第 3 章

2006 ACM-ICPC 世界总决赛试题解析

试题 3–1　最小费用的飞机旅行（Low Cost Air Travel）

【问题描述】

一张机票的价格是由多个因素决定的，它通常与飞行距离没有直接关系。许多旅行者于是在这方面变得非常有创意，当飞机在多个城市停靠时，只使用机票的一部分，以实现低花费的旅行。然而，航空公司也了解这种行为，并通常要求一张票所包含的站点必须按顺序旅行，而且不允许中途加入其他线路。例如，你手中有一张从城市 1 到城市 2 然后再到城市 3 的机票，你不能仅使用机票中城市 2 到城市 3 的部分，你必须从机票上的第一个城市出发；此外，也不允许你从城市 1 到城市 2，然后去一些其他地方并返回，再继续你从城市 2 到城市 3 的旅途。

为此，我们举一个例子，假设你可以购买以下 3 种机票：

机票#1：城市 1 至城市 3 至城市 4　　$225.00

机票#2：城市 1 至城市 2　　$200.00

机票#3：城市 2 至城市 3　　$50.00

假设你需要从城市 1 到城市 3，仅使用这些机票，你有两种走法可以选择：

1）用$225.00 购买机票#1，但仅使用机票的第一段行程。

2）用$200.00 购买机票#2，并用$50.00 购买机票#3。

显然，选择第一种是最便宜的。

给出一组优惠的机票，以及一条或多条旅游路线，你要确定为了使旅行费用最少，应该如何购买机票。注意需要保证每一条路线都是可行的。

输入：

输入包含多组测试用例，每组测试用例描述了一组优惠机票和一组旅行路线。

每组测试用例的第一行是一个数字 *NT*，表示优惠机票的数量。再给出 *NT* 行，每行描述一种优惠机票，其中，对每种机票的描述包括若干个正整数，分别表示票价、途中的城市数量，以及途经的城市，每个城市由一个任意且唯一的数字来标识。可能存在若干张机票的路线是同一条路线。

接下来的一行给出一个数字 *NI*，表示需要求出最小旅行费用的旅行路线的数量。在接下来的 *NI* 行描述这些路线，其中，每行先给出路线中城市的数量（包括出发的城市），然后按路线上到达的先后顺序给出这些城市的标号。

本题设定在每个测试用例中优惠机票的数量不超过 20 张，或者旅行路线的数量不超过 20 条；每一张机票和每一条旅行路线包含 2～10 个城市；机票的价格不超过$10 000。机票路线或旅行路线中相邻的城市不相同。在每个测试用例中，机票和旅行路线从 1 开始连续地进行编号。

输入最后一组测试用例后，给出一行，包含一个数字 0。

输出：

对于每条旅行路线，输出两行，包括测试用例编号、路线编号、路线的最小花费；然后按使用顺序输出本次旅行所使用的机票编号。具体输出格式见样例，保证答案唯一。

样例输入	样例输出
3 225 3 1 3 4 200 2 1 2 50 2 2 3 1 2 1 3 3 100 2 2 4 100 3 1 4 3 200 3 1 2 3 2 3 1 4 3 3 1 2 4 0	Case 1, Trip 1: Cost = 225 Tickets used: 1 Case 2, Trip 1: Cost = 100 Tickets used: 2 Case 2, Trip 2: Cost = 300 Tickets used: 3 1

试题解析

在本题中，旅行路线上的城市是需要一个一个访问的，因此可以尝试使用动态规划的算法。

此时状态应包括目前已经途经路线中的前几个城市、目前旅行者所在的城市。每一个状态记录此时的最小花费。

具体的状态转移是以递推的方式进行的：记当前状态为 f[n, m]，枚举此时使用机票 t 可途经当前路线上的城市：若使用机票 t（机票价为 cost[t]）可最后到达城市 m′且途经路线上的前 n′个城市，则尝试更新。形式上可以写为

$$f[n', m'] = \min\{f[n, m]+cost[t], f[n', m']\}$$

注意：这里可能有 n′=n 的情况，可能存在第 n 层内部之间更新的情况。考虑到题目范围较小，在第 n 层可以递推若干次，直到无法继续更新为止，再进入下一层。

程序清单

```
#include <stdio.h>
const int Max_City = 200;                  //所有旅行路线包含城市数的上限
const int Max_Ticket = 20;                 //优惠机票数的上限
const int Max_TicketLength = 10;           //每一张机票和每一条旅行路线包含城市数的上限
const int Max_TripLength = 10;
struct info                                //优惠机票的结构类型
{
    int cost,n;                            //记录票价和途经的城市数
    int city[Max_TicketLength];            //途经的城市序列
};
int cases,NT,NI,i,j,k,l,tmp,trip,change,sum,n;
//测试用例编号为 cases，优惠机票数为 NT，最小旅行费用的路线数为 NI，更新标志为 change，路线编号
//为 trip，sum 为总的城市数，n 为某条旅游路线的长度
int city[Max_City];                        //城市序列
```

```
int list[Max_TripLength];                  //当前最小旅行费用路线上第 i 个到达的城市为 list[i]
info ticket[Max_Ticket];                   //优惠机票序列
int f [Max_TripLength][Max_City][4];       //途经当前路线的前 n 个城市、最后到达城市 m 时，最小
                                           //花费为 f[n][m][0]，最后使用的机票编号为 f[n]
                                           //[m][1]，使用本机票前途经的城市数为 f[n][m][2]，
                                           //开始使用本机票的城市为 f[n][m][3]
int route[Max_TripLength * Max_City];      //当前路线使用的机票序列
int find(int x)                            //找到城市 x 的编号，若不存在则更新城市列表
{
    int i;
    for (i=0;i<sum;i++) if (city[i]==x) return i;
    city[sum] = x;
    sum ++;
    return (sum - 1);
}
int main()
{
    cases = 0;                             //测试用例编号初始化
    while (1)
    {
        scanf("%d",&NT); sum = 0;          //输入优惠机票的数量
        if (NT==0) break;
        else cases ++;                     //若输入 0，则结束；否则计算测试用例编号
        for (i=0;i<NT;i++)                 //输入每种优惠机票的票价和途经的城市数
        {
            scanf("%d %d",&ticket[i].cost,&ticket[i].n);
            for (j=0;j<ticket[i].n;j++)    //输入途经的城市序列
            {
                scanf("%d",&tmp);
                ticket[i].city[j] = find(tmp);
            }
        }
        scanf("%d",&NI);                   //输入需求的最小旅行费用的路线数
        for (trip=0;trip<NI;trip++)
        {
            scanf("%d",&n);                //输入第 i 条最小旅行费用路线
            for (i=0;i<n;i++) { scanf("%d",&tmp); list[i] = find(tmp); }
            for (i=0;i<n;i++)
            for (j=0;j<sum;j++)
            f[i][j][0] = -1;               //初始化
            for (i=0;i<4;i++)
            f[0][list[0]][i] = 0;
            for (i=0;i<n;i++)              //递推路线上到达的城市数
            {
                while (1)                  //一直更新到不能更新为止
                {
                    change = 0;            //更新标志初始化
                    for (j=0;j<sum;j++)    //枚举旅行者目前所在的城市
                        if (f[i][j][0]>=0) //若已求出到达城市 j 的最小花费
                            for (k=0;k<NT;k++)//枚举在城市 j 开始换用的优惠机票 k
                                if (ticket[k].city[0]==j)
                                {
                                    tmp=i+1;
                                    //从当前路线的第 i+1 个城市出发，枚举使用优惠机票 k 可到达路线上
                                    //的每个城市
```

```
                                for (l=1;l<ticket[k].n;l++)
                                {               //若使用机票 k 可使路线延长至前 tmp 个城市且未
                                                //求出费用或费用更小，则更新
                                   if (tmp<n && list[tmp]==ticket[k].city[l]) tmp ++;
                                   if(f[tmp-1][ticket[k].city[l]][0]<0|| f[tmp-1]
[ticket[k].city[l]][0]>f[i][j][0]+ticket[k].cost)
                                   {
                                      f[tmp - 1][ticket[k].city[l]][0] = f[i][j][0] +
ticket[k].cost;
                                      f[tmp - 1][ticket[k].city[l]][1] = k;
                                      f[tmp - 1][ticket[k].city[l]][2] = i;
                                      f[tmp - 1][ticket[k].city[l]][3] = j;
                                      change ++;        //设更新标志
                                   }
                                }
                             }
                             if (change==0) break;      //若未更新，则退出循环
                          }
                       }
                       //输出测试用例编号、路线编号和路线的最小花费
                       printf("Case %d, Trip %d: Cost = %d\n",cases,trip + 1,f[n - 1][list[
n - 1]][0]);
                       l = 0;                           //使用的机票数初始化
                       i = n - 1; j = list[n - 1];      //从最后一个状态出发倒推机票序列
                       while (i || j!=list[0])
                       {
                          route[l] = f[i][j][1];        //记下当前的机票编号
                          k = f[i][j][2];               //记下使用本机票前到过的城市数
                          j = f[i][j][3];               //记下使用本机票前旅行者所在的城市
                          i = k;
                          l ++;                         //机票数+1
                       }
                       printf("  Tickets used:");      //输出本次旅行所使用的机票编号
                       for (i=l-1;i>=0;i--)
                       printf(" %d",route[i] + 1);
                       printf("\n");
                    }
                 }
     }
```

试题 3–2　订购冰激凌薄饼片！（Remember the A La Mode!）

【问题描述】

Hugh Samston 拥有所谓的“你想要，Hugh 就给你”的餐饮服务，于是在今年的 ICPC 世界总决赛中就要求 Hugh 为参赛选手提供甜点。Hugh 将按计划持续一个星期在各种社交场合里提供上面浇有冰激凌的薄饼片。就像其他商家一样，Hugh 要提供尽可能最好的服务，于是他订购了各种各样的薄饼片和冰激凌，以满足顾客各种各样的口味。

Hugh 计划为每个薄饼片浇上一勺冰激凌，配成固定的组合供顾客们挑选。当然，与其他商家一样，Hugh 希望从这笔生意中获得尽量多的利润。他预先知道了薄饼片和冰激凌的每种组合分别能赚取的利润，还知道有哪些薄饼和冰激凌是不能组合在一起的（例如，薄荷香蕉块冰激凌和酸橙薄饼）。

给出这些信息以及每种薄饼的块数和每种冰激凌的份数，Hugh 想知道他可以获得的最小利润和最大利润。因为他希望以后的世界总决赛都由他来承办餐饮，他希望有一个通用的程序来解决当前和以后的问题。

输入：

输入包含多组测试用例。每组测试用例的第一行给出两个整数 P（P≤50）和 I（I≤50），分别表示薄饼和冰激凌的种类数。第二行给出 P 个整数，表示每种薄饼的片数；下一行给出 I 个整数，表示每种冰激凌的份数。薄饼的片数之和总是与冰激凌的份数之和相等，并且可以设定所有的薄饼和冰激凌都会被用完。

每组测试用例给出 P 行，每行包含 I 个浮点数，表示每一种薄饼和冰激凌的组合可以获得的利润：第一个数表示第 0 种薄饼配上第 0 种冰激凌所获得的利润；下一个数表示第 0 种薄饼配上第 1 种冰激凌所获得的利润，依次类推。利润值为“–1”表示该种薄饼和冰激凌配料从来都不会组合在一起出售。其他所有的整数（每种薄饼的片数和每种冰激凌的份数）都小于或等于 100，每种薄饼和冰激凌的组合所获得的利润（除“–1”外）都大于 0 且小于或等于 10，小数点后最多保留两位小数。

输入最后一个测试用例后，给出两个 0 表示输入结束。

输出：

对于每组测试用例，输出测试用例的编号（从 1 开始），然后输出最小利润和最大利润，按照样例输出所示的格式输出。所有的数值均保留两位小数，所有的测试用例都保证用完所有的薄饼和冰激凌，至少会有一个解。

样例输入	样例输出
2 3 40 50 27 30 33 1.11 1.27 0.70 −1 2 0.34 4 4 10 10 10 10 10 10 10 10 1.01 −1 −1 −1 −1 1.01 −1 −1 −1 −1 1.01 −1 −1 −1 −1 1.01 0 0	Problem 1: 91.70 to 105.87 Problem 2: 40.40 to 40.40

试题解析

这道题可转化为最小费用最大流模型。网络流图共有（P+I+2）个点，其中左边有 P 个点表示 P 种馅饼，右边有 I 个点表示 I 种冰激凌配料，还有一个源点和一个汇点。从源点出发有 P 条边分别指向每一种馅饼，边的容量为这种馅饼的块数，边的费用为 0。有 I 条边分别从每一种冰激凌指向汇点，边的容量为这种冰激凌配料的份数，边的费用为 0。中间的边表示某种馅饼和冰激凌可以相配，边的容量为无穷大，费用为这种组合所获得的利润。图 3.2-1 为第一个样例数据构造出来的网络流图。

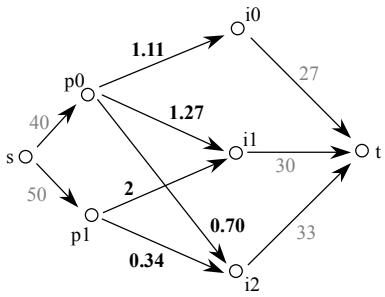


图 3.2-1

图中灰色的数据表示边的容量（这些边的费用为 0），加黑的数据表示边的费用（这些边的容量为无穷大）。

构造出带权网络流图后，计算一遍最小费用最大流，就可以得到总利润的最小值。将边的费用值改为原费用值的相反数，再求一遍最小费用最大流，就可以得到总利润的最大值。最小费用最大流的具体实现方法见程序清单。

程序清单

```
    #include <cstdio>
    #include <memory>
    int profit[55][55],pies[55],ices[55],pies_left[55],ices_left[55],profit_p[55],
profit_i[55],combins[55][55],from_p[55],from_i[55],n,m,i,j,k,new_profit,sum,cases,inf,
updated;//每种薄饼和冰激凌的组合可获得利润为 profit[][]，每种馅饼的片数为 pies[]，每种冰激凌的份数为
        //ices[55]，每种馅饼的剩余数为 pies_left[]，每种冰激凌的剩余数为 ices_left[]，馅饼所在节
        //点的增广路费用为 profit_p[]，冰激凌所在节点的增广路费用为 profit_i[]，馅饼和冰激凌组合的
        //流量为 combins[][],馅饼所在节点的增广路来源为 from_p[],冰激凌所在节点的增广路来源为 from_i[],
        //新的增广路费用为 new_profit，馅饼数为 sum，测试用例编号为 cases，负无穷大为 inf，更新标志
        //为 updated
    double total_profit,x;                    //总费用
    void go()                                 //求最小费用最大流
    {
       memset(combins,0,sizeof(combins));     //初始化馅饼和冰激凌组合的边的流量
       total_profit=0;                        //初始化总费用
       for(i=0;i<sum;i++)                     //对每块馅饼各求一次最小费用增广路，每次流量增加 1
       {
          for(j=0;j<m;j++)
             profit_i[j]=inf;                 //初始化到达冰激凌所在节点的增广路费用
          for(j=0;j<n;j++)
          {
             from_p[j]=-1;                    //初始化馅饼所在节点的增广路来源
             profit_p[j]=inf;                 //初始化馅饼所在节点的增广路费用
             if(pies_left[j])
             profit_p[j]=0;                   //如果当前这种馅饼还有剩余（源点到这种馅饼所在节点
                                              //的边容量未满），则到达这个点的费用为 0
          }
          updated=1;                          //初始化增广路费用是否有更新的标记
          while(updated)                      //在增广路费用有更新的时候进入循环
          {
             updated=0;
             for(j=0;j<n;j++)                 //逐一检查馅饼所在的节点
```

```
            if(profit_p[j]>inf)          //如果有增广路到达了这个点
                for(k=0;k<m;k++)         //逐一检查冰激凌所在的节点
                    if(profit[j][k]>0)   //如果馅饼 j 和冰激凌 k 可以组合
                    {
                        new_profit=profit_p[j]+profit[j][k];   //计算新的增广路费用
                        if(profit_i[k]==inf||new_profit<profit_i[k])
                                         //与原有的增广路费用比较
                        {
                            profit_i[k]=new_profit;    //更新冰激凌所在节点的增广路费用
                            from_i[k]=j;               //更新冰激凌所在节点的增广路来源
                            updated=1;
                        }
                    }
                for(j=0;j<m;j++)                       //逐一检查冰激凌所在的节点
                    if(profit_i[j]>inf)                //如果有增广路到达了这个点
                        for(k=0;k<n;k++)               //逐一检查馅饼所在的节点
                            if(combins[k][j])          //如果馅饼 k 和冰激凌 j 可以组合
                {
                    new_profit=profit_i[j]-profit[k][j];       //计算新的增广路费用
                    if(profit_p[k]==inf||new_profit<profit_p[k])
                                                               //与原有的增广路费用比较
                    {
                        profit_p[k]=new_profit;        //更新馅饼所在节点的增广路费用
                        from_p[k]=j;                   //更新馅饼所在节点的增广路来源
                        updated=1;
                    }
                }
        }
        k=0;                                   //增广路费用最小的冰激凌节点的标号
        for(j=1;j<m;j++)                       //逐一检查每个冰激凌节点的增广路费用
            if(profit_i[k]==inf||!ices_left[k]||profit_i[j]<profit_i[k]&&ices_left[j])
            k=j;   //如果当前节点的增广路费用比先前节点的最小费用还小，则更新费用最小的节点的标号
            ices_left[k]--;                    //使用了第 k 种冰激凌配料，剩余数量减 1
            total_profit+=profit_i[k];         //把增广路的费用值累加到总数
            while(1)                           //逐步寻找最小费用增广路的来源
            {
                j=from_i[k];                   //把与第 k 种冰激凌组合的馅饼标号赋值给 j
                combins[j][k]++;               //第 j 种馅饼和第 k 种冰激凌组合，更新流量，增量为 1
                if(from_p[j]<0)
                break;                         //如果第 j 种馅饼所在点没有来源了，则终止循环
                k=from_p[j];                   //否则把第 j 种馅饼所在节点的增广路来源赋值给 k
                combins[j][k]--;               //取消原来的组合，更新反向弧的流量，减量为 1
            }
            pies_left[j]--;                    //使用了第 j 种馅饼，剩余数量减 1
        }
    }
    int main()
    {
        inf=-(1<<30);                          //负无穷大的记号
        while(scanf("%d%d",&n,&m)&&n)          //反复输入薄饼和冰激凌的种类数，直至输入两个 0 为止
        {
            sum=0;                             //初始化馅饼的总数量
            for(i=0;i<n;i++)
            {
                scanf("%d",&pies[i]);          //输入每种馅饼的片数
```

```
            pies_left[i]=pies[i];                    //可用馅饼的数量初始化
            sum+=pies_left[i];                       //统计馅饼的总数量
        }
        for(i=0;i<m;i++)
        {
            scanf("%d",&ices[i]);                    //输入每种冰激凌的份数
            ices_left[i]=ices[i];                    //可用的冰激凌配料的份数
        }
        for(i=0;i<n;i++)                             //输入每种薄饼和冰激凌的组合可以获得的利润
            for(j=0;j<m;j++)
            {
                scanf("%lf",&x);
                profit[i][j]=int(x*100+0.1);         //把利润扩大 100 倍化为整数
            }
        go();                                        //求最小费用最大流

        printf("Problem %d: %.2lf to ",++cases,total_profit/1e2);
                                                     //输出测试用例编号和最小利润
        memcpy(pies_left,pies,sizeof(pies));         //恢复初始时每种馅饼的片数和冰激凌的份数
        memcpy(ices_left,ices,sizeof(ices));
        for(i=0;i<n;i++)                             //把利润值倒置（将边费用值改为原费用值的相反数）
            for(j=0;j<m;j++)
                if(profit[i][j]>0)profit[i][j]=1001-profit[i][j];
        go();                                        //再求一遍最小费用最大流
        printf("%.2lf\n",(sum*1001-total_profit)/1e2);     //输出最大利润
    }
    return 0;
}
```

试题 3-3　稳态的雕塑（Ars Longa）

【问题描述】

你有了灵感，并为你们当地的博物馆的门厅设计一个美丽的新的艺术雕塑。由于一个很重要的艺术上的因素，你使用了一些十分特殊的材料进行设计。然而，你不清楚你的设计是否满足物理上的原理，你的雕塑可以稳定地站着吗？

这件雕塑是由很多件每个重量为 1 千克的球形接头和连接这些球形接头的竿子（重量忽略不计）组成的。这些竿子不可以被拉伸或压缩，也不能脱离球形接头。但是，它们可以绕着球形接头任意地旋转。在地面上的球形接头是被胶水固定住的，所有其他的球形接头都可以随意移动。为了简单起见，可以忽略互相交叉的竿子之间的影响，每根竿子仅作用于和它相接的两个球形接头；而且，任何悬空的球形接头都至少有一根与它相连的不平行于地面的竿子。这避免了这样的情况：在一个刚性雕塑中，一个球形接头仅被水平的竿子支撑。在现实中，球形接头会略微下沉。

请编写一个程序来判断雕塑是否是静态的（也就是说，不会因重力作用而马上变形）。需要注意的是，每根竿子都可以沿着竿子的方向传输任意大的张力，“静态”意味着这些张力使得每个球形接头受力平衡。

如果雕塑是静态的，还要判断它是否是稳态的（也就是说，不会因轻轻推动球形接头而变形）。

输入：

输入文件包含多组雕塑的描述。每个描述的第一行是两个整数 J 和 R，分别表示球形接头的

个数和竿子的根数，球形接头的标号从 1 到 J；后面的 J 行，每行描述了一个球形接头，给出 3 个浮点数 x, y, z，描述球形接头的坐标；然后的 R 行，每行描述了一根竿子，给出两个不同的整数，描述了竿子两端所连接的球形接头。

每根连接球形接头的竿子的长度都是它的真实长度。z 坐标总是非负的，z 坐标为 0 表示相应的球形接头是被固定好了的。至多有 100 个球形接头和 100 根竿子。

最后一组测试用例后的一行包含两个整数 0。

输出：

对于每组雕塑的描述，输出 NON-STATIC，UNSTABLE 或 STABLE，具体格式见样例输出。

样例输入	样例输出
4 3 0 0 0 −1.0 −0.5 1.0 1.0 −0.5 1.0 0 1.0 1.0 1 2 1 3 1 4 4 6 0 0 0 −1.0 −0.5 1.0 1.0 −0.5 1.0 0 1.0 1.0 1 2 1 3 1 4 2 3 2 4 3 4 7 9 0 0 0 −1.0 −0.5 1.0 1.0 −0.5 1.0 0 1.0 1.0 −1.0 −0.5 0 1.0 0.5 0 0 1.0 0 1 2 1 3 1 4 2 3 2 4 3 4 2 5 3 6 4 7 0 0	Sculpture 1: NON-STATIC Sculpture 2: UNSTABLE Sculpture 3: STABLE

试题解析

本题是一道解方程题。每根竿子提供的张力大小是未知数。对于每个球形接头，如果它不是贴在地面上（z 坐标大于 0），它就要保持静止不动，必须满足受力平衡的条件：

1）在 x 方向上所受张力的合力为 0。

2）在 y 方向上所受张力的合力为 0。

3）在 z 方向上所受张力的合力为 1，方向向上（假设重力为 1，方向向下）。

于是我们列出 $3r$ 个含有 m 个未知数方程。其中 r 是不贴地面的球形接头个数，m 是竿子的根数。采用高斯消元法检查这 $3r$ 个方程是否有解：如果有解，就说明受力平衡的条件可以全部满足，雕塑是静态的；否则就说明雕塑是非静态的。

如果雕塑是静态的，我们还要判断它是否稳定。一个简单的判断方法就是对每个球形接头施加一个随机方向、随机大小的力。例如，将满足受力平衡的条件改为

1）在 x 方向上所受张力的合力为 $\sin(3i)$。

2）在 y 方向上所受张力的合力为 $\sin(3i+1)$。

3）在 z 方向上所受张力的合力为 $\sin(3i+2)$。

其中 i 是球形接头的整数标号。利用 sin 函数产生–1 到 1 之间的随机数。

我们再次采用高斯消元法检查这 $3r$ 个方程是否有解：如果有解，就说明受力平衡的条件可以全部满足，雕塑是稳定的；否则就说明雕塑是不稳定的。

其余细节见程序清单。

程序清单

```
#include <cstdio>
#include <memory>
#include <math.h>
double x[101],y[101],z[101],a[300][101],b[300][101],eps,c;
            //球形接头坐标为 x[]、y[]、z[]，各球形接头受到竿子的张力为 a[][]，方程组为 b[][]
int n,m,r,i,j,k,l,t;                        //定义球形接头数为 n，竿子数为 m，方程数为 r
bool ok()                                   //使用高斯消元法判断方程组是否有解
{
   l=0;
   for(i=m;i;i--)                           //逐一消去 m 个未知数
   {
      k=l;
      for(j=l+1;j<r;j++)                    //对于第 i 个未知数，找到系数绝对值最大的方程
         if(fabs(b[j][i])>fabs(b[k][i]))
         k=j;
      if(fabs(b[k][i])<eps)
      continue;                             //如果所有方程的第 i 个未知数的系数为 0，则跳过
      memcpy(x,b[l],sizeof(x));             //将第 i 个未知数系数绝对值最大的方程交换到第 l 行
      memcpy(b[l],b[k],sizeof(x));
      memcpy(b[k],x,sizeof(x));
      for(j=l+1;j<r;j++)                    //消去其余方程的第 i 个未知数
      {
         c=b[j][i]/b[l][i];
         for(k=0;k<i;k++)
            b[j][k]-=c*b[l][k];
         b[j][i]=0;
      }
```

```
            l++;
        }
        for(i=l;i<r;i++)
            if(fabs(b[i][0])>eps)
            return false;
        return true;
    }
    int main()
    {
        eps=0.1;
        while(scanf("%d%d",&n,&m)&&n)          //反复读入球形接头数、竿子数，直至读入两个 0 为止
        {
            memset(a,0,sizeof(a));
            for(i=0;i<n;i++)                   //读入球形接头坐标
                scanf("%lf%lf%lf",&x[i],&y[i],&z[i]);
            for(i=1;i<=m;i++)
            {
                scanf("%d%d",&j,&k);                          //读入竿子所连接的球形接头的标号 j、k
                j--;k--;                                      //将 1 到 n 的标号改成 0 到 n-1
                if(z[j])                                      //如果球形接头 j 不贴地面
                {
                    a[j*3][i]=x[k]-x[j];                      //球形接头 j 在 x 方向受到竿子 i 的张力
                    a[j*3+1][i]=y[k]-y[j];                    //球形接头 j 在 y 方向受到竿子 i 的张力
                    a[j*3+2][i]=z[k]-z[j];                    //球形接头 j 在 z 方向受到竿子 i 的张力
                }
                if(z[k])                                      //如果球形接头 k 不贴地面
                {
                    a[k*3][i]=x[j]-x[k];                      //球形接头 k 在 x 方向受到竿子 i 的张力
                    a[k*3+1][i]=y[j]-y[k];                    //球形接头 k 在 y 方向受到竿子 i 的张力
                    a[k*3+2][i]=z[j]-z[k];                    //球形接头 k 在 z 方向受到竿子 i 的张力
                }
            }
            r=0;                                              //r 变量记录方程的个数
            for(i=0;i<n;i++)                                  //逐一检查每个球形接头
                if(z[i]>0)                                    //如果该球形接头不贴地面
                {
                    memcpy(b[r++],a[i*3],sizeof(x));          //根据 x 方向的受力添加一个方程到 b 数组
                    memcpy(b[r++],a[i*3+1],sizeof(y));        //根据 y 方向的受力添加一个方程到 b 数组
                    memcpy(b[r++],a[i*3+2],sizeof(z));        //根据 z 方向的受力添加一个方程到 b 数组
                    b[r-1][0]=1;                              //z 方向所受合力为 1
                }
            printf("Sculpture %d: ",++t);
            if(!ok())                                         //判断方程组是否有解
            {
                puts("NON-STATIC");
                continue;
            }
            r=0;
            for(i=0;i<n;i++)                                  //再次逐一检查每个球形接头
                if(z[i]>0)                                    //如果该球形接头不贴地面
                {
                    memcpy(b[r++],a[i*3],sizeof(x));          //根据 x 方向的受力添加一个方程到 b 数组
                    memcpy(b[r++],a[i*3+1],sizeof(y));        //根据 y 方向的受力添加一个方程到 b 数组
                    memcpy(b[r++],a[i*3+2],sizeof(z));        //根据 z 方向的受力添加一个方程到 b 数组
                    b[r-1][0]=sin(i*3+1);                     //x 方向所受合力为 sin(i*3+1)
                    b[r-2][0]=sin(i*3+2);                     //y 方向所受合力为 sin(i*3+2)
```

```
            b[r-3][0]=sin(i*3+3);                    //z 方向所受合力为 sin(i*3+3)
        }
        if(!ok())                                    //判断方程组是否有解
        {
            puts("UNSTABLE");
            continue;
        }
        puts("STABLE");
    }
    return 0;
}
```

试题 3–4　二段数（Bipartite Numbers）

【问题描述】

你所在公司的执行官要求员工互相发送加密过的消息。相对于现成的加密软件如 PGP，他们更希望把这个任务交给 IT 员工，让他们来处理加密问题。这些 IT 员工决定采取一个需要“公共的” 和“私人的”整数密钥的解决方案。也就是说，所有人都可以看到你的公共密钥，但是只有你才知道自己的私人密钥。

你公司里最好的朋友是一个很好的人，但也是一个不那么好的程序员。他创造了一个公私组合的密钥，方案如下：一个公共的密钥可以是任意的正整数，相应的私人密钥是一个比公共密钥大而且是公共密钥倍数的最小的二段数（bipartite number）。

二段数是这样的正整数：恰好包含两种不同的十进制数字 *s* 和 *t*，*s* 不是 0，并且 *s* 的所有出现均排列在所有的 *t* 的前面。例如，44444411 是二段数（*s* 是 4，*t* 是 1），41、10000000 和 5555556 也是。但 4444114 和 44444 都不是二段数。

注意那些很大的二段数，如 88888888888800000 可以简洁地表述成 12 个 8 后面接 5 个 0。可以使用 4 个数表述任意的二段数：*m s n t*。数字 *s* 和 *t* 是如上所述的头部和尾部的数字，*m* 是数字 *s* 在二段数中出现的次数，*n* 是数字 *t* 在二段数中出现的次数。

你朋友的方案存在的问题是，如果你知道公共密钥，计算私人密钥是不太难的。你要劝说你的朋友，让他知道他的公私组合密钥是不够强的，以免让他的错误决定使他失去了工作。因此请编写一个程序，以公共密钥作为输入，输出相应的私人密钥。

输入：

输入包含多组测试用例。每组测试用例是单独的一行，包含一个范围为 1 到 99999 的公共密钥。

最后一组测试用例后的一行是整数 0。

输出：

对于每一组测试用例，输出公共密钥，接着是 4 个整数 *m s n t*，其中 *m*、*n*、*s* 和 *t* 是上面提到的数。

样例输入	样例输出
125 17502 2005 0	125: 1 5 2 0 17502: 4 7 4 8 2005: 3 2 3 5

试题解析

预先计算好两个数列。

数列 1：$a(1)=1$，$a(i+1)=(a(i)*10+1) \bmod N$，$i=1$，2，3，…。$a(i)$就是连续 i 个 1 除以 N 的余数。

数列 2：$b(0)=1$，$b(i+1)=b(i)*10 \bmod N$，$i=0$，1，2，…。$b(i)$就是 10 的 i 次方除以 N 的余数。

有了这两个数列，只需用 O（1）的时间就可以计算任何二段数除以 N 的余数。二段数 m，s，n，t 除以 N 的余数等于（$a(m)*b(n)*s+a(n)*t$）$\bmod N$。

由于这题的时间限制比较宽松，接下来只要枚举 m，s，n，t 就可以了。按照（$m+n$）的值从小到大枚举，（$m+n$）确定后枚举 m，则 n 可以直接计算出来，不需要枚举。m 和 n 确定之后枚举 s 和 t。一旦找到解，后面的（$m+n$）值就不需要继续枚举下去了。为了减少不必要的枚举，可以先进行判断。例如，N 是 10 的倍数时，t 只能取 0。

其他具体细节见程序清单。

程序清单

```
#include <stdio.h>
int ones[9999],tens[999],n,i,j,k,s,t,best_i,best_j,best_s,best_t;
                //a 数列为 ones[]，b 数列为 tens[]，二段数中头部的数字为 best_s，出现次数为
                //best_j+1；尾部的数字为 best_t，出现次数为 best_i-best_j
bool ck()       //检查当前枚举到的二段数是否大于 N
{
   int p,q,r;
   if(i>5)return 1;
   p=s;
   r=t;
   for(q=0;q<j;q++)
   p=p*10+s;
   for(q=0;q<i-j;q++)
   p=p*10;
   for(q=1;q<i-j;q++)
   r=r*10+t;
   return p+r>n;
}
int main()
{
   while(scanf("%d",&n)*n)          //反复输入公共密钥，直至输入 0 为止
   {
      printf("%d: ",n);             //输出公共密钥
      if(n==1)                      //输出 1 的二段数
      {
         puts("1 1 1 0");
         continue;
      }
      ones[0]=1;                          //初始化 a 数列的第 1 项
      tens[0]=1;                          //初始化 b 数列的第 1 项
      for(i=1;i<9999;i++)                 //生成 a 数列
         ones[i]=(ones[i-1]*10+1)%n;
      for(i=1;i<999;i++)                  //生成 b 数列
         tens[i]=tens[i-1]*10%n;
```

```
            for(i=1,best_s=0;i<9999;i++)      //枚举二段数的总长度
            {
                k=0;
                if((n%10==0||n%25==0)&&i>11)
                k=i-11;                       //对第一段数的长度范围作限制，以提高枚举的效率
            for(j=k;j<i;j++)                  //枚举第一段的长度
                for(s=1;s<10;s++)             //枚举 s
                   for(t=0;t<(n%10?10:1);t++) //枚举 t。如果 N 是 10 的倍数，则 t 只能取 0
                      if(t!=s&&(((long long)ones[j])*tens[i-j]*s+ones[i-j-1]*t)%n==0&&
ck()&&(!best_s||s<best_s||s==best_s&&j>best_j&&s<best_t))
                                          //如果 t 和 s 不相等，且二段数除以 N 的余数为 0，且二段数大
                                          //于 N，则这个二段数符合要求，把它与最佳结果比较
                      {
                         best_i=i;        //更新最佳结果
                         best_j=j;
                         best_s=s;
                         best_t=t;
                      }
                if(best_s)
                break;                    //如果找到解了，则不需要枚举更长的二段数了
            }
            printf("%d %d %d %d\n",best_j+1,best_s,best_i-best_j,best_t);
                                          //输出 n 的两段数对应的 4 个整数
        }
        return 0;
    }
```

试题 3–5　压缩二进制消息（Bit Compressor）

【问题描述】

数据压缩的目的在于减少数据存储与通信的冗余。这增加了有效数据的密度，并加速了数据转换的速度。一个可能的压缩二进制消息的方法如下：

只要能使消息的长度减少，就把最长的连续 n 个 1 替换为 n 的二进制表示。

例如，可以把数据 11111111001001111111111111110011 压缩为 10000010011110011。原始数据有 32 位长，然而压缩以后的数据仅有 17 位。

这个方法的缺点在于，有时候解压缩过程可以从同一条数据解压出多个可能的原始数据，这会导致无法获得真正的原始消息。请编写一个程序，给出原始消息的长度（L），原始数据中 1 的个数（N）和被压缩以后的数据，来判断是否能从被压缩的数据获得原始的消息。原始数据的长度不大于 16 千字节，压缩以后的数据长度不大于 40 位。

输入：

输入文件包含多组测试用例。每组数据有两行。第一行给出 L 和 N，第二行给出了被压缩后的数据。

最后一组测试用例以后的一行仅包括两个 0。

输出：

对于每组测试测试用例，输出一行，包括测试用例编号（从 1 开始）以及消息 YES、NO 或 NOT UNIQUE。YES 表示可以获得原始的消息；NO 表示被压缩的数据被破坏，无法获得原始的消息；NOT UNIQUE 表示可以获得多条原始消息。具体格式见样例输出。

样例输入	样例输出
32 26 10000010011110011 9 7 1010101 14 14 111111 0 0	Case #1: YES Case #2: NOT UNIQUE Case #3: NO

试题解析

该题无技巧可循，只能枚举所有可能的解压方案，判断是否符合要求。

对于一段压缩过的编码，0 可以看做分隔符。对于一段两端以 0 隔开的以 1 开头的编码，枚举两种情况：

情况 1——这段编码是原始的消息。

情况 2——这段编码是经过压缩的编码。

对于情况 1，要求这段编码全部是 1，中间不含有 0，而且长度不大于 2。对于情况 2，不要求这段编码全是 1，中间可以包含 0，而且长度是不固定的，对于任何可能的长度均尝试解码。需要注意的是，长度为 2 的“11”是不会压缩成“10”的，所以不要对“10”进行解码。

在枚举的过程中，一旦找到一个符合要求的解压方案，就计入总数。当符合要求的解压方案数大于 1 时即可停止枚举，输出“NOT UNIQUE”。如果所有可能的解压方案均不符合要求，输出“NO”；否则输出“YES”。

具体实现过程见程序清单。

程序清单

```
#include <stdio.h>
char a[44];                                          //被压缩后的数据
int cases,inf,n,m,len,total,ones,solutions;          //测试用例编号为 cases；解压后的编码长度上限
                                                     //为 inf；原始消息长度为 n，其中 1 的个数为 m；
                                                     //被压缩过的原始编码长度为 len；解压缩长度为
                                                     //total，其中 1 的个数为 ones；符合要求的解压
                                                     //方案数为 solutions
void go(int i)                                       //枚举所有可能的解压方案
{
    int j,k,tmp_len,all_ones;
    if(total>n||ones>m||solutions>1)
    return;          //解压后的编码过长、1 的个数超过了给定值或者已经找到多于 1 个符合要求的解压方
                     //案，均不用继续往下递归
    if(i>=len)                                       //如果解压完毕
    {
        if(total+ones==n+m)
        solutions++;                                 //如果解压后的编码长度和 1 的个数均符合要求，则
                                                     //这个解压方案是符合要求的，计入总数
        return;
    }
    if(a[i]=='1')                                    //对以 1 开头的编码段进行枚举
        for(j=i,k=0,all_ones=1;j<len;j++)            //对所有可能的长度均尝试解压
        {
```

```
            if(k<inf)
            k=k*2+a[j]-'0';                    //计算当前编码片段所表示的 1 的个数
            all_ones&=a[j]-'0';                //对这段编码是否全是 1 做标记
            if(a[j+1]!='1')                    //如果这段编码后面是以 0 隔开的
            {
               if(k>2)                         //如果这段编码的解压缩长度大于 2
               {
                  total+=k;                    //累计解压缩长度和其中 1 的个数
                  ones+=k;
                  go(j+1);                     //进入下一层递归
                  total-=k;                    //递归完毕，还原长度和其中 1 的个数
                  ones-=k;
               }
               if(all_ones&&j-i<2)             //如果这段编码全是 1 且长度不超过 2，则可以理解
                                               //为原始消息中的 1
               {
                  tmp_len=j-i+1;               //计算这段编码的长度
                  total+=tmp_len;              //累计解压缩长度和其中 1 的个数
                  ones+=tmp_len;
                  go(j+1);                     //进入下一层递归
                  total-=tmp_len;              //递归完毕，还原长度和其中 1 的个数
                  ones-=tmp_len;
               }
            }
         }
      else                                     //如果当前位是 0，它就是原始消息中的 0
      {
         total++;                              //总长度增加 1
         go(i+1);                              //进入下一层递归
         total--;                              //递归完毕，还原长度
      }
   }
   int main()
   {
      inf=99999;                               //解压后的编码长度上限
      for(cases=1;scanf("%d%d\n",&n,&m)&&n+m;cases++)
                  //反复输入原始消息长度和其中 1 的个数，计算测试用例编号，直至输入两个 0 为止
      {
         gets(a);                              //输入了被压缩后的数据
         for(len=0;a[len]>' ';len++);          //被压缩过的原始编码长度记为 len
         total=0;                              //解压后的编码长度初始化
         ones=0;                               //解压后编码中 1 的个数初始化
         solutions=0;                          //符合要求的解压方案数初始化
         go(0);                                //以递归的方式枚举所有可能的解压方案
         printf("Case #%d: %s\n",cases,solutions?solutions-1?"NOT UNIQUE":"YES":"NO");
                                               //输出可获得多条原始消息或可否获得原始消息的结果
      }
      return 0;
   }
```

试题 3-6　构造一个时钟（Building a Clock）

【问题描述】

在 Prague（布拉格）的旧城广场（Old Town Square）有一座制造于 1410 年的美丽的天文钟，

如图 3.6-1 所示。几个世纪以来，制表工艺都是用一个以已知速率转动的轴和一些齿轮相连接，通过齿轮之间精确的咬合来驱动其他的轴，使得两个轴以分针和时针的速率旋转。

旧城广场的天文钟

图　3.6-1

请编写一个程序，给出驱动轴旋转速率和一个齿轮的集合，输出一个将齿轮连接在一起构造成一只钟的方案，即有两个轴可以分别驱动分针和时针。你可以使用任意多个轴，但每个轴上最多连接 3 个齿轮。在同一个轴上的所有齿轮都以相同的速率旋转。若一个有 T_1 个齿，并以 R_1 速度旋转的齿轮与另一个有 T_2 个齿的齿轮相咬合，那么第二个齿轮的旋转速率是 R_1（T_1/T_2）。解答中必须包含以下两个轴：一个顺时针旋转并且每小时转一周的分针轴和一个顺时针旋转且每 12 小时转一周的时针轴。你的解答不要求给出所有使用的齿轮。

输入：

输入包含多组测试用例，每组测试用例一行。每一行先给出一个整数 N（$3 \leqslant N \leqslant 6$），表示制造钟可以用的齿轮数；然后给出另一个整数 R（$-3600 \leqslant R \leqslant 3600$，$R \neq 0$），表示驱动轴的旋转速率，即驱动轴每 24 小时旋转几周（正数表示顺时针旋转，负数表示逆时针旋转）；接下来给出 N 个齿轮的描述，每个齿轮以一个二元组表示：一个字符来表示齿轮的名称，接下来一个整数 T（$6 \leqslant T \leqslant 120$），表示齿轮的齿数。齿轮的名称和齿数均以空格隔开，如样例输入所示。

输入最后一组测试用例后给出一行，仅包含一个整数 0。

输出：

对于每组测试用例，如样例输出所示，首先输出测试用例编号；如果使用给出的齿轮可以构造一只钟，接下来输出两行，分别给出分针轴和时针轴；否则，如样例输出所示输出“IS IMPOSSIBLE”。

描述分针轴的一行以“Minutes:”开头，然后输出驱动轴通过一系列齿轮来驱动分针轴的方案。方案给出一个以连字符相连的轴的序列，每个轴由一个或两个字符组成，第一个字符表示被驱动齿轮（在当前的轴上与前一个轴的齿轮相咬合的齿轮）的名称；如果是驱动轴，则用星号（*）来替代默认的被驱动齿轮；描述轴的第二个字符表示驱动齿轮（在当前轴上与下一个轴上的齿轮相咬合的齿轮）的名称；被驱动齿轮和驱动齿轮可以是同一个，这种情况下轴用一个字符表示，这个字符就是这个齿轮的名称。最后轴是分针轴，用一个字符表示，这个字符就是它的被驱动齿轮的名称。

描述时针轴的一行以“Hours:”开头，输出驱动轴如何通过一系列齿轮来驱动时针轴的方案。输出格式与分针轴相同。

虽然每个齿轮在一只钟里只能出现一次，但分针轴和时针轴的构造方案中最初的部分可以有公共的部分。在最初的公共部分中出现的齿轮可以在分针轴和时针轴的构造方案中同时出现。同样的，某个特定的轴也可以在时针轴和分针轴的构造方案中同时出现。在两个方案中同时出现的一个轴在两个方案中的描述可能相同，也可能不同。比如说，一个仅包含齿轮 A 的轴在两个方案中都用 A 表示，但一个包含 3 个齿轮 A，B 和 C 的轴可能在分针轴中以 AB 表示（B 是该方案中的驱动齿轮），而在时针轴中以 AC 表示（C 是该方案中的驱动齿轮）。

下面是一些合法输出的例子。

Hours：*A-BC-D　驱动轴中包含 1 个齿轮，与一个有两个齿轮的中间轴相咬合，中间轴的齿轮咬合在时针轴的一个齿轮上。

Minutes：*A-B-C　驱动轴中包含 1 个齿轮，与有一个齿轮的中间轴相咬合，中间轴的齿轮咬合在分针轴的一个齿轮上。

Minutes：*在这个方案中不需要其他齿轮，因为驱动轴就在以分针轴的速率旋转。

如果可以有多种方案使用给定齿轮构造一个钟，就输出使用最少的轴的方案。若仍有多解，则输出使用最少齿轮的方案。若还有多解，则输出解答的字符串描述字典序最小的方案。解答的字符串描述是把它的分针方案放在前，时针方案放在后，去掉星号和连字符以后连接在一起得到的字符串。例如，分针轴是*A-B，时针轴是*A-BC-D-E，此时的字符串描述为 ABABCDE。

在两组测试用例之间输出一个空行。

样例输入	样例输出
6 40 P 7 Q 84 R 50 A 40 B 30 C 14 6 40 P 7 Q 84 R 45 A 40 B 30 C 14 0	Trial 1 Minutes: *B-A-R Hours: *B-A-RP-C-Q Trial 2 IS IMPOSSIBLE

试题解析

试题给出了驱动轴的旋转速率和 n 个齿轮的齿数，要求以转轴数为第 1 关键字、齿轮数为第 2 关键字、方案的字典序为第 3 关键字，分别计算驱动分针轴和驱动时针轴的最佳方案。有两种求解方法。

方法 1：回溯+剪枝优化

很显然，本题是一道搜索题，可以采用回溯的方法求解，关键在于剪枝优化。对于齿轮之间关系形成的路径，我们可以看成是一个三叉结构，分别从公共前缀串（分针方案+时针方案）和两个子串（分针方案、时针方案）的角度来考虑问题。

我们先考虑公共串 s 的情况。首先是原始的转轴，我们知道，在原始转轴上放两个公共齿轮是没有必要的。同时，如果遇到分支，必然只会有两个齿轮在同一转轴上，所以，如果在公共串上出现连续 3 个齿轮显然是不优的。

对于子串的情况，我们可以用 sm 和 sh 两个子串分别表示当前的分针方案与时针方案。与公共串相同的是，子串上也是不可能出现连续 3 个齿轮的。此外还有一处可行的剪枝：如果公共串是在第三个位置出现分支，那么必然不会再出现齿轮了。

上述算法较为简洁，可在竞赛允许的时限内通过所有数据。我们采用的正是这种方法。具体实现见程序清单。

方法 2：直接枚举组合方案

回溯+剪枝优化的方法之所以行得通，是因为可用齿轮的个数最多为 6 个，可能的组合方案并不多。假设 6 个齿轮的标号分别为 A、B、C、D、E、F，在不考虑哪个标号对应哪个齿轮的前提下，驱动轴顺时针旋转（R>0）时，有 12 种组合方案；驱动轴逆时针旋转（R<0）时，有 10 种组合方案。

（1）驱动轴顺时针旋转（R>0）时产生的 12 种组合方案如下所示：

分针轴	时针轴	轴数	齿轮数	以转轴数为第 1 关键字、齿轮数为第 2 关键字、字典序为第 3 关键字排序
*	*A-B-C	3	3	1
*A-B-C	*	3	3	2
*	*A-BC-D	3	4	3
*A-BC-D	*	3	4	4
*A-B-C	*A-B-D	4	4	5
*A-BC-D	*A-BC-E	4	5	6
*A-B-C	*A-BD-E	4	5	7
*A-BC-D	*A-B-E	4	5	8
*A-BC-D	*A-BE-F	4	6	9
*A-B-C	*D-E-F	5	6	10
*A-B-C	*A-B-CD-E-F	5	6	11
*A-B-CD-E-F	*A-B-C	5	6	12

（2）驱动轴逆时针旋转（R<0）时产生的 10 种组合方案如下所示：

分针轴	时针轴	轴数	齿轮数	以转轴数为第 1 关键字、齿轮数为第 2 关键字、字典序为第 3 关键字排序
*A-B	*A-C	3	3	1
*A-B	*C-D	3	4	2
*A-B	*A-B-CD-E	4	5	3
*A-B-CD-E	*A-B	4	5	4
*A-B	*A-BC-D-E	4	5	5
*A-BC-D-E	*A-B	4	5	6
*A-B	*A-BC-DE-F	4	6	7
*A-BC-DE-F	*A-B	4	6	8
*A-BC-D-E	*A-BC-D-F	5	6	9
*A-B-CD-E	*A-B-CD-F	5	6	10

于是，我们换另一种方法：可以根据驱动轴的旋转速率、齿轮的齿数和驱动轴的旋转方向（R>0 和 R<0）选择对应的组合表，顺序查找第一个满足分针“每小时转一周、时针每天转两周”的合法方案。显然，方法 2 的运行效率要明显优于方法 1，但编程要复杂得多，远不如方法 1 简洁。

程序清单

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <map>
#include <string>
using namespace std ;
#define PB push_back
char ch[7] ;
map<char,int> T ;                          //齿轮名为 s 的齿数为 T[S]
//最优解。其中驱动分针轴的最佳方案为 AnswerM，驱动时针轴的最佳方案为 AnswerH，转轴数为 AnswerShaft，
//齿轮数为 AnswerGear
string AnswerM , AnswerH ;
int AnswerShaft , AnswerGear ;
int N , R ;                                //齿轮数和旋转速率
bool used[7] ;                             //齿轮的使用状态
int Time( string s ) {                     //驱动分针轴（或时针轴）的方案为 s，计算其旋转速率
int Rs = R , Rt = 1 , lasT ;
for ( int i = 1 ; i < s.size() ; i ++ ) {
   if ( s[i] == '-' ) ;
   else if ( s[i-1] == '-' ) {
      Rs *= -lasT ;
      Rt *= T[s[i]] ;
      lasT = T[s[i]] ;
      } else lasT = T[s[i]] ;              //记下当前转轴尾齿轮的齿数
   }
   if ( Rs % Rt == 0 ) return Rs/Rt ;      //若齿轮间咬合，则返回方案的旋转速率
      else return 0 ;
   }
   void Refresh( string sm , string sh , int shaft , int gear ) { //更新最佳方案
   if ( AnswerM == "X" ) {                 //若目前驱动分针轴的最佳方案为空，则当前方案调整为最佳方案
      AnswerM = sm ; AnswerH = sh ; AnswerShaft = shaft ; AnswerGear = gear ;
      return ;
   }
   string sold , snew ;                    //最佳方案的公共串和当前方案公共串初始化
   sold.clear() ; snew.clear() ;
                                           //将最佳方案中驱动分针轴和驱动时针轴的齿轮存入公共串 sold
   for ( int i=0 ;i<AnswerM.size() ; i ++ )
      if ( AnswerM[i] != '*' && AnswerM[i] != '-' )
      sold.PB( AnswerM[i] ) ;
   for ( int i = 0 ; i < AnswerH.size() ; i ++ )
   if ( AnswerH[i] != '*' && AnswerH[i] != '-' )
   sold.PB( AnswerH[i] ) ;
         //将当前方案中驱动分针轴和驱动时针轴的齿轮存入公共串 snew
   for ( int i = 0 ; i < sm.size() ; i ++ )
   if ( sm[i] != '*' && sm[i] != '-' )
   snew.PB( sm[i] ) ;
for ( int i = 0 ; i < sh.size() ; i ++ )
   if ( sh[i] != '*' && sh[i] != '-' )
   snew.PB( sh[i] ) ;
if ( shaft < AnswerShaft ) {
         //以转轴数为第 1 关键字、齿轮数为第 2 关键字、方案的字典序为第 3 关键字调整最佳方案
   AnswerM = sm ; AnswerH = sh ;
   AnswerShaft = shaft ; AnswerGear = gear ;
```

```
        }
        else if (shaft == AnswerShaft && gear < AnswerGear) {
            AnswerM = sm; AnswerH = sh;
            AnswerShaft = shaft; AnswerGear = gear;
            }
            else if (shaft == AnswerShaft && gear == AnswerGear && snew < sold) {
                AnswerM = sm; AnswerH = sh;
                AnswerShaft = shaft; AnswerGear = gear;
            }
    }
    void Dfs(string sm,string sh,int shaft,int gear) {
          //递归计算分支（驱动分针轴的方案为 sm，驱动时针轴的方案为 sh，第 1 关键字为转轴数 shaft，
          //第 2 关键字为齿轮数 gear）
    if (sm.size() == 3 || sh.size() == 3)
    return;                                          //分支不可能在第三个位置出现齿轮
    if (sm.size()>=4 && sm[sm.size()-1]!='-'&& sm[sm.size()-2]!='-' && sm[sm.size()-3] !=
'-') return;                                         //驱动时针轴的方案中不能出现连续 3 个齿轮

    if (sh.size()>=4&&sh[sh.size()-1]!= '-'&&sh[sh.size()-2]!='-'&& sh[sh.size()-3]!='-')
    return;                                          //驱动时针轴的方案中不能出现连续 3 个齿轮
    if (shaft>AnswerShaft||(shaft==AnswerShaft && gear>AnswerGear))
    return;                                          //若当前方案劣于最优方案，直接退出
    //若分针每小时转一周、时针每天转两周，则更新最佳方案
    if (Time(sm) == 24 && Time(sh) == 2) Refresh(sm,sh,shaft,gear);
       else if (Time(sm) == 24) {                    //若分针每小时转一周，则处理时针
           if (sm.size() >= 3 && sm[sm.size()-2] != '-')
           return;                                   //若驱动分针轴的尾转轴未咬合一个齿轮，则退出
               for(int i=1;i<=N;i++)                 //枚举每一个未使用过的齿轮，置该齿轮使用标志
                  if (!used[i]) {
                     used[i] = true;
                     Dfs(sm,sh+ch[i],shaft,gear+1);          //该齿轮安置在同一个转轴上
                     if (sh.size()>1)                //新增一个转轴，与上一个齿轮连接
                        Dfs(sm,sh+'-'+ch[i],shaft+1,gear+1);
                     used[i] = false;                //撤去该齿轮的使用标志
           }
       } else {                                      //处理分针
             for (int i = 1; i <= N; i ++)           //枚举每一个未使用过的齿轮，置该齿轮使用标志
             if (!used[i]) {
                used[i] = true;
                Dfs(sm+ch[i],sh,shaft,gear+1);                //该齿轮安置在同一个转轴上
                if(sm.size()>1)                               //新增一个转轴，与上一个齿轮连接
                   Dfs(sm+'-'+ch[i],sh,shaft+1,gear+1);
                used[i] = false;                              //撤去该齿轮的使用标志
             }
       }
    }
    void Dfs(string s,int shaft,int gear){  //递归计算公共部分 s(转轴数为 shaft,齿轮数为 gear)
       if (s.size() == 3)
       return;                                    //原始转轴上不可能放有 2 个公共转轴
       if (s.size()>=4 && s[s.size()-1]!='-'&&s[s.size()-2]!= '-' && s[s.size()-3]!= '-')
       return;                                    //公共路径上不可能有连续的 3 个齿轮
       if (shaft>AnswerShaft||(shaft==AnswerShaft && gear>AnswerGear))
       return;                                    //转轴个数和齿轮个数已经使用过多
       Dfs(s, s, shaft, gear);                    //递归计算分支（驱动分针轴和时针轴的方案为 s）
       for (int i = 1; i <= N; i ++)              //枚举未使用的下一个齿轮
```

```
        if (!used[i]) {
            used[i] = true;
            Dfs(s+ch[i],shaft,gear+1);                    //该齿轮安置在同一个转轴上
            if (s.size()>1)                               //新增一个转轴，与上一个齿轮连接
                Dfs(s+'-'+ch[i],shaft+1,gear+1);
            used[i] = false;                              //恢复齿轮 i 的未使用状态
        }
}
intmain() {
    int Case = 0;                                         //测试用例编号初始化
    while (cin >> N && N) {                               //反复输入齿轮数，直至输入 0 为止
        if (Case > 0) cout << "\n";                       //数据组间空一行
            cin >> R;                                     //输入旋转速率
        int x;
        for (int i = 1; i<= N; i++)
        used[i] = false;                                  //所有齿轮未使用
        for (int i = 1; i<= N; i++) {                     //输入每个齿轮的名称和齿数
            cin >> ch[i] >> x;
            T[ch[i]] = x;
        }
            //最佳方案初始化：驱动分针轴和时针轴的方案为空；使用的转轴数和齿轮数足够大
    AnswerM = "X"; AnswerH = "X";
    AnswerShaft=100000000,AnswerGear=100000000;
    Dfs("*",1,0);                                         //从驱动轴出发，计算最佳方案
//若不存在驱动分针轴的最佳方案，则输出失败信息；否则输出驱动分针轴和时针轴的最佳方案
    if (AnswerM == "X") cout << "Trial " << ++Case<< " IS IMPOSSIBLE\n";
        else cout<<"Trial"<<++Case<<"\nMinutes:"<<AnswerM<<"\nHours:"<<AnswerH <<"\n";
    }
}
```

试题 3–7　朝圣（Pilgrimage）

【问题描述】

Jack 和他的朋友们正在做长途徒步旅行，他们沿着古老的朝圣之路，从 Vézelay 出发，前往 Santiago de Compostela。Jack 管理大家的经费。他的管理很简单：无论何时，只要有需要集体出钱的时候（如 60 欧元），他就会支付这笔钱，然后在他的小册子上记录：PAY 60。

在需要的时候，Jack 也会要求包括他自己在内的每个组员一起支付（如 50 欧元），然后在他的小册子上记录：COLLECT 50。如果他们的小组有 7 个人，那么他就一共收集到 350 欧元。

然而，Jack 小组的有些组员不能全程参与这次旅行，所以他的小组人员有时会扩大，有时则会减少。Jack 是如何在有人加入或退出时管理集体的经费呢？例如，假设组内有 7 人，并且 Jack 手中有 140 欧元现金，即平均每个人有 20 欧元。若其中两个人离开，每个人就收到 20 欧元，然后 Jack 会在小册子中记录：OUT 2。如果在同样情况下，有 3 个新人加入，那么他们每人都要交付 20 欧元，然后 Jack 记录：IN 3。

在上面的例子中，现金的数量恰好可以被整除。巧合的是，在整个旅行过程中一直是这样，因此 Jack 从来都不需要考虑分数运算。

临近旅行的终点，所有的人都加入了 Jack 的小组，因为没有人愿意错过旅行的光荣结束。这时，Jack 试图去回忆在旅行中的每一段小组里有多少个人，但他不记得了。

给出 Jack 的小册子的一页，你可以计算出小组在这一页开始时的人数吗？

输入：

输入包含多组测试用例。每组测试用例都是 Jack 的小册子中的连续的若干行。每个测试用例的第一行给出一个整数 N（$0 < N \leqslant 50$），然后给出 N 行，每行格式如下：

```
<关键字> <数字>
```

其中<关键字> = PAY | COLLECT | IN | OUT；而<数字>为一个正整数，满足如下约束：

```
IN k          k ⩽ 20
OUT k         k ⩽ 20
COLLECT k     k ⩽ 200
PAY k         k ⩽ 2000
```

输入最后一组测试用例后给出一行，包含一个整数 0。

输出：

对于每组测试用例，输出一行，给出该小组在开始时的人数。这一行为：

- 如果测试数据是不一致的，则输出 IMPOSSIBLE。
- 如果人数是唯一确定的，则输出一个整数，表示开始时的人数。
- 如果有多解但答案是有限的，则输出若干个数字，用空格隔开，按照升序输出，表示可能的小组人数。
- 如果解答数是无限多的，则以 SIZE >= N 的格式，输出 Jack 小组人数的下界；这里要注意，不等式 SIZE >= 1 总是成立的，因为至少有 Jack 本人在旅行。

样例输入	样例输出
5 IN 1 PAY 7 IN 1 PAY 7 IN 1 7 IN 1 COLLECT 20 PAY 30 PAY 12 IN 2 PAY 30 OUT 3 3 IN 1 PAY 8 OUT 3 1 OUT 5 0	IMPOSSIBLE 2 3 7 SIZE >= 6

试题解析

题目中，对于人数的限制有两条：

限制 1：在 IN 和 OUT 发生时，钱数是人数的倍数。

限制 2：在任何时候，组内至少有一人。

以下分别分析 4 种操作。

1. 分析 COLLECT 操作

COLLECT 操作是要求组内的每个人缴纳一定费用。对于限制 1，由于收上来的钱一定是人数的倍数，COLLECT 操作并不影响总钱数与当前人数的倍数关系；对于限制 2，COLLECT 操作与人数没有任何关系。因此，我们可以直接忽视 COLLECT 操作。

2. 分析 IN 和 OUT 操作

两个操作的本质是相同的，我们可以同时考虑。对于限制 1， IN 和 OUT 在发生前后钱数都是人数的倍数；对于限制 2，要求在累计 OUT 最多的时候，至少剩下一个人，根据这个条件可以得到人数的下界。

3. 分析 PAY 操作

为了简便起见，我们可以把连续的多个 PAY（在忽略 COLLECT 操作以后）压缩到一起，例如 PAY 30 和 PAY 50 可以压缩为一次 PAY 80。分析题意及两条限制，这样压缩并没有任何问题，因此不妨压缩起来以方便我们解题。这里要注意如下 3 个问题：

1）在 IN 或 OUT 之前的 PAY 操作是应该忽视的，因为在初始时钱数当然可以不是人数的倍数，只是在 IN 或 OUT 之前的那次 PAY 恰好使得倍数关系成立。因此，在 IN 或 OUT 发生前的 PAY 是没有意义的。

2）在最后一个 IN 或 OUT 以后发生的 PAY 操作也是应该忽视的，这个操作列表仅仅是一部分，在这一次 PAY 操作以后当然可以跟着若干 PAY 操作使得倍数关系重新成立。因此，最后的 PAY 操作是没有意义的。

3）考虑其他的 PAY。假设在 IN 或 OUT 操作后，发生了 PAY X 操作。由于在 IN 或 OUT 操作以后，倍数关系一定成立，因此 PAY 操作以后必然跟着一个 IN 或 OUT 操作（COLLECT 操作已经被忽略），此时倍数关系也应该成立。由此可得 PAY X 并不影响人数和钱数的倍数关系，不难知道 X 就是人数的倍数。X 是一个有限数，它的因子也一定是有限个。因此，可以根据 PAY 操作来确定人数的候选值。

综上所述，算法大致如下：

首先，忽略所有的 COLLECT 操作和无效的 PAY 操作，并将连续的 PAY 合并为一个。接下来分析 PAY 操作：

若还存在 PAY 操作，则说明有有限个解，从中抽取满足条件的所有 PAY，并且在 OUT 最多时至少保持一人的若干解输出即可（无解，则输出 IMPOSSIBLE）。

若不存在 PAY 操作，则说明只要满足在人数最低时至少有一人的限制即可，此时求出其下限，输出 SIZE >= N。

程序清单

```
#include <stdio.h>
#include <cstring>
const int MaxP = 2000 * 50;              //队伍人数的上限
```

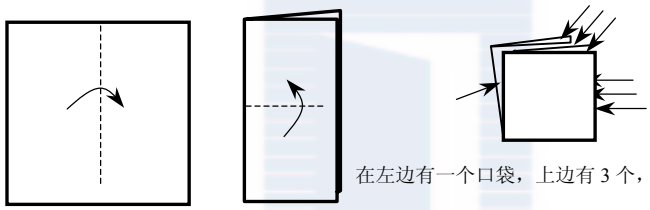
```
char s[20];                                  //操作的关键字
int n, m, i, j, k, P, sum, delta, min;       //Jack 小册子中的行数为 n，当前人数相比初始人数的变
                                             //化为 delta，队伍人数在低谷时比初始人数的变化为
                                             //min，PAY 操作的次数为 P
int list[50][2];                             //记录删除、合并后的操作序列，其中 list[i][0]为第
                                             //i 个操作的操作类型(0 : in  1 : out  2 : pay)，
                                             //list[i][1]为操作数
int a[MaxP+1];                               //队伍满足 PAY 限制的次数
int main()
{
   while (scanf("%d", &n) && n)              //反复输入 Jack 小册子中的行数
   {
      sum = 0; P = 0;                        //list 表长和 PAY 操作的数字与初始化
      for (i=0; i<n; i++)                    //输入每行信息
      {
         scanf("%s %d", s, &m);              //输入操作的关键字和数字
         if (s[0]=='C') continue;            //忽视所有的 COLLECT 操作
         if (s[0]=='P') P += m;
         else {                              //将 PAY 累计起来。以 list 记录删除、合并后的操作序
                                             //列,其中 list[i][0]表示第 i 个操作的操作类型,list
                                             //[i][1]表示操作数
            if (P && sum) { list[sum][0] = 2; list[sum][1] = P; sum ++; }
         //在前面出现过 IN 或 OUT 操作，且正在输入新的 IN 或 OUT 操作时，中间的 PAY 操作才是有效的
            if (s[0]=='I') list[sum][0] = 0;
            else list[sum][0] = 1;
            list[sum][1] = m;
            sum ++;
            P = 0;
         }
      }
      memset(a, 0, sizeof(a));               //a[x]=k 表示初始时有 x 个人的队伍可以满足 k 次 PAY 限制。
                                             //最后的答案应该满足所有的 PAY 限制
      delta = 0; min = 0; P = 0;             //delta 记录当前人数相比初始人数的变化，min 记录队
                                             //伍人数在低谷时比初始人数的变化，P 记录有几次 PAY 操作
      for (i=0; i<sum; i++)
      {
         if (list[i][0]==0) delta += list[i][1];     //更新 delta 以及 min
         if (list[i][0]==1) delta -= list[i][1];
         if (min>delta) min = delta;
         if (list[i][0]!=2) continue;
         P ++;
         for (j=1; j<=list[i][1]; j++)
         if (j>delta && list[i][1]%j==0)
         a[j-delta]++;                       //将 PAY X 操作中 X 的所有因子 Y 还原到初始人数 Z，标记 Z
                                             //能多满足一次 PAY 限制
      }
         if (P==0)
         printf("SIZE >= %d\n", 1 - min);    //若没有有效的 PAY 操作，则输出人数下限即可
         else {
            k = 0;                           //可能的小组人数初始化
            for (i=1-min; i<=MaxP; i++)
            if (a[i]==P)                     //按照递增顺序枚举满足所有 PAY 操作的小组人数
            {
               if (k) printf(" ");           //输出最后一个满足条件的小组人数
               k = i;
               printf("%d", k);
```

```
                }
                    if (!k) printf("IMPOSSIBLE");
                                        //若始终没有满足条件的小组人数，则输出 IMPOSSIBLE
                    printf("\n");
            }
        }
    }
```

试题 3–8　口袋数（Pockets）

【问题描述】

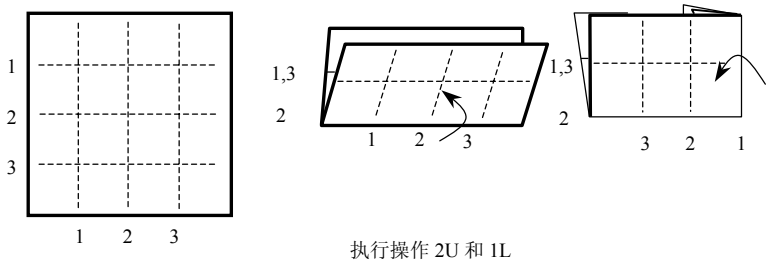
折纸，或者说折叠纸张艺术，经常使用到纸做的“口袋”，特别在用很多张纸制作复杂的模型的时候，有时要把某一片折好的纸套入另一张纸的口袋里面。在本题中，请计算在一张平的纸上折叠正方形所产生的口袋数。所谓口袋，就是指任何一个可以从被折叠的纸片的边上看到的开口（在纸的两个面之间）。这里要注意，因为一个开口的每一条边都算做一个口袋，从边界上的开口可能数出若干个口袋。图 3.8-1 就是一个例子，可以看到有 3 个口袋出自“中间”的那个开口（在纸的第二层和第三层之间）。



口袋的一个例子

图　3.8-1

假设纸片一开始被放置在一个平面上，并且在折纸的过程中不会完全离开该平面。折纸沿着水平方向或垂直方向进行。折叠线仅沿着间距相等的折痕线，在每个方向上等距离地分出了 N 条线。在还没开始折纸时，折痕和边从上到下、从左到右编号，如图 3.8-2 所示。每次折叠都会使纸片变成更小的矩形，而在最后一次折纸以后就产生一个单位长度为 1 的正方形。具体的折叠以折痕和方向来描述。比如说，2 U 意味着将底边沿着折痕 2 向上（Up）折叠；1 L 意味着将右边沿着折痕 1 向左（Left）折叠（如图 3.8-2 所示）。在若干次折叠后，折痕可能重叠在一起（例如，图 3.8-2 中的折痕 1 和折痕 3）。此时其中的任何数字都可以用来表示沿着那条线进行的折叠（即在图 3.8-2 中，在第一次折叠以后，1 D 和 3 D 的执行效果是相同的）。你需要计算最后那个单位正方形上的口袋数。一旦进行了一次折叠，这次折叠就不可以被撤销。所有的折痕都不考虑纸片的厚度，从上到下经过纸片的所有表面。



执行操作 2U 和 1L

图　3.8-2

输入：

输入包含多组测试用例。每组测试用例的第一行是两个整数 *N* 和 *K*，其中 *N* 表示水平和垂直方向上折痕的个数。折痕从上到下、从左到右以 1，2，…，*N* 标号。*K* 为折纸的次数。*N* 和 *K* 均不大于 64。

接下来给出 *K* 次折纸的描述。每个描述包括折痕编号 *C* 和折纸方向 U，D，L 或 R（分别表示上、下、左、右），以空格分开。每个折痕描述前后也以空格隔开。

每组测试用例均使得经过折叠后的纸片为 1 单位长度的正方形。

输入最后一组测试用例后给出一行，包括两个整数 0。

输出：

对于每组测试用例，输出测试用例编号，以及最后的单位正方形上的口袋数。具体格式参见样例输出。

样例输入	样例输出
1 2 1 R 1 U 3 5 2 U 1 L 3 D 3 R 2 L 0 0	Case 1: 7 pockets Case 2: 17 pockets

试题解析

本题是一道模拟题，通过模拟折纸的步骤，计算最终的“口袋”数。为了便于理解，读者不妨找一张纸，根据样例输入，实际折一下。

首先，考虑口袋是怎么形成的：在矩形纸片的某条边上，有若干张纸片重叠在一起。最终的口袋数＝“外露”的纸片数–1。如图 3.8-3 所示，考虑底边，“外露”的纸片有两张：最里面的纸片 A 和折叠起来的纸片 B。由题意不难知道被 B 折在内部的纸片 C 对于口袋数是没有贡献的。计算最终的口袋数，不妨从“外露”的纸片数入手，考虑它与哪些相关。

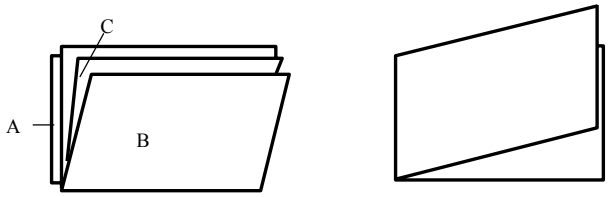


图　3.8-3

由于本题中的操作仅有“折叠”，考虑这一操作造成的影响。

如图 3.8-3 所示，纸片折叠时，会使纸片发生两种性质的折叠：

第 1 种折叠方式：纸片的某一条内部折痕外露，对应上图中纸片的左侧部分。

第 2 种折叠方式：纸片的两个不同的折痕重合在一起，对应上图中纸片的右侧、上侧和下

侧部分。

考虑每一单位长度的折痕。若能正确统计所有外部折痕处外露的纸片数，就能轻松得到最终的口袋数。是否只要知道所有外部折痕上的外露纸片数，就能得到答案了呢？

分别分析两种性质的折叠，对于第 2 种折叠方式，的确只要知道上一步外露折痕上的相关信息，就能得到新图形上的相应信息。但对于第 1 种折叠方式，由于是将内部折痕外翻，因此必须知道这条折痕上的纸片信息，才能确定新的外部折痕上的外露纸片数。那么需要记录内部折痕上的哪些信息呢？

图 3.8-4 描述了一个内部折痕上的一个可能的情况：当纸片自左向右折叠时，外露纸片有两张，即最下边的纸片及深灰色的长纸片；而当纸片自右向左折叠时，外露纸片有 3 张，即最上方的两个纸片及浅灰色的长纸片。

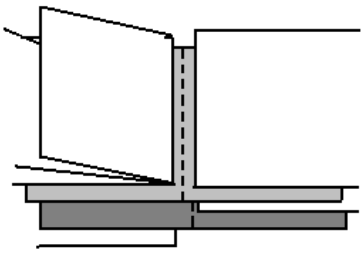


图 3.8-4

不难发现，夹在两张长纸片之间的纸片，必然不会成为外露纸片，因为无论往哪个方向折叠，都会被某一条长纸条夹在中央；已经夹在某纸片之间的纸片也必然不会外露，因为不可能拆开已经折叠好的纸片。由此可知，上面这些量均不需要记录，可能需要记录的量只有最下方的长纸片以下的纸片数 Down，以及最上方的长纸片以上的纸片数 Up。

由于无论向哪个方向折叠，这条折痕外翻后的外露纸片数均为最下方的长纸片以下的纸片数 Down，似乎只要记录 Down 即可。但考虑如下例子：沿着第九条折痕向右折叠一个 10×10 的纸片，此时有一大块纸片被翻到右边以后上下颠倒，Down_New＝Up，Up_New＝Down。因此，这两个量均需要记录。

至此，可以得到本题算法的大致思想：统计每条外部折痕上的外露纸片数 O（Out），统计每条内部纸片上方以及下方的外露纸片数 U（Up）及 D（Down），在每次折叠时，对这些值进行维护，就可以得到最终的口袋数。

首先，假设已经知道所有折痕上的相关信息（初始时，所有内部折痕上、下的纸片数为 0，外部折痕的纸片数为 1，如图 3.8-5 所示）。考虑某一次折叠，可能有以下几种情况：

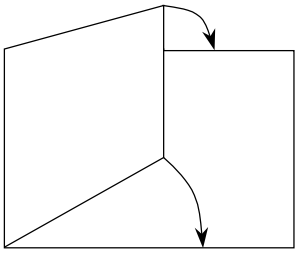
1）某外部折痕折在空处。此时，$O_{New}=O_{Old}$。

2）某外部折痕折 A 到某内部折痕 B 上。此时，需要考虑纸片折叠的方向，若位于内部折痕 B 上方，则 $U_{New}=U_B+O_A$，$D_{New}=D_B$；若位于内部折痕 B 下方，则 $D_{New}=D_B+O_A$，$U_{New}=U_B$。

3）某两条外部折痕 A 和 B 重叠到一起。此时，直接将纸片数相加即可：$O_{New}=O_B+O_A$。

4）某内部折痕 A 外露。此时，无论如何折叠，均将最下方的若干纸片外翻，即 $O_{New}=D_A+1$。

5）某内部折痕 A 与内部折痕 B 重叠到一起。不妨设 A 折叠到 B 上方。此时，$U_{New}=U_A$，$D_{New}=D_B$。



纸片折叠

图　3.8-5

至此，一个完整的算法浮出水面：计算所有折痕的 O、U 及 D 值，并在每一次折叠时，分别对上述情况进行维护，最终将剩下的 4 个折痕上的 O 值累加并减掉 4 即可得到最终的口袋数。

程序清单

```
#include <stdio.h>
#include <string>
struct info                              //折痕的结构类型
{
   int sign;                             //sign=1 表示是外部折痕，sign=0 表示不是外部折痕
   int up, down, uCover, dCover;         //若该折痕是内部折痕，则以 up、down 记录上下分别有多少外
                                         //露纸片
   int sum;                              //若该折痕是外部折痕，则记录有几张外露纸片
};
const int MaxN = 64;              //同一方向上折叠次数的上限
int Cases, N, n, m, K, ans;       //测试用例编号为 Cases，原始纸片的大小为 N，折叠到当前状态下
                                  //纸片的大小为 n*m，垂直方向上尚待折叠的次数为 K，口袋数为 ans
char dir;                         //方向字符
info hor[MaxN + 2][MaxN + 1], ver[MaxN + 1][MaxN + 2];
      //折痕编号应为[1, n-1]，这里折痕 0 记录最上（左）方的折痕，折痕 n 记录最下（右）方的折痕，
      //其中 hor[][]记录每一个水平折痕上的相关信息，ver[][]记录每一个竖直折痕上的相关信息
info Nhor[MaxN+2][MaxN+1], Nver[MaxN+1][MaxN+2];//记录折叠一次以后，新的折痕信息
int Lhor[MaxN + 1], Lver[MaxN + 1];     //Lhor[i]=k 表示编号为 i 的水平折痕在当前状态下对
                                        //应自上而下第 k 条折痕；同理定义 Lver
void init()                             //初始化
{
   int i;
   memset(hor, 0, sizeof(hor));
   memset(ver, 0, sizeof(ver));
   for (i=0; i<n; i++)                  //标记所有折痕为外部折痕，外露纸片数为 1
   {
      hor[0][i].sign = hor[n][i].sign = 1;
      hor[0][i].sum = hor[n][i].sum = 1;
      ver[i][0].sign = ver[i][n].sign = 1;
      ver[i][0].sum = ver[i][n].sum = 1;
   }
   for (i=0; i<=n; i++)
   Lhor[i] = Lver[i] = i;               //按照自上而下、由左而右顺序为折痕编号
}
int get(int x, int s, int line, char dir)   //当前方向共有 s 条折痕，将第 x 条折痕沿着第
                                            //line 条折痕按照 dir 方向折叠，求这条折痕在新
                                            //图形中对应哪条折痕
```

```
{
    if (dir=='D' || dir=='R')
    return labs(x - line);
    else if (line+line>=s)
    {
        if (x<=line)
        return x;
        return line - (x - line);
    } else {
        if (x>=line)
        return s - x;
        return (s - line) - line + x;
    }
}
void work()                                    //模拟当前一步折叠
{
    int i, j, k, nNew, mNew, x, y;
    int line;
    char dir;
    scanf("%d%c", &line, &dir);                //输入当前折纸的折痕编号和折纸方向
    while (dir!='U' && dir!='D' && dir!='R' && dir!='L')
    scanf("%c", &dir);                         //略去多余空格

    //判断沿第几条折痕折叠，计算新的纸片大小，并按照折叠后的情况给所有折痕重新编号
    if (dir=='U' || dir=='D')
    {
           line = Lhor[line];
           mNew = m;
           if (line>n-line)
           nNew = line;
           else nNew = n - line;
           for (i=0; i<=N; i++)
           Lhor[i] = get(Lhor[i], n, line, dir);
    } else {
           line = Lver[line];
           nNew = n;
           if (line>m-line)
           mNew = line;
           else mNew = m - line;
           for (i=0; i<=N; i++)
           Lver[i] = get(Lver[i], m, line, dir);
    }
    memset(Nhor, 0, sizeof(Nhor)); memset(Nver, 0, sizeof(Nver));
    //记录新图形中有哪些折痕位于外部
    for (i=0; i<nNew; i++)
        Nver[i][0].sign = Nver[i][mNew].sign = 1;
    for (i=0; i<mNew; i++)
        Nhor[0][i].sign = Nhor[nNew][i].sign = 1;
    //每次折叠，除情况 4 外，其他均为上一状态的某些值相加，仅情况四中需将结果额外加一。这里先处理
    //加一操作，记其外露纸片数为 1，再将其他值累加进来
    if (dir=='U')
    for (i=0; i<mNew; i++)
    Nhor[nNew][i].sum = 1;
    if (dir=='D')
    for (i=0; i<mNew; i++)
    Nhor[0][i].sum = 1;
```

```
        if (dir=='L')
        for (i=0; i<nNew; i++)
        Nver[i][mNew].sum = 1;
        if (dir=='R')
        for (i=0; i<nNew; i++)
        Nver[i][0].sum = 1;

            for (i=0; i<=n; i++)
            for (j=0; j<m; j++)      //处理所有水平折痕
            {
               x = i; y = j;         //判断旧纸片上(i, j)-(i, j+1)折痕，在折叠后的编号
               if (dir=='U' || dir=='D')
               x = get(x, n, line, dir);
               else y = get(y, m, line, dir);
               if ((dir=='R' && j<line) || (dir=='L' && j>=line))
               y --;
               //坐标为(x, y)-(x, y+1)的纸片在折叠后，坐标可能为(x', y')-(x', y'-1)，此时新的 y
               //值应该是 y'-1 而不是 y'

            if (Nhor[x][y].sign)                    //考虑某条折痕经过折叠后变成了外部折痕
            {
               if (hor[i][j].sign)
               Nhor[x][y].sum += hor[i][j].sum;   //若本身就是外部折痕，则直接将外露纸片数相加；
                                                  //若本身是内部纸片，则最下方的外露纸片外翻
               else Nhor[x][y].sum += hor[i][j].down;
            } else {                              //考虑某条折痕经过折叠后为内部折痕
                  if (hor[i][j].sign)             //某外部折痕折叠后掉入内部
                  {
                     //分别处理原外部折痕上的纸片落入了新折痕的上方还是下方的两种情况
                     if ((i<line && dir=='U') || (i>=line && dir=='D'))
                     Nhor[x][y].down += hor[i][j].sum;
                     else Nhor[x][y].up += hor[i][j].sum;
                  } else {                        //若某内部折痕折叠后仍为内部折痕
                    if ((i<line && dir=='U') || (i>=line && dir=='D') || (j<line && dir=='L')
|| (j>=line && dir=='R'))
                     {     //在折叠至新折痕下方的情况下，维护外露纸片数，并记录这条折痕的下方已被长
                           //纸条覆盖
                            Nhor[x][y].down = hor[i][j].down;
                            Nhor[x][y].dCover = 1;
                               //若这条折痕的上方未被长纸条覆盖，则原折痕上方的外露纸片仍外露，将
                               //两个值累加起来
                            if (!Nhor[x][y].uCover)
                            Nhor[x][y].up += hor[i][j].up;
                        } else {
                               //与上面类似，处理折叠至折痕上方的情况
                            Nhor[x][y].up = hor[i][j].down;
                            Nhor[x][y].uCover = 1;
                            if (!Nhor[x][y].dCover)
                            Nhor[x][y].down += hor[i][j].up;
                        }
                     }
                  }
        }
        for (i=0; i<n; i++)
        for (j=0; j<=m; j++)                        //与上面类似，处理所有竖直折痕
        {
```

```
            x = i; y = j;
               if (dir=='U' || dir=='D')
               x = get(x, n, line, dir);
               else y = get(y, m, line, dir);
            if ((dir=='U' && i>=line) || (dir=='D' && i<line))
            x --;

            if (Nver[x][y].sign)
            {
               if (ver[i][j].sign)
               Nver[x][y].sum += ver[i][j].sum;
               else Nver[x][y].sum += ver[i][j].down;
            }else {
               if (ver[i][j].sign)
               {
                  if ((j<line && dir=='L') || (j>=line && dir=='R'))
                  Nver[x][y].down+= ver[i][j].sum;
                  else Nver[x][y].up += ver[i][j].sum;
               }else {
                     if ((i<line && dir=='U') || (i>=line && dir=='D') || (j<line && dir=='L')
|| (j>=line && dir=='R'))
                     {
                        Nver[x][y].down = ver[i][j].down;
                        Nver[x][y].dCover = 1;
                        if (!Nver[x][y].uCover)
                        Nver[x][y].up += ver[i][j].up;
                     }else {
                     Nver[x][y].up = ver[i][j].down;
                      Nver[x][y].uCover = 1;
                     if (!Nver[x][y].dCover)
                     Nver[x][y].down += ver[i][j].up;
                  }
               }
            }
         }
         memcpy(hor, Nhor, sizeof(hor));                    //用折叠后的数据替换旧数据
         memcpy(ver, Nver, sizeof(ver));
         n = nNew; m = mNew;
      }
      int main()
      {
         while (scanf("%d %d", &n, &K) && (n || K))         //反复输入水平和垂直方向上折痕的个数，直
                                                            //至输入两个 0 为止
         {
            n ++; N = m = n;                                //以 N 记录原始纸片的大小，n、m 分别记录
                                                            //折叠到当前状态下纸片的大小
            init();                                         //初始化
            for (; K; K--)
            work();                                         //模拟每一步折叠
            //最后单位正方形上每条折痕上的口袋数为外露纸片数减一，将结果累加并输出
            ans = hor[0][0].sum - 1 + hor[1][0].sum - 1 + ver[0][0].sum - 1 + ver[0][1].sum
- 1;
            printf("Case %d: %d pockets\n", ++ Cases, ans);
```

```
        }
    }
```

试题 3-9　隔离度（Degrees of Separation）

【问题描述】

在这个世界中，人与人之间的联系越来越频繁，已经推断出世界上的任何一个人和其他人之间的隔离度不多于 6 个。请编写一个程序，在一个由人组成的网络中，找出人的最大的隔离度。

对于任何两个人，隔离度是联系两个人需要通过的关系的最少数目。对于一个网络，最大隔离度（Maximum Degree of Separation）是在所有的两人之间的隔离度中取的最大值。如果网络中的两个人之间没有关系连接，则称这个网络是不连通的。

如图 3.9-1 所示，一个网络可以用一个对称关系的集合来表示，每个关系连接两个人。一条线表示两个人之间的关系。网络 A 给出一个最大隔离度为 2 的网络的实例，网络 B 是不连通的。

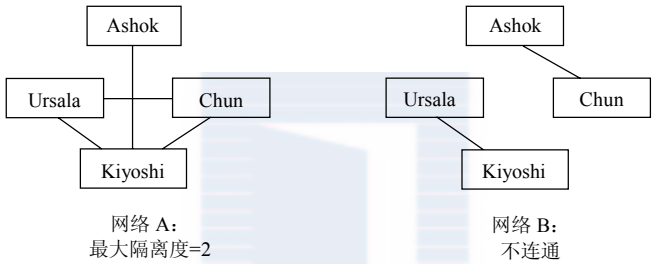


图　3.9-1

输入：

输入包含多组描述由人组成的网络的测试用例。每组测试用例的第一行是两个整数 P（$2 \leqslant P \leqslant 50$）和 R（$R \geqslant 1$），分别表示网络中包含的人数和网络中的关系数。接下来输入 R 对关系，每对关系由两个字符串组成，表示网络中这对关系所联系的两个人的名字。在本题中，名字是唯一的，且不包含空白字符。因为一个人可能与多个人有关系，所以同一个名字可能在一个测试用例中出现若干次。

输入最后一组测试用例后给出一行，包含两个 0，表示输入结束。

输出：

对于每个测试用例，输出测试用例的编号和最大隔离度。如果该网络是不连通的，则输出“DISCONNECTED”。在每组测试用例处理以后输出一个空行。具体格式见样例输出。

样例输入	样例输出
4 4 Ashok Kiyoshi Ursala Chun Ursala Kiyoshi Kiyoshi Chun 4 2 Ashok Chun Ursala Kiyoshi 6 5 Bubba Cooter Ashok Kiyoshi Ursala Chun Ursala Kiyoshi Kiyoshi Chun 0 0	Network 1: 2 Network 2: DISCONNECTED Network 3: DISCONNECTED

试题解析

由于网络中至多 50 人，因此算法是很简单的：

若记相关的两人之间的路程为 1，其他人之间的路程为无穷大，则使用一次时间复杂度为 $O(n^3)$ 的 floyd 算法即求出所有人之间的最短路，最后从中取最大值即可。关键是要注意输入串的处理。

程序清单

```
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;
const int MaxP = 50;
int P,R,cases,i,j,k,sum,ans;    //网络的人数为 P、关系数为 R，测试用例编号为 cases，最大隔离度
                                //为 ans
string name[MaxP],s1,s2;        //name[]为姓名列表，下标对应该名字的节点序号，表长为 sum
int a[MaxP][MaxP];              //相邻矩阵
int find(string s)              //输入名字，返回节点序号。如果名字没出现过，则更新姓名列表
{
   int i;
   for (i=0;i<sum;i++)
   if (name[i]==s)
   return i;
   name[sum] = s; sum ++;
   return (sum - 1);
}
bool init()                     //输入当前测试用例
{
   scanf("%d %d",&P,&R);        //输入网络的人数和关系数
   if (P==0 && R==0)
   return false;                //若输入两个 0，则退出程序
   sum = 0;                     //不同名的人数初始化
   memset(a,0,sizeof(a));       //相邻矩阵初始化
   for (i=0;i<R;i++)            //输入 R 条关系，构造相邻矩阵
   {
      cin >> s1 >> s2;
      j = find(s1); k = find(s2);
      a[j][k] = a[k][j] = 1;
   }
   return true;
}

int main()
{
   cases = 0;
      while (init())            //反复输入测试用例，直至输入两个 0 为止
      {
         for (k=0;k<P;k++)      //使用 floyd 算法求任意节点对之间的最短路（两人间的隔离度）
            for (i=0;i<P;i++)
               for (j=0;j<P;j++)
                  if (a[i][k] && a[k][j])
```

```
                    if (a[i][j]==0 || a[i][j]>a[i][k]+a[k][j])
                    a[j][i] = a[i][j] = a[i][k] + a[k][j];
            ans = 0;                //最大隔离度初始化
            for (i=0;i<P;i++)       //若出现不可达的节点对，则图不连通；否则最大隔离度为所有节点
                                    //对之间最短路长的最大值
               for (j=i+1;j<P;j++)
                  if (a[i][j]==0)
                  ans = P + 10;
                  else if (a[i][j]>ans) ans = a[i][j];
               if (ans>P)
               printf("Network %d: DISCONNECTED\n\n",++cases);       //输出结果
               else printf("Network %d: %d\n\n",++cases,ans);
      }
   }
```

试题 3–10　通信路线（Routing）

【问题描述】

随着越来越多的公司或人通过网络来交流，安全通信的问题也越来越被人们重视。因特网密码协议公司（Internet Cryptographic Protocol Company，ICPC）专门从事网络上的安全商务交易。ICPC 开发的软件系统以一种很独特的方式在网络上进行安装。

像因特网这样的网络是可以用一个有向图来建模的：节点表示机器或路由器，边则对应直接的连接，数据沿着边的方向传输。两个节点之间的通信，是将数据沿着有向路径从一个节点传输到另一个节点，然后从另一个节点传输回第一个节点。

图 3.10-1 对应第一个输入样例。从图中可以看出，从节点 x 到节点 y 的箭头意味着从 x 到 y 可以直接通信，但不表示从 y 到 x 可以直接通信。若将软件系统在节点 1、2、7 和 8 安装，那么节点 1 与节点 2 互相之间就可以通信。存在其他某些配置也可以使两个节点互相通信，但上述方法花费最小。

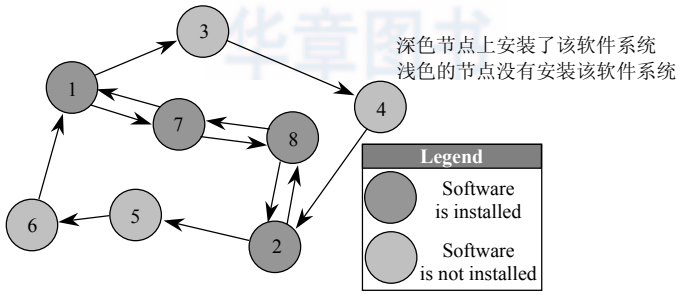


图　3.10-1

为了进行安全的交易，ICPC 系统软件要求安装软件不仅要在两个需要通信的节点安装，而且在连接这两个节点的通信路径上所有的中间节点都要安装。因为 ICPC 是根据软件安装的数量来向客户收费的，因此要开发一个程序，对于任何的网络及通信节点对，应能找到最便宜的安装软件的方法，使得给定的两个节点之间能互相通信。

输入：

输入包含多组测试用例。每组测试用例的第一行是两个整数 N 和 M（$2 \leqslant N \leqslant 100$），分别表示网络的节点数和边数。网络中的节点标号为 1，2，…，N，其中节点 1 和节点 2 是需要通

信的节点。然后给出 M 行，每行给出两个整数 X 和 Y（$1 \leqslant X, Y \leqslant N$），表示网络中有一条从 X 到 Y 的有向边。

输入最后一组测试用例后给出一行，包括两个 0，表示输入结束。

输出：

对于每组测试用例，按测试用例的顺序输出编号，然后输出最少需要安装软件系统的数量，使得存在两条仅经过已安装软件系统的节点的路径，分别从节点 1 到节点 2、节点 2 到节点 1（这里要注意，一个节点可以同时出现在两条路径中，而一条路径可以不包含所有的节点）。计算安装软件的数量应包括节点 1 和节点 2。

若节点 1 和节点 2 不能互相通信，输出 IMPOSSIBLE。

具体格式参见样例输出，每组测试用例处理后，要输出一个空行。

样例输入	样例输出
8 12 1 3 3 4 4 2 2 5 5 6 6 1 1 7 7 1 8 7 7 8 8 2 2 8 2 1 1 2 0 0	Network 1 Minimum number of nodes = 4 Network 2 IMPOSSIBLE

试题解析

本题要求一个从节点 1 到节点 2 节再返回 1 的路径，要求路径上不同的点最少。

若不考虑重复的点所降低的花费，那么本题就是简单的最短路径问题。考虑到重复经过某些点不增加总花费，可以由重复点入手解决问题。

假设最终的路径形如：

1→a1→A→b1→2；

1←a2←A←b2←2。

其中，a1 为 1→‥→A 的子路径；b1 为 A→‥→2 的子路径；a2 为 1←‥←A 的子路径；b2 为 A←‥←2 的子路径，A 为重复经过的点。

若 a1 与 b2、a2 与 b1 中没有重复的点，那么最终答案就是节点 1 到节点 A 再返回节点 1、节点 A 到节点 2 再返回节点 A 的路径中不同的节点的个数。此时问题通过点 A 分解为两个部分。

但 a1 与 b2 之间很可能有重复的节点，此时最终的路径形如：

1→a11→B→a12→ A → b1 →2；

1← a2 ←A←b21←B←b22←2。

不难发现，路径 a12 与路径 b21 都是由 B 到 A 的子路径，因此，花费最小的情况下必然有 a12＝b21。

此时，上述路径可以表示为：

1→a11→B→C→…→A→b1 →2；

1←a2 ←A←…←C←B←b22←2。

若 a11 与 b22、b1 与 a2 之间仍有重复的点，则采取相同的步骤，最终保证它们之间不存在重复的点。

通过上述分析，可以确定最终路径的形式如下：

1→…→A→…→B→C→D→…→E→…→F→…→2；

1←…←A←…←D←C←B←…←E←…←F←…←2。

即若干“块”重复的点将两条路径分成了互不重复的部分。

确定了路径的形式，就可以尝试求出具体路径。

首先，不妨以 ans[x][y]表示节点 1‥→节点 x 的路径与节点 y‥→节点 1 的路径中不重复的点的最小数目。不难知道 ans[2][2]即为所求答案。

为了便于编写代码，我们采用了递推的方式，假设 ans[x][y]的值已知，用 ans[x][y]的值来更新其他值。

递推时有如下 5 种情况：

1）x 的下一个点不是重复点。记 x 的下一个点为 i，则必然有 i 与 x、y 都不相等。此时，ans[i][y]=min{ans[i][y], ans[x][y] + 1}。

2）y 的上一个点不是重复点。与情况 1 类似，ans[x][i]=min{ans[x][i], ans[x][y] + 1}。

3）x 与 y 均为重复点，类似于上面路径中的 B 与 D。此时，ans[y][x]=min{ans[y][x], ans[x][y] + dis[x][y] −1}，其中 dis[x][y]表示由 x 到 y 的最短路长。

4）x 是重复点，且 y 的上一个点是 x。此时，ans[x][x]=min{ans[x][x], ans[x][y]}。

5）与情况 4）类似，y 是重复点也是 x 的下一个点。此时，ans[y][y]=min{ans[y][y], ans[x][y]}。

根据上述 5 种情况，加上初值 ans[1][1]=1，就可以用递推的方式求出 ans[2][2]的值。

在递推时，应该从 ans 值最小的出发递推，类似 Dijkstra 求最短路径。本题采用线段树来求 ans 的最小值并维护。

程序清单

```
#include <stdio.h>
const int MaxN = 100;                  //节点数的上限
const int Max = 1000;
int Cases, n, m, i, j, k, x, y;        //测试用例编号为 Cases，网络的节点数为 n、边数为 m
int dis[MaxN][MaxN];                   //dis[x][y]表示从 x 到 y 的最短路径的长度
int tr[MaxN * MaxN * 3];               //线段树，用于求 ans 的最小值并在 ans 更新时进行维护
int ans[MaxN][MaxN];                   //ans[x][y]表示从 1‥→x、y‥→1 的路径中至少有几个不同
                                       //的点，存储在线段树节点 tr[t]中，其中节点编号 t=x*n+y，
                                       //代表区间[t/n，t%n]的最小值


void Make(int n, int x, int y)         //从线段树的根节点 n(对应区间[x,y])出发，将树中所有节点
                                       //的初值置为最大值 Max
{
   tr[n] = Max;
```

```
    if (x==y)
    return;
    Make(n + n, x, (x + y) / 2);
    Make(n + n + 1, (x + y) / 2 + 1, y);
}
void Change(int n, int x, int y, int ch, int data)
                //从线段树的根节点 n(对应区间[x,y])出发,将树中节点 ch 的大小更新为 data 并维护
{
    if (x==y)
    {
        tr[n] = data;
    } else if ( (x+y)/2 >= ch)
    {
        Change(n + n, x, (x + y) / 2, ch, data);
        if (tr[n+n]<tr[n+n+1])
        tr[n] = tr[n + n];
        else tr[n] = tr[n + n + 1];
    } else {
        Change(n + n + 1, (x + y) / 2 + 1, y, ch, data);
        if (tr[n+n]<tr[n+n+1])
        tr[n] = tr[n + n];
        else tr[n] = tr[n + n + 1];
    }
}
int Get_Min(int n, int x, int y)            //求线段树中最小值的位置
{
    if (x==y)
    return x;
    if (tr[n+n]<tr[n+n+1])
    return Get_Min(n + n, x, (x + y) / 2);
        else return Get_Min(n + n + 1, (x + y) / 2 + 1, y);
}
int main()
{
    Cases = 0;                              //测试用例编号初始化
    while (scanf("%d %d", &n, &m) && n)     //反复输入网络的节点数和边数
    {
    for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    dis[i][j] = Max;    //任意节点对的最短路长初始化，注意节点编号范围是 0 到 n-1 而不是 1 到 n，
                        //因此代码中所有节点编号均比题目中小 1
        for (i=0; i<m; i++)                 //输入每条有向边的两个端点，构造相邻矩阵
    {
        scanf("%d %d", &x, &y);
        dis[x - 1][y - 1] = 1;
    }
    for (i=0; i<n; i++)
    dis[i][i] = 0;      //使用 floyd 算法计算任意节点对之间的最短路长
    for (k=0; k<n; k++)
        for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (dis[i][j]>dis[i][k]+dis[k][j])
            dis[i][j] = dis[i][k] + dis[k][j];

        Make(1, 0, n * n - 1);              //将线段树中所有节点的初值置为最大值 Max
```

```
        Change(1, 0, n * n - 1, 0, 1);          //记 ans[0][0]的初值为 1，其他值为 Max
        for (i=0; i<n; i++)
        for (j=0; j<n; j++)
        ans[i][j] = Max;
        ans[0][0] = 1;
        while (tr[1]<Max)
        {
            x = Get_Min(1, 0, n * n - 1);       //求 ans 值最小的节点编号
            y = x % n; x /= n;                  //还原区间[x，y]
            if (x==1 && y==1)
            break;        //若求出节点 1 到节点 2 再返回 1 的路径上最少的不同点数，则退出循环对应于
                          //情况 3（x 与 y 均为重复点）
               if (x!=y && ans[y][x]<=Max && ans[x][y]+dis[x][y]-1<ans[y][x])
               {
                  ans[y][x] = ans[x][y] + dis[x][y] - 1;
                  Change(1, 0, n * n - 1, y * n + x, ans[y][x]);
               }

               for (i=0; i<n; i++) if (i!=x && i!=y)    //枚举与 x 和 y 不重复的每个点
               {
                  if (dis[x][i]==1 && ans[i][y]<=Max && ans[x][y]+1<ans[i][y])
                  { //对应于情况 1（x 的后续点不是重复点）
                     ans[i][y] = ans[x][y] + 1;
                     Change(1, 0, n * n - 1, i * n + y, ans[i][y]);
                  }
                  if (dis[i][y]==1 && ans[x][i]<=Max && ans[x][y]+1<ans[x][i])
                  { //对应于情况 2 （y 的前驱点不是重复点）
                     ans[x][i] = ans[x][y] + 1;
                     Change(1, 0, n * n - 1, x * n + i, ans[x][i]);
                  }
               }
               if (dis[x][y]==1 && ans[y][y]<=Max && ans[x][y]<ans[y][y])
               {  //对应于情况 4（x 是重复点且为 y 的前驱点）
                  ans[y][y] = ans[x][y];
                  Change(1, 0, n * n - 1, y * n + y, ans[x][y]);
               }
               if (dis[x][y]==1 && ans[x][x]<=Max && ans[x][y]<ans[x][x])
               { //对应于情况 5（y 是重复点且为 x 的后续点）
                  ans[x][x] = ans[x][y];
                  Change(1, 0, n * n - 1, x * n + x, ans[x][y]);
               }

               //标记已用来递推过的 ans 值
               ans[x][y] = Max + 1;
               Change(1, 0, n * n - 1, x * n + y, ans[x][y]);
            }
            printf("Network %d\n", ++ Cases);//输出测试用例编号
            if (ans[1][1]==Max)
            printf("Impossible\n\n");          //若未产生一条途经节点 0 和节点 1 的回路，则输出
                                               //无解信息；否则输出回路中至少有多少个不同的节点
            else printf("Minimum number of nodes = %d\n\n", ans[1][1]);
        }
}
```