第 2 章

2005 ACM-ICPC 世界总决赛试题解析

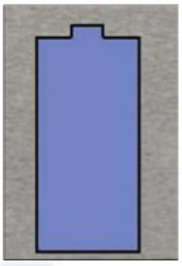
试题 2–1　眼球弯曲（Eyeball Benders）

【问题描述】

“眼球弯曲（Eyeball Benders）”是一个很流行的解谜游戏，它的规则是玩家通过物体的一个部分的特写视图来识别出这个常见的物体。举例来说，一幅规则排列的彩色圆锥体的图像可能是一个打开的新粉笔盒的一部分。图 2.1-1 给出一个例子，其中图(2)为谜题，图(1)为解答。

解答（一张软盘）

(1)

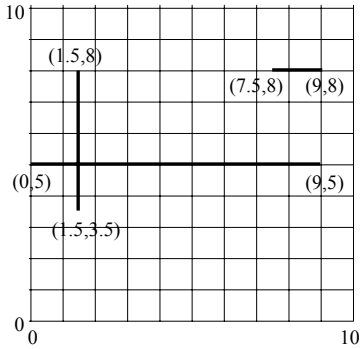


一个简单的眼球弯曲的谜题

(2)

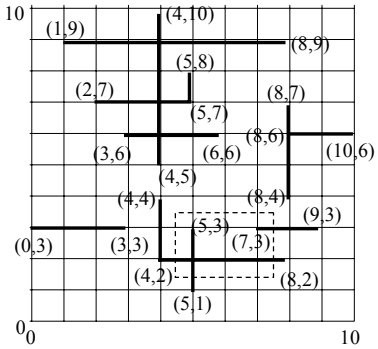
图　2.1-1

请验证“眼球弯曲”谜题图简化版的解答图。给出若干对图片，每张图片均包含一个线段的集合。所有的线段都是水平或竖直的，并且包括线段的端点。图 2.1-2 给出了一个实例。



解答：右图中虚线所框出的区域放大三倍的效果

(1)



谜题

(2)

图　2.1-2

对于给出的两张图片，请判断是否能构成合法的一对，即第一张图片是否是第二张图片某

些部分的放大版本。在两张图中线段的厚度是 0。对于一对合法的谜题图和解答图，至少有一个谜题图中的线段端点也是解答图中的线段端点。

在一张图中，坐标是相对的位置坐标，并有一定的比例。一张图与另一张图可以不使用相同的原点或比例。谜题图相对于解答图的放大倍数大于或等于 1，即谜题图是解答图的放大版。对于图 2.1-2，所编程序要判断这是一对合法的谜题/解答图片。

输入：

本题包含多组测试用例。每组测试用例的开头为两个正整数 M 和 N（$1 \leqslant M$，$N \leqslant 50$），M 为谜题图中的线段数，N 为待判断的解答图中的线段数；后面的行给出 $M+N$ 对端点。前 M 对端点是谜题图中线段的端点，剩余的 N 对端点是解答图中的线段的端点。每对 x 坐标和 y 坐标均满足$-100 \leqslant x$，$y \leqslant 100$，并且小数点以后至多保留三位。所有的输入数据用空白字符隔开（空格或换行符）。

任何两个不同的点之间的距离不会小于 0.005（按照图片中的比例），并且所有的线段长度至少为 0.005。任何两条垂直线段均不相交，任何两条水平线段也均不相交。但是垂直线段和水平线段有可能交叉，并且交点既可以是线段的端点也可以是在线段内部。

最后一组测试用例后跟着一行，包含两个整数 0 0。

输出：

对于每组测试用例，若解答图匹配谜题图中的一个封闭的矩形子区域（包括至少一个端点），且通过不小于 1 倍的放大以及一定的平移能与谜题图相匹配，并且未出现在谜题图中的线段距离该矩形区域至少有 0.005，则输出测试用例编号数（1，2，…）以及“valid puzzle”。

如果匹配条件未能成立，则输出“impossible”。具体格式参见样例输出。

样例输入	样例输出
3 12 9 8 7.5 8 1.5 8 1.5 3.5 0 5 9 5 4 2 8 2 5 7 2 7 10 6 8 6 8 7 8 4 1 9 8 9 9 3 7 3 4 10 4 5 4 2 4 4 5 8 5 7 3 6 6 6 0 3 3 3 5 1 5 3 4 12 −50 −5 50 −5 0 10 0 −10 50 5 −50 5 −50 0 50 0 4 2 8 2 5 7 2 7 10 6 8 6 8 7 8 4 1 9 8 9 9 3 7 3 4 10 4 5 4 2 4 4 5 8 5 7 3 6 6 6 0 3 3 3 5 1 5 3 0 0	Case 1: valid puzzle Case 2: impossible

试题解析

本题的主要难点在于，谜题经过了放大以及平移，无法直接与解答图比较。

对于平移问题，题目中多次强调谜题中至少有一条线段的端点也是解答中的端点，因此只要枚举谜题中的每个端点分别与解答中的哪个端点对应，就可以得到谜题的平移量。

接下来讨论放大倍数的问题，分以下两种情况：

1. 谜题中仅包含一条线段

题目要求解答中的矩形区域距离其他线段至少为 0.005，因此，在同样包含某一条线段的情况下，矩形越小越容易满足这一条件。显然，可以直接假设对应的解答中的线段长度为最短的 0.005，使得矩形最小化。此时的比例为线段长度 L/0.005。

2. 谜题中包含多条线段

首先将谜题中的线段标号为 0 至 n–1，并且将被固定的端点所在的线段标号为 0。由于谜题与解答在平移、缩放后是一一对应的，因此若解答合法且固定端点正确，那么对于谜题中的任意一条线段，在解答图中都有其对应的线段。枚举谜题中标号为 1 的线段所匹配的解答中的那条线段，得出谜题中固定端点到线段 1 的距离 L_1、解答中固定端点到枚举的线段的距离 L_2，由此计算出缩放比例 L_2/L_1。

有了固定点以及缩放比例以后，就可以将谜题还原到解答图中的相应位置，比较矩形区域内部是否重合、矩形区域外部的线段距离矩形是否有 0.005 的距离，就可以知道解答图是否满足谜题的要求。

枚举所有的固定点和匹配线段：若存在一个合法的方案满足题目条件，则说明解答图是正确的；否则说明无法在满足题目要求的情况下使二者匹配。

其他细节见程序清单。

程序清单

```
#include <stdio.h>
#include <math.h>
struct line                          //线段的结构类型
{
    double x1, y1, x2, y2;
};
struct point                         //坐标的结构类型
{
    double x, y;
};
const double delta = 1e-6;           //精度
bool ok;                             //成功标志
int T, n, m, i, j, k;                //测试用例编号为 T，小图和大图的线段数分别为 m 和 n
double scale;                        //记录谜题到解答的缩小倍数
point pL, pS;                        //记录大图（描述解答图片）上的固定端点以及小图（描述谜题图片）
                                     //上的固定端点
line tL;                             //记录小图上的线段 1 对应的大图上的线段
line L[60], S[60], SN[60];           //L 记录大图上的线段，S 记录小图上的线段，SN 记录小图在经过按
                                     //比例缩小、平移后的线段
line in()                            //输入线段，这里统一处理为 x1≤x2，y1≤y2，方便编程
{
    double tmp;
    line t;
    scanf("%lf %lf %lf %lf", &t.x1, &t.y1, &t.x2, &t.y2);
    if (t.x1>t.x2+delta)
    {
        tmp = t.x1; t.x1 = t.x2; t.x2 = tmp;
    }
    if (t.y1>t.y2+delta)
```

```
    {
        tmp = t.y1; t.y1 = t.y2; t.y2 = tmp;
    }
    return t;
}
bool same(double x, double y)  //比较浮点数 x 和 y 是否相等，应处理精度误差
{
    return (y-delta<x && x<y+delta);
}
bool findS(line p)             //判断小图中的线段 p 是否为大图的子线段
{
    int i;
for (i=0; i<n; i++)            //枚举大图中的每条线段
    {                          //若小图中的线段 p 为大图中的线段 i 的一部分，则返回 true
        if (same(p.x1, p.x2) && same(L[i].x1, L[i].x2) && same(p.x1, L[i].x1))
            if (L[i].y1<=p.y1 && p.y2<=L[i].y2) return true;
        if (same(p.y1, p.y2) && same(L[i].y1, L[i].y2) && same(p.y1, L[i].y1))
            if (L[i].x1<=p.x1 && p.x2<=L[i].x2) return true;
    }
    return false;              //小图中的线段 p 非大图的子线段
}
bool findL(line p)             //判断大图中的线段 p 是否为小图中的子线段，与 findS 函数类似
{
    int i;
    for (i=0; i<m; i++)        //枚举小图中的每条线段
    {                          //若大图中的线段 p 为小图中线段 i 的一部分，则返回 true
        if (same(p.x1, p.x2) && same(SN[i].x1, SN[i].x2) && same(p.x1, SN[i].x1))
            if (SN[i].y1<=p.y1 && p.y2<=SN[i].y2) return true;
        if (same(p.y1, p.y2) && same(SN[i].y1, SN[i].y2) && same(p.y1, SN[i].y1))
            if (SN[i].x1<=p.x1 && p.x2<=SN[i].x2) return true;
    }
    return false;              //大图中的线段 p 非小图的子线段
}
double dis(double x, double y) //计算（x,y）与原点的距离
{
    return sqrt(x * x + y * y);
}
bool check()                   //检查在当前比例及固定端点的情况下大图与小图是否匹配
{
    double MaxX, MaxY, MinX, MinY;
    int i, j, k;
    MaxX=MinX=pL.x;            //记录矩形区域在 X 方向的最大值和最小值
    MaxY=MinY=pL.y;            //记录矩形区域在 Y 方向的最大值和最小值
    for (i=0; i<m; i++)        //枚举小图的每条线段，计算缩放、平移后的坐标，并调整矩形区域在
                               //两个方向上的最大值和最小值
    {
        SN[i].x1 = (S[i].x1 - pS.x) * scale + pL.x; SN[i].y1 = (S[i].y1 - pS.y) * scale + pL.y;
        SN[i].x2 = (S[i].x2 - pS.x) * scale + pL.x; SN[i].y2 = (S[i].y2 - pS.y) * scale + pL.y;
        if (!findS(SN[i])) return false; //判断小图中的线段是否在大图中出现过
        if (SN[i].x1<MinX) MinX = SN[i].x1; if (SN[i].x2>MaxX) MaxX = SN[i].x2;
        if (SN[i].y1<MinY) MinY = SN[i].y1; if (SN[i].y2>MaxY) MaxY = SN[i].y2;
    }
    for (i=0; i<n; i++)        //枚举大图的每条线段
    {                          //以下分别讨论线段是水平的或竖直的、线段与矩形区域的位置关系，
```

```
                        //来计算矩形外线段到矩形区域是否不小于 0.005，以及矩形内线段
                        //是否在小图中有匹配线段。由于细节处理比较多且代码简单，这里
                        //不再一一注释
if (same(L[i].x1, L[i].x2))//若大图的线段 i 是竖直线段
{
    if (L[i].x1>MaxX)    //若大图的线段 i 严格在矩形区域右方或左方，且该线段到矩形区域
                         //的距离小于 0.005，则返回 false
    {
        if (L[i].y1>MaxY)
        {
            if (dis(L[i].x1-MaxX, L[i].y1-MaxY)<0.005) return false;
        } else if (L[i].y2<MinY)
        {
            if (dis(L[i].x1-MaxX, L[i].y2-MinY)<0.005) return false;
        } else if (L[i].x1-MaxX<0.005) return false;
    } else if (L[i].x2<MinX)
    {
        if (L[i].y1>MaxY)
        {
            if (dis(L[i].x2-MinX, L[i].y1-MaxY)<0.005) return false;
        } else if (L[i].y2<MinY)
        {
            if (dis(L[i].x2-MinX, L[i].y2-MinY)<0.005) return false;
        } else if (MinX-L[i].x2<0.005) return false;
    } else {
        if (L[i].y2<MinY-0.005 || L[i].y1>MaxY+0.005) continue;
        if (L[i].y2<MinY || L[i].y1>MaxY) return false;
        tL = L[i];          //截取大图的线段 i 在矩形区域内的部分。若与小图不匹配，则返回 false
        if (tL.y1<MinY) tL.y1 = MinY; if (tL.y2>MaxY) tL.y2 = MaxY;
        if (!findL(tL)) return false;
    }
} else {                 //大图的线段 i 是水平线段
    if (L[i].y1>MaxY)    //若大图的线段 i 严格在矩形区域上方或下方，且该线段到矩形区域
                         //的距离小于 0.005，则返回 false
    {
        if (L[i].x1>MaxX)
        {
            if (dis(L[i].x1-MaxX, L[i].y1-MaxY)<0.005) return false;
        } else if (L[i].x2<MinX)
        {
            if (dis(L[i].x2-MinX, L[i].y1-MaxY)<0.005) return false;
        } else if (L[i].y1-MaxY<0.005) return false;
    } else if (L[i].y2<MinY)
    {
        if (L[i].x1>MaxX)
        {
            if (dis(L[i].x1-MaxX, L[i].y2-MinY)<0.005) return false;
        } else if (L[i].x2<MinX)
        {
            if (dis(L[i].x2-MinX, L[i].y2-MinY)<0.005) return false;
        } else if (MinY-L[i].y2<0.005) return false;
    } else {
        if (L[i].x2<MinX-0.005 || L[i].x1>MaxX+0.005) continue;
        if (L[i].x2<MinX || L[i].x1>MaxX) return false;
```

```
                tL = L[i];    //截取大图的线段 i 在矩形区域内的部分。若与小图不匹配，则返回 false
                if (tL.x1<MinX) tL.x1 = MinX;
                if (tL.x2>MaxX) tL.x2 = MaxX;
                if (!findL(tL)) return false;
            }
        }
    }
    return true;                        //大图的所有线段与小图匹配
}
void work()                             //判断是否为一对合法的谜题/解答图片
{
    int k;
    if (m==1)                           //若小图仅一条线段
    {
        scale = 0.005 / (S[0].x2 - S[0].x1 + S[0].y2 - S[0].y1);  //计算此时的缩放比例
        ok = scale<=1 && check();       //返回是否成功（缩放比例不大于 1 且与大图匹配）的标志
    } else for (k=0; k<n; k++)          //枚举大图中所有与小图线段 1 匹配的线段
        if ((L[k].x1-L[k].x2)*(S[1].x1-S[1].x2) || (L[k].y1-L[k].y2)*(S[1].y1-S[1].y2))
        {
          if (S[1].x1!=S[1].x2)         //处理该线段水平时的情况
          {
            if (same(S[1].y1, pS.y)) //若固定端点与线段在同一水平线，则计算 X 方向的距离
            {
                if (!same(S[1].x1, pS.x)) scale = (L[k].x1 - pL.x) / (S[1].x1 - pS.x);
                                        //若与线段左端点不重合，则按照与左端点的距离计算比例；在
                                        //与左端点重合的情况下，按照与右端点的距离计算比例
                else scale = (pL.x - L[k].x2) / (pS.x - S[1].x2);
            } else scale = (L[k].y1 - pL.y) / (S[1].y1 - pS.y);
                                        //若固定端点与线段不在同一水平线，则直接以 Y 方向的距离
                                        //计算比例
        } else {                        //处理该线段竖直时的情况
            if (same(S[1].x1, pS.x)) //若固定端点与线段在同一条垂直线上，则分端点按照 Y 方向的
                                        //距离计算比例；否则按照 X 方向的距离计算比例
                        {
              if (!same(S[1].y1, pS.y)) scale = (L[k].y1 - pL.y) / (S[1].y1 - pS.y);
                else scale = (pL.y - L[k].y2) / (pS.y - S[1].y2);
            } else scale = (L[k].x1 - pL.x) / (S[1].x1 - pS.x);
        }
        ok = scale<=1 && scale>delta && check();
                                        //计算是否成功（缩放比例为不大于 1 的正整数且与大图匹配）
                                        //的标志。若缩放比例为 0 或负数，则可能导致小图在翻转或缩
                                        //小为一个点后与大图匹配，因此应保证缩放比例为正
        if (ok) return;                 //返回大小图是否匹配成功的标志
    }
}
int main()                              //主算法
{
    T = 0;                              //测试用例编号初始化
    while (scanf("%d %d", &m, &n) && (m || n))
                                        //反复输入小图和大图的线段数，直至输入两个 0 为止
    {
        for (i=0; i<m; i++) S[i] = in();      //输入小图的每条线段并进行初步处理
        for (i=0; i<n; i++) L[i] = in();      //输入大图的每条线段并进行初步处理
        ok = false;                           //成功标志初始化
        for (i=0; i<n; i++)
        for (j=0; j<m; j++)
        if (!ok)                              //枚举固定端点所在的线段
        {
            tL = S[0]; S[0] = S[j]; S[j] = tL; //强制将小图中的线段 j 标记为 0，方便后续处理
```

```
                pL.x = L[i].x1; pL.y = L[i].y1;    //假设匹配端点是左（下）端点
                pS.x = S[0].x1; pS.y = S[0].y1;
                work();                            //判断是否为一对合法的谜题/解答图片
                if (ok) break;                     //若匹配成功，退出循环
                pL.x = L[i].x2; pL.y = L[i].y2;    //假设匹配端点是右（上）端点
                pS.x = S[0].x2; pS.y = S[0].y2;
                work();                            //判断是否为一对合法的谜题/解答图片
                if (ok) break;                     //若匹配成功，退出循环
            }
            if (ok)                                //输出是否存在一对合法的谜题/解答图片的信息
                printf("Case %d: valid puzzle\n\n", ++T);
            else
            printf("Case %d: impossible\n\n", ++T);
        }
    }
```

试题 2–2　GSM 网络的简化模型（Simplified GSM Network）

【问题描述】

近 10 年来，手机戏剧性地改变了人们的生活方式。手机以多种协议互相联系，全球移动通信系统（Global System for Mobile Communication，GSM）网络是其中最流行的手机网络之一。

在一个典型的 GSM 网络中，一部手机和最近的收发基站（Base Transceiver Station，BTS）相连。一个基站中心（Base Station Center，BSC）控制若干个 BTS。一个移动服务交换中心（Mobile Services Switching Center，MSC）控制若干个 BSC，并且这个 MSC 和一些其他的 MSC、公共交换电信网（Public Switched Telecom Network，PSTN）和综合业务数字网（Integrated Services Digital Network，ISDN）保持连接。

在本题中，我们用常规的 GSM 网络的简化模型。我们的简化网络由至多 50 个 BTS 塔组成。在网络运作中，手机通常与离它最近的 BTS 相连。被一个 BTS 塔所覆盖的区域称为一个单元。当一个运行状态中的手机移动并穿越单元边界时，它必须无缝地从一个 BTS 转换到另一个。给出一个由城市、道路和 BTS 塔组成的地图，请计算从一个城市到另一个城市至少需要转换几次 BTS？

在图 2.2-1 中，城市以方框来表示，BTS 塔以梯形来表示，直线表示道路。虚线划分了 9 个不同的单元。从城市 1 到城市 6 所需的最小的转换次数为（2+1+0）=3。注意城市 7 是孤立的，无法到达。

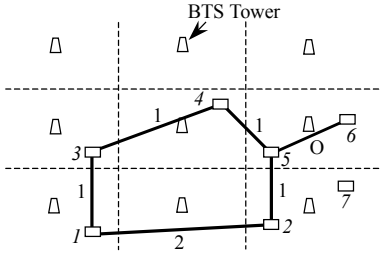


图　2.2-1

可以认为每个塔和城市的坐标是一个二维笛卡尔坐标系的一个点。如果在两个城市之间有一条路，可以认为它是一条连接这两个城市的直线线段。举个例子，在图中，从城市 1 至城市 2 之间的路需要穿越两个单元边界，因此需要两次转换。从城市 2 到城市 5 需要穿过一个单元边界，从城市 5 到城市 6 不需要转换。按照这个路线从城市 1 走到城市 6 一共需要 3 次转换。注意任何一条从城市 1 到城市 6 的路径需要的转换次数都不小于 3。如果两个城市之间有多条路径，那么所编程序就要找到最佳路线。

输入：

输入文件中包含多组测试用例。每组测试用例的第一行是 4 个整数：BTS 塔的数量 B（$1 \leqslant B \leqslant 50$），城市的数量 C（$1 \leqslant C \leqslant 50$），路的数量 R（$0 \leqslant R \leqslant 250$），询问的数量 Q（$1 \leqslant Q \leqslant 10$）。

然后给出 B 行，每行两个浮点数 x 和 y，表示一个 BTS 塔的笛卡尔坐标。接下来的 C 行，每行包括两个浮点数 x_i 和 y_i，表示第 i 个（$1 \leqslant i \leqslant C$）城市的笛卡尔坐标。接下来的 R 行，每行两个整数 m 和 n（$1 \leqslant m, n \leqslant C$），表示在第 m 个城市和第 n 个城市之间有一条路。最后给出 Q 行，每行两个整数 s 和 d（$1 \leqslant s, d \leqslant C$），表示出发城市和目的地城市。

保证所有坐标的绝对值在 1000 以内，没有两个塔在同一个位置，没有两个城市在同一个位置，也没有城市在单元的边界上。没有一条路与单元边界重叠，或包含多个（三个或以上）单元的交界处的点。

输入以一行 4 个 0 结束。

输出：

对于每组测试用例，输出 Q+1 行，如下表所示。第一行为测试用例编号数，接下来的 Q 行，每行对应一个询问，每个询问包含一个整数 r，表示从城市 s 到城市 d 所需的最少的转换次数。如果城市 s 和城市 d 不可达，输出“Impossible”。

样例输入	样例输出
9 7 6 2 5 5 15 5 25 5 5 15 15 15 25 15 5 25 15 25 25 25 8 2 22 3 8 12 18 18 22 12 28 16 28 8 1 2 1 3 2 5 3 4 4 5 5 6 1 6 1 7 0 0 0 0	Case 1: 3 Impossible

试题解析

在本题中，如果能把每条路所需要的转换次数求出来，则可以将该题转化为一个简单的最短路径问题了。其中，图的顶点是城市，图的边是路径，边的长度就是经过这条路径所需的转

换次数。由于仅有 50 个点，因此可以直接采用 floyd 算法计算最短路径。以下仅说明怎样求每条路径所需的转换次数。

首先可以尝试对任意的一组 BTS 塔，画出每个单元。如图 2.2-2 给出了 6 个点的 BTS 塔所划分出的单元。

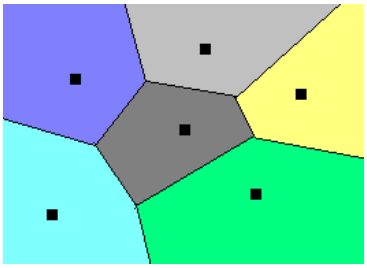


图　2.2-2

可以发现，划分出的单元都是凸的。事实上不可能存在凹的单元，因为画单元时是用两个塔之间的垂直平分线来作为边界线，此时这条线的左边与右边分别属于两个塔，也就是说，对于某个区域，任何区域内两点之间的连线也在这个区域内，而这正是凸多边形的性质之一。计算转换次数的时候也可以从这一性质入手。

下面，给出计算城市 A 到城市 B 之间的转换次数的方法：

首先枚举城市 A 和城市 B 分别所在的单元。如果在同一个单元，那么说明整条线段均在这个单元中，转换次数为 0。否则二分，把一条线段分成两条较短的线段，进行分治处理。若某条线段的长度小于某个很小的长度，我们可以认为这是一个点，并且是一个分界点。伪代码如下（其中 area（x, y）返回点（x, y）所在的单元，dis（$x1, y1, x2, y2$）返回点（$x1, y1$）和点（$x2, y2$）之间的距离）：

```
int get_switch(double x1,double y1,double x2,double y2)
{
   if (area(x1, y1)==area(x2, y2)) return 0;
   if (dis(x1, y1, x2, y2)<1e-6) return 1;
   return get_switch(x1, y1, (x1+x2)/2, (y1+y2)/2) + get_switch((x1+x2)/2, (y1+y2)/2,
x2, y2);
}
```

其他细节见程序清单。

程序清单

```
#include <stdio.h>
#include <math.h>
#include <memory>
struct coor                          //坐标的结构类型
{
   double x,y;
};
const int MaxB = 50;                 //单元（BTS 塔）数的上限
const int MaxC = 50;                 //城市数的上限
const int MaxS = 50 * 50;            //由于从城市 A 到城市 B 至多经过 49 条路，每条路至多穿过 49 个单
                                     //元，因此如果可以到达的话，答案小于 MaxS
int B,C,R,Q,i,j,k,cases;             //BTS 塔数为 B、城市数为 C、路径数为 R、询问数为 Q，测试用例编
                                     //号为 cases
coor BTS[MaxB],city[MaxC];           //单元（BTS 塔）序列和城市序列
int S[MaxC][MaxC];                   //最短路矩阵
double dis(double x1,double y1,double x2,double y2)  //计算线段(x1,y1)(x2,y2)的长度
{
   return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}
int area(double x,double y)                          //计算距离(x,y)最近的单元编号
{
```

```
    int i,k;
    double min,tmp;
    min = dis(x,y,单元[0].x,单元[0].y);        //初始时，最小距离为第 0 个单元与(x,y)的距离
    k = 0;                                     //最近的单元编号初始化
    for (i=1;i<B;i++)                          //枚举每个单元塔
    {
       tmp = dis(x,y,单元[i].x,单元[i].y);     //根据第 i 个单元与(x,y)的距离调整 min 和 k
       if (tmp<min) { min = tmp; k = i; }
    }
    return k;                                  //返回最近的单元编号
    }
       int get_switch(double x1,double y1,double x2,double y2)
    //采用二分法计算(x1,y1)至(x2,y2)的路径上经过的单元数：若距离(x1,y1)和(x2,y2)最近的单元为同一
    //单元，则返回 0；若(x1,y1)和(x2,y2)重合，则返回 1；否则分别递归左半路径和右半路径，并累加两条路
    //径经过的单元数
    {
       if (area(x1,y1)==area(x2,y2)) return 0;
       if (dis(x1,y1,x2,y2)<1e-6) return 1;
       return get_switch(x1,y1,(x1+x2) / 2,(y1+y2)/2)+get_switch((x1+x2) / 2,(y1+y2)/2,
x2,y2);
    }
    bool init()                                //输入 GSM 网络并初始化
    {
       scanf("%d %d %d %d",&B,&C,&R,&Q);       //反复输入 BTS 塔数、城市数、路径数、询问数，直至输
                                               //入 4 个 0 为止
       if (B==0 && C==0 && R==0 && Q==0) return false;
       for (i=0;i<B;i++) scanf("%lf %lf",&BTS[i].x,&BTS[i].y);    //输入每个 BTS 塔的坐标
       for (i=0;i<C;i++) scanf("%lf %lf",&city[i].x,&city[i].y); //输入每个城市的坐标
       memset(S,0,sizeof(S));
       for (i=0;i<R;i++)                       //输入每条路径的两个端点城市
       {
          scanf("%d %d",&j,&k); j --; k --; //由于标号从 0 到 C-1，因此需要-1
          S[j][k] = S[k][j] = 1;
       }
       for (i=0;i<C;i++)
       for (j=i+1;j<C;j++)                     //计算从城市 i 到城市 j 之间的路（如果存在的话）穿过
                                               //几个单元，若 i 和 j 之间没路，则赋为最大值
       {
       if (S[i][j]==0) S[i][j] = MaxS;
       else S[i][j] = get_switch(city[i].x,city[i].y,city[j].x,city[j].y);
       S[j][i] = S[i][j];
       }
       for (i=0;i<C;i++) S[i][i] = 0;          //往返于城市 i 的回路穿过的单元数为 0
       return true;
    }
    int main()
    {
       cases = 0;                              //测试用例编号初始化
       while (init())                          //反复输入 GSM 网络并初始化，直至输入 4 个 0 为止
       {
          printf("Case %d:\n",++cases);        //输出测试用例编号
          for (k=0;k<C;k++)          //采用 floyd 算法计算任一对城市间的最短路径
            for (i=0;i<C;i++)
              for (j=0;j<C;j++)
```

```
            if (S[i][j]>S[i][k]+S[k][j]) S[i][j] = S[i][k] + S[k][j];
        for (i=0;i<Q;i++)          //顺序输入每个询问，根据最短路径矩阵 S 回答一对城市间的最短路
                                   //径，或指出最短路径不存在
        {
            scanf("%d %d",&j,&k); j --; k --;
            if (S[j][k]>=MaxS) printf("Impossible\n"); else printf("%d\n",S[j][k]);
        }
    }
}
```

试题 2-3　裁判员的旅行问题（The Traveling Judges Problem）

【问题描述】

一群裁判要一起去裁判一场在一个特定的城市举行的比赛，他们需要计算出一个把他们送到比赛场地去的最便宜的租车方案。他们发现如果一些裁判在路上全程或部分地共用一辆车可能便宜些，因为那样可以降低总成本。请确定那些裁判要走的路线，使得总租金最少。

本题有如下设定：

- 租车的费用与车的行进距离成正比。多人乘坐、燃料、保险或异地下车并不收费。
- 所有被租的车辆每英里所收的费用是相同的。
- 被租的车辆可以载任意多的乘客。
- 任何两个城市之间至多有一条路直接相连。每一条路都是双向的，长度为一个大于 0 的整数。
- 从每个裁判出发城市到比赛举办城市都有至少一条行进路线。
- 所有路过某个城市或来自这一城市的裁判一起从该城市去比赛场（一个裁判可以搭一辆车到一个城市，并搭另一辆车离开该城市）。

输入：

输入包含若干组测试用例。每组测试包括一个路线地图、要举办比赛的目的地城市和开始时裁判所在的城市。

每组测试用例以一串被空格和（或）换行所隔开的整数组成。这些整数的顺序和意义如下：

- *NC*——路线图中出现的城市数，不超过 20。
- *DC*——目标城市的编号，假设城市从 1 至 *NC* 标号。
- *NR*——路线图中道路的数量。每条路连接一对不同的城市。
- 对于每条路，以 3 个整数 *C*1、*C*2 和 *DIST* 来描述。*C*1 和 *C*2 为被这条路所连起来的两个城市的编号，*DIST* 为路的长度。
- *NJ*——裁判的数量，不超过 10。
- *NJ* 个整数，每个整数对应一个裁判，表示该裁判最初所在城市的编号。

最后一组测试用例后跟着一个整数-1，表示输入结束。

输出：

对于每组测试用例，输出测试用例编号（1、2、…）和运送裁判到比赛场地的所有被租车辆的最短行驶距离。然后输出裁判们所走的路线，每条路线占一行，每一条路线以裁判所在的城市开始，到比赛的举办地结束，按照访问的先后顺序输出相应的裁判必须经过的城市。路线上的其他的城市应按照裁判访问的顺序输出。城市之间以“-”隔开，在每条路线前输出 3 个空格。

如果有多条距离最短的路线，则输出经过的城市最少的那条路线；如果还有相同的，输出以城市编号排列时字典序最小的那个（例如，取{2，3，6}优先于{2，10，5}）；如果还有多条可行路线，输出任何一组满足条件的。

参考样例输出的格式。

样例输入	样例输出
5 3 5 1 2 1 2 3 2 3 4 3 4 5 1 2 4 2 2 5 1 4 4 3 1 3 1 2 3 2 3 4 2 2 1 2 3 3 3 1 2 2 1 3 3 2 3 1 2 2 1 -1	Case 1: distance = 6 5-4-2-3 1-2-3 Case 2: distance = 5 1-3-4 2-3-4 Case 3: distance = 3 2-3 1-2-3

试题解析

对本题来说，如果把最优情况下所有裁判的路线画出来，就应该构成一棵树。因为一旦出现环，那么总可以删去其中一段。若某裁判需要通过那一段，则从剩下的链中“绕”过去，可使得总路程缩短。因此在已知经过了哪些城市的情况下，可以用最小生成树把这些城市连接起来。

这样，一个蛮力但有效的算法浮出水面：枚举所有经过了的城市，然后用最小生成树计算其花费。总的时间复杂度为 $O(2^{NC}*NC^2)$，而实际上，由于目标城市和有裁判出现的城市是必须“经过”的，因此实际花费的时间要少得多，是完全可以接受的。

程序清单

```
#include <stdio.h>
#include <memory>
const int MaxC = 20;            //节点数的上限
const int MaxJ = 10;            //起始点数的上限
```

```
int cases,NC,NJ,T,DC,map,best_map,ans,i,k;
//节点数为 NC，目标节点数为 DC，起始点数为 NJ，T 记录哪些城市必须经过（包括所有裁判的起点和比赛举办地），
//map 记录选取了哪些城市，最佳方案为 best_map，运送裁判到比赛场地的所有被租车辆的最短行驶距离为 ans
int road[MaxC][MaxC],judges[MaxJ],fa[MaxC];
//相邻矩阵为 road[][]，起始点序列为 judges[]，生成树中每个节点的父指针为 fa[]
bool init()                          //由于城市的标号为 0～NC-1，因此读入的城市标号均要-1
{
    int NR,i,C1,C2;
    scanf("%d",&NC);                 //输入节点数
    if (NC==-1) return false;        //返回输入结束标志
    scanf("%d",&DC); DC --;          //输入目标节点
    T = 1 << DC;                     //T 为 2^DC，即第 DC 二进制位设 1，其余为 0
    scanf("%d",&NR);                 //输入边数
    memset(road,0,sizeof(road));
    ans = 1000;                      //ans 的初值为所有边长的和加 1000
    for (i=0;i<NR;i++)
    {
        scanf("%d %d",&C1,&C2); C1 --; C2 --;//输入连接第 i 条边的两个节点及其边长
        scanf("%d",&road[C1][C2]);
        road[C2][C1] = road[C1][C2];          //无向图的相邻矩阵对称
        ans += road[C1][C2];                  //第 i 条边的边长累计入总边长
    }
    scanf("%d",&NJ);                          //输入起始点的数目
    for (i=0;i<NJ;i++)
    {
        scanf("%d",&judges[i]); judges[i] --;//输入第 i 个起始点的编号
        T = 1 << judges[i];                   //将 T 对应的二进制位设 1
    }
    return true;
}
void check(int map) //被选取的节点组成二进制数 map，对包含这些节点的子图计算一次最小生成树
{
    int i,j,k,sum,min;
                                              -2  节点i与生成树无边连接
    int d[MaxC];                        //d[i]= -1  节点i进入生成树
                                              >0  节点i与生成树的最短边长（距离值）
    for (i=0;i<MaxC;i++) d[i] = -2;       //初始时，所有节点未在生成树
    d[DC] = -1;                           //拓展目标节点
    for (i=0;i<MaxC;i++)                  //枚举每个已选取且与 DC 有边的节点，该节点的距离值设为边长，
                                          //父节点设为 DC
    if (((1<<i)&map) && road[DC][i]) { d[i] = road[DC][i]; fa[i] = DC; }
    sum = 0;
    while (1)                             //使用 prime 算法计算最小生成树
    {
        min = ans + 10;                   //最小距离值初始化
          for (i=0;i<MaxC;i++)            //计算生成树外距离值最小的节点 k
            if (d[i]>0 && d[i]<min) { min = d[i]; k = i; }
          if (min>ans) break;             //若没有节点可扩展，则退出循环
            sum += min; d[k] = -1;//k 进入最小生成树，距离值累计入最小生成树的边权和
            for (i=0;i<MaxC;i++) //若节点 i 被选取，且与节点 k 有边，且未计算过与生成树的距离值
                                 //或(k,i)的边长小于节点 i 的距离值，则将节点 i 的距离值调整为
                                 //(k,i)的边长，i 节点的父指针设为 k
              if ((1<<i)&map && road[k][i])
```

```
            if (d[i]==-2 || d[i]>road[k][i]) { d[i] = road[k][i]; fa[i] = k; }
        }
        for (i=0;i<MaxC;i++)
        if (((1<<i)&map) && d[i]==-2) return; //若选取的节点无法扩展，则说明是一个非法方案
        if(sum<ans)    //若最小生成树的边权和小于 ans，则边权和调整为 ans，被选取的节点记为最佳方案；
                       //若最小生成树的边权和等于 ans，则计算目前最优方案选取的节点数 j 和当前方案
                       //选取的节点数 k。若 j 大于 k，则当前选取的节点记为最优方案
        { ans = sum; best_map = map; }
        else if (sum==ans)
        {
          j = 0; k = 0;
          for (i=0;i<MaxC;i++)
          {
            if ((1<<i)&best_map) j++;
            if ((1<<i)&map) k++;
          }
          if (j>k) { best_map = map; return; }
          if (j<k) return; //按照编号递增的顺序枚举每个节点：若节点 i 在最优方案中被选取但未被当前
                           //方案选取，则退出；若节点 i 未被最优方案选取但在当前方案中被选取，则当前
                           //方案调整为最优方案并退出
          for (i=0;i<MaxC;i++)
          {
            if (((1<<i)&best_map) && !((1<<i)&map)) return;
            if (!((1<<i)&best_map) && ((1<<i)&map)) { best_map = map; return; }
          }
        }
    }
    int main()                          //主算法
    {
        cases = 0;                      //测试用例编号初始化
        while (init())                  //反复输入测试用例，直至输入 0 为止
        {
          best_map = 0;                 //最优方案初始化
          for (map=0;map<(1<<NC);map++) //枚举节点被选取的所有可能情况
            if ((map&T)==T) check(map); //若这一选取情况包括了所有应该有的节点，则计算一次最小生成树
        printf("Case %d: distance = %d\n",++cases,ans);
                                        //输出运送裁判到比赛场地的所有被租用车辆的最短行驶距离
        check(best_map); //对最佳方案中包含选取节点的剩图再计算一次最小生成树，以得出路径方案
        for (i=0;i<NJ;i++)              //枚举每个裁判
        {
            printf("  %d",judges[i] + 1);//输出裁判序号
            k = judges[i];              //从该裁判出发，沿父指针向上追溯一条至目标节点 DC（最小
                                        //生成树的根）的路径，顺序输出路径上的节点
            while (k!=DC)
            {
                printf("-%d",fa[k] + 1);
                k = fa[k];
            }
            printf("\n");
          }
          printf("\n");
        }
    }
```

试题 2–4　纸牌戏法（cNteSahruPfefrlefe）

【问题描述】

Preston Digitation 是一名专攻纸牌戏法的魔术师。Preston 有一件无法做好的事，就是完美洗牌。一次完美洗牌，就是把 52 张纸牌分为两半，然后完美地交叉到一起，这样，原来的下半部分的第一张纸牌在洗牌后位于最上方。如果我们把纸牌从 0（顶部的牌）到 51（底部的牌）标号，一次完美洗牌后我们会看到如下结果：

26　0　27　1　28　2　29　3　30　4　31　5　32　6　…　51　25

Preston 发现他在一次洗牌中至多犯一次错误。例如，纸牌 2 和纸牌 28 可能交换了，这样在一次洗牌以后会看到如下结果：

26　0　27　1　2　28　29　3　30　4　31　5　32　6　…　51　25

这两张相邻纸牌的交换就是 Preston 所犯的唯一错误。当一次洗牌以后，他可以很容易地发现他是否犯了错，以及在哪里犯了错，但当洗过很多次牌以后判断是否犯错就变得很难了。请编写一个程序来找到他的错误（如果存在的话）。

输入：

输入包含多个测试用例。第一行是一个整数，表示有多少个测试用例。每个测试用例一行，是这副纸牌在洗了 1 至 10 次牌以后的排列顺序。所有纸牌的数量都是 52。

注意：对于每个测试用例，下面的样例输入数据中均包含若干行，但实际的输入数据仅包含一行。

输出：

对于每个测试用例，输出测试用例的编号数（从 1 开始），然后输出洗牌的次数。如果在洗牌过程中没有出错，则输出一行“No error in any shuffle”；否则按如下格式输出若干行：“Error in shuffle *n* at location *m*”，其中 *n* 表示发生错误的洗牌次数编号，*m* 表示错误发生的位置。洗牌次数从 1 开始，发生的位置以交换了的两张牌中靠前的那张牌为准（这副牌最上方为位置 0）。在下面的例子中，位置 4 和 5 的卡片（纸牌的编号为 2 和 28）是错误的，因此此时 *m* 为 4。按 *n* 递增的顺序输出所有的错误。如果有一些洗牌中没有发生错误，不要输出它们的情况。如果有多组答案，输出错误最少的那种（所有的测试用例中均包含唯一的最小解）。

样例输入	样例输出
3	Case 1
26 0 27 1 2 28 29 3 30 4 31 5 32 6 33 7 34	Number of shuffles = 1
8 35 9 36 10 37 11 38 12 39 13 40 14 41 15	Error in shuffle 1 at location 4
42 16 43 17 44 18 45 19 46 20 47 21 48 22	
49 23 50 24 51 25	Case 2
26 0 27 1 28 2 29 3 30 4 31 5 32 6 33 7 34	Number of shuffles = 1
8 35 9 36 10 37 11 38 12 39 13 40 14 41 15	No error in any shuffle
42 16 43 17 44 18 45 19 46 20 47 21 48 22	
49 23 50 24 51 25	Case 3
49 26 43 40 37 34 31 28 25 22 19 16 13 10	Number of shuffles = 9
7 4 1 51 48 45 42 39 36 33 24 27 30 21 18	Error in shuffle 3 at location 3
15 12 9 6 3 0 50 47 44 41 38 35 32 29 46	Error in shuffle 7 at location 11
23 20 17 2 11 8 5 14	Error in shuffle 8 at location 38

试题解析

如果洗牌完全不出错，那么得到的扑克牌序列记为 *a*。同样次数的有错误的洗牌得到的序列记为 *b*。定义一个差异函数 *d* 来衡量 *a* 和 *b* 的差异。差异函数 *d* 的值表示序列 *b* 的元素至少需要交换多少次才能得到序列 *a*，每次交换的两个元素不要求相邻。

洗牌的次数是很好求出来的。我们假设每次洗牌都不出错，每洗一次牌都求一下当前的扑克牌序列与给出的扑克牌序列的差异函数。如果差异函数值不大于洗牌次数，则说明给出的扑克牌序列的洗牌次数就是当前统计到的洗牌次数。这样做的前提是 10 次洗牌得出来的序列中的任何两个序列都不相似。实际上这 10 个序列中相同位置上的元素两两不相等，即任何两个序列的差异函数都达到了最大值 51。这就保证了任何一个序列在 20 次交换之内是不能得到另一个序列的，所以可以采取前面的简单做法求洗牌的次数。

剩下的问题，就是要找出发生错误的洗牌次数和出错的位置。一种可行的做法是从给出的序列开始，枚举出错的位置，把相邻两张牌的位置交换，然后反向执行一次洗牌操作，再次枚举出错的位置并交换，继续反向执行一次洗牌操作……重复这个过程，直到把扑克牌还原成初始序列。每次反向操作后都比较一下当前的序列和正确的序列的差异函数值，如果大于剩余的洗牌次数，则说明继续枚举下去是无法还原成初始序列的，停止枚举并返回上一步操作。这样就使得错误枚举的层数不超过 5 层，可以在规定的时间限制内找到最佳答案。

实现过程和具体细节见程序清单。

程序清单

```
#include <cstdio>
#include <memory>
#include <math.h>
int go[52],correct[11][52],card[52],next[52],err[11],ans[11],n,i,j,dfs,errors,best,
cases,s; //元素的去向为 go[]，每次洗牌的正确序列为 correct[][]，当前洗牌序列为 card[52]，下次洗牌的
      //序列为 next[52]，第 i 次洗牌发生错误的位置为 err[]，最后得出结果为 ans[]，差异函数值为 dfs，
      //当前错误总次数为 errors，最少错误总次数为 best，测试用例编号为 cases，测试用例数为 s
void anti_shuffle(int n)                    //反向洗牌
{
   int tmp[52],i;
   for(i=0;i<52;i++)
      tmp[correct[n][i]]=next[i];           //把反向洗牌后的序列存到临时数组
   memcpy(next,tmp,sizeof(tmp));            //更新序列
}
int differents(int *a, int *b)              //求序列 a 和序列 b 的差异函数值
{
   int i,j,s;
   int step[52],go[52];
   memset(step,0,sizeof(step));             //初始化遍历标记
   for(i=0;i<52;i++)
      go[a[i]]=b[i];                        //元素 a[i]的去向
   s=0;                                     //初始化差异函数值
   for(i=0;i<52;i++)                        //逐个元素检查
      if(!step[i])                          //若元素 i 未遍历，则设元素 i 为遍历的起点
      {
         j=i;
         while(!step[j])                    //若未回到起点，则继续遍历
         {
```

```
                step[j]=1;                          //标记遍历过的元素
                j=go[j];                            //遍历下一个元素
                s++;                                //统计循环长度
            }
            s--;                                    //差异函数值的增量为循环长度减 1
        }
    return s;                                       //返回 a 和 b 的差异函数值
}
void trytrytry(int i)                               //以递归的方式枚举每一次洗牌的错误位置
{
    int j,save[52];
    dfs=differents(card,correct[i]);                //求当前的扑克牌序列和正确的序列的差异函数值
    if(dfs>i)return;                                //若差异函数值比剩余的洗牌次数大，则停止递归
    if(!i)                                          //若递归完毕
    {
        if(errors<best)                             //若错误总次数最少，则更新最佳答案
        {
            best=errors;                            //记下错误次数
            memcpy(ans,err,sizeof(err));            //记下当前错误位置
        }
        return;
    }
    memcpy(save,card,sizeof(card));                 //把当前的扑克牌序列保留一份
    err[i]=-1;                                      //错误位置为-1，表示这次洗牌没有出错
    for(j=0;j<51;j++)                               //逐一枚举错误位置
    {
        memcpy(next,card,sizeof(card));             //初始化新序列
        next[j]+=next[j+1];                         //交换相邻两张扑克牌
        next[j+1]=next[j]-next[j+1];
        next[j]-=next[j+1];
        dfs=differents(next,correct[i]);            //求新序列和正确的序列的差异
        anti_shuffle(1);                            //反向洗牌一次
        memcpy(card,next,sizeof(card));             //更新当前序列
        errors++;                                   //累计一次错误次数
        err[i]=j;                                   //错误位置为 j
        if(errors+dfs<best) trytrytry(i-1);         //当错误次数有可能比最佳答案好的时候进入下一层递归
        errors--;                                   //递归完毕，还原错误次数、错误位置和序列
        err[i]=-1;
        memcpy(card,save,sizeof(card));
    }
    //本次洗牌没有出错
    memcpy(next,card,sizeof(card));                 //初始化新序列
    anti_shuffle(1);                                //反向洗牌一次
    memcpy(card,next,sizeof(card));                 //更新当前序列
    trytrytry(i-1);                                 //进入下一层递归
    memcpy(card,save,sizeof(card));                 //递归完毕，还原序列
}
int main()
{
    for(i=0;i<52;i++)                               //初始化序列
    {
        go[i]=26*(1-i%2)+i/2;                       //第 i 张牌洗牌后的去向
        correct[0][i]=i;                            //初始的扑克牌序列
```

```
    }
    for(i=1;i<11;i++)                          //正确地洗 10 次牌，把结果记录到 correct 数组中
        for(j=0;j<52;j++)
            correct[i][j]=correct[i-1][go[j]];
    scanf("%d",&s);                            //读测试用例数
    for(cases=1;cases<=s;cases++)
    {
        for(i=0;i<52;i++)
            scanf("%d",&card[i]);              //读这副纸牌洗了 1 至 10 次后的排列顺序
        for(i=0;i<11;i++)                      //和每个正确的洗牌序列比较
        {
            dfs=differents(card,correct[i]);                    //求差异函数值
            if(dfs<=i) break;                  //若差异函数值不大于洗牌次数，说明给出的扑克牌
                                               //序列的洗牌次数就是当前的洗牌次数
        }
        printf("Case %d\nNumber of shuffles = %d\n",cases,i);  //输出测试用例编号
        if(!dfs)                //若给出的序列和正确序列没有差异，则输出洗牌过程中未出错信息
        {
            puts("No error in any shuffle\n");
            continue;           //继续测试下一组数据
        }
        n=i;                    //洗牌次数记为 n
        best=dfs+1;             //初始化最小错误次数
        errors=0;               //初始化累计错误次数
        trytrytry(i);           //对于每一次洗牌，枚举错误位置
        for(i=1;i<=n;i++)
            if(ans[i]>-1)       //第 i 次洗牌发生错误，输出错误发生的位置
               printf("Error in shuffle %d at location %d\n",i,ans[i]);
        puts("");
    }
    return 0;
}
```

试题 2-5　阳光普照（Lots of Sunlight）

【问题描述】

住宅施工管理部门（Apartment Construction Management，ACM）在上海的郊区有一些新的高层公寓。由于经济景气，ACM 预期公寓租赁有可观的利润。由于这些公寓可以受到更多的阳光直射，所以公司声称他们的公寓比在其他地方的公寓要好，没有其他的建筑物遮挡 ACM 的高层建筑的阳光。

ACM 希望通过告诉他们的潜在客户某一房间确切能照射到多少阳光来证实他们的说法。为了向客户提供一个日照时间的典型例子，公司希望在广告上说明 2005 年 4 月 6 日的日照时间。在那一天的上海，太阳在早上 5:37 升起，下午 6:17 落下。

如图 2.5-1 所示，公寓处于一系列自东向西排列的建筑物之中。公寓的最后两位数字标识了它所在的建筑物，自东向西从 01 开始编号。其他数字表示了公寓的层数，第一层标识为 1。

太阳从东方升起，在天空中以恒定速率通过，直到在西方落下。所有的阴影都是由建筑物造成的，也就是说，每栋建筑物可能在一栋或若干栋建筑物上投下阴影。当某间公寓的东墙或西墙完全被阳光覆盖，或太阳在它的正上方的时候，就认为这间公寓能接受到阳光。

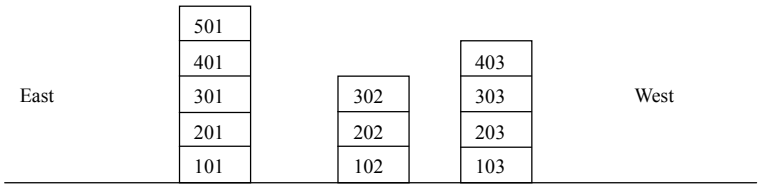


图　2.5-1

输入：

输入包含一系列公寓住宅群的描述。每组描述的第一行是一个整数 n（$1 \leqslant n < 100$），表示住宅群中公寓楼栋数。下一行给出两个整数 w（公寓的从东到西方向的宽度）和 h（每间公寓的高度），单位为米；接下来有一列整数 m（1），d（1），m（2），d（2），…，d（n−1），m（n），其中 m（i）表示建筑物 i 中有多少间公寓，d（i）表示建筑物 i 和建筑物 i+1 之间的距离，以米为单位。

在每组住宅群的描述后，给出一列公寓的整数，表示查询该间公寓的日照时间，以 0 结束。以仅包括一个整数 0 的一行作为输入结束。

输出：

对于每组住宅群的描述，按描述的序列先输出它的编号。接下来对于每个询问，以 24 小时制的格式输出相应的日照时间，把所有的时间四舍五入至最近的秒。如果询问了一间不存在的公寓，就指出该公寓是不存在的。具体格式见输出样例。

样例输入	样例输出
3 6 4 5 6 3 3 4 302 401 601 303 0 4 5 3 4 5 7 8 5 4 3 101 302 503 0 0	Apartment Complex: 1 Apartment 302: 10:04:50 - 13:23:47 Apartment 401: 05:37:00 - 17:13:57 Apartment 601: Does not exist Apartment 303: 09:21:19 - 18:17:00 Apartment Complex: 2 Apartment 101: 05:37:00 - 12:53:32 Apartment 302: 09:08:55 - 14:52:47 Apartment 503: 09:01:12 - 18:17:00

试题解析

对于任意一个给定的房间，以图 2.5-1 中第 2 栋楼的 202 室为例，在太阳位于正上方的时候肯定可以完全被阳光照射。在这之前有一个刚开始被照射的最小角度，最小角度取决于这栋楼左边的楼房对太阳光的遮挡。对左边的楼房一栋一栋地检查，以右上角的顶点为基准，连一条线到所求房间的左下角，求这条线的倾斜角。倾斜角最大的线就是使得所求房间刚开始被太阳

光照射到的最小角度。同理，在这之后有一个刚结束被照射的最大角度，最大角度取决于右边的楼房对太阳光的遮挡。同样对右边的楼房一栋一栋地检查，以左上角的顶点为基准，连一条线到所求房间的右下角，求这条线的倾斜角。倾斜角最小的线就是使得所求房间刚结束被太阳光照射到的最大角度，如图 2.5-2 所示。最后将这两个角度转化成时间并输出。

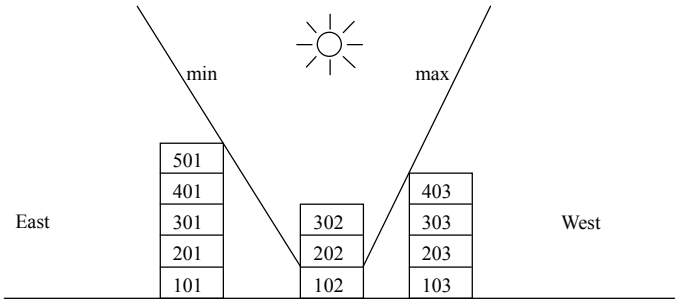


图 2.5-2

为了计算方便，角度应该使用弧度制。在这道题中，从点（x_1, y_1）照射到（x_2, y_2）的照射角度为：

$$a=\text{acos}\left(\frac{x_2-x_1}{dis}\right)$$

其中 *dis* 为两个点的距离。

由于太阳是按照弧度匀速行进的，所以照射角度 *a* 对应的时间为：

$$T_a=s+\frac{a}{\pi}*(d-u)$$

其中 *u* 是太阳从地平线升起的时间，*d* 是太阳从地平线落下的时间。

其他具体细节见程序清单。

程序清单

```
#include <cstdio>
#include <math.h>
int n,h[111],d[111],width,height,sum,i,di,num,level,id,cases,dh,dw,h1,m1,h2,m2;
         //各栋楼的楼层数为 h[]、左方水平位置为 d[]，公寓的宽度为 width、高度为 height，询问的
         //公寓编号为 num，楼层为 level，楼的编号为 id，日照时间为 h1：m1～h2：m2(小时和分)，
double min,max,angle, Half_pi; //min 和 max 取下整后分别对应开始时间和结束时间的秒，Half_pi
                               //存储π/2，angle 存储照射角
int main()                     //主算法
{
   Half_pi=acos(0);                          //圆周率π/2 的精确表达式
   while(scanf("%d",&n)&&n)                  //反复读公寓的栋数，直至输入 0 为止
   {
      scanf("%d%d",&width,&height);          //读公寓的宽度和高度
      sum=0;                                 //统计当前楼房到第一栋楼房的水平距离
      for(i=0;i<n;i++)
      {
         scanf("%d",&h[i]);                  //读第 i 栋楼的楼层数
         d[i]=sum;                           //确定第 i 栋楼左边的水平位置
         if(i<n-1)
         {
```

```
            scanf("%d",&di);                    //读第 i 栋楼和第 i+1 栋楼之间的距离
            sum+=di+width;                      //累计水平距离
        }
    }
    printf("Apartment Complex: %d\n",++cases);      //输出测试用例编号
    while(scanf("%d",&num)&&num)                //反复输入查询日照时间的公寓编号，直至输入 0 为止
    {
        printf("Apartment %d: ",num);           //输出查询日照时间的公寓编号
        level=num/100-1;                        //计算该公寓的楼层和楼的编号
        id=num%100-1;
        if(id<0||id>=n||level<0||level>=h[id])
                                                //若询问了一间不存在的公寓，则输出失败信息
        {
            puts("Does not exist");
            continue;                           //继续下一个查询
        }
        min=0;                                  //开始时刻和结束时刻被照射的角度初始化为 0
        max=0;
        for(i=0;i<id;i++)                       //逐一检查左边的楼房
        {
            dw=d[id]-d[i]-width;                //计算水平距离和垂直距离
            dh=height*(h[i]-level);
                if(dh>0)                        //不低于当前房间高度的楼房才有可能遮挡阳光
            {
                angle=asin(dh/sqrt(dh*dh+dw*dw));     //求照射角
                if(angle>min) min=angle;              //与当前的最值比较
            }
        }
        for(i=id+1;i<n;i++)                             //对右边的楼房逐一检查
        {
            dw=d[i]-d[id]-width;                        //计算水平距离和垂直距离
            dh=height*(h[i]-level);
            if(dh>0)                                    //不低于当前房间高度的楼房才有可能遮挡阳光
            {
                angle=asin(dh/sqrt(dh*dh+dw*dw));       //求照射角
                if(angle>max) max=angle;                //与当前的最值比较
            }
        }
        min=20220+22800*min/ Half_pi;           //把开始的照射角换算成时间，以秒为单位
        max=65820-22800*max/ Half_pi;           //把结束的照射角换算成时间，以秒为单位
        h1=int(min/3600);                       //根据开始的秒数求出开始的小时数
        min-=h1*3600;                           //求剩余的秒数
        m1=int(min/60);                         //根据剩余的秒数求出开始的分钟数
        min-=m1*60;                             //求剩余的秒数
        h2=int(max/3600);                       //根据结束的秒数求出结束的小时数
        max-=h2*3600;                           //求剩余的秒数
        m2=int(max/60);                         //根据剩余的秒数求出结束的分钟数
        max-=m2*60;                             //求剩余的秒数
      printf("%02d:%02d:%02d - %02d:%02d:%02d\n",h1,m1,int(min),h2,m2,int(max));
                                                //输出 num 公寓的日照时间
    }
  }
  return 0;
}
```

试题 2-6 交叉的街道（Crossing Streets）

【问题描述】

Peter Longfoot 是 Suburbia 大学的学生。每天早上，Peter 离开家步行去学校。尽管许多其他的学生喜欢开车或骑自行车去学校，但 Peter 更喜欢步行，以尽可能地避免城市中混乱的交通。

然而，因为步行到学校需要横穿街道，Peter 并不能完全避免交通带来的影响。现在，Peter 想知道怎样选择路线能使得横穿街道的次数最少。例如，在图 2.6-1 所示的地图中，街道以水平或垂直的线来描述。为了从家步行到大学，Peter 至少需要两次横穿街道。Peter 不能在两条街道的交叉点上一次横穿两条街道，也不能顺着街道走。

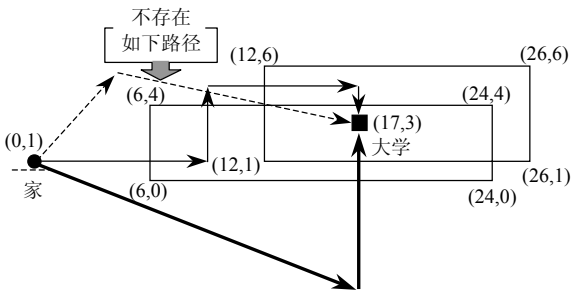


图 2.6-1

在图 2.6-1 中，街道以直线来描述，箭头表示了 Peter 的一个可能的行走路线。黑色箭头展示了一条 Peter 仅需要两次横穿街道的路径。灰色箭头展示了一条 Peter 需要 4 次横穿街道的路径。虚线所示的路径是不合法的，因为它在一个交叉口穿过了两条街道。该图对应第一组样例输入。

Peter 知道城市中所有道路的位置，但他在寻找去大学的最佳路径时遇到了麻烦。所以请编写一个程序来帮助他。

输入：

输入包含若干组城市的描述。每组描述的第一行是一个整数 n（$1 \leqslant n \leqslant 500$），表示城市中街道的数量。接下来的 n 行，每行 4 个整数 x_1，y_1，x_2，y_2，表示一条从坐标（x_1, y_1）到（x_2, y_2）的街道。因为城市是在一些正方形格子上建造的，所有街道均与 X 轴或 Y 轴平行。街道可能覆盖，此时覆盖部分仅认为是一条街道。城市的描述用一行 4 个整数 x_h，y_h，x_u，y_u 结束，坐标（x_h, y_h）表示 Peter 家，坐标（x_u, y_u）表示大学的坐标。Peter 的家和大学都不会在街道上。街道可以被认为是没有宽度的直线线段。尽管端点是整数，Peter 不会沿着与 X 轴或 Y 轴平行的平行线行走，他可以按照任何他想走的方向行走。题目中所有整数均小于 2×10^9。

输入以仅包括一个整数 0 的一行结束。

输出：

对于所有的城市描述，首先按顺序输出描述的编号，然后输出 Peter 从家到学校至少需要横穿几条街道。

具体格式见样例输出。

样例输入	样例输出
8 6 0 24 0 24 0 24 4 24 4 6 4 6 4 6 0 12 1 26 1 26 1 26 6 26 6 12 6 12 6 12 1 0 1 17 3 1 10 10 20 10 1 1 30 30 0	City 1 Peter has to cross 2 streets City 2 Peter has to cross 0 streets

试题解析

本题中可以把街道作为墙，先从 Peter 所在的位置“倒水”进去，作为第 0 层；如果此时能到达终点，说明不必穿越街道；否则把所有隔开第 0 层水的墙拆掉，继续倒满水，作为第 1 层；如果到达目标，则说明要穿越一次街道；否则继续拆墙，以此类推。

但若要保存所有的边界，处理起来是很麻烦的，这里不使用这种方法。

那么，能否用一些代表性的点来表示方格呢？答案是肯定的。

首先，每个 1×1 的方格中的水可以用它的中点来表示，由于坐标是整数，只要某个格子的中间有水，就说明整个格子都是满的。而在穿墙的时候，由于 Peter 不能从角落走过去，而在某面墙的某部位打洞通过与从墙中部打洞通过是等效的，因此也可以通过中点之间的渗透来完成。

此时，就可以通过对点的存储来完成对“水”的“边界”的存储，而且“拆墙”也是很容易实现的。

尽管题目中坐标范围达到 2×10^9，蛮力搜索是不可行的，但我们可以离散化处理，将坐标的数量级缩小至 1000×1000，此时的时间复杂度是完全可以承受的。

实现过程和具体细节见程序清单。

程序清单

```
#include <stdio.h>
#include <algorithm>
using namespace std;
const int MaxS = 500;                    //街道数的上限
const int MaxN = 500 * 2 + 4;            //记录新坐标的最大范围，因为需要多记录上下边界、
                                         //起点和终点，比 2 × MaxS 多 4
const int add[4][2] = {0,1,  0,-1,  -1,0,  1,0};   //上下左右四个方向的坐标增量
int City,xh,yh,xu,yu,n,N,q1,q2,q3,ans,i;
//城市编号为 City，坐标范围为 N，Peter 家的坐标为（xh，yh），大学的坐标为（xu，yu），在 BFS 中本层
//节点的起点为 q1、终点为 q2，下一层节点的终点为 q3，Peter 从家到学校至少横穿的街道数为 ans
int coor[2][MaxS * 2 + 4];               //记录所有出现过的坐标
int w[MaxS * 2 + 4][MaxS * 2 + 4];       //记录某格子是否被水覆盖了
int list[MaxN * MaxN][2];                //按水覆盖的先后顺序记录那些格子
```

```
int map[MaxN][MaxN];                    //用 4 位二进制数记录每格四个方向上是否有墙
int find(int type,int T)                //二分查找 coor[type]中坐标值为 T 的边序号
{
    int l,r,m;
    l = 0; r = n + n + 1;
    while (l!=r)
    {
        m = (l + r) / 2;
        if (coor[type][m]<T) l = m + 1;
        else r = m;
    }
    return l;
}
bool init()
{
    int i,j,k,tmp;
    int streets[MaxS][4];               //记录街道的坐标
    scanf("%d",&n);                     //输入街道数
    if (n==0) return false;             //若输入 0，则退出
    for (i=0;i<n;i++)                   //输入每条街道的原始坐标
    {
        for (j=0;j<4;j++) scanf("%d",&streets[i][j]);
        coor[0][i] = streets[i][0];
        coor[1][i] = streets[i][1];
        coor[0][i + n] = streets[i][2];
        coor[1][i + n] = streets[i][3];
    }
    scanf("%d %d %d %d",&xh,&yh,&xu,&yu);//输入 Peter 家的坐标和大学的坐标
    coor[0][n + n] = xh;
    coor[1][n + n] = yh;
    coor[0][n + n + 1] = xu;
    coor[1][n + n + 1] = yu;
    sort(coor[0],coor[0] + n + n + 2);
    sort(coor[1],coor[1] + n + n + 2);
//求出离散化后的新坐标，由于边上同样需要留出 Peter 行走的空位，全体向右上方移动一格
    for (i=0;i<n;i++)                   //这里保证 x1<x2 或 y1<y2，方便处理
    {
        if (streets[i][0]<streets[i][2])
        {
            tmp = find(0,streets[i][0]) + 1;
            streets[i][2] = find(0,streets[i][2]) + 1;
            streets[i][0] = tmp;
        } else {
            tmp = find(0,streets[i][2]) + 1;
            streets[i][2] = find(0,streets[i][0]) + 1;
            streets[i][0] = tmp;
        }
        if (streets[i][1]<streets[i][3])
        {
            tmp = find(1,streets[i][1]) + 1;
            streets[i][3] = find(1,streets[i][3]) + 1;
            streets[i][1] = tmp;
        } else {
            tmp = find(1,streets[i][3]) + 1;
            streets[i][3] = find(1,streets[i][1]) + 1;
```

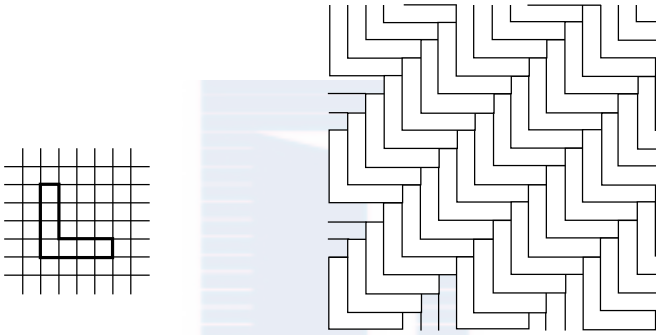
```
            streets[i][1] = tmp;
        }
    }
    xh = find(0,xh) + 1; yh = find(1,yh) + 1;
    xu = find(0,xu) + 1; yu = find(1,yu) + 1;
    N = n + n + 3;                              //实际坐标范围为[0, n+n+3]×[0, n+n+3]
    //把街道画在地图上，以二进制串表示 4 个方向上是否有墙
    memset(map,0,sizeof(map));
    for (i=0;i<n;i++)
        if (streets[i][0]==streets[i][2])     //分别处理竖直方向和水平方向的墙
        {
            for (j=streets[i][1];j<streets[i][3];j++)
            {
                map[streets[i][0]][j] |= 4;
                map[streets[i][0] - 1][j] |= 8;
            }
        } else {
            for (j=streets[i][0];j<streets[i][2];j++)
            {
                map[j][streets[i][1]] |= 2;
                map[j][streets[i][1] - 1] |= 1;
            }
        }
        return true;
}
void go(int x,int y,int t)
{
    int i,h,l;
    for (i=0;i<4;i++) if (!(map[x][y]&(1<<i)) || t)   //尝试往 4 个方向浇水
    {
        h = x + add[i][0]; l = y + add[i][1];
        if (h<0 || l<0 || h>N || l>N || w[h][l]) continue;
        w[h][l] = 1;
        list[q3][0] = h; list[q3][1] = l;
        q3 ++;
        if (map[x][y]&(1<<i)) go(h,l,t - 1);
        else go(h,l,t);
    }
}
int main()
{
    City = 0;
    while (init())
    {
        memset(w,0,sizeof(w)); w[xh][yh] = 1;
        list[0][0] = xh; list[0][1] = yh;
        q3 = 1; go(xh,yh,0);     //从起点开始浇水，此时不能穿墙
        q1 = 0; q2 = q3;         //BFS 中，q1 记录本层结点的起点，q2 记录本层结点的终点，
                                 //q3 记录下一层结点的终点
        ans = 0;
        while (1)
        {
            if (w[xu][yu]) break;
            for (i=q1;i<q2;i++)  //从上一层的每个点开始浇水，此时可以穿一次墙
                go(list[i][0],list[i][1],1);
            ans ++;
```

```
            q1=q2; q2=q3;
        }
    printf("City %d\n",++City);//输出城市编号和 Peter 从家到学校至少横穿的街道数
    printf("Peter has to cross %d streets\n",ans);
    }
}
```

试题 2-7 铺满平面（Tiling the Plane）

【问题描述】

对于一个多边形，如果可以通过使用一个由同样的多边形组成的集合不重不漏地覆盖一个没有限制的二维平面，我们称这个多边形是“铺满平面”的。如图 2.7-1 所示，图(1)给出了一个 L 型多边形，图(2)给出了使用这种多边形平铺出的平面的一部分。请编写一个程序来判断一个给出的多边形是否可以铺满平面。



(1) (2)

图 2.7-1

每组测试用例由一个闭合的多边形组成，这个多边形所有的角均为直角，每一条边的长度均为单位长度的整数倍。可以随意地复制多边形，也可以在平面中平移这些多边形，但不可以旋转或翻转任何一个多边形。

以下信息可能对编程有用：众所周知，只有两种本质不同的铺满平面的情况，用正方形铺满平面（棋盘堆砌）或用正六边形铺满平面（蜂巢堆砌）。一个多边形当且仅当满足下列两个条件之一时可以铺满平面：

1）在多边形的边界线上按顺序存在 4 个点 A，B，C，D（这些点不一定是多边形的顶点），使得从 A 到 B 和从 D 到 C 的多边形的边界线与从 B 到 C 和从 A 到 D 的多边形的边界线恰好一致地接合。这样就存在一种与棋盘堆砌等价的堆砌方案。

2）在多边形的边界线上按顺序存在 6 个点 A，B，C，D，E，F，使得边界线对 AB 和 ED，BC 和 FE，CD 和 AF 恰好一致地接合，这样就存在一种与蜂巢堆砌等价的堆砌方案。

输入：

输入包含若干个多边形的描述，每个描述一行。每个描述以一个整数 n（$4 \leqslant n \leqslant 50$）开始，表示多边形的边数。接下来按逆时针顺序给出了 n 条线段的描述。每条线段的描述为一个字母和一个数字。字母为“N”，“E”，“S”或“W”，表示线段的方向分别为北，东，南，西。数字表示线段的长度为单位长度的多少倍。所描述的多边形不会与其本身相连或相交。

输入以仅包括一个整数 0 的一行结束。

输出：

对于输入中的每个多边形，输出一行。首先输出多边形的编号，接下来如果可以铺满平面则输出“Possible”，否则输出“Impossible”。具体格式见输出样例。

样例输入	样例输出
6 N 3 W 1 S 4 E 4 N 1 W 3 8 E 5 N 1 W 3 N 3 E 2 N 1 W 4 S 5 0	Polygon 1: Possible Polygon 2: Impossible

试题解析

对于此题，并无特别好的方法解决，只能按照题目给出的两种判断规则来判断：如果能够把给出的方向序列分成 4 段或 6 段，其中相对应的序列段互相匹配，则可以铺满平面。

下面以方向序列 SSEEESSEENNWNNNNWWSSWW 为例，分析具体的匹配方法：

先把方向序列分成长度相同的两段 SSEEESSEENN 和 WNNNNWWSSWW，把其中一段的每个元素均改成原来方向的反方向，得到 SSEEESSEENN 和 ESSSSEENNEE；然后再各分成两段或三段，得到 SSE,EESS,EENN 和 ESS,SSEE,NNEE；再把其中一部分的各段倒置，得到 SSE,EESS,EENN 和 SSE,EESS,EENN。此时两部分的各段对应相等，说明匹配成功。图 2.7-2 清晰地描述出上述匹配过程。

剩下的问题就是如何枚举分割点了。由于原始序列展开后可能很长，但一般连续一段序列都是同一个方向，所以用 combo 数组记录从某一个元素开始，后面连续相同的元素个数。采用这样的数据结构枚举，可大大提高匹配的效率。

枚举分割点之前，首先要保证分成的两段中，把其中一段的各个元素反向后，两段中的 4 个方向的数目是相同的；否则这两段无论如何分割都不可能相等，这样就不需要枚举分割点了。另外，分成四段和分成六段的匹配规则是一样的，仅仅只是段数不同，所以无需分开处理，使用同一个枚举过程的不同参数即可。

具体枚举过程和其他细节见程序清单。

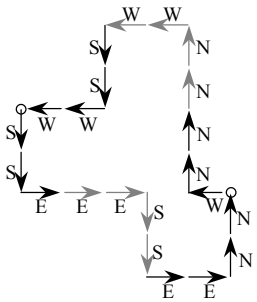


图　2.7-2

程序清单

```
#include <stdio.h>
#include <memory>
char c[55],ch[5555],sequence_a[2777],sequence_b[2777];
//线段的方向序列为 c[]，待匹配的两段序列为 sequence_a[]和 sequence_b[]
int num[55],combo_a[2777],combo_b[2777],total[4],n,i,j,k,len,count,reverse,ok,cases;
//线段的长度序列为 num[]，待匹配的两段序列中当前元素连续重复的次数分别为 combo_a[]和 combo_b[]，
//各方向的频率为 total[]，某元素连续重复的次数为 count，反向存储时 sequence_b 的指针为 reverse，
//有解标志为 ok，测试用例编号为 cases
void tr(int i,int point)                    //分割过程
{
    int j,min,step;
    if(point>=len)                          //如果整段序列均匹配完毕，则可铺满平面
    {
        ok=1;                               //标志有解并退出枚举
        return;
    }
```

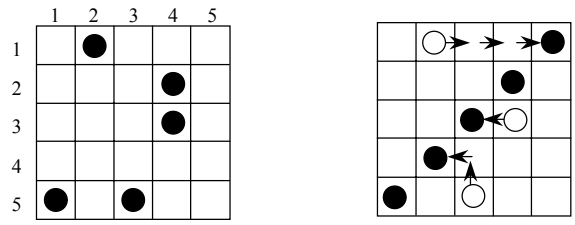
```
    if(i>2)return;                          //如果已经枚举完三段，则退出枚举
    for(j=0;j<len-point;j++)                //枚举下一个分割点
    {
        step=0;                             //初始化当前检测到的位置
        while(sequence_a[point+step]&&sequence_a[point+step]==sequence_b[j+step])
        //检查两段序列是否匹配
        {
            if(combo_a[point+step]>combo_b[j+step]) min=combo_b[j+step];
                                            //以重复次数较少的那部分序列作为下一个检测点
            else min=combo_a[point+step];
            step+=min+1;                    //更新检测位置
        }
        if(j+step>=len-point)tr(i+1,point+step);  //如果两段序列匹配，则枚举下一个分割点
    }
}
int main()
{
    while(scanf("%d",&n)&&n)                        //反复输入多边形的边数，直至输入 0 为止
    {
        memset(ch,0,sizeof(ch));                    //初始化展开的方向序列
        memset(sequence_a,0,sizeof(sequence_a));    //初始化第一部分序列
        memset(sequence_b,0,sizeof(sequence_b));    //初始化第二部分序列
        for(i=0;i<n;i++)                            //输入每条线段的方向和长度
            scanf(" %c %d",&c[i],&num[i]);
        ok=0;                                       //初始化有解标记
        for(i=0;i<n/2;i++)                          //枚举两部分序列的分界点
        {
            len=0;                                  //初始化方向序列展开后的长度
            for(j=0;j<n;j++)                        //把方向序列展开
            {
                count=(i+j)%n;                      //序列偏移量为 i
                for(k=0;k<num[count];k++)ch[len++]=c[count];//把当前的方向重复给定的次数
            }
            count=0;                                //初始化当前元素的连续重复次数
            for(j=0;j<4;j++)total[j]=0;             //初始化各个方向出现的个数
            for(j=len/2-1;j>=0;j--)                 //为第一部分序列赋值并统计
            {
                sequence_a[j]=ch[j];                //前 len/2 个字符是第一部分序列
                if(ch[j]=='N') total[0]++;          //累计“N”的个数
                if(ch[j]=='S') total[1]++;          //累计“S”的个数
                if(ch[j]=='W') total[2]++;          //累计“W”的个数
                if(ch[j]=='E') total[3]++;          //累计“E”的个数
                if(j<len/2-1&&ch[j]==ch[j+1])count++;//累计当前元素的连续重复次数
                else count=0;
                combo_a[j]=count;                   //把当前元素的连续重复次数存入 combo 数组
            }
            count=0;                                //初始化当前元素连续重复次数
            for(j=len/2;j<len;j++)                  //为第二部分序列赋值并统计
            {
                reverse=len-j-1;                    //直接把第二部分序列反向存储
                if(ch[j]=='N')                      //把各方向取反并统计，下同
                {
                    sequence_b[reverse]='S';
                    total[1]--;
                }
                if(ch[j]=='S')
                {
```

```
                sequence_b[reverse]='N';
                total[0]--;
            }
            if(ch[j]=='W')
            {
                sequence_b[reverse]='E';
                total[3]--;
            }
            if(ch[j]=='E')
            {
                sequence_b[reverse]='W';
                total[2]--;
            }
            if(j>len/2&&ch[j]==ch[j-1])count++; //累计当前元素的连续重复次数
            else count=0;
            combo_b[reverse]=count;             //把当前元素的连续重复次数存入 combo 数组
        }
        for(j=0;j<4;j++)                        //判断两个序列的各个方向的个数是否相同
            if(total[j])break;                  //如果不同，退出循环
        if(j>3)                                 //如果各个方向的个数都相同
        {
            len/=2;                             //把长度赋值为第一段的长度
            tr(0,0);                            //枚举分割点
            if(ok) break;                       //找到匹配方案就退出循环
        }
    }
    printf("Polygon %d: ",++cases);             //输出测试用例编号和可否铺满平面的结论
    if(ok) puts("Possible");
    else puts("Impossible");
  }
  return 0;
}
```

试题 2–8　长城游戏（The Great Wall Game）

【问题描述】

Hua 和 Shen 发明了一个简单的单人棋盘游戏，他们取名为“长城游戏（The Great Wall Game）”。这个游戏是用 n 颗石头在一个 $n \times n$ 的棋盘上玩的。石头被随机地摆放在格子内，每个格子至多放置一颗石头。每一个单步，任何一颗石头可以水平或垂直地移动一格，到下一个没有放置其他石头的空格子中。这一游戏的目标是造一堵“墙”，也就是移动最少的步数把 n 颗石头排列成一行、一列或者一条对角线。例如，在图 2.8-1 中，图(1)中给出 n=5 的例子。在图(2)中现实了如何用 6 步把所有的石子排成一条对角线。不可能用更少的步数来完成五子连珠。（还有其他的 6 步完成目标的方法，例如，我们可以用 6 步把所有的石子移动到第三列。）



5×5 的起始棋盘　　6 步完成的方案

(1)　　(2)

图　2.8-1

现在存在一个问题——对于任何一个给出的起始状态，Hua 和 Shen 不知道最小的移动步数是多少。他们希望你能写一个程序帮助他们，对于任何一个给出的起始情况，计算造一堵墙的最少步数。

输入：

输入包含多组测试用例。每组测试用例的第一行是一个整数 n（$1 \leqslant n \leqslant 15$）。下一行给出第一颗石子所在的行号和列号，接下来是第二颗石子的，以此类推。行和列的编号方式如图 2.8-1 所示。输入最后一组测试用例后，给出一行，包含一个数字 0。

输出：

对于每组测试用例，输出测试用例编号（1，2，…）和将 n 颗石子排成直线所需的最小步数。格式参考样例输出。

样例输入	样例输出
5 1 2 2 4 3 4 5 1 5 3 2 1 1 1 2 3 3 1 1 2 2 2 0	Board 1: 6 moves required. Board 2: 0 moves required. Board 3: 1 moves required.

试题解析

首先，我们必须明确一个道理：**石子不重合的条件对排成直线的步数大小并没有影响**。

证明：假设石子 A 将进入石子 B 所在的格子 G。而最终状态下格子 G 中至多有一颗石子，也就是说，石子 A 和 B 中有一颗石子必须离开这个格子。如果石子 B 需要离开，那么可以通过让石子 B 先离开、石子 A 后进入的方式使得它们之间不重合（不可能存在石子 B 要进入石子 A 的格子，而石子 B 要进入石子 A 的格子的情况，因为进行了交换以后棋盘整体并没有改变，那一步一定是无用的）。如果是石子 B 不要离开，那么此时石子 A 必须离开。假设它的目标是格子 H，那么此时可以让石子 B 去格子 H，而把石子 A 的目标变成格子 G。这样对整个棋盘并没有影响，因为我们并不在乎长城中某颗石子原来在哪个位置，而我们可以通过改变行动顺序来在使得结果不变差的情况下满足题目要求的条件。

综上，在解题中，石子不允许重合这一条件是可以忽略的。

如果石子可以任意移动，要怎么求出最小步数呢？

一个比较简单的方法是，首先，需要考虑最终状态是什么（哪一行、哪一列或哪条对角线）。在此情况下，我们的任务就是求出在此情况下，每一颗石子最后所在的位置，使得总移动步数最小。

不妨考虑二分图的匹配形式，以石子原来的 n 个位置为左边的结点，以后来目标状态的 n 个位置为右边的结点，两两之间连边，边的权值为相应的从初始状态到目标状态的距离。那么此时再做一次最佳匹配，就可以知道在此状态下的最小步数了。

程序清单

```
#include <stdio.h>
#include <string>
const int MaxN = 15;
```

```
int n, ans, cases;
int a[MaxN+1][MaxN+1], Lx[MaxN+1], Ly[MaxN+1], matchY[MaxN+1];
                        //二分图的相邻矩阵为 a[][]，可行坐标为 Lx[]和 Ly[]，匹配边为 matchY[]
bool usedX[MaxN+1],usedY[MaxN+1];//X 集合和 y 集合的节点访问标志分别为 usedX[]和 usedY[]
bool path(int r)                    //返回“相等子图”中存在由节点 r 出发的可增广路的标志
{
    int i;
    usedX[r]=true;
    for ( i=0; i<n; ++i )
    if ( Lx[r]+Ly[i]==a[r][i] )
    if ( !usedY[i] )
    {
        usedY[i]=true;
        if ( matchY[i]<0 || path(matchY[i]) ) { matchY[i]=r; return true; }
    }
    return false;
}
void work()                                          //求最佳匹配
{
    int i, j, k, res = 0;
    for ( i=0; i<n; ++i )                            //计算 Lx[]的初始值
    {
        Lx[i]=-2147483647;
        for ( j=0; j<n; ++j )
        if ( a[i][j]>Lx[i] )
        Lx[i]=a[i][j];
    }
    memset(Ly, 0, sizeof(Ly));                       //LY 清零
    memset(matchY, -1, sizeof(matchY));              //匹配边集为空
    for ( i=0; i<n; ++i )
    for ( ; ; )
    {
        memset(usedX, false, sizeof(usedX));         //X 集和 Y 集的所有节点未访问
        memset(usedY, false, sizeof(usedY));
        if ( path(i) ) break;   //若在“相等子图”中存在由 i 出发的可增广路，则退出当前循环
        int delta = 2147483647, v;                   //计算顶标的可改进量 delta
        for ( j=0; j<n; ++j )
        if ( usedX[j] )
            for ( k=0; k<n; ++k )
            if( !usedY[k] )
            {
                v=Lx[j]+Ly[k]-a[j][k];
                if ( v<delta ) delta=v;
            }
        for ( j=0; j<n; ++j )                        //修改顶标
        if ( usedX[j] ) Lx[j]-=delta;
        for ( j=0; j<n; ++j )
        if ( usedY[j] ) Ly[j]+=delta;
    }
    for ( i=0; i<n; ++i )
    if ( matchY[i]>=0 ) res+=a[matchY[i]][i];        //计算匹配边的权和
    if (res>ans) ans = res;                          //调整最佳解
}
int dis(int x,int y)                                 //求点(x,y)到原点的距离
{
    if ( x*y>=0 ) x += y;
```

```
    else x -= y;
    if ( x>0 ) return x;
    else return -x;
}
int main()
{
    int i, j, k;
    int P[MaxN + 1][2];
    while (1)
    {
        scanf("%d", &n);                                    //输入石头数
        if ( n==0 ) break;                                  //若输入 0，则结束
        ans = - (n * n + n);
        for ( i=1; i<=n; i++ ) scanf("%d %d", &P[i][0], &P[i][1]);
        //这里的最佳匹配求的是最大值，因此将所有边权设为负值，最大的负值取绝对值后就变成最小值了
        for ( i=1; i<=n; i++ )                              //计算行情况
        {
          for ( j=1; j<=n; j++ )                            //构造第 i 行横线对应的带权二分图
            for ( k=1; k<=n; k++ )
            a[j - 1][k - 1] = - dis(P[j][0] - i,P[j][1] - k);
           work();                                          //计算最佳匹配并调整最佳解
        }
        for (i=1;i<=n;i++)                                  //计算列情况
        {
            for ( j=1; j<=n; j++ )
            for ( k=1; k<=n; k++ )                          //构造第 i 列竖线对应的带权二分图
                a[j - 1][k - 1] = - dis(P[j][0] - k,P[j][1] - i);
            work();                                         //计算最佳匹配并调整最佳解
        }
        for (j=1;j<=n;j++)
        for (k=1;k<=n;k++)                                  //构造主对角线对应的带权二分图
            a[j - 1][k - 1] = - dis(P[j][0] - k,P[j][1] - k);
        work();                                             //计算最佳匹配并调整最佳解
        for (j=1;j<=n;j++)
        for (k=1;k<=n;k++)                                  //构造对角线对应的带权二分图
            a[j - 1][k - 1] = - dis(P[j][0] - k,P[j][1] - (n - k + 1));
        work();                                             //计算最佳匹配并调整最佳解
        printf("Board %d: %d moves required.\n\n", ++cases, -ans);  //输出最佳解
    }
}
```

试题 2-9　讨论会（Workshops）

【问题描述】

第 1 届加州整体论会议（The first Californian Conference on Holism）于 1979 年在旧金山举行。“加州”这个范围是有一些夸大了的，因为 23 名与会者实际上都住在旧金山。若干年后，在 1987 年，因为 337 名与会者来自加州各地，这个讨论会变成货真价实的“加州会议”。从此以后，参与人数不断增长。1993 年，这一会议更名为美国整体论会议（American Conference on Holism）（2549 参与者），1997 年，会议进行了第二次更名为世界整体论会议（World Conference on Holism），当时世界各地的参会者已经增长至 9973 人。2003 年，在讨论了是否拒绝接纳银河系外的生命以后，这一会议获得了一个新名字——银河整体论会议（Galactic Conference on Holism）并沿用至今。尽管到了下一年度，所有注册参加会议的代表还是地球人——虽然少数与会者坚持说，他们感到有外星人出席会议。

专题讨论的数量和与会者的人数一起增长。为了即将召开的会议，组织者必须实际面对实际的却也是难处理的日程问题。对于 2005 年的会议，管理部门决定同一时刻至多进行 1000 个专题讨论。尽管如此，他们还是需要租他们所能租到的礼堂或教室，而其中的一些仅在特定的时间可以使用。

在第一天的早晨，开幕式在一个足球场举行，下午与会者们要参加专题讨论。在午饭前，每个与会者都要指明在下午他/她想参加的专题讨论。组织者有所有专题讨论的列表，包括每个专题讨论的持续时间和所能容纳的人数。他们也有所有可以使用的房间的清单，包括每个房间必须清场的时刻和能容纳的人数。他们还有可使用的房间的列表，每个房间的容量，以及特定的房间可以使用的时间。通过这些信息，组织人员必须把每个专题讨论安排在一个容量足够大和可用时间足够的房间里。由于这个问题并不是一定有解的，不足的容量可以用足球场中的帐篷代替。这些帐篷足够大，但又热又吵，让人不舒服。所以组织人员希望让那些帐篷里的专题讨论，即不能安排在房间里的专题讨论，最少。如果有多种方案使得帐篷里的专题讨论最少，组织人员希望让那些参与帐篷专题讨论的人数最少。

请给出这样的日程安排（最好在午餐结束前）。

输入：

输入文件包含若干组测试用例。每组测试用例包含两部分：专题讨论列表和被租来的房间列表。

专题讨论列表的第一行为一个整数 w（$0<w\leqslant 1000$），表示专题讨论的数量。接下来 w 行中每行有两个数字，描述了一个专题讨论。第一个数字 p（$0<p\leqslant 100$），表示参与者的人数，第二个整数为持续时间 d（$0<d\leqslant 300$），以分钟为单位表示。为了方便起见，专题讨论的其他细节省略。所有的专题讨论都在 14:00 开始。

被租来的房间列表的第一行是一个整数 r（$0<r\leqslant 1000$），表示房间数。接下来 r 行描述那些房间。每行包括一个整数 s（$0<s\leqslant 100$），表示房间中的座位数，然后是房间可以被使用的时刻，以 hh:mm 的形式来描述，这里 hh 表示小时，mm 表示分钟，以二十四小时制表示。所有房间在 14:00 开始可用。所有可用的时间在 14:01 至 23:59 范围内（包括这两个时刻）。

输入以仅包括一个整数 0 的一行结尾。

输出：

对于每组测试用例，输出应包含测试用例编号、帐篷中的专题讨论的数量和参与帐篷专题讨论的人数。具体格式见样例输出。

样例输入	样例输出
1 20 60 1 30 16:00 2 20 60 50 30 1 30 14:50 0	Trial 1: 0 0 Trial 2: 2 70

试题解析

显然，如果本题中仅有人数或仅有时间的限制，则可以直接采用贪心策略。如果两个限制同时存在，怎么办？我们认为，贪心策略也未尝不可，当然，这里采取的贪心方案并不是“显然”的，因此下面加了一段证明，说明贪心是正确的。证明方式是：任何一个最佳策略，总是通过调整变成贪心策略的，而且效果不会变差。也就是说，贪心策略得出的答案也是“最佳答案”。

首先把房间按持续时长排序，假设第 1 个到 k–1 个房间都处理好了，下面我们尝试处理第 k 个房间。此时可以在所有能装进当前房间的专题讨论里挑一个人数最多的装进去。这样一定是最优的。

证明如下：

假设这个房间为 A，容量为 P，时间为 T，装进里面的专题讨论为 B，容量为 $P1$，时间为 $T1$。因为对于任何一个方案，无外乎三种情况：

1）如果房间 A 是空的，专题讨论 B 的可能情况是：

- 在其他房间里。
- 没有装进去。

此时把专题讨论 B 从原来的地方拎出来放到房间 A 里，至少答案不会变差。

2）房间 A 装了专题讨论 C（容量为 $P2$，时间为 $T2$），而专题讨论 B 在另一个房间 A′中（容量为 P'，时间为 T'）。此时由于贪心的顺序、容纳的关系等，有 $T1$、$T2 \leqslant T \leqslant T'$，$P2 \leqslant P1 \leqslant P$、$P'$。换言之，两个房间的专题讨论是可以互换的，答案不会变差。

3）如果房间 A 装了专题讨论 C，专题讨论 B 没有在任何房间中，那么可以把专题讨论 C 拎出来，把专题讨论 B 放进去，由于房间 A 装专题讨论时挑的是人数最多的，因此专题讨论 C 的人数一定不大于专题讨论 B 的人数，答案不会变差。

因此，任何一个最佳方案，总可以通过一些不使答案变差的调整方式使之与贪心策略求出的方案相同。换言之，上述贪心策略是正确的。

程序清单

```
#include <stdio.h>
#include <algorithm>
using namespace std;
struct info                          //专题讨论和房间的结构类型
{
    int p,t;                         //人数和时间
};
const int MaxW = 1000;
const int MaxR = 1000;
int w,r,cases,sum_w,sum_p,i,j,l;     //专题讨论数为 w，房间数为 r，测试用例编号为 cases，尚待
                                     //安排的讨论数和人数分别为 sum_w、sum_p，当前房间可选取
                                     //的讲习班为 l
info workshop[1100],room[1100];      //专题讨论序列和房间序列
bool cmp(info x,info y)              //返回递增标志，作为排序的比较函数
{
    return (x.t<y.t);
}
bool init()
{
    int i,hh,mm;
```

```
    scanf("%d",&w);                    //输入专题讨论数
    if (w==0) return false;
    sum_w = w; sum_p = 0;              //尚待安排的讨论数和人数初始化
    for (i=0;i<w;i++)                  //输入每个专题讨论的参与人数和持续时间 d，累计参与的总人数
    {
        scanf("%d %d",&workshop[i].p,&workshop[i].t);
        sum_p += workshop[i].p;
    }
    scanf("%d",&r);                    //输入房间数
    for (i=0;i<r;i++)                  //输入每个房间的座位数和可以被使用的时刻
    {
        scanf("%d %d:%d",&room[i].p,&hh,&mm);
        room[i].t = (hh - 14) * 60 + mm;          //计算房间 i 可被使用的分钟数
    }
    sort(workshop,workshop + w,cmp);              //按照持续时间递增的顺序排列专题讨论
    sort(room,room + r,cmp);                      //按照被使用时间递增的顺序排列房间
    return true;
}
int main()
{
    cases = 0;
    while (init())
    {
        for (i=0;i<r;i++)                         //按照可使用时间递增的顺序安排每个房间
        {
            l = -1;                               //可选取的专题讨论编号初始化
            for (j=0;j<w;j++)
            if (workshop[j].t>room[i].t) break;   //一旦发现有时间上不能安排在房间 i 的专题
                                                  //讨论，则退出循环，因为后面讨论的时间更
                                                  //长，再也不可能放入 i 房间
            else if (workshop[j].p<=room[i].p && workshop[j].p>=0)
            //若专题讨论 j 没有安排房间（小于 0 表示已安排）且满足房间 i 的人数限制
            {           //若房间 i 没有安排讨论或安排专题讨论 j 的人数更多，则安排专题讨论 j
                if (l<0 || workshop[l].p<workshop[j].p) l = j;
            }
            if (l>=0) { sum_w --; sum_p -= workshop[l].p; workshop[l].p = -1; }
                        //若房间 i 安排好，则计算尚待安排的讨论数和人数，该讨论设已安排房间标志
        }
        printf("Trial %d: %d %d\n\n",++cases,sum_w,sum_p);
                        //输出未安排在租房内（即安排在帐篷内）的讨论数和人数
    }
}
```

试题 2-10　通信服务区（Zones）

【问题描述】

通过电话线进行通信的有线电话已经是一项过时的技术了。现在我们可以通过手机在任何地方给任何人打电话，而不必依赖电话线。但是手机通话存在的问题是，如果附近没有服务塔，手机便没有任何作用。

如果没有丘陵和山地，一个服务塔的服务可以覆盖一个圆形区域。无线电话公司必须计划要在该区域建造服务塔，而不是放置电话线杆。如果塔与塔之间的距离太远，会导致存在服务空洞的区域，使得用户的抱怨增加；如果塔与塔之间的距离太近，那么服务塔就显得多余而且

低效，因为有大量的区域会被多个服务塔的服务覆盖。

国际移动电话公司（International Cell Phone Company，ICPC）正在开发一个确定服务塔最佳放置的网络布置方案。因为大多数顾客已经用手机替代了旧的有线电话，所以这一放置方案尽量多地覆盖那些用户的住宅。

图 2.10-1 显示了 ICPC 企划部门计划于今年建造的 5 座服务塔所能覆盖的服务区域。塔 5 为 24 位顾客服务，其中 6 位顾客也接受塔 4 的服务。塔 1，2 和 3 的公共服务区共同服务 3 位顾客。

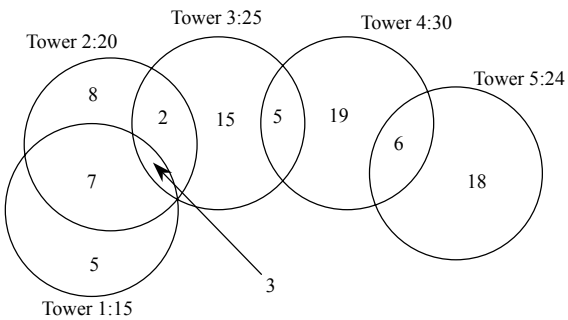


图 2.10-1

在这 5 个塔的计划提出不久之后，高层经理决定停止建造所有的塔。抗议的顾客强迫他们修正这一计划，要他们从 5 个中挑选 3 座塔来建造，当然，这 3 座塔要为尽可能多的顾客服务。找出最佳选择方案并不繁琐（最优方案是选择塔 2，塔 4 和塔 5，为 68 位顾客服务）。

请编写一个程序来帮助 ICPC 在这样的情况下选择建造哪些塔。如果有若干种选择可以为同样多的顾客服务，选择包含塔 1 的那个方案。如果还有多种，选择包含塔 2 的，以此类推。

输入：

输入文件包含若干组测试用例。每组测试用例的第一行是两个正整数：计划计划建造的塔的数量 n（$n \leqslant 20$），和实际能造的塔的数量；下一行有 n 个整数，给出每座塔能为多少位顾客服务（第一个数字对应第一座塔，以此类推），没有一座塔能为多于 100 万位顾客服务；下一行为一个整数 m（$m \leqslant 10$），表示有 m 个共同服务的区域；然后给出 m 行，每行是一个共同服务区域的描述：以一个数字 t（$t>1$）开头，表示有多少座塔为这个区域服务，然后 t 个数字表示这些塔，最后一个数字是这个区域中的顾客人数。输入文件的最后一行是两个整数 0 0。

输出：

对于每组测试用例，输出服务的顾客人数，以及最佳的塔的选择。具体格式见样例输出。

样例输入	样例输出
5 3 15 20 25 30 24 5 2 1 2 7 3 1 2 3 3 2 2 3 2 2 3 4 5	Case Number 1 Number of Customers: 68 Locations recommended: 2 4 5 Case Number 2 Number of Customers: 75 Locations recommended: 1 3 5

（续）

样例输入	样例输出
2 4 5 6 5 3 25 25 25 25 25 4 2 1 2 5 2 2 3 5 2 3 4 5 2 4 5 5 5 3 25 25 25 25 25 0 0 0	Case Number　3 Number of Customers: 75 Locations recommended: 1 2 3

试题解析

由于本题中 n 和 m 的范围均比较小，因此不妨直接枚举：

首先枚举选取哪些塔，然后看一共覆盖了几个公共区域，减去重复计算的，即可得到覆盖的用户数，从中取最大值即可。

若以二进制串 i 表示选取情况，则不妨从 1 枚举至 2^{20}，然后看二进制串中 1 的个数。如果恰好与实际能造的塔数相同，那么求具体覆盖了的用户数。

此时，可以逐个判断公共区域，如果区域包含的塔的情况为 k，那么看 $i\&k$ 有几个 1，即可知道有几个塔出现了，也就知道了这一部分算了几次。设有 T 个塔出现了，那么说明这个区域一共算了 T 次，总人数减去住户数 ×（T–1）即可。这样就能算出覆盖的精确人数了，这个覆盖人数即为二进制串 i 对应的服务顾客数。

在枚举 i（$1 \leqslant i \leqslant 2^{20}$）的过程中，不断调整覆盖人数的最大值。若两个二进制数 T 和 i 对应的服务顾客数都是相同的最大值，则由右而左搜索 T 和 i，最先出现 1 的二进制数即为最佳的选塔方案，因为最佳解是以服务的顾客数为第 1 关键字、选塔的字典序为第 2 关键字的。

程序清单

```
#include <stdio.h>
const int MaxN = 20;              //塔数的上限
const int MaxM = 10;              //公共服务区域数的上限
int cases,n,m,r,tot,ans,T,i,j; //测试用例编号为 cases，计划建造的塔数为 n，实际能造的塔数为 r，
                                  //公共服务区域数为 m，当前覆盖的人数为 tot，最多服务的顾客数为
                                  //ans，最佳选塔方案为 T
int sum[MaxN];          //各塔服务的人数为 sum[]
int common[MaxM][2];    //各个公共区域选中的塔为 common[][1]，覆盖的用户数为 common[][2]
int S[1 << MaxN];       //二进制串 k 中 1 的个数为 S[k]
bool init()             //输入信息
{
    int i,j,k,tmp;
    scanf("%d %d",&n,&r);                    //输入计划建造的塔数和实际能造的塔数
    if (n==0 && r==0) return false;          //若输入两个 0，则退出
    for (i=0;i<n;i++) scanf("%d",&sum[i]);   //输入每座塔服务的顾客数
    scanf("%d",&m);                          //输入共同服务的区域数
```

```
    for (i=0;i<m;i++)
    {
        scanf("%d",&k);                          //输入共同服务区 i 内的塔数
        common[i][0] = 0;                        //输入区域内的每座塔，构造对应的二进制状态
        for (j=0;j<k;j++) { scanf("%d",&tmp); common[i][0] |= 1 << (tmp - 1); }
        scanf("%d",&common[i][1]);               //输入区域内的顾客数
    }
    return true;
}
int main()
{
    cases = 0;
    for (i=0;i<(1<<20);i++)                      //计算二进制串 i 中 1 的个数
    for (j=0;j<20;j++)
    if (i&(1<<j)) S[i] ++;
    while (init())                               //反复输入初始信息，直至输入 2 个 0 为止
    {
       ans = -1;
       for (i=0;i<(1<<n);i++)
       if (S[i]==r)                              //枚举满足 1 的个数恰为实际能造的塔数的二进制数 i
       {
          tot = 0;
          for (j=0;j<n;j++)
          if (i&(1<<j)) tot += sum[j];           //统计这些塔服务的顾客总数
          for (j=0;j<m;j++)     //通过计算每个公共区域出现的塔数得出总覆盖人数
             if (common[j][0]&i)    tot -= common[j][1] * (S[common[j][0] & i] - 1);
          if (tot>ans) { ans = tot; T = i; } else     //若总覆盖人数更多，则更新最优解
          if (tot==ans)          //若总覆盖人数与原最佳解相同
             for (j=0;j<n;j++) //按照序号递增顺序搜索每座塔:若 i 状态中的塔 j 不在原最佳方案 T 中，
                               //则最佳方案改为 i 状态；若原最佳方案 T 中的塔 j 不在 i 状态中，
                               //则最佳方案 T 不变
             {
                if (((1<<j)&i) && !((1<<j)&T)) { T = i; break }
                if (((1<<j)&T) && !((1<<j)&i)) break;
             }
       }
       printf("Case Number  %d\n",++cases);       //输出测试用例编号
       printf("Number of Customers: %d\n",ans);   //输出服务的顾客数
       printf("Locations recommended:");          //输出最佳方案选择了哪些塔
       for (i=0;i<n;i++)
       if ((1<<i)&T) printf(" %d",i + 1);
       printf("\n\n");
    }
}
```