿着所有主要街道和大路种树。为了知道这一城市改进项目需要花多少钱，市议会希望能确定要种多少棵树。树是按如下两条规则种植的：

1）在一条街道上，两棵树之间的距离至少为 50 米。这是为了给树提供足够的成长空间，同时也是为了将这一项目的花费控制在合理的范围内。

2）为了安全考虑，在距离交叉路口 25 米以内不得种树。这是为了保证车辆驾驶员在接近交叉路口时可以方便地看到彼此，不致因为能见度降低而影响交通安全。

在这一项目中，所有街道都是直的，没有转弯或弯曲的道路。

市议会需要知道在上述两条限制规则之下至多能种多少棵树。

输入：

输入包含若干组街道地图。每组描述的第一行是一个整数 n（$1 \leqslant n \leqslant 100$），表示地图中街道的数量；接下来的 n 行，每行描述了一条街道，这条街道可以认为是笛卡尔平面中的一条线段。一条街道以 4 个整数 x_1、y_1、x_2 和 y_2 来描述，这些数字表示这条街道从点（x_1,y_1）到点（x_2,y_2）。坐标 x_1、y_1、x_2 和 y_2（$0 \leqslant x_1$、y_1、x_2、$y_2 \leqslant 100\,000$）以米为单位。每条街道的长度都是正数。每一个端点仅在一条街道上。

对于每条街道，邻接的交叉路口之间的距离，邻接的交叉路口和端点之间的距离，以及邻接的端点之间的距离不是 25 的整数倍。更确切地说，这个距离与 25 的整数倍之间至少相差 0.001

米。在每个交叉路口，只有两条街道交叉。

街道地图的输入以包含 0 的一行结束。

输出：

对于输入中的每张街道地图，首先按顺序输出它的编号，然后输出按上述限制规则最多能种树的数目。具体格式见样例输出。

样例输入	样例输出
3 0 40 200 40 40 0 40 200 0 200 200 0 4 0 30 230 30 0 200 230 200 30 0 30 230 200 0 200 230 3 0 1 121 1 0 0 121 4 0 4 121 0 0	Map 1 Trees = 13 Map 2 Trees = 20 Map 3 Trees = 7

试题解析

首先，不妨将题目中的条件整理一下：

1）不可能存在两条线段的端点重合。

2）不可能存在三条线段交于一点。

3）计算精度在 0.001 即可。

这样对于每条线段，我们不妨求出它与其他所有线段的交点（如果存在的话）。这些交点把线段分割成若干段，对于每一段，设其长度为 *len*，这一段最多种 *k* 棵树 $\left(k=\left\lfloor\frac{len}{50}\right\rfloor\right)$。端点和邻接的交点的距离可以特判，也可以把端点拉远 25 米作为一个虚拟的交点处理。

求线段交点时，不妨先用叉乘法判断是否有交点，然后列出两条线段的标准型并解方程。具体见程序清单，这里不再赘述。

程序清单

```
#include <stdio.h>
#include <math.h>
#include <algorithm>
using namespace std;
struct line                          //线段的结构类型
{
    int x1,y1,x2,y2;                 //线段的两个端点为(x1,y1)和(x2,y2)
};
int cases,n,ans,s,i,j,k;                 //测试用例编号为 cases，线段数为 n，树的最大棵数为 ans
```

```
line a[110];                          //线段序列
double b[200];                        //以 b 记录某条线段与其他线段的交点到某端点的距离
double len;
bool check(int x1,int y1,int x2,int y2,int x3,int y3)
                                      //判断 p(x1,y1)(x2,y2) 是否在 p(x1,y1)(x3,y3) 的顺时针方向
{
    long long t1,t2;
    t1=(long long)x1*(long long)y2-(long long)x2*(long long)y1;
    t2=(long long)x1*(long long)y3-(long long)x3*(long long)y1;
    return((t1>=0&&t2<=0)||(t1<=0&&t2>=0));
}
double dis(double x,double y)         //求出点(x,y)到原点的距离
{
    return sqrt(x*x+y*y);
}
void cross(line A,line B)             //求线段 A 和 B 的交点
{
    double A1,B1,C1,A2,B2,C2,x,y;     //线段 A 所在的直线方程为 A1*x+B1*y+C1=0，线段 B 所在的
                                      //直线方程为 A2*x+B2*y+C2 =0；两者的交点为点(x,y)
    A1=A.y2-A.y1; B1=A.x1-A.x2; C1 =-(A.x1*A1+A.y1*B1);
    A2=B.y2-B.y1; B2=B.x1-B.x2; C2 =-(B.x1*A2+B.y1*B2);
    x=-(C1*B2-C2*B1) / (A1*B2-A2*B1);
    y=-(A1*C2-A2*C1) / (A1*B2-A2*B1);
    b[s]=dis(x-a[i].x1,y-a[i].y1);    //该交点与第 i 条线段的端点间的距离记入 b[s]
}
bool init()                           //输入当前组的街道地图；若返回 false，则整个输入结束
{
    int i;
    scanf("%d",&n);                   //输入街道（线段）数
    if(n==0)
    return false;                     //整个输入结束
    for(i=1;i<=n;i++)                 //依次输入每条线段的两个端点坐标
        scanf("%d %d %d %d",&a[i].x1,&a[i].y1,&a[i].x2,&a[i].y2);
    return true;
}
int main()                            //主算法
{
    cases=0;                          //街道地图的编号初始化
    while(init())                     //依次输入各组的街道地图
    {
        ans=0;                        //树的最大棵数初始化
        for(i=1;i<=n;i++)
        {
           s = 2;
        b[1]=-25; b[2]=dis(a[i].x2-a[i].x1,a[i].y2-a[i].y1)+25;
                                      //把两个端点分别拉远 25 米作为虚拟的交点处理
        for(j=1;j<=n;j++)
            if(i!=j)                  //判断第 i 条线段与第 j 条线段是否相交，即一条线段的两个端
                                      //点是否都在另一条线段的两侧
            if(check(a[i].x2-a[i].x1,a[i].y2-a[i].y1,a[j].x1-a[i].x1,a[j].y1-a[i].y1,
a[j].x2-a[i].x1,a[j].y2-a[i].y1))
            if(check(a[j].x2-a[j].x1,a[j].y2-a[j].y1,a[i].x1-a[j].x1,a[i].y1-a[j].y1,
a[i].x2-a[j].x1,a[i].y2-a[j].y1))
            {
                s++; cross(a[i],a[j]);                    //交点数+1，更新 b 表
```

```
            }
                sort(b+1,b+s+1);                          //按距离排序
                for(j=1;j<s;j++)                          //对于每一段，求出能种多少棵树
                {
                    len=b[j+1]-b[j];                      //计算当前段的长度
                    k=(int)(len/50);//计算当前段种的树的最多棵树 k=⌊len/50⌋，精度误差不超过 10^-6
                    while (len-k*50>1e-6)k++;
                    while (len-k*50<1e-6)k--;
                    ans+=k;                               //累计树的总棵树
                }
            }
            printf("Map %d\nTrees=%d\n",++cases,ans);  //输出树的最大棵数
        }
    }
```

试题 1-9 悬吊！（Suspense!）

【问题描述】

Jan 和 Tereza 住在两栋相邻的建筑物中，并且她们的房间是面对面的。为了完成学校的科研项目，她们要用绳索、细线以及纸板做一个微型吊桥连接她们所住的两栋建筑。她们把两条长度一样的绳索做成吊桥的主吊索，并将这两根吊索绑到她们的窗户的底部；桥的硬纸板“车行道”被许多细线连接到主吊索上。桥面是水平的，并且正好位于吊索最低点以下一米处。考虑到审美的因素，桥面至少比两个学生的窗户的底边低两米。根据物理定律，每条吊索都是一条抛物线。

尽管 Jan 和 Tereza 不会走这个模型桥，但还有一个严重的问题：楼里的有些住户养宠物猫，有些养宠物鸟。Jan 和 Tereza 要保证她们的吊桥不会为宠物猫袭击宠物鸟提供通道。经过观察，Jan 和 Tereza 确定猫至多向上能跳 0.5 米，向下可以跳 3 米。因此，只要桥面至少高于猫所在的窗户的底部 0.5 米，或至少低于窗户底部 3 米，猫就不能跳上桥去。类似地，只要桥面比鸟所在的窗户的底部高 3 米，或低 0.5 米，吊桥上的猫就不能袭击鸟。宠物猫们只关心能否袭击到鸟，它们并不关心如何回家。

图 1.9-1 给出了 Jan 的房间（标记为“J”）和 Tereza 的房间（标记为“T”），有一条绳索连接了这两个房间的窗户的底部，桥面在绳索的最低点以下 1 米处。二楼的猫可以通过吊桥袭击二楼的鸟。

请编写一个程序，计算 Jan 和 Tereza 需要准备多长的绳索，才能使得吊桥不会使两栋楼里任何一只鸟受到威胁。

输入将给出两栋建筑物的距离（以米为单位），Jan 和 Tereza 所住的楼层层数（每栋楼最低层，或者说在地面上的那一层记为 1 楼），生活在 Jan 所住的建筑中的宠物的种类（到 Jan 所住的楼层为止），生活在 Tereza 所住的建筑中的宠物的种类（到 Tereza 所住的楼层为止）。所编写的程序需要计算出悬挂在两栋建筑物之间的吊桥的绳索的最长长度，要保证没有一只猫能够利用桥到一只鸟的身边。桥面比地面高 1 米，恰好在绳索的最低点的 1 米以下，并且桥面距离 Jan 和 Tereza 中较低者的窗户的底部至少 2 米。建筑物中的所有房间均恰好 3 米高，所有的窗户均高 1.5 米，窗户的底部离房间底部恰好为 1 米。

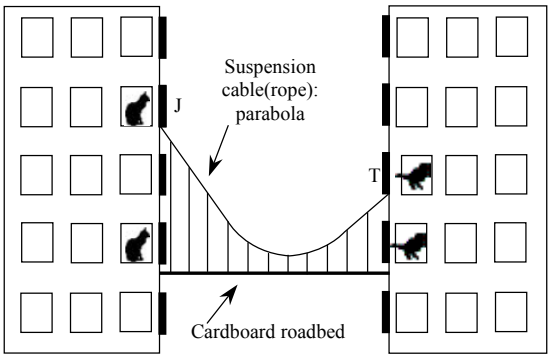

图 1.9-1

输入：

输入包含若干测试用例，每个测试用例有 3 行。第一行是：两个正整数 j 和 t（$2 \leqslant j,t \leqslant 25$），分别表示 Jan 住的楼层和 Tereza 住的楼层；一个实数 d（$1 \leqslant d \leqslant 25$），表示两栋建筑物之间的距离。第二行包括 j 个以空格分开的大写字母 l1,l2,…,lj；若 Jan 居住的建筑物的 k 楼有一只鸟，则字母 lk 为 B；若住着猫则为字母 C，否则以字母 N 来标记两者均不是。第三行与第二行类似，以 t 个大写字母来描述 Tereza 居住的建筑中住户所拥有的宠物种类。最后一组测试用例后是 3 个 0。

输出：

对于每组测试用例，输出测试用例的编号数（1,2,…）以及最大值 c，使得两条长度为 c 的绳索可以用来连接 Jan 和 Tereza 的窗户，以保证桥面恰好比绳索的最低点低 1 米，距离两人的窗户至少 2 米，并且不会让猫去袭击鸟。结果保留小数点后三位。如果无法建造一座满足条件的吊桥，输出“impossible”。两组测试用例之间输出一个空行，输出格式参考样例输出。

样例输入	样例输出
4 3 5.0 N C N C N B B 4 3 5.0 C B C C B C B 0 0 0	Case 1: 14.377 Case 2: impossible

试题解析

显然，要使得 Jan 和 Tereza 准备的绳索最长，也就意味着路基的高度最低。因此可以将题目分解为两个部分：

（1）求路基的最低高度

我们不妨以一些特殊的点把高度分为若干个区间来讨论（这里以 H_i 表示第 i 层的窗户的高度）。经过初步分析，考虑到可能的特殊点有：

1）窗户的高度 H_i；

2）恰好跳不上某个窗户的高度为 H_i–0.5；

3）恰好不能从某窗户跳上去的高度为 H_i+0.5；

4）恰好不能从窗户跳下桥的高度为 H_i–3；

5）恰好不能从桥上跳下去的高度为 H_i+3。

以楼层高度 3 米为一个周期，讨论[H_i–0.5，H_i+2.5]之间的高度：

高度	可能出现猫的楼层	可能被袭击的鸟的楼层
H_i–0.5	第 i 层	第 i–1 层
(H_i–0.5, H_i)	第 i 层	第 i–1 层，第 i 层
H_i	第 i 层	第 i 层
(H_i, H_i+0.5)	第 i 层，第 i+1 层	第 i 层
H_i+0.5	第 i+1 层	第 i 层
(H_i+0.5, H_i+2.5)	第 $i \neq$ +1 层	第 i 层

可以看出，对于每个开区间（A）和它的左端点（B）：若 B 不可行，则 A 一定不可行；若 B 可行，则 A 也不一定可行且高度一定大于 B，因此 A 一定不是最优解。因此，我们不必关心 A，只要考虑所有的点 B 就可以了。换言之，只要讨论每个周期中的 3 个特殊点 H_i–0.5、H_i和 H_i+0.5 即可。

通过枚举所有的特殊点，判断是否可行，即可得到路基的最低高度。

（2）求某条抛物线的长度

当知道了最低高度以后，如何求抛物线的长度呢？

首先，不妨以 $Y = A(x-B)^2 + C$ 来表示抛物线方程。显然，C 为抛物线的最低点，即路基的最低高度＋1。抛物线经过两个点：$(0, H_j)$，(d, H_t)，即方程组 $H_j = AB^2 + C$，$Ht = A(B-d)^2 + C$。

当 $H_j = H_t$ 时，由对称性可知 $B = \frac{d}{2}$，否则可将上述两式两端分别减去 C，再相除，可得

$$\frac{H_j - C}{H_t - C} = \frac{B^2}{(B-d)^2}$$

其中 H_j、H_t、C、d 均为已知项。整理可得关于 B 的一个一元二次方程：

$$(H_t - H_j)B^2 + 2d(H_j - C) - d^2(H_j - C) = 0$$

这里 B 可能有两组解，但考虑到最低点一定位于两栋建筑物之间，可知 $0 \leqslant B \leqslant d$，通过这一条件即可求出 B 的唯一解。

当 B 求出以后，由 $H_j = AB^2 + C$，可以解出 $A = \frac{H_j - C}{B^2}$。

求出抛物线方程以后，再求曲线的长度，有两种方法：

方法 1：把曲线分割得非常细，然后将每一段近似他认为是一条直线，把每一条直线段的长度累加起来即可得出曲线的长度。这种方法简洁明了，写起来也很方便，但会造成一定的精度误差，无法保证答案的精确度。

方法 2：套用公式 $L = \int_0^d \sqrt{x'^2(t) + y'^2(t)}\,\mathrm{d}t$，其中 $x(t) = t$，$y(t) = A(t-B)^2 + C$，代入 x 和 y，可得 $L = \int_0^d \sqrt{1 + 4A^2(t-B)^2}\,\mathrm{d}t = \int_0^B \sqrt{1 + 4A^2t^2}\,\mathrm{d}t + \int_0^{d-B} \sqrt{1 + 4A^2t^2}\,\mathrm{d}t$。此时套用公式 $\int \sqrt{ax^2 + c}\,\mathrm{d}x = \frac{x}{2}\sqrt{ax^2 + c} + \frac{c}{2\sqrt{a}}\ln(x\sqrt{a} + \sqrt{ax^2 + c}),(a > 0)$ 即可求出曲线的长度。下面的程序采用的是第二种方法。

程序清单

```
#include <stdio.h>
#include <string>
#include <math.h>
char ch;                          //窗口信息
int C, Tj, Tt, i, j, k;           //Jan 和 Tereza 住的楼层分别为 Tj、Tt
double d, ans;                    //两栋楼之间的距离为 d，连接两人窗户的绳索长度为 ans
int a[30], b[30];                 //Jan 所在建筑物的各楼层信息为 a[],Tereza 所在建筑物的各楼层
                                  //信息为 b[]

double calc_2(double a, double c, double x)
//计算积分公式 ∫√(ax²+c)dx = x/2·√(ax²+c) + c/(2√a)·ln(x√a+√(ax²+c))，(a>0)
{
    return x/2*sqrt(a*x*x+c)+c/(2*sqrt(a))*log(x*sqrt(a)+sqrt(a*x*x+c));
}
double calc_1(double C)           //求出抛物线公式 Y=A（x-B)²+C，并计算长度
{
    double A, B, a, b, c;         //考虑到直接以原始数据解一元二次方程求 B 比较麻烦，所以以 a，b，
                                  //c 代替方程中的 3 个系数
    if (Tt==Tj) B=d/2;
    else {
        a=(Tt-Tj)*3;b=2*d*(Tj*3-2-C); c=-d*d*(Tj*3-2-C);    //这里 Hj=3*Tj-2,Ht=3*Tt-2
        B=(-b+sqrt(b*b-4*a*c))/(2*a);
        if(B<0||B>d) B=(-b-sqrt(b*b-4*a*c))/(2*a);
    }
    A=(Tj*3-2-C)/(B*B);
        return calc_2(4*A*A,1,B)+calc_2(4*A*A,1,d-B);
}

int in()                          //判断当前宠物的属性，1 表示猫，2 表示鸟，0 表示其他
{
    scanf("%c", &ch);             //读入某个窗口的信息
    while(ch!='C' && ch!='B' && ch!='N') scanf("%c", &ch);//忽略空白字符
    if(ch=='C') return 1;
    else if(ch=='B') return 2;
    else return 0;
}
int main()                        //主算法
{
    C=0;
    while(1)
    {
        scanf("%d %d %lf", &Tj, &Tt, &d);//输入 Jan 和 Tereza 住的楼层以及两栋楼之间的距离
        if(Tj==0&&Tt==0&&d<1e-6) break;
        memset(a, 0, sizeof(a)); memset(b, 0, sizeof(b));
        for(i=1; i<=Tj; i++) a[i]=in();  //输入 Jan 所在建筑物的各楼层信息
        for(i=1; i<=Tt; i++) b[i]=in()   //输入 Tereza 所在建筑物的各楼层信息
        ans=-1;
        for(i=1;i<Tt && i<Tj;i++)        //自下而上搜索 Jan 和 Tereza 所住楼层下方的每一楼层
        {
            if(i>1)                      //离地至少 1 米，因此 1 楼窗口往下 0.5 米不可行
                if((a[i]!=1 && b[i]!=1)||(a[i-1]!=2 && b[i-1]!=2))
                                         //若没有猫能上吊桥绳索，或没有鸟被袭击，则可行
```

```
            {
                ans=calc_1(i*3-2-0.5+1);            //计算和输出连接两人窗户的绳索长度
                if(C) printf("\n");                 //若非首个测试用例，则换行
                printf("Case %d: %.3lf\n", ++C,ans);
                break;
            }
            if((a[i]!=1 && b[i]!=1) || (a[i]!=2 && b[i]!=2))
            {
                if(C) printf("\n");                 //若非首个测试用例，则换行
                ans=calc_1(i*3-2+1);                //计算和输出连接两人窗户的绳索长度
                printf("Case %d: %.3lf\n", ++C, ans);
                break;
            }
            if((a[i+1]!=1 && b[i+1]!=1) || (a[i]!=2 && b[i]!=2))
            {
                if(C) printf("\n");                 //若非首个测试用例，则换行
                ans=calc_1(i*3-2+0.5+1);            //计算和输出连接两人窗户的绳索长度
                printf("Case %d: %.3lf\n", ++C, ans);
                break;
            }
        }
        if (ans<0)                                  //输出无法建造吊桥的信息
        {
            if(C) printf("\n");                     //若非首个测试用例，则换行
            printf("Case %d: impossible\n", ++C);
        }
    }
}
```

试题 1–10　地面飞行控制中心（Air Traffic Control）

【问题描述】

为了避免飞机在半空中相撞，大多数的商业飞机都要由地面飞行控制中心来监控，地面飞行控制中心用雷达跟踪监测飞机的位置。在本题中，给出一个飞机集合和一个控制中心集合的信息，请计算哪些飞机是被哪些控制中心所控制的。每个飞机的位置唯一地以一对坐标（x, y）来表示。在本题中，可以忽略飞机的高度（海拔高度）。

由于人员和设备的变化，一个控制中心可以监控的飞机数目经常发生变化。在任何时刻，每个控制中心都会按照如下规则监控尽可能多的飞机：

1）相对于离它较远的飞机，控制中心会监控离它较近的。

2）如果两架飞机与控制中心的距离相同，并且控制中心只能监控其中的一架飞机，它会选择监控在正北方向较远的那架飞机（正 y 轴正方向）。

3）如果两架飞机距离相等且 y 坐标值相同，控制中心会选择监控在正东方向较远的那驾飞机（正 x 轴正方向）。

在任何时候，每个控制中心都有一个圆形的“控制范围”，这一控制范围的半径就是该控制中心与被监控飞机之间的最大距离。所有在控制范围内的飞机都被控制中心监控。在控制范围边界上的飞机被监控与否则取决于控制中心的容量，并根据上面所列出的优先级进行取舍。控制边界上可以没有飞机被控制。

控制中心的具体位置没有提供，但作为替代，可提供每个控制中心正在监控的飞机的数量，

以及控制中心的当前控制范围边界上的两个点。基于这些信息，可以推算出每个控制中心的位置，以及它在监控哪些飞机。如果有多种可能的控制范围，应该选择监控在正北方向较远的那架飞机的方案；如果还有多种方案，则应选择监控在正东方向较远的那驾飞机的方案。

图 1.10-1 给出了四架飞机及两个控制中心的例子。每个控制中心用一个圆形的控制范围以及这个范围的边界上的两个点表示，这两个点分别记为 *A* 和 *B*；*P*1、*P*2、*P*3、*P*4 分别表示四架飞机。在本例中，*P*1 和 *P*4 分别被一个控制中心所监控，*P*3 则被两个控制中心所监控，而 *P*2 没有被任何控制中心监控。

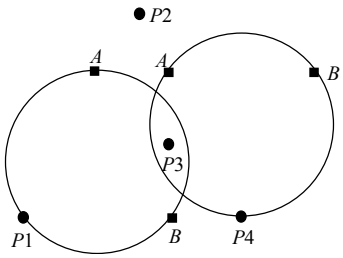

图 1.10-1

输入：

输入包含若干测试用例。每组测试用例的第一行给出两个整数 *NP*（0<*NP*<100）和 *NC*（0<*NC*<10），分别表示飞机的数量以及控制中心的数量。后面 *NP* 行，每行包括一对浮点数，表示某架飞机的（*x*,*y*）坐标。接下来的 *NC* 行，每行描述了一个控制中心，包括一个在 0 至 *NP*（含）范围内的整数表示所监控的飞机数量，以及两个浮点数对，表示边界上的两个点的坐标（*x*, *y*）（均不会与飞机位置重叠）。如果两个距离误差在 0.000 001 内，就可以认为二者相等。

最后一组测试用例后，以两个整数 0 结束。

输出：

对于每组测试用例，分别计算被 0 个、1 个、……、*NC* 个控制中心所监控的飞机数量。输出数据包含测试用例编号和一个有 *NC* + 1 个整数的数字串，其中第 *i* 个整数表示被 *i*–1 个控制中心所控制的飞机数量。如果输入的数据是矛盾的，则以“Impossible”替代该数字串输出。具体格式见样例输出，每组测试用例处理以后输出一个空行。

样例输入	样例输出
4 2 3.0 0.0 0.0 0.0 1.6 2.8 2.0 1.0 2 1.0 2.0 2.0 0.0 2 2.0 2.0 4.0 2.0 2 1 0.0 0.5 0.0 –0.5 0 –1.0 0.0 1.0 0.0 0 0	Trial 1: 1 2 1 Trial 2: Impossible

试题解析

本题的题意表述得比较模糊，有些必要条件没有直接给出，因此需要重新简述一下题意：有若干飞机和若干控制中心，问飞机被控制中心所控制的情况。

首先定义飞机优先级：飞机 A 的优先级大于飞机 B，等价于飞机 A 的 y 坐标大于飞机 B 的 y 坐标；或二者 y 坐标相同且 A 的 x 坐标大于 B 的 x 坐标。

当控制范围确定时，被控制的飞机有：范围内的全部飞机，范围边界上的优先级最高的若干架飞机。

当有若干个控制范围备选时，优先选择半径较小的（这一条件在题目中并未明文给出）；半径相同时，选择两个控制范围中优先级最高的飞机所在的范围；若仍相同，则选择次高的飞机所在的范围，以此类推。

其他细节条件在题目中描述得较清晰，这里不再赘述。下面讨论解题过程。

首先，不难发现不同控制中心之间是不会互相干涉的，因此可以单独处理每个控制中心。对于某个控制中心，已知其控制范围上的两个点，又已知该控制范围通过某架飞机。由于 3 点可以确定一个圆，因此只要枚举该控制范围通过哪架飞机，加上给定的点，即可求出控制范围。具体过程为：

1）枚举控制范围所过的飞机 X。

2）与给定的 A、B 共同确定控制范围。

3）根据上述优先级，确定该控制范围控制哪些飞机。

4）确定所控飞机后，与目前为止的最优解比较，根据上述条件确定选择哪个方案。

5）返回步骤 1），继续枚举。

在确定了每个控制中心分别控制了哪些飞机以后，分别统计每架飞机被几个中心所控制，输出答案即可，这里不再赘述。以下详细分析步骤 2）～4）。

1. 分析步骤 2）

给出 3 个点，确定一个圆，其算法如下：

假设 3 个点分别是 A、B、C。由数学知识可知，这个圆的圆心就是线段 AB、BC、AC 的垂直平分线的交点。因此，可以分为两个步骤：

1）求线段 AB、AC 的垂直平分线的直线方程。

2）根据两条直线的方程求交点。

求线段 AB 的垂直平分线方程，过程如下：

假设其中点为 P，有 $P=\frac{A+B}{2}$。两直线垂直，斜率乘积为–1，可得斜率 $k'=-\frac{1}{k}$（其中 k 为线段 AB 的斜率，$k=\frac{A.y-B.y}{A.x-B.x}$）。已知某线段过点 P 且斜率为 k'，则有直线方程：$y=k'*(x-P.x)+P.y$，即 $k'*x-y+(P.y-k'*P.x)=0$。将 k' 和 P 分别以 A 和 B 代入，即可得到其直线方程。为了避免除法运算，可以将整个式子乘以其分母。最终可得其直线方程为 $Ax+By+C=0$，其中 $A=B.x-A.x$，$B=B.y-A.y$，$C=(A.y^2-B.y^2+A.x^2-B.x^2)/2$。

同理可得 AC 的垂直平分线方程。

有两条直线的方程 $A_1x+B_1y+C_1=0$、$A_2x+B_2y+C_2=0$，求其交点，有多种方法：解二元一次方程组、求行列式等。最终可得其交点坐标：

$$x=\frac{B_2*C_1-B_1*C_2}{B_1*A_2-B_2*A_1};\quad y=\frac{C_2*A_1-C_1*A_2}{B_1*A_2-B_2*A_1}$$

2. 分析步骤 3）

在已知控制范围的前提下，通过下述方法求出被控制的飞机：

1）将所有飞机按照优先级排序。

2）通过枚举判断有几架飞机在控制范围内，有几架飞机在边界上。以 *in* 记录范围内的飞机，*on* 记录边界上的飞机，*m* 记录该控制中心当前控制几架飞机。根据 *in*、*on* 的值，可以很容易地判断控制范围的合法性。

3）若该控制范围合法，则计算具体有哪些飞机被控制：

首先计算边界上真实的飞机数 $on'=m-in$，然后按照优先级从高到低的顺序枚举所有的飞机：若该飞机在控制范围内，则被覆盖；若该飞机在控制边界上，且边界上还有飞机余量，则该飞机被控制，且余量减一；否则该飞机不被覆盖。

这样就可以得到有哪些飞机被该控制中心所控制。

3. 分析步骤 4）

比较两个控制范围，按照下述方法选择较优的那个。

首先比较半径，选择半径较小的那个。

若半径相同，则按照优先级从高到低枚举所有飞机：若两个控制范围均覆盖或均不覆盖当前飞机，则比较下一个；否则，若这架飞机被 X 控制但不被 Y 控制，则说明范围 X 较优，应选择范围 X 作为答案；同理，若这架飞机被 Y 控制但不被 X 控制，则选择 Y。

为了便于理解，可以打个比方：假设以一个长为 *NP* 的 01 串表示被控飞机，1 表示覆盖，0 表示不覆盖，则上述比较相当于比较两个 01 串的字典序。

本题对于算法的要求并不高，主要难点在于计算三点确定的圆以及两直线交点，接下来只要理清题意，按照题目要求和计算步骤按部就班地写代码即可。需要注意的是，本题的输出格式有一些奇特的地方：冒号后面应该是两个空格；每个数字之间应该有两个空格；在最后一个数字以后输出两个空格；Impossible 后没有空格。否则，返回的结果是“Presentation Error”而不是“Accepted”。

程序清单

```
#include <stdio.h>
#include <algorithm>
#include <cstring>
#include <math.h>
using namespace std;
const double eps=1e-5;
struct coor                                    //点坐标
{
    double x, y;
};
double r;
int C, NP, NC, m, i, j, k, ok;                 //飞机数为 NP，控制中心数为 NC，合法标志为 ok
int covers[100], ans[10], cov[100], Tc[100]; //covers[i]记录飞机 i 被几个控制中心所
                                               //控制；ans[i]记录被第 i 个控制中心所控
                                               //制的飞机有几架；cov 记录对于某个控制中
                                               //心，当前的最佳控制范围；Tc 记录对于某个
                                               //控制范围，飞机的覆盖情况
```

```
coor A, B;                                  //坐标点
coor P[100];                                //飞机的坐标序列
coor operator -(coor a, coor b)             //将 a 点平移至 b 点
{
    a.x -=b.x;
    a.y -=b.y;
    return a;
}
bool cmp(coor a, coor b)                    //比较 a 点和 b 点的优先级
{
    return a.y>b.y+eps || (a.y>b.y-eps && a.x>b.x);
          //y 坐标较大的优先级较高；y 相同时，x 较大的优先级较高。这里要注意精度误差
}

double len(coor a)                          //计算 a 点至原点的距离
{
    return sqrt(a.x*a.x+a.y*a.y);
}

bool cross(coor A,coor B,coor C,coor &O)    //已知 3 个点 A，B，C，求确定的圆的圆心 O
{
    double A1, B1, C1, A2, B2, C2, tmp;
    A1=B.x-A.x;
    B1=B.y-A.y;
    C1=(A.y*A.y-B.y*B.y+A.x*A.x-B.x*B.x)/2.0; //求 AB 的垂直平分线方程
    A2=C.x-A.x;
    B2=C.y-A.y;
    C2=(A.y*A.y-C.y*C.y+A.x*A.x-C.x*C.x)/2.0; //求 AC 的垂直平分线方程
    tmp = B1*A2-B2*A1;
    if (fabs(tmp)<eps)
    return false;                           //若两直线平行或重合，无交点
    O.y=(C2*A1-C1*A2)/tmp;                  //求出交点坐标，返回结果
    O.x=(C1*B2-C2*B1)/tmp;
    return true;
}
bool check()//比较目前为止的最优解 cov 和当前控制范围 Tc，若 Tc 优于 cov 则返回 true，否则返回 false
{
    int i;
    for(i=0; i<NP; i++)
    {
        if(cov[i]&&!Tc[i])
        return false;
        if(!cov[i]&&Tc[i])
        return true;
    }
    return true;
}
void work(coor A,coor B,coor C)     //根据边界坐标 A、B 和飞机坐标 C 计算最佳控制范围
{
    coor O;
    double Tr;
    int i, in, on;
    if(!cross(A,B,C,O))
    return;                         //若 A、B、C 无法确定控制范围，则返回；否则计算圆心 O
    Tr=len(A-O);                    //计算半径
```

```
    in=0; on=0;
    for(i=0; i<NP; i++)                    //计算控制范围内的飞机数 in，边界上的飞机数 on
        if(len(O-P[i])<Tr-eps) in++;
        else if(len(O-P[i])<Tr+eps) on++;
    if(in>m || in+on<m) return;            //若非法则返回
    on=m-in;                               //计算边界上的飞机数的余量
    for(i=0; i<NP; i++)                    //标记每架飞机是否被控制
        if(len(O-P[i])<Tr-eps) Tc[i]=1;
            else if(len(O-P[i])<Tr+eps&&on){ Tc[i]=1; on--; }
                else Tc[i]=0;
    if(r<0||Tr<r-eps || (Tr<r+eps && check()))
                                           //与目前的最优解比较，判断是否更新。注意精度误差
    {
        r=Tr;
        for(i=0; i<NP; i++) cov[i]=Tc[i];
        }
    }
    int main()                             //主算法
    {
        while(scanf("%d %d", &NP, &NC) && NP)
                                           //反复输入飞机数和控制中心数，直至输入两个 0 为止
    {
        for(i=0;i<NP;i++)
        scanf("%lf %lf",&P[i].x,&P[i].y);      //输入飞机坐标
        sort(P, P+NP, cmp);                    //按照优先级排序
        memset(covers, 0, sizeof(covers));
        ok=1;
        for(i=0; i<NC; i++)
        {
            scanf("%d %lf %lf %lf %lf", &m, &A.x, &A.y, &B.x, &B.y);
                                               //输入控制的飞机数及边界坐标
            r=-1;
            for(j=0;j<NP;j++)
            work(A, B, P[j]);                  //枚举每架飞机，求最佳控制范围
            for(j=0; j<NP; j++)
            covers[j]+=cov[j];                 //累计每架飞机的覆盖数
            if(r<0)
            ok=0;                              //若对于某个控制中心，所有的控制范围均不合法，
                                               //记该数据非法
        }
        memset(ans, 0, sizeof(ans));
        for(i=0; i<NP; i++)
        ans[covers[i]]++;                      //计算被 i 个控制中心所覆盖的飞机数 ans[i]
            printf("Trial %d:  ", ++C);        //输出结果。注意空格、空行等输出格式
        if(ok)
            for(i=0;i<=NC;i++)
        printf("%d  ",ans[i]);
        else printf("Impossible");
        printf("\n\n");
    }
}
```