# 第 4 章　最短路径问题

有向网或无向网中一个典型的问题是**最短路径问题**(Shortest Path Problem)。最短路径问题要求解的是：如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径，使得沿此路径各边上的权值总和(即从源点到终点的距离)达到最小，这条路径称为**最短路径**(Shortest Path)。

根据有向网或无向网中各边权值的取值情形及问题求解的需要，最短路径问题分为以下 4 种情形，分别用不同的算法求解。

(1) 求单源最短路径(边的权值非负)——Dijkstra 算法，所谓**单源最短路径**(Single Source Shortest Path)就是固定一个顶点为源点，求源点到其他每个顶点的最短路径。

(2) 求单源最短路径(边的权值允许为负值，但不存在负权值回路)——Bellman-Ford 算法。

(3) Bellman-Ford 算法的改进——SPFA 算法。

(4) 求所有顶点之间的最短路径(边的权值允许为负值，但不存在负权值回路)——Floyd 算法。

本章介绍这 4 个算法。注意，以上 4 个算法均适用于无向网和有向网，本章以有向网为例介绍这 4 个算法。另外，本章还介绍了求最短路径的算法思想在求解差分约束系统中的应用。

## 4.1　边上权值非负情形的单源最短路径问题——Dijkstra 算法

### 4.1.1　算法思想

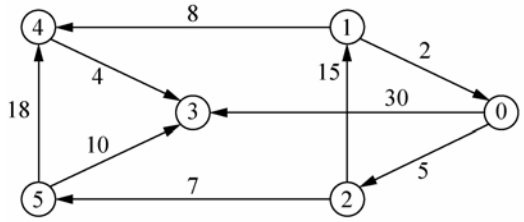
问题的提出：给定一个带权有向图(即有向网)$G$ 和源点 $v_0$，求 $v_0$ 到 $G$ 中其他每个顶点的最短路径。限定各边上的权值大于或等于 0。

例如，在图 4.1(a)中，假设源点为顶点 0，则源点到其他顶点的最短距离分别如下。

顶点 0 到顶点 1 的最短路径距离是：20，其路径为 $v_0 \rightarrow v_2 \rightarrow v_1$。

顶点 0 到顶点 2 的最短路径距离是：5，其路径为 $v_0 \rightarrow v_2$。

顶点 0 到顶点 3 的最短路径距离是：22，其路径为 $v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_3$；



(a) 有向网

$$\mathbf{Edge}=\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

(b) 邻接矩阵

图 4.1　Dijkstra 算法：有向网及其邻接矩阵

顶点 0 到顶点 4 的最短路径距离是：28，其路径为 $v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_4$；

顶点 0 到顶点 5 的最短路径距离是：12，其路径为 $v_0 \rightarrow v_2 \rightarrow v_5$。

这些最短路径距离及对应的最短路径是怎么求出来的？

为求得这些最短路径，Dijkstra 提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从源点 $v_0$ 到其他各顶点的最短路径全部求出为止。

例如在图 4.1(a)中，要求顶点 0 到其他各顶点的最短距离分别是多少。其求解过程如图 4.2 所示。具体如下。

| 源点 | 终点 | 最短路径 | | | | 最短路径长度 | | | |
|---|---|---|---|---|---|---|---|---|---|
| $v_0$ | $v_1$ | — | $v_0 \rightarrow v_2 \rightarrow v_1$ | | | ∞ | 20 | | |
| | $v_2$ | $v_0 \rightarrow v_2$ | | | | 5 | | | |
| | $v_3$ | $v_0 \rightarrow v_3$ | | $v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_3$ | | 30 | | 22 | |
| | $v_4$ | — | | $v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_4$ | $v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_4$ | ∞ | ∞ | 30 | 28 |
| | $v_5$ | — | $v_0 \rightarrow v_2 \rightarrow v_5$ | | | ∞ | 12 | | |

图 4.2　Dijkstra 算法的求解过程

(1) 首先求出长度最短的一条最短路径，即顶点 0 到顶点 2 的最短路径，其长度为 5，其实就是顶点 0 到其他各顶点的直接路径中最短的路径($v_0 \rightarrow v_2$)。

(2) 顶点 2 的最短路径求出来以后，顶点 0 到其他各顶点的最短路径长度有可能要改变。例如从顶点 0 到顶点 1 的最短路径长度由∞缩短为 20，从顶点 0 到顶点 5 的最短路径长度也由∞缩短为 12。这样长度次短的最短路径长度就是在还未确定最终的最短路径长度的顶点中选择最小的，即顶点 0 到顶点 5 的最短路径长度，为 12，其路径为($v_0 \rightarrow v_2 \rightarrow v_5$)。

(3) 顶点 5 的最短路径求出来以后，顶点 0 到其他各顶点的最短路径长度有可能要改变。例如从顶点 0 到顶点 3 的最短路径长度由 30 缩短为 22，从顶点 0 到顶点 4 的最短路径长度也由∞缩短为 30。这样长度第 3 短的最短路径长度就是在还未确定最终的最短路径长度的顶点中选择最小的，即顶点 0 到顶点 1 的最短路径长度，为 20，其路径为($v_0 \rightarrow v_2 \rightarrow v_1$)。

(4) 此后再依次确定顶点 0 到顶点 3 的最短路径($v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_3$)，其长度为 22；以及顶点 0 到顶点 4 的最短路径($v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_4$)，其长度为 28。

Dijkstra 算法的具体实现方法如下。

(1) 设置两个顶点的集合 $T$ 和 $S$。

① $S$ 中存放已找到最短路径的顶点，初始时，集合 $S$ 中只有一个顶点，即源点 $v_0$。

② $T$ 中存放当前还未找到最短路径的顶点。

(2) 在 $T$ 集合中选取当前长度最短的一条最短路径($v_0, \cdots, v_k$)，从而将 $v_k$ 加入到顶点集合 $S$ 中，并修改源点 $v_0$ 到 $T$ 中各顶点的最短路径长度；重复这一步骤，直到所有的顶点都加入到集合 $S$ 中，算法就结束了。

Dijkstra 算法的原理为：可以证明，$v_0$ 到 $T$ 中顶点 $v_k$ 的最短路径，要么是从 $v_0$ 到 $v_k$ 的直接路径；要么是从 $v_0$ 经 $S$ 中某个顶点 $v_i$ 再到 $v_k$ 的路径，如图 4.3 所示。

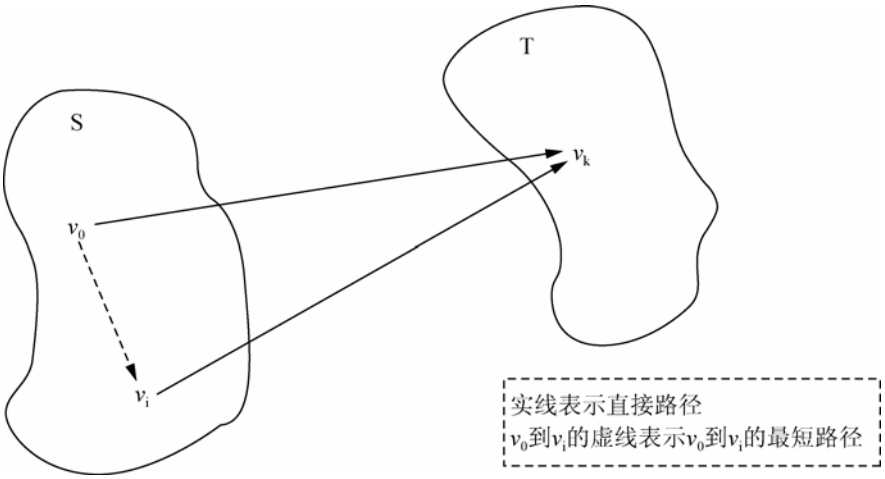



图 4.3　Dijkstra 算法的原理

### 4.1.2　算法实现

在 Dijkstra 算法里，为了求源点 $v_0$ 到其他各顶点 $v_i$ 的最短路径及其长度，需要设置 3 个数组。

(1) dist[$n$]：dist[$i$]表示当前找到的从源点 $v_0$ 到终点 $v_i$ 的最短路径的长度，初始时，dist[$i$]为 Edge[$v_0$][$i$]，即邻接矩阵的第 $v_0$ 行。

(2) $S$[$n$]：$S$[$i$]为 0 表示顶点 $v_i$ 还未加入到集合 $S$ 中，$S$[$i$]为 1 表示 $v_i$ 已经加入到集合 $S$ 中。初始时，$S$[v0]为 1，其余为 0，表示最初集合 $S$ 中只有顶点 $v_0$。

(3) path[$n$]：path[$i$]表示 $v_0$ 到 $v_i$ 的最短路径上顶点 $v_i$ 的前一个顶点序号。采用“倒向追踪”的方法，可以确定 $v_0$ 到顶点 $v_i$ 的最短路径上的每个顶点。

在 Dijkstra 算法里，重复做以下 3 步工作。

(1) 在数组 dist[$n$]里查找 $S$[$i$] != 1，并且 dist[$i$]最小的顶点 $u$。

(2) 将 $S$[$u$]改为 1，表示顶点 $u$ 已经加入进来了。

(3) 修改 $T$ 集合中每个顶点 $v_k$ 的 dist 及 path 数组元素值：当 S[$k$] != 1，且顶点 $u$ 到顶点 $v_k$ 有边(Edge[$u$][$k$]<MAX)，且 dist[$u$] + Edge[$u$][$k$] < dist[$k$]，则修改 dist[$k$]为 dist[$u$] + Edge[$u$][$k$]，修改 path[$k$]为 $u$。

在例 4.1 的代码中，可以很清晰地观察到这 3 步工作；并且对这 3 步工作，在例 4.1 的代码中特意用边框标明了。

因此 Dijkstra 算法的**递推公式**(求源点 $v_0$ 到各顶点的最短路径)为：

初始：dist [$k$]=Edge[$v_0$][$k$]，$v_0$ 是源点
递推：$u$=min{ dist[$t$] }，$v_t \in T$
　　$u$ 表示当前 $T$ 集合中 dist 数组元素值最小的顶点的序号；此后 $u$ 加入到集合 $S$ 中。
　　dist [$k$]=min{ dist [$k$], dist [$u$] + Edge[$u$][$k$] }, $v_k \in T$ (只修改 $T$ 集合中顶点的 dist 值)

**例 4.1**　利用 Dijkstra 算法求图 4.1(a)中顶点 0 到其他各顶点的最短路径长度，并输出对应的最短路径。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 $n$，然后输入每条边的数据。每条边的数据格式为：$u$ $v$ $w$，分别表示这条边的起点、终点和边上的权值。顶点序号从 0 开始计起。最后一行为−1 −1 −1，表示输入数据的结束。

**分析：**

Dijkstra 算法的 3 个数组：S、dist、path 初始状态如图 4.4(a)所示(源点为顶点 0)。

(1) S[0]为 1，其余为 0，表示初始时，$S$ 集合中只有源点 $v_0$，其他顶点都是在集合 $T$ 中。

(2) dist 数组各元素的初始值就是邻接矩阵的第 $v_0$ 行对应的元素值。

(3) 在 path 数组中，path[2]和 path[3]为 0，表示当前求得的顶点 0 到顶点 2 的最短路径上，前一个顶点是顶点 0；顶点 0 到顶点 3 的最短路径上，前一个顶点也是 0；其余元素值均为−1，表示顶点 0 到其余顶点没有直接路径。

在 Dijkstra 算法执行过程中，以上 3 个数组各元素值的变化如图 4.4(b)～图 4.4(f)所示。以图 4.4(b)为例加以解释。首先在 dist 数组的各元素 dist[$t$]中，选择一个满足 $S$[$t$]为 0 且 dist[$t$]取最小值，求得的最小值为 5，用粗体、斜体标明，对应顶点 $v_2$；然后将 $S$[2]修改成 1，用粗体、下划线标明，表示将顶点 $v_2$ 加入到集合 $S$ 中；最后修改 dist 数组和 path 数组中某些元素的值。

(1) 修改条件是：$S$[$k$]为 0，顶点 $v_2$ 到顶点 $v_k$ 有直接路径，且 dist[2]+Edge[2][$k$]<dist[$k$]。

(2) 修改方法是：将 dist[$k$]修改成 dist[2] + Edge[2][$k$]，将 path[$k$]修改成 2。所有修改过的值用粗体、下划线标明。

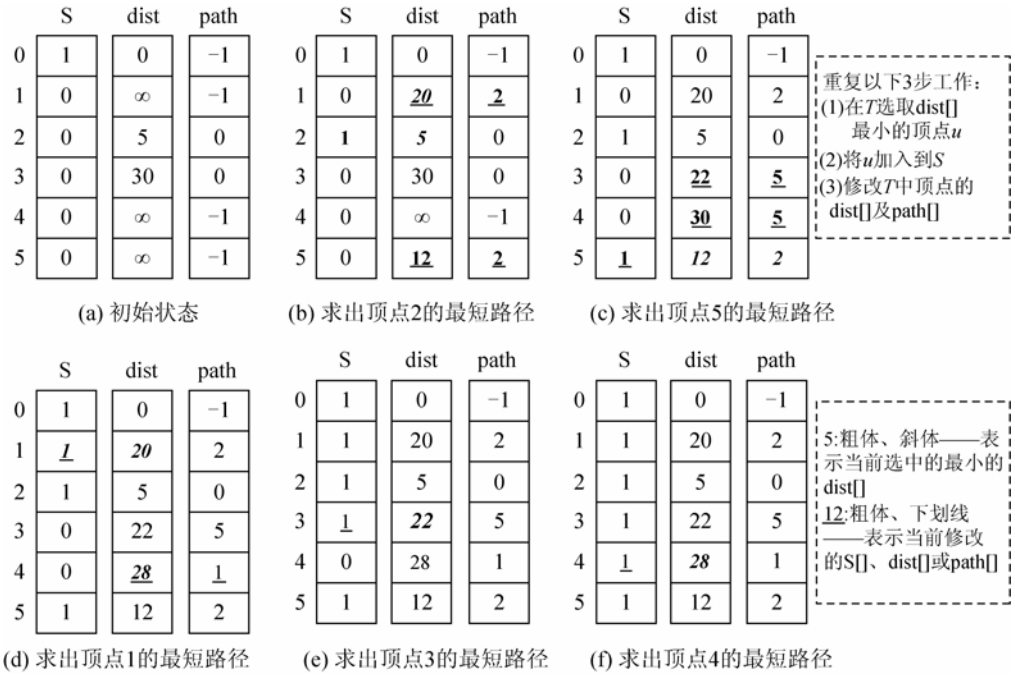



图 4.4　Dijkstra 算法的实现过程

当顶点 0 到其他各顶点的最短路径长度求解完毕后，如何根据 path 数组求解顶点 0 到其他各顶点 $v_k$ 的最短路径？方法是：从 path[$k$]开始，采用“倒向追踪”方法，一直找到源点 $v_0$。

举例说明：设 $k$ 为 4，则因为 path[4]=1，说明最短路径上顶点 4 的前一个顶点是顶点 1；path[1]=2，说明顶点 1 的前一个顶点是顶点 2；path[2]=0，说明顶点 2 的前一个顶点是顶点 0，这就是源点；所以从源点 $v_0$ 到终点 $v_4$ 的最短路径为：$v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_4$，最短路径长度为 dist[4]=28。

在下面的代码中，Dijkstra( int v0 )函数实现了求源点 $v_0$ 到其他各顶点的最短路径。如果要求顶点 0 到其他各顶点的最短路径，在主函数中只要调用 Dijkstra(0)即可。

另外，在主函数中，Dijkstra(0)执行完毕后，输出源点 0 到其他每个顶点的最短路径长度及最短路径上的每个顶点，shortest 数组的作用是在“倒向追踪”查找最短路径时用于存储最短路径上的各个顶点的序号。

代码如下：

```
#define INF 1000000          //无穷大
#define MAXN 20              //顶点个数的最大值
int n;                       //顶点个数
int Edge[MAXN][MAXN];        //邻接矩阵
int S[MAXN];                 //Dijkstra 算法用到的 3 个数组
int dist[MAXN];              //
int path[MAXN];              //
void Dijkstra( int v0 )      //求顶点 v0 到其他顶点的最短路径
{
    int i, j, k;             //循环变量
    for( i=0; i<n; i++ )
    {
        dist[i]=Edge[v0][i];  S[i]=0;
        if( i!=v0 && dist[i]<INF )  path[i]=v0;
        else path[i]=-1;
    }
    S[v0]=1;  dist[v0]=0;    //顶点 v0 加入到顶点集合 S
    for( i=0; i<n-1; i++ )   //从顶点 v0 确定 n-1 条最短路径
    {
        int min=INF, u=v0;
                 //选择当前集合 T 中具有最短路径的顶点 u
        for( j=0; j<n; j++ )                                    //(1)
        {
            if( !S[j] && dist[j]<min )
            {
                u=j;  min=dist[j];
            }
        }
        S[u]=1; //将顶点 u 加入到集合 S，表示它的最短路径已求得      //(2)
        //修改 T 集合中顶点的 dist 和 path 数组元素值                 //(3)
        for( k=0; k<n; k++ )
        {
            if( !S[k] && Edge[u][k]<INF && dist[u]+Edge[u][k]<dist[k] )
```

```
                {
                    dist[k]=dist[u]+Edge[u][k];  path[k]=u;
                }
            }
        }
    }
    int main( )
    {
        int i, j;           //循环变量
        int u, v, w;        //边的起点和终点及权值
        scanf( "%d", &n );  //读入顶点个数 n
        while( 1 )
        {
            scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
            if( u==-1 && v==-1 && w==-1 )  break;
            Edge[u][v]=w;                  //构造邻接矩阵
        }
        for( i=0; i<n; i++ )
        {
            for( j=0; j<n; j++ )
            {
                if( i==j ) Edge[i][j]=0;
                else if( Edge[i][j]==0 )  Edge[i][j]=INF;
            }
        }
        Dijkstra( 0 );          //求顶点 0 到其他顶点的最短路径
        int shortest[MAXN];     //输出最短路径上的各个顶点时存放各个顶点的序号
        for( i=1; i<n; i++ )
        {
            printf( "%d\t", dist[i] ); //输出顶点 0 到顶点 i 的最短路径长度
            //以下代码用于输出顶点 0 到顶点 i 的最短路径
            memset( shortest, 0, sizeof(shortest) );
            int k=0;                   //k 表示 shortest 数组中最后一个元素的下标
            shortest[k]=i;
            while( path[ shortest[k] ]!=0 )
            {
                k++; shortest[k]=path[ shortest[k-1] ];
            }
            k++; shortest[k]=0;
            for( j=k; j>0; j-- )
                printf( "%d→", shortest[j] );
            printf( "%d\n", shortest[0] );
        }
        return 0;
    }
```

该程序的运行示例如下。

**输入：**

```
6
0 2 5
0 3 30
1 0 2
1 4 8
2 5 7
2 1 15
4 3 4
5 3 10
5 4 18
-1 -1 -1
```

**输出：**

```
20      0->2->1
5       0->2
22      0->2->5->3
28      0->2->1->4
12      0->2->5
```

### 4.1.3　关于 Dijkstra 算法的进一步讨论

1. Dijkstra 算法的复杂度分析

算法的时间复杂度分析：在 Dijkstra 算法中，最主要的工作是求源点到其他 $n$–1 个顶点的最短路径及长度，要把其他 $n$–1 个顶点加入到集合 S 中来。每加入一个顶点，首先要在 $n$–1 个顶点中判断每个顶点是否属于集合 $T$，且最终要在 dist 数组中找元素值最小的顶点；然后要对其他 $n$–1 个顶点，要判断每个顶点是否属于集合 $T$ 以及是否需要修改 dist 和 path 数组元素值。所以算法的时间复杂度是 $O(n^2)$。

2. Dijkstra 算法与 Prim 算法的对比分析

Dijkstra 算法的思想和 Prim 算法的思想有很多类似之处。以下对这两个算法的执行过程做对比分析。

(1) Dijkstra 算法的执行过程：对 S[$k$] != 1 的顶点，选择具有最小 dist[ ]值的顶点，设为 $u$；修改其他顶点 $v_k$ 的 dist[$k$]值：取 dist[$k$]=min{dist[$k$], dist[$u$] + Edge[$u$][$k$]}；Dijkstra 算法多了一个 path 数组，从而可以求得每个顶点的最短路径。

(2) Prim 算法的执行过程：对 nearvex[$j$] != -1 的顶点，选择具有最小 lowcost[ ]值的顶点，设为 $v$；修改其他顶点 $v_k$ 的 lowcost[$k$]值：取 lowcost[$k$]=min{ lowcost[$k$], Edge[$v$][$k$] }。

### 4.1.4　例题解析

以下通过两道例题的分析，再详细介绍 Dijkstra 算法的基本思想及其实现方法。

**例 4.2**　多米诺骨牌效应(Domino Effect)

**题目来源：**

Southwest Europe 1996, ZOJ1298, POJ1135

**题目描述：**

你知道多米诺骨牌除了用来玩多米诺骨牌游戏外，还有其他用途吗？多米诺骨牌游戏：取一些多米诺骨牌，竖着排成连续的一行，两张骨牌之间只有很短的空隙。如果排列得很

好，当推倒第 1 张骨牌，会使其他骨牌连续地倒下(这就是短语“多米诺效应”的由来)。

然而当骨牌数量很少时，这种玩法就没多大意思了，所以一些人在 20 世纪 80 年代早期开创了另一个极端的多米诺骨牌游戏：用上百万张不同颜色、不同材料的骨牌拼成一幅复杂的图案。他们开创了一种流行的艺术。在这种骨牌游戏中，通常有多行骨牌同时倒下。

试编写程序，给定这样的多米诺骨牌游戏，计算最后倒下的是哪一张骨牌、在什么时间倒下。这些多米诺骨牌游戏包含一些“关键牌”，它们之间由一行普通骨牌连接。当一张关键牌倒下时，连接这张关键牌的所有行都开始倒下。当倒下的行到达其他还没倒下的关键骨牌时，则这些关键骨牌也开始倒下，同样也使得连接到它的所有行开始倒下。每一行骨牌可以从两个端点中的任何一张关键牌开始倒下，甚至两个端点的关键牌都可以分别倒下，在这种情形下，该行最后倒下的骨牌为中间的某张骨牌。假定骨牌倒下的速度一致。

**输入描述：**

输入文件包含多个测试数据，每个测试数据描述了一个多米诺骨牌游戏。每个测试数据的第 1 行为两个整数：$n$ 和 $m$，$n$ 表示关键牌的数目，$1 \leqslant n < 500$；$m$ 表示这 $n$ 张牌之间用 $m$ 行普通骨牌连接。$n$ 张关键牌的编号为 1～$n$。每两张关键牌之间至多有一行普通牌，并且多米诺骨牌图案是连通的，也就是说从一张骨牌可以通过一系列的行连接到其他每张骨牌。

接下来有 $m$ 行，每行为 3 个整数：$a$、$b$ 和 $t$，表示第 $a$ 张关键牌和第 $b$ 张关键牌之间有一行普通牌连接，这一行从一端倒向另一端需要 $t$ 秒。每个多米诺骨牌游戏都是从推倒第 1 张关键牌开始的。

输入文件最后一行为 $n$=$m$=0，表示输入结束。

**输出描述：**

对输入文件中的每个测试数据，首先输出一行"System #k"，其中 $k$ 为测试数据的序号；然后再输出一行，首先是最后一块骨牌倒下的时间，精确到小数点后一位有效数字，然后是最后倒下骨牌的位置，这张最后倒下的骨牌要么是关键牌，要么是两张关键牌之间的某张普通牌。输出格式如样例输出所示。如果存在多个解，则输出任意一个。每个测试数据的输出之后输出一个空行。

**样例输入：** **样例输出：**

```
2 1          System #1
1 2 27       The last domino falls after 27.0 seconds, at key domino 2.
3 3
1 2 5        System #2
1 3 5        The last domino falls after 7.5 seconds, between key dominoes 2 and 3.
2 3 5
0 0
```

**分析：**

样例输入中两个测试数据所描述的多米诺骨牌图案如图 4.5(a)和图 4.5(b)所示。在图 4.5(a)中，先推倒第 1 张关键骨牌，这样第 2 张关键骨牌最后倒下，用时为 27 秒。在图 4.5(b)中，先推倒第 1 张骨牌，则第 2、3 张骨牌同时倒下，最后倒下的是第 2、3 张关键骨牌中间的某张普通骨牌，其用时为 7.5 秒。

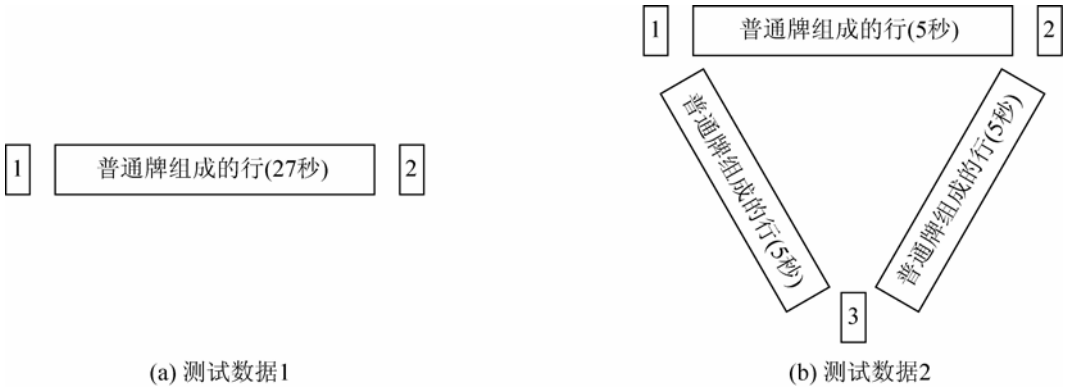



图 4.5　多米诺骨牌游戏

本题要求的是推倒第 1 张关键牌后，最后倒下的牌的位置及时间。如前所述，最后倒下的牌有两种情形：① 最后倒下的牌是关键牌，其时间及位置就是第 1 张关键牌到其他关键牌中最短路径的最大值及对应的关键牌；② 最后倒下的牌是两张关键牌之间的某张普通牌，其时间为这两张关键牌倒下时间的一半再加上这一行倒下时间的一半，位置为这两张牌之间的某张普通牌(不一定恰好是该行正中间的那张牌，但题目并不需要具体求出是哪张牌)。

本题的方法步骤如下。

(1) 先计算每一张关键牌倒下的 time[*i*]。这需要利用 Dijkstra 算法求第 1 张关键牌到其他每张关键牌的最短路径。然后取 time[*i*]的最大值，设为 maxtime1。

(2) 计算每一行完全倒下的时间。设每一行的两端的关键牌为 *i* 和 *j*，则这一行完全倒下的时间为(time[*i*] + time[*j*] + Edge[*i*][*j*])/2.0，其中 Edge[*i*][*j*]为连接第 *i*、*j* 两张关键牌的行倒下所花的时间。取所有行完全倒下时间的最大值，设为 maxtime2。

(3) 如果 maxtime2 > maxtime1，则是第(2)种情形；否则是第(1)种情形。

代码如下：

```
#define MAXN 500
#define INF 1000000     //无穷大
int n, m;               //关键牌的数目、关键牌之间连接的数目
int Edge[MAXN][MAXN];   //邻接矩阵
int caseno=1;           //测试数据序号
int time[MAXN]; //tim[i]为第 i 张关键牌倒下的时间(最先推倒第 0 张骨牌,存储时序号已减 1)
int S[MAXN];            //S[i]表示关键牌 i 的倒下时间是否已计算
void solve_case( )
{
    int i, j, k;    //循环变量
    for( i=0; i<n; i++ )
    {
        time[i]=Edge[0][i];  S[i]=0;
    }
    time[0]=0;  S[0]=1;
    for( i=0; i<n-1; i++ )      //Dijkstra 算法：从顶点 0 确定 n-1 条最短路径
    {
        int min=INF, u=0;
```

```
            for( j=0; j<n; j++ )    //选择当前集合 T 中具有最短路径的顶点 u
            {
                if( !S[j] && time[j]<min )
                {
                    u=j;  min=time[j];
                }
            }
            S[u]=1;              //将顶点 u 加入到集合 S，表示它的最短路径已求得
            for( k=0; k<n; k++ )    //修改 T 集合中顶点的 dist 和 path 数组元素值
            {
                if( !S[k] && Edge[u][k]<INF && time[u]+Edge[u][k]<time[k] )
                {
                    time[k]=time[u]+Edge[u][k];
                }
            }
        }
        double maxtime1=-INF;  int pos;  //最后倒下的关键牌时间及位置
        for( i=0; i<n; i++ )
        {
            if( time[i]>maxtime1 )
            {
                maxtime1=time[i];  pos=i;
            }
        }
        double maxtime2=-INF, t;    //每一行中间普通牌倒下的时间最大值及位置
        int pos1, pos2;
        for( i=0; i<n; i++ )
        {
            for( j=0; j<n; j++ )
            {
                t=(time[i]+time[j]+Edge[i][j])/2.0;
                if( Edge[i][j]<INF &&  t>maxtime2 )
                {
                    maxtime2=t;  pos1=i;  pos2=j;
                }
            }
        }
        printf( "System #%d\n", caseno++ );    //输出
        printf( "The last domino falls after " );
        if( maxtime2>maxtime1 )
            printf( "%.1f seconds, between key dominoes %d and %d.\n\n",
                maxtime2, pos1+1, pos2+1 );
        else printf( "%.1f seconds, at key domino %d.\n\n", maxtime1, pos+1 );
    }
    int read_case( )    //读入数据
    {
        int i, j;       //循环变量
```

```
    int v1, v2, t;  //每对连接的两张关键牌序号、时间
    scanf( "%d %d", &n, &m );
    if( n==0 && m==0 )  return 0;
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )  Edge[i][j]=INF;  //INF 表示没有连接
    }
    for( i=0; i<m; i++ )
    {
        scanf( "%d %d %d", &v1, &v2, &t);
        v1--;  v2--;
        Edge[v1][v2]=Edge[v2][v1]=t;
    }
    return 1;
}
int main( )
{
    while( read_case( ) )
        solve_case( );
    return 0;
}
```

**例 4.3**　成语游戏(Idiomatic Phrases Game)

**题目来源：**

Zhejiang Provincial Programming Contest 2006, ZOJ2750

**题目描述：**

Tom 正在玩一种成语接龙游戏。成语是一个包含多个中文字符、具有一定含义的短语。游戏规则是：给定 Tom 两个成语，他必须选用一组成语，该组成语中第 1 个和最后一个必须是给定的两个成语。在这组成语中，前一个成语的最后一个汉字必须和后一个成语的第一个汉字相同。在游戏过程中，Tom 有一本字典，他必须从字典中选用成语。字典中每个成语都有一个权值 $T$，表示选用这个成语后，Tom 需要花时间 $T$ 才能找到下一个合适的成语。试编写程序，给定字典，计算 Tom 至少需要花多长时间才能找到一个满足条件的成语组。

**输入描述：**

输入文件包含多个测试数据。每个测试数据包含一本成语字典。字典的第 1 行是一个整数 $N$，$0<N<1\,000$，表示字典中有 $N$ 个成语；接下来有 $N$ 行，每行包含一个整数 $T$ 和一个成语，其中 $T$ 表示 Tom 走出这一步所花的时间。每个成语包含多个(至少 3 个)中文汉字，每个中文汉字包含 4 位十六进制位(即 0～9，A～F)。注意，字典中第 1 个和最后一个成语为游戏中给定的起始和目标成语。输入文件最后一行为 $N$=0，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，输出一行，为一个整数，表示 Tom 所花的最少时间。如果找不到这样的成语组，则输出-1。

| **样例输入：** | **样例输出：** |
| --- | --- |
| 5 | 17 |
| 5 12345978ABCD2341 | -1 |

```
5 23415608ACBD3412
7 34125678AEFD4123
15 23415673ACC34123
4 41235673FBCD2156
2
20 12345678ABCD
30 DCBF5432167D
0
```

**分析：**

假设用图中的顶点代表字典中的每个成语，如果第 *i* 个成语的最后一个汉字跟第 *j* 个成语的第 1 个汉字相同，则画一条有向边，由顶点 *i* 指向顶点 *j*，权值为题目中所提到的时间 *T*：选用第 *i* 个成语后，Tom 需要花时间 *T* 才能找到下一个合适的成语。这样，样例输入中两个测试数据所构造的有向网如图 4.6(a)和图 4.6(b)所示。

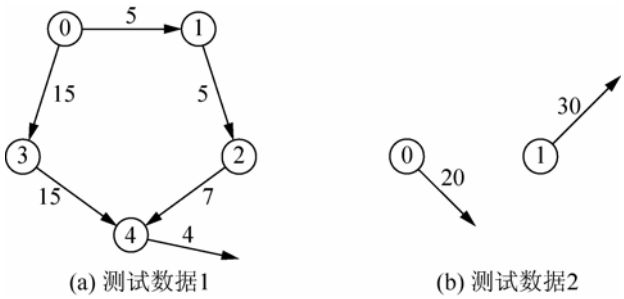



图 4.6　成语接龙游戏

构造好有向网后，问题就转化成求一条从顶点 0 到顶点 *N*-1 的一条最短路径，如果从顶点 0 到顶点 *N*−1 没有路径，则输出−1。例如，在图 4.6(a)中，顶点 0 到顶点 4 的最短路径长度为 17，所以输出 17；而在图 4.6(b)中从顶点 0 到顶点 1 不存在路径，所以输出−1。

因为源点是固定的，即顶点 0，所以本题采用 Dijkstra 算法求源点到第 *N*−1 个顶点之间的最短路径长度。

代码如下：

```
#define INF 1000000000 //无穷大
#define MAXN 1000      //顶点个数最大值
struct idiom
{
    char front[5], back[5]; //存储成语第 1 个和最后一个汉字
    int T;              //选用这个成语后，Tom 需要花时间 T 才能找到下一个合适的成语
};
idiom dic[MAXN];        //字典
int Edge[MAXN][MAXN];   //邻接矩阵
int dist[MAXN];         //求得的源点 0 到每个顶点的最短路径长度
int S[MAXN];            //S[i]=1 表示顶点 i 的最短路径已求得
int N;                  //成语个数
int main( )
```

```
{
    int i, j, k;    //循环变量
    char s[100];    //读入的每个成语(取其第1个和最后一个汉字)
    int len;        //成语的长度
    while( scanf( "%d", &N ) != EOF )
    {
        if( N==0 )  break;  //输入结束
        for( k=0; k<N; k++ )
        {
            scanf( "%d%s", &dic[k].T, s );
            len=strlen(s);
            for( i=0,j=len-1; i<4; i++, j-- )//取前4个字符、后4个字符
            {
                dic[k].front[i]=s[i];
                dic[k].back[3-i]=s[j];
            }
            dic[k].front[4]=dic[k].back[4]='\0';
        }
        for( i=0; i<N; i++ )    //建图
        {
            for( j=0; j<N; j++ )
            {
                Edge[i][j]=INF;
                if( i==j )  continue;
                if( strcmp( dic[i].back, dic[j].front )==0 )
                    Edge[i][j]=dic[i].T;
            }
        }
        //Dijkstra算法
        for( i=0; i<N; i++ )    //初始化
        {
            dist[i]=Edge[0][i];  S[i]=0;
        }
        S[0]=1;  dist[0]=0;     //顶点0加入到顶点集合S
        for( i=0; i<N-1; i++ )  //确定N-1条最短路径(Dijkstra算法)
        {
            int min=INF, u=0;
            //选择当前集合T中具有最短路径的顶点u
            for( j=0; j<N; j++ )
            {
                if( !S[j] && dist[j]<min )
                {
                    u=j;  min=dist[j];
                }
            }
            S[u]=1; //将顶点u加入到集合S，表示它的最短路径已求得
            for( k=0; k<N; k++ )    //修改T集合中顶点的dist数组元素值
```

```
            {
                if(!S[k] && Edge[u][k]<INF && dist[u]+Edge[u][k]<dist[k])
                    dist[k]=dist[u]+Edge[u][k];
            }
        }
        if( dist[N-1]==INF )  printf( "-1\n" );
        else  printf( "%d\n", dist[N-1] );
    }
    return 0;
}
```

## 练　习

4.1　纽约消防局救援(FDNY to the Rescue!), ZOJ1053, POJ1122

**题目描述：**

纽约消防局一直以他们对纽约市火警的反应时间而自豪，但是他们还想让反应时间更快。请帮助他们改进他们的反应时间。他们想确保总局指挥中心知道一旦有火警，哪个消防站离火警位置最近。你被雇来编写这个软件，以维护消防局的自豪。给定火警位置、所有消防站的位置、街道交叉路口、通过每条连接交叉路口之间道路所需的时间，该程序必须根据这些信息计算每个消防站到达一个指定火警位置所需时间。这些时间必须按从小到大的顺序进行排序，这样指挥中心就可以选择一个最近的消防站，并派出足够多的消防队员、携带足够多的设备去灭火。

**输入描述：**

输入文件包含多个测试数据。输入文件的第 1 行为一个整数 $T$，表示输入文件中测试数据的数目。然后是一个空行。空行后面是 $T$ 个测试数据。测试数据之间用一个空行隔开。

每个测试数据的格式如下。

第 1 行为一个整数 $N(N<20)$，表示城市中交叉路口的数目。

第 2～$N$+1 行：为 $N\times N$ 的矩阵，矩阵中的元素用一个或多个空格隔开。矩阵中的元素 $t_{ij}$，表示从第 $i$ 个交叉路口到第 $j$ 个交叉路口所需的时间(分钟)，如果 $t_{ij}$ 值为−1，则表示从第 $i$ 个交叉路口到第 $j$ 个交叉路口没有直接路径。

第 $N$+2 行首先是一个整数 $n(n\leqslant N)$，代表火警位置所在的交叉路口；然后有一个或多个整数，表示消防站所处的交叉路口。

注意：

(1) 行和列的序号都是从 1～$N$。

(2) 测试数据中所有数据都是整数。

(3) 测试数据保证每个消防站都是可以到达火警位置的。

(4) 消防站到火警的距离为消防站所处的交叉路口到火警位置所在的交叉路口的距离。

**输出描述：**

每个测试数据的输出格式如下。

第 1 行：列表的标题行，内容和格式如样例输出所示。

从第 2 行开始每一行描述了一个消防站的信息，这些信息按消防站到达火警位置所需

时间从小到大排列。这些信息包括：消防站的位置(初始位置)、火警位置(目标位置)、所需时间以及最短路径上的每个交叉路口。

注意：

(1) 列与列之间用 Tab 键隔开。

(2) 如果多个消防站到达的时间一样，则这些消防站之间的输出顺序任意。

(3) 如果从某个消防站到火警位置所需的时间为最短时间的路径不止一条，则输出任意一条。

(4) 如果火警位置和某个消防站的位置恰好是同一个交叉路口，则输出时，该消防站的输出行中，初始位置和目标位置是同一个交叉路口，所需时间为 0，最短路径上只有一个交叉路口序号，就是火警位置的序号。

每两个测试数据的输出之间用一个空行隔开。

**样例输入：**

```
1

6
0 3 4 -1 -1 -1
-1 0 4 5 -1 -1
2 3 0 -1 -1 2
8 9 5 0 1 -1
7 2 1 -1 0 -1
5 -1 4 5 4 0
2 4 5 6
```

**样例输出：**

```
Org     Dest    Time    Path
5       2       2       5       2

4       2       3       4       5       2
6       2       6       6       5       2
```

### 4.2　运输物质(Transport Goods), ZOJ1655

**题目描述：**

HERO 国正被其他国家攻打。入侵者正在攻打 HERO 国的首都，因此 HERO 国的其他城市必须运输物质到首都，以支援首都。城市之间有一些道路，物质必须沿着这些道路进行运输。

根据道路的长度和物质的重量，在运输途中需要花费一定的费用。每条道路的费用率为运输费用与货物重量的比率。费用率都是小于 1 的。

另外，每个城市必须等运输到这个城市的所有物质到齐后，才能将这些物质、连同它本身提供的物质一起运输到下一个城市。一个城市的所有物质只能运输到其他一个城市。

试计算可以运输到首都的物质的最大重量。

**输入描述：**

输入文件中包含多个测试数据。每个测试数据的第 1 行为两个整数 $N(2\leqslant N\leqslant 100)$和 $M$，其中 $N$ 表示城市的数目，包括首都(首都的编号为 $N$，其他城市的编号为 $1\sim N$)，$M$ 为道路的数目。接下来有 $N-1$ 行，第 $i$ 行$(1\leqslant i\leqslant N-1)$行为一个整数 $W(\leqslant 5\,000)$，表示该城市有 $W$ 重的物质运输到首都。接下来有 $M$ 行，描述了 $M$ 条道路。每行有 3 个整数：$A$、$B$ 和 $C$，表示从城市 $A$ 到城市 $B$ 有一条道路，其运输费用率为 $C$。

测试数据一直到文件尾。

**输出描述：**

对输入文件中的每个测试数据，输出一行，为可以运输到首都的物质的最大重量，精确到小数点后两位有效数字。

**样例输入：**

```
5 6
10
10
10
10
1 3 0
1 4 0
2 3 0
2 4 0
3 5 0
4 5 0
```

**样例输出：**

```
40.00
```

4.3 超级马里奥的冒险(Adventure of Super Mario), ZOJ1232

**题目描述：**

在救出美丽的公主后，超级马里奥需要找到一条回家的路，带着公主。他很熟悉“超级马里奥的”世界，所以他不需要地图，他只需要最好的路线来节省时间。

在“超级马里奥”的世界里，有 $A$ 个村子和 $B$ 个城堡。村庄被标号位 1～$A$，城堡被标号为 $A$+1～$A$+$B$。马里奥住在村子 1，他出发的城堡标号为 $A$+$B$。当然，村子 1 和城堡 $A$+$B$ 之间有双向的通道。两个地方至多有一条路，并且任何地方都没有连接自己的路。马里奥已经计算好每条路的长度，但他们不想一直走下去，因为他们单位时间只能走单位距离(太慢了)。

幸运的是，在救公主的城堡里，马里奥找到一个神奇的靴子。如果他穿上它，他能跑得超快，从一个地方到另一个地方只需一瞬间。(不要担心公主，马里奥找到了一个办法，当他跑得超快时可以带上公主，但他不会告诉你。)

因为在城堡里有圈套，马里奥不能用超快速度通过城堡。他总是在到城堡时停下来。当然，他开始/停止“超快速度”只能在村子(或城堡)里。

不幸的是，神奇的靴子太旧了，所以他不能一次使用它超过 $L$ 千米，他也不能使用它总共超过 $K$ 次。当他回家时，他能修理它，使它能再次使用。

**输入描述：**

输入文件第 1 行为一个整数 $T$，表示测试数据的个数，1≤$T$≤20。每个测试数据的第 1 行为 5 个整数：$A$、$B$、$M$、$L$ 和 $K$，分别表示村子的个数、城堡个数，1≤$A$、$B$≤50；以及路的条数，每次能使用的最大距离；靴子能使用的最多次数，0≤$K$≤10。接下来有 $M$ 行，每行包括 3 个整数：$X_i$ $Y_i$ $L_i$，表示第 $i$ 条路连接 $X_i$ 和 $Y_i$，距离为 $L_i$，所以走的时间也要 $L_i$，1≤$L_i$≤100。

**输出描述：**

对输入文件中的每个测试数据，输出占一行，为一个整数，表示超级马里奥带着漂亮的公主回到家所需的最短时间。每个测试数据可以使超级马里奥总能回到家。

**样例输入：**

```
1
4 2 6 9 1
4 6 1
5 6 10
4 5 5
3 5 4
2 3 4
1 2 3
```

**样例输出：**

```
9
```

4.4　邀请卡(Invitation Cards), ZOJ2008, POJ1511

**题目描述：**

在电视时代，没有太多的人再去剧院看演出。Malidinesia 国家的古代喜剧演员们(ACM)已经意识到这样的现状。他们想宣传戏剧，特别是古代喜剧。他们印制了一些邀请卡，上面是喜剧的演出信息。他们雇了一些学生志愿者来向人们发放邀请卡。每个学生被分派到一个公交站点，他(或她)整天就待在那，向乘坐公交车的人们发放邀请卡。在派往公交站点前，每个学生都经过培训，训练他们如何吸引人们而不是强行向人们发放邀请卡。

公交系统很特别：所有的线路都是单向的、只连接两个站点。公交车搭载乘客，每半个小时从始发站发车。到达目的站点后，空车返回到始发站，等待下一个半小时，例如 $X$：00 或 $X$：30，其中 $X$ 表示整点。两个站点之间的车费由一个列表指定，在站点支付。整个公交系统的路线是这样设计的：每个往返旅途，也就是说从一个站点出发，并回到同一个站点，通过中央检查站(CCS)，在这里，每个乘客都必须通过安检。

所有学生早上从 CCS 站点出发，每个学生到达一个指定的站点，邀请乘客。学生的人数跟站点的数目一样。每天结束的时候，所有学生乘车回到 CCS。试编写程序，计算 ACM 每天需要支付给所有学生车费的最小值。

**输入描述：**

输入文件包含 $N$ 个测试数据。输入文件第 1 行为一个正整数 $N$。接下来是 $N$ 个测试数据。每个测试数据第 1 行为两个整数：$P$ 和 $Q$，$1 \leqslant P$、$Q \leqslant 1\,000\,000$，其中 $P$ 为公交站点的数目，包括 CCS；$Q$ 为公交线路的数目。接下来有 $Q$ 行，每行描述了一条公交线路。每行有 3 个数：起点站点、目标站点、车费。CCS 站点编号为 1。车费为正整数，所有价格总和不超过 1 000 000 000。假定每一个站点都可以到达其他每个站点。

**输出描述：**

对输入文件中的每个测试数据，输出一行，为 ACM 每天需要支付给学生车费的最小值。

**样例输入：**

```
2
1
4 6
1 2 10
2 1 60
1 3 20
3 4 10
2 4 5
4 1 50
```

**样例输出：**

```
210
```

## 4.2 边上权值为任意值的单源最短路径问题—Bellman-Ford 算法

### 4.2.1 算法思想

1. Dijkstra 算法的局限性

4.1 节介绍了 Dijkstra 算法，该算法要求网络中各边上的权值大于或等于 0。如果有向网中存在带负权值的边，则采用 Dijkstra 算法求解最短路径得到的结果有可能是错误的。

例如，对图 4.7(a)所示的有向网，采用 Dijkstra 算法求得顶点 $v_0$ 到顶点 $v_2$ 的最短距离是 dist[2]，即 $v_0$ 到 $v_2$ 的直接路径，长度为 5。但从 $v_0$ 到 $v_2$ 的最短路径应该是$(v_0, v_1, v_2)$，其长度为 2。

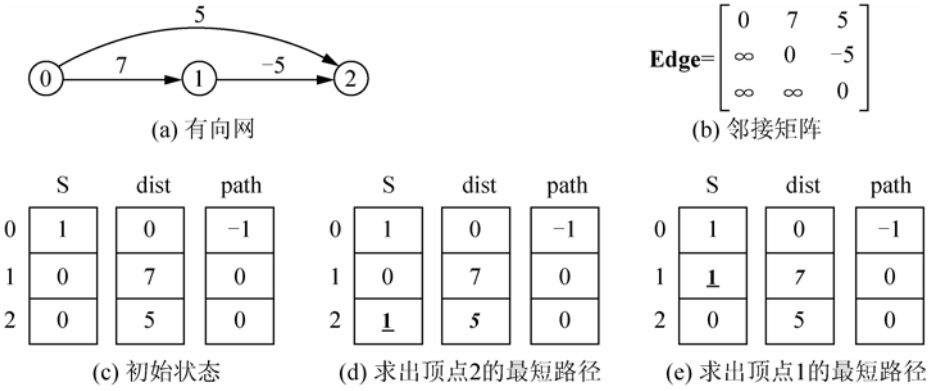



图 4.7 有向网中存在带负权值的边，用 Dijkstra 算法求解是错误的

如果把图 4.7(a)中边<1, 2>的权值由-5 改成 5，则采用 Dijkstra 算法求解最短路径，得到的结果是正确的，如图 4.8 所示。

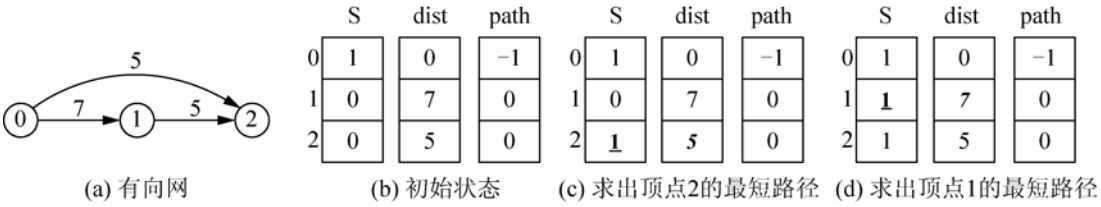



图 4.8 边上权值非负的有向网，用 Dijkstra 算法求解是正确的

为什么当有向网中存在带负权值的边时，采用 Dijkstra 算法求解得到的最短路径有时是错误的？答案是：Dijkstra 算法在利用顶点 $u$ 的 dist[ ]值去递推 $T$ 集合各顶点的 dist[$k$]值时，前提是顶点 $u$ 的 dist[ ]值是当前 $T$ 集合中最短路径长度最小的。如果图中所有边的权值都是正的，这样推导是没有问题的。但是如果有负权值的边，这样推导是不正确的。例如，在图 4.7(d)中，第 1 次在 $T$ 集合中找到 dist[ ]最小的是顶点 2，dist[2]等于 5；但是顶点 0 距离顶点 2 的最短路径是$(v_0, v_1, v_2)$，长度为 2，而不是 5，其中边<1, 2>是一条负权值边。

2. Bellman-Ford 算法思想

为了能够求解边上带有负权值的单源最短路径问题，Bellman(贝尔曼)和 Ford(福特)提出了从源点逐次途经其他顶点，以缩短到达终点的最短路径长度的方法。该方法也有一个限制条件：要求图中不能包含权值总和为负值的回路。

例如图 4.9(a)所示的有向网中，回路($v_0$, $v_1$, $v_0$)包括了一条具有负权值的边，且其路径长度为-1。当选择的路径为($v_0$, $v_1$, $v_0$, $v_1$, $v_0$, $v_1$, $v_0$, $v_1$, …)时，路径的长度会越来越小，这样顶点 0 到顶点 2 的路径长度最短可达-∞。如果存在这样的回路，则不能采用 Bellman-Ford 算法求解最短路径。

如果有向网中存在由带负权值的边组成的回路，但回路权值总和非负，则不影响 Bellman-Ford 算法的求解，如图 4.9(b)所示。


图 4.9　Bellman-Ford 算法：有向网中存在负权值的边

权值总和为负值的回路在本书中称为**负权值回路**，在 Bellman-Ford 算法中判断有向网中是否存在负权值回路的方法，详见 4.2.3 节中的第 3 点。

假设有向网中有 $n$ 个顶点且不存在负权值回路，从顶点 $v_1$ 和到顶点 $v_2$ 如果存在最短路径，则此路径最多有 $n$−1 条边。这是因为如果路径上的边数超过了 $n$−1 条时，必然会重复经过一个顶点，形成回路；而如果这个回路的权值总和为非负时，完全可以去掉这个回路，使得 $v_1$ 到 $v_2$ 的最短路径长度缩短。下面将以此为依据，计算从源点 $v_0$ 到其他每个顶点 $u$ 的最短路径长度 dist[$u$]。

Bellman-Ford 算法构造一个最短路径长度数组序列：$\text{dist}^1[u]$，$\text{dist}^2[u]$，$\text{dist}^3[u]$，…，$\text{dist}^{n-1}[u]$。其中：

$\text{dist}^1[u]$为从源点 $v_0$ 到终点 $u$ 的只经过一条边的最短路径长度，并有 $\text{dist}^1[u]=\text{Edge}[v_0][u]$。

$\text{dist}^2[u]$为从源点 $v_0$ 出发最多经过不构成负权值回路的两条边到达终点 $u$ 的最短路径长度。

$\text{dist}^3[u]$为从源点 $v_0$ 出发最多经过不构成负权值回路的 3 条边到达终点 $u$ 的最短路径长度。

……

$\text{dist}^{n-1}[u]$为从源点 $v_0$ 出发最多经过不构成负权值回路的 $n$−1 条边到达终点 $u$ 的最短路径长度；

算法的最终目的是计算出 $\text{dist}^{n-1}[u]$，为源点 $v_0$ 到顶点 $u$ 的最短路径长度。

采用递推方式计算 $\text{dist}^k[u]$。

设已经求出 $\text{dist}^{k-1}[u]$, $u$=0, 1, …, $n$−1，此即从源点 $v_0$ 最多经过不构成负权值回路的 $k$−1 条边到达终点 $u$ 的最短路径的长度。

从图的邻接矩阵可以找到各个顶点 $j$ 到达顶点 $u$ 的(直接边)距离 Edge[$j$][$u$]，计算 $\min\{\ \text{dist}^{k-1}[j] + \text{Edge}[j][u]\ \}$，可得从源点 $v_0$ 途经各个顶点，最多经过不构成负权值回路的

$k$ 条边到达终点 $u$ 的最短路径的长度。

比较 $dist^{k-1}[u]$和 $\min\{ dist^{k-1}[j] + Edge[j][u] \}$，取较小者作为 $dist^{k}[u]$的值。

因此 Bellman-Ford 算法的**递推公式**(求源点 $v_0$ 到各顶点 $u$ 的最短路径)为：

初始：$dist^{1}[u]=Edge[v_0][u]$，$v_0$ 是源点

递推：$dist^{k}[u]=\min\{ dist^{k-1}[u], \min\{ dist^{k-1}[j] + Edge[j][u] \} \}$，

$j=0, 1, \cdots, n-1, j\neq u$; $k=2, 3, 4, \cdots, n-1$

### 4.2.2 算法实现

Bellman-Ford 算法在实现时，需要使用以下两个数组。

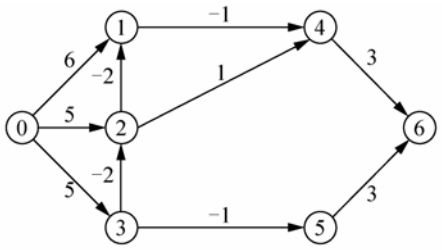
(1) 使用同一个数组 dist[$n$]来存放一系列的 $dist^{k}[n]$，其中 $k=1, 2, \cdots, n-1$；算法结束时 dist[$u$]数组中存放的是 $dist^{n-1}[u]$；

(2) path[$n$]数组含义同 Dijkstra 算法中的 path 数组。

Bellman-Ford 算法具体实现代码详见例 4.4。

**例 4.4** 利用 Bellman-Ford 算法求图 4.10(a)中顶点 0 到其他各顶点的最短路径长度，并输出对应的最短路径。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 $n$，然后输入每条边的数据。每条边的数据格式为：$u\ v\ w$，分别表示这条边的起点、终点和边上的权值。顶点序号从 0 开始计起。最后一行为−1 −1 −1，表示输入数据的结束。




(a) 带负权值的有向网

$$Edge = \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{bmatrix} 0 & 6 & 5 & 5 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & -1 & \infty & \infty \\ \infty & -2 & 0 & \infty & 1 & \infty & \infty \\ \infty & \infty & -2 & 0 & \infty & -1 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix} \end{array} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix}$$

(b) 邻接矩阵

| $k$ | $dist^k[0]$ | $dist^k[1]$ | $dist^k[2]$ | $dist^k[3]$ | $dist^k[4]$ | $dist^k[5]$ | $dist^k[6]$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 6 | 5 | 5 | ∞ | ∞ | ∞ |
| 2 | 0 | ***3*** | ***3*** | 5 | ***5*** | ***4*** | ∞ |
| 3 | 0 | ***1*** | 3 | 5 | ***2*** | 4 | ***7*** |
| 4 | 0 | 1 | 3 | 5 | ***0*** | 4 | ***5*** |
| 5 | 0 | 1 | 3 | 5 | 0 | 4 | ***3*** |
| 6 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |

在递推$dist^k[u]$时,有更新的$dist^k[u]$用粗体、斜体标明，$u$=1,2,3,4,5,6

(c) $dist^k[\ ]$的递推过程

图 4.10 Bellman-Ford 算法的求解过程

**分析：**

如图 4.10(c)所示，$k$=1 时，dist 数组各元素的值 dist[$u$]就是 Edge[0][$u$](见图 4.10(b))。

在 Bellman-Ford 算法执行过程中，dist 数组各元素值的变化如图 4.10(c)所示。在图 4.10(c)中，dist[$u$]的值如果有更新，则用粗体、斜体标明，$u$=1, 2, 3, 4, 5, 6。以 $k$=2，$u$=1 加以解释。求 $dist^2[1]$的递推公式是：

$$dist^2[1] = \min\{ dist^1[1], \min\{dist^1[j] + Edge[j][1]\}\}，j = 0, 2, 3, 4, 5, 6$$

所以，在程序中 $k = 2$ 时，dist[1]的值为：

$$\begin{aligned} dist[1] &= \min\{ 6, \min\{ dist[0] + Edge[0][1], dist[2] + Edge[2][1], \\ &\quad dist[3] + Edge[3][1], dist[4] + Edge[4][1], \\ &\quad dist[5] + Edge[5][1], dist[6] + Edge[6][1] \} \} \\ &= \min\{ 6, \min\{ 0+6, 5+(-2), 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty \} \} \\ &= 3 \end{aligned}$$

此时 dist[1]的值为从源点 $v_0$ 出发，经过不构成负权值回路的两条边到达顶点 $v_1$ 的最短路径长度，其路径为$(v_0, v_2, v_1)$。

在 Bellman-Ford 算法执行过程中，path[$n$]数组的变化与 Dijkstra 算法类似，所以在图 4.10 中并没有列出 path[$n$]数组的变化过程。当顶点 0 到其他各顶点的最短路径长度求解完毕后，如何根据 path 数组求解顶点 0 到其他各顶点 $v_k$ 的最短路径？方法跟 Dijkstra 算法中的方法完全一样：从 path[$k$]开始，采用“倒向追踪”方法，一直找到源点 $v_0$。

在下面的代码中，Bellman( int $v_0$ )函数实现了求源点 $v_0$ 到其他各顶点的最短路径。在主函数中调用 Bellman( 0 )，则求解的是从顶点 0 到其他各顶点的最短路径。另外，主函数中的 shortest 数组用来保存最短路径上各个顶点的序号，其作用与例 4.1 代码中 shortest 数组的作用一样。

代码如下：

```
#define INF 1000000          //无穷大
#define MAXN 8               //顶点个数最大值
int n;                       //顶点个数
int Edge[MAXN][MAXN];        //邻接矩阵
int dist[MAXN];              //
int path[MAXN];              //
void Bellman( int v0 )       //求顶点 v0 到其他顶点的最短路径
{
    int i, j, k, u;          //循环变量
    for( i=0; i<n; i++ )     //初始化
    {
        dist[i]=Edge[v0][i];
        if( i!=v0 && dist[i]<INF )  path[i]=v0;
        else path[i]=-1;
    }
    for( k=2; k<n; k++ ) //从 dist(1)[u]递推出 dist(2)[u], …,dist(n-1)[u]
    {
        for( u=0; u<n; u++ ) //修改每个顶点的 dist[u]和 path[u]
        {
            if( u!=v0 )
            {
                for( j=0; j<n; j++ )//考虑其他每个顶点
```

```
                {
                    //顶点 j 到顶点 u 有直接路径，且途经顶点 j 可以使得 dist[u]缩短
                    if( Edge[j][u]<INF && dist[j]+Edge[j][u]<dist[u] )
                    {
                        dist[u]=dist[j]+Edge[j][u];
                        path[u]=j;
                    }
                }//end of for
            }//end of if
        }//end of for
    }//end of for
}
int main( )
{
    int i, j;           //循环变量
    int u, v, w;        //边的起点和终点及权值
    scanf( "%d", &n );  //读入顶点个数 n
    while( 1 )
    {
        scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
        if( u==-1 && v==-1 && w==-1 )  break;
        Edge[u][v]=w;       //构造邻接矩阵
    }
    for( i=0; i<n; i++ )    //设置邻接矩阵中其他元素的值
    {
        for( j=0; j<n; j++ )
        {
            if( i==j ) Edge[i][j]=0;
            else if( Edge[i][j]==0 )  Edge[i][j]=INF;
        }
    }
    Bellman( 0 );           //求顶点 0 到其他顶点的最短路径
    int shortest[MAXN];     //输出最短路径上的各个顶点时存放各个顶点的序号
    for( i=1; i<n; i++ )
    {
        printf( "%d\t", dist[i] ); //输出顶点 0 到顶点 i 的最短路径长度
        //以下代码用于输出顶点 0 到顶点 i 的最短路径
        memset( shortest, 0, sizeof(shortest) );
        int k=0;    //k 表示 shortest 数组中最后一个元素的下标
        shortest[k]=i;
        while( path[ shortest[k] ]!=0 )
        {
            k++; shortest[k]=path[ shortest[k-1] ];
        }
        k++; shortest[k]=0;
        for( j=k; j>0; j-- )
            printf( "%d→", shortest[j] );
```

```
            printf( "%d\n", shortest[0] );
        }
        return 0;
    }
```

该程序的运行示例如下。

**输入：**

```
7
0 1 6
0 2 5
0 3 5
1 4 -1
2 1 -2
2 4 1
3 2 -2
3 5 -1
4 6 3
5 6 3
-1 -1 -1
```

**输出：**

```
1       0→3→2→1
3       0→3→2
5       0→3
0       0→3→2→1→4
4       0→3→5
3       0→3→2→1→4→6
```

### 4.2.3　关于 Bellman-Ford 算法的进一步讨论

1. Bellman-Ford 算法的本质思想

在从 $dist^{k-1}[\ ]$递推到 $dist^{k}[\ ]$的过程中，Bellman-Ford 算法的本质是对每条边<$u$, $v$>(或($u$, $v$))进行判断：设边<$u$, $v$>的权值为 $w(u, v)$，如图 4.11 所示，如果边<$u$, $v$>的引入会使得 $dist^{k-1}[v]$的值再减小，则要修改 $dist^{k-1}[v]$，即：如果 $dist^{k-1}[u] + w(u, v) < dist^{k-1}[v]$，则要将 $dist^{k}[v]$的值修改成 $dist^{k-1}[u] + w(u, v)$。本文将修改 $dist^{k-1}[v]$的运算称为一次**松弛**(Slack)。

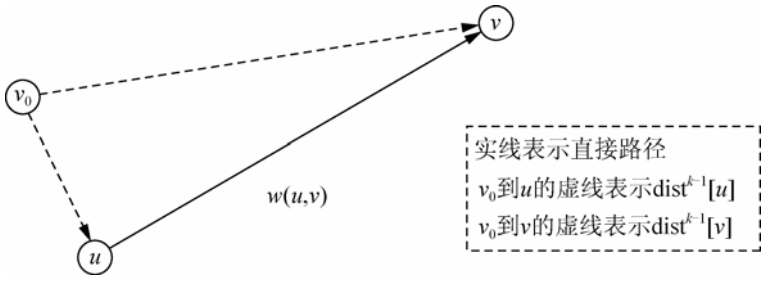



图 4.11　Bellman-Ford 算法的本质思想

按照这样的思想，Bellman-Ford 算法的递推公式应该改为(求源点 $v_0$ 到各顶点 $v$ 的最短路径)：

| |
|---|
| 初始：$dist^{0}[v] = \infty$, $dist^{0}[v_0]=0$, $v_0$ 是源点, $v \neq v_0$　　(dist 数组的初始值详见后面的分析) |
| 递推：对每条边$(u, v)$, $dist^{k}[v]=\min\{dist^{k-1}[v], dist^{k-1}[u] + w(u, v)\}$,　　$k=1, 2, 3, \cdots, n-1$ |

理解了这一点，就能理解 Bellman-Ford 算法的复杂度分析、Bellman-Ford 算法的优化等。

2. Bellman-Ford 算法的时间复杂度分析

在例 4.4 的 Bellman-Ford 算法代码中，有一个三重嵌套的 for 循环，如果使用邻接矩阵存储有向网，最内层的 if 语句的总执行次数为 $n^3$，所以算法的时间复杂度是 $O(n^3)$。

如果使用邻接表存储有向网，内层的两个 for 循环可以改成一个 while 循环，可以使算法的时间复杂度降为 $O(n\times m)$，其中 $n$ 为有向网中顶点个数，$m$ 为边的数目。这是因为，邻接表里直接存储了边的信息，浏览完所有的边，复杂度是 $O(m)$；而邻接矩阵是间接存储边，浏览完所有的边，复杂度是 $O(n^2)$。

使用邻接表存储思想实现 Bellman-Ford 算法的具体过程为：将每条边的信息(两个顶点 $u$、$v$ 和权值 $w$，可以用一个结构体，如 eg，来实现)存储到一个数组 edges 中，从 $dist^{k-1}[\ ]$ 递推到 $dist^{k}[\ ]$的过程中，对 edges 数组中的每条边<$u$, $v$>，判断一下边<$u$, $v$>的引入，是否会缩短源点 $v_0$ 到顶点 $v$ 的最短路径长度。(这种方法的具体应用详见例 4.10。)

根据上面的分析，可以将例 4.4 代码中的 bellman 函数简化成如下代码。

```
void Bellman( int v0 )      //求顶点 v0 到其他顶点的最短路径
{
    int i, k;               //循环变量
    for( i=0; i<n; i++ )    //初始化 dist[ ]数组，n 为顶点数目
    {
        dist[i]=INF;  path[i]=-1;   //INF 为代表∞的常量
    }
    dist[v0]=0;
    for( k=1; k<n; k++ ) //从 dist(0)[u]递推出 dist(1)[u], …,dist(n-1)[u]
    {
        //判断第 i 条边<u,v>的引入，是否会缩短源点 v0 到顶点 v 的最短路径长度
        for( i=0; i<m; i++ )     //m 为边的数目，即 edges 数组中元素个数
        {
            if( dist[edges[i].u] != INF && edges[i].w + dist[edges[i].u] <
                dist[edges[i].v] )
            {
                dist[edges[i].v]=edges[i].w+dist[edges[i].u];
                path[edges[i].v]=edges[i].u;
            }
        }//end of for
    }//end of for
}
```

其中，dist 数组各元素中，除源点 $v_0$ 外，其他顶点的 dist[ ]值都初始化为∞，这样 Bellman-Ford 算法需要多递推一次，详见后面的分析。

3. Bellman-Ford 算法负权值回路的判断方法

如果存在从源点可达的负权值回路，则最短路径不存在，因为可以重复走这个回路，使得路径无穷小。在 Bellman-Ford 算法中，判断是否存在从源点可达的负权值回路的方法如下。在求出 $dist^{n-1}[\ ]$之后，再对每条边<$u$, $v$>判断一下：加入这条边是否会使得顶点 $v$ 的最短路径值再缩短。即判断：

$$dist[u] + Edge[u][v] < dist[v]$$

是否成立，如果成立，则说明存在从源点可达的负权值回路。代码如下：

```
for( i=0; i<n; i++ )    //采用邻接矩阵
{
    for ( j=0; j<n; j++ )
    {
        if( Edge[i][j] < INF && dist[i]+Edge[i][j]<dist[j] )
            return 0;   //存在从源点可达的负权值回路
    }
}
return 1;   //不存在从源点可达的负权值回路
```

或：

```
for( i=0; i < m; i++ ) //每条边存储在数组 edges[ ]中，m 为边的数目
{
    if(dist[edges[i].u]!=INF && edges[i].w+dist[edges[i].u]<dist[edges[i].v])
        return 0;       //存在从源点可达的负权值回路
}
return 1;               //不存在从源点可达的负权值回路
```

具体应用详见例 4.10。

4. 关于 Bellman-Ford 算法中数组 dist 的初始值

在 4.2.1 节关于 Bellman-Ford 算法的描述中，dist 数组的初始值为邻接矩阵中源点 $v_0$ 所在的行，实际上还可以采用以下方式对 dist 数组初始化：除源点 $v_0$ 外，其他顶点的最短距离初始为∞(在程序实现时可以用一个权值不会达到的一个大数表示)；源点 dist[$v_0$]=0。这样 bellman-Ford 算法的第 1 重 for 循环要多执行一次，即要执行 $n$−1 次；且执行第 1 次后，dist 数组的取值为邻接矩阵中源点 $v_0$ 所在的行。

5. Bellman-Ford 算法的改进

Bellman-Ford 算法是否一定要循环 $n$−2 次，$n$ 为顶点个数，即是否一定需要从 $dist^1[u]$ 递推到 $dist^{n-1}[u]$？

答案是未必！其实只要在某次循环过程中，考虑每条边后，都没能改变当前源点到所有顶点的最短路径长度，那么 Bellman-Ford 算法就可以提前结束了。这种思路的具体应用详见例 4.11。

6. Dijkstra 算法与 Bellman-Ford 算法比较

Dijkstra 算法和 Bellman-Ford 算法的求解过程有很大的区别，具体如下。

(1) Dijkstra 算法在求解过程中，源点到集合 $S$ 内各顶点的最短路径一旦求出，则之后不变了，修改的仅仅是源点到 $T$ 集合中各顶点的最短路径长度。

(2) Bellman-Ford 算法在求解过程中，每次循环每个顶点的 dist[ ]值都有可能要修改，也就是说源点到各顶点最短路径长度一直要到 Bellman-Ford 算法结束才确定下来。

7. Bellman-Ford 算法的其他用途

Bellman-Ford 算法不仅可以用来求最短路径，还可以用来求最长路径。其思路是：dist[ ]

的含义是从源点 $v_0$ 到其他每个顶点的最长路径长度，初始时，各顶点的 dist[ ]值为 0；在从 $dist^{k-1}$[ ]递推到 $dist^k$[ ]过程中，对每条边<$u$, $v$>，设其权值为 $w(u, v)$，如果边<$u$, $v$>的引入会使得 $dist^{k-1}[v]$的值增加，则要修改 $dist^{k-1}[v]$，即：如果 $dist^k[u] + w(u, v) > dist^k[v]$，则要将 $dist^k[v]$的值设置成 $dist^{k-1}[u] + w(u, v)$。其具体应用详见例 4.5。

### 4.2.4 例题解析

以下通过两道例题的分析，再详细介绍 Bellman-Ford 算法的基本思想及其实现方法。

**例 4.5** 套汇(Arbitrage)

**题目来源：**

University of Ulm Local Contest 1996, ZOJ1092, POJ2240

**题目描述：**

套汇是利用汇率之间的差异，从而将一单位的某种货币，兑换回多于 1 单位的同种货币。例如，假定 1 美元兑换 0.5 英镑，1 英镑兑换 10.0 法郎，1 法郎兑换 0.21 美元，那么，在兑换货币过程中，一个聪明的商人可以用 1 美元兑换到 0.5 × 10.0 × 0.21=1.05 美元，这样就有 5%的利润。

试编写程序，读入货币之间的汇率列表，判断是否存在套汇。

**输入描述：**

输入文件包含多个测试数据。每个测试数据的第 1 行为一个整数 $n$，$1 \leq n \leq 30$，表示有 $n$ 种不同的货币。接下来有 $n$ 行，每行是一种货币的名称，货币名称是一个不包含空格的字符串。接下来一行是一个整数 $m$，代表汇率列表种有 $m$ 种汇率。接下来有 $m$ 行，每行格式为：$c_i\ r_{ij}\ c_j$，其中 $c_i$ 为源货币，$c_j$ 表示目标货币，实数 $r_{ij}$ 表示从 $c_i$ 到 $c_j$ 的汇率。汇率列表中没有出现的兑换表示不能进行的兑换。测试数据用空行隔开。

输入文件最后一行为 $n$=0，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，输出一行：如果获得套汇，则输出"Case i: Yes"，否则输出"Case i: No"，其中 $i$ 为测试数据的序号。

**样例输入：**

```
3
USDollar
BritishPound
FrenchFranc
3
USDollar 0.5 BritishPound
BritishPound 10.0 FrenchFranc
FrenchFranc 0.21 USDollar

3
USDollar
BritishPound
FrenchFranc
6
```

**样例输出：**

```
Case 1: Yes
Case 2: No
```

```
USDollar 0.5 BritishPound
USDollar 4.9 FrenchFranc
BritishPound 10.0 FrenchFranc
BritishPound 1.99 USDollar
FrenchFranc 0.09 BritishPound
FrenchFranc 0.19 USDollar

0
```

**分析：**

假设用图 4.12 中的顶点代表每一种货币，如果第 $i$ 种货币能够兑换成第 $j$ 种货币，汇率为 $c_{ij}$，则画一条有向边，由顶点 $i$ 指向顶点 $j$，权值为 $c_{ij}$。这样，样例输入中两个测试数据所构造的有向网如图 4.12(a)和图 4.12(b)所示。在图 4.12(a)中，从 1 美元(顶点 0)出发，兑换回 0.5×10.0×0.21=1.05 美元，所以存在套汇，应该输出"Yes"。而在图 4.12(b)中，从 1 美元(顶点 0)出发，能兑换回 0.5×1.99=0.995 美元，或者能兑换回 0.5×10×0.09×1.99=0.8955 美元，或者能兑换回 0.5×10×0.19×4.9×0.09×1.99=0.8337105 美元，不存在套汇；从其他货币出发，也不存在套汇，所以应该输出"No"。

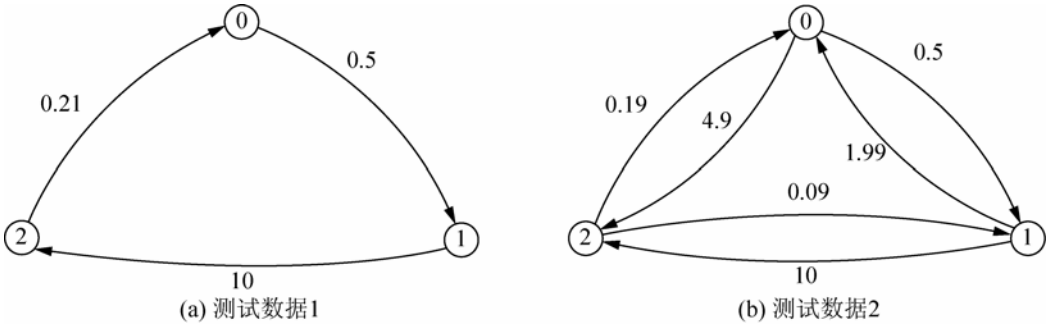



图 4.12　套汇：求最长路径

本题的求解思路是：构造好有向网后，问题就转化成判断图中是否存在某个顶点，从它出发的某条回路上权值乘积是否大于 1，大于 1 则表示存在套汇。具体求解时，可以用 Bellman-Ford 算法求从源点 $v_0$ 出发到各个顶点 $v$(包括 $v_0$ 本身)最长路径长度，只不过这里路径长度并不是权值之和，而是权值乘积。

假设 maxdist 数组存储的是源点 $v_0$ 到其他每个顶点 $v$(包括 $v_0$ 本身)的最长路径长度，其递推公式为(c($u$, $v$)表示从第 $u$ 种货币到第 $v$ 种货币的汇率)：

初始：$\text{maxdist}^0[v]=0$, $\text{maxdist}^0[v_0]=1$, $v_0$ 是源点
递推：对每条边($u$, $v$), $\text{maxdist}^k[v]=\max\{\ \text{maxdist}^{k-1}[v], \text{maxdist}^{k-1}[u] * c(u, v)\ \}$;
　　$k=1, 2, 3, \cdots, n$.

关于 maxdist 数组初始值和 Bellman-Ford 算法递推次数的说明如下。

(1) maxdist 数组的初始值：因为要取最大值，所以 maxdist[$v$]的初始值应该是一个较小的值，所以取为 0。

(2) Bellman-Ford 算法递推次数：在 4.2.1 节介绍 Bellman-Ford 算法思想时，曾经提到

从源点 $v_0$ 到顶点 $v$ 的最短路径最多有 $n-1$ 条边；但在本题中，由于要找一条回路，回到源点 $v_0$，所以最多有 $n$ 条边，且多于 $n$ 条边是没有必要的，因为如果存在套汇的话，$n$ 条边构成的回路就能形成套汇；因此，应该从 $maxdist^0[v]$递推到 $maxdist^1[v]$, $maxdist^2[v]$, …, $maxdist^{(n)}[v]$。

Bellman-Ford 算法执行完毕后，如果 maxdist[v0]>1，则表示从源点 $v_0$ 出发，存在套汇。依次将第 $i$ 个顶点作为源点执行 Bellman-Ford 算法，如果某个顶点存在套汇，则不必判断下去了。

需要说明的是，本题也可以采用 4.3 节介绍的 Floyd 算法求解。

代码如下：

```
#define maxn 50              //顶点数最大值
#define maxm 1000            //边数最大值
#define max(a,b) ( (a)>(b) ? (a):(b) )
struct exchange              //汇率关系
{
    int ci, cj;
    double cij;
}ex[maxm];                   //汇率关系数组
int i, j, k;                 //循环变量
int n, m;                    //货币种类的数目、汇率的数目
char name[maxn][20], a[20], b[20]; //货币名称
double x;                    //读入的汇率
double maxdist[maxn];        //源点 i 到其他每个顶点(包括它本身)的最长路径长度
int flag;                    //是否存在套汇的标志，flag=1 表示存在
int kase=0;                  //测试数据序号
int readcase( )              //读入数据
{
    scanf( "%d", &n );
    if( n==0 )  return 0;
    for( i=0; i<n; i++ )     //读入 n 个货币名称
        scanf( "%s", name[i] );
    scanf( "%d", &m );
    for( i=0; i<m; i++ )     //读入汇率
    {
        scanf( "%s %lf %s", a, &x, b );
        for( j=0; strcmp( a, name[j] ); j++ )
            ;
        for( k=0; strcmp( b, name[k] ); k++ )
            ;
        ex[i].ci=j;  ex[i].cij=x;  ex[i].cj=k;
    }
    return 1;
}
//Bellman-Ford 算法：以顶点 v0 为源点，求它到每个顶点(包含它本身)的最大距离
void bellman( int v0 )
{
```

```
        flag=0;
        memset( maxdist, 0, sizeof(maxdist) ); //初始化 maxdis 数组
        maxdist[v0]=1;
        for( k=1; k<=n; k++ )   //从 maxdist(0)递推到 maxdist(1),...,maxdist(n)
        {
            for( i=0; i<m; i++ )    //判断每条边,加入它是否能使得最大距离增加
            {
                if( maxdist[ex[i].ci]*ex[i].cij>maxdist[ex[i].cj] )
                {
                    maxdist[ex[i].cj]=maxdist[ex[i].ci]*ex[i].cij;
                }
            }
        }
        if( maxdist[v0]>1.0 ) flag=1;
    }
    int main( )
    {
        while( readcase( ) )    //读入货币种类的数目
        {
            for( i=0; i<n; i++ )
            {
                bellman( i );   //从第 i 个顶点出发求最长路径
                if( flag )  break;
            }
            if( flag )  printf( "Case %d: Yes\n", ++kase );
            else  printf( "Case %d: No\n", ++kase );
        }
        return 0;
    }
```

例 4.6　门(The Doors)

**题目来源：**

Mid-Central USA 1996, ZOJ1721, POJ1556

**题目描述：**

在本题中，试在一个布置了障碍墙的房间里找一条最短路径。房间的边界为 $x$=0, $x$=10, $y$=0, 以及 $y$=10。路径的起始位置为(0, 5)，目标位置为(10, 5)。在房间布置了一些竖直的墙，墙的数目范围为 0～18，每堵墙有两个门。图 4.13(a)和图 4.13(b)描述了这样的房间，并显示了最短路径。

**输入描述：**

输入文件包含多个测试数据，每个测试数据描述了一个房间。每个测试数据的格式如下。

第 1 行为一个整数 $n$，表示房间内竖直墙的数目。

接下来有 $n$ 行，每行描述了一堵竖直的墙，用 5 个实数表示：第 1 个实数为墙的 $x$ 坐标，0<$x$<10；其他 4 个实数为这堵墙上两扇门的两个端点的 $y$ 坐标。$n$ 堵墙以 $x$ 坐标的升序顺序给出，每堵墙的数据中，4 个 $y$ 坐标也以升序顺序出现。

输入文件最后一行为 $n$=–1，表示输入结束。

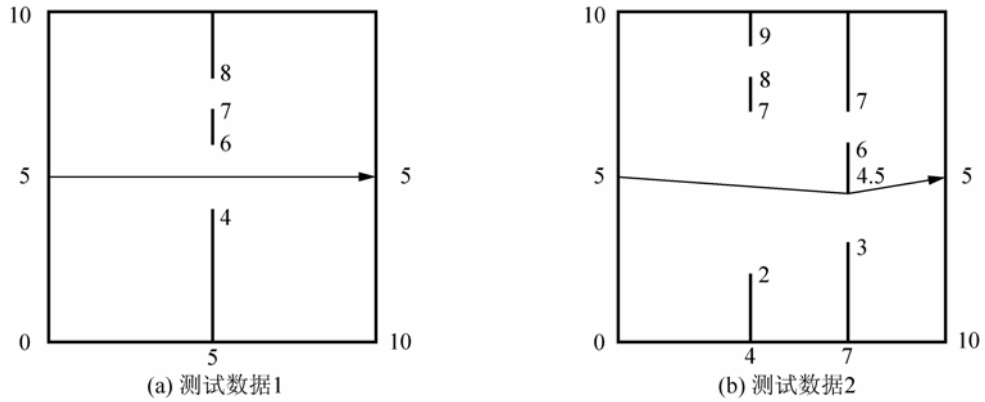



图 4.13　布满墙的房间

**输出描述：**

对输入文件中的每个测试数据，输出占一行，为求得的最短路径长度，精确到小数点后两位有效数字。

**样例输入：**

```
1
5 4 6 7 8
2
4 2 7 8 9
7 3 4.5 6 7
-1
```

**样例输出：**

```
10.00
10.06
```

**分析：**

本题样例输入中两个测试数据所描述的房间如图 4.13(a)和图 4.13(b)所示。

本题要求平面上两个点(0,5)和(10,5)之间的最短距离。如果要采用本章介绍的最短路径算法进行求解，关键是如何构造一个有向网(或无向网)。

在本题中，构造无向网的方法为：以(0,5)为第 0 个顶点，以(10,5)为最后一个顶点，对每堵墙中的两扇门，每个端点对应无向网中的一个顶点，这样整个无向网的顶点数为 4n+2；在判断顶点 $a$ 和顶点 $b$ 之间是否需要连接时，需要判断这两个顶点是否被它们之间的某堵墙阻挡了，如果被某堵墙阻挡了，则不能连线。

例如在图 4.14 中，点(0,5)和点(10,5)就不能连线，因为尽管它们没被第 1 堵墙阻挡，但被第 2 堵墙阻挡了。每堵墙实际上是由 3 段组成的，例如图 4.14 中，第 2 堵墙由(7,0)~(7,3)、(7,4.5)~(7,6)、(7,7)~(7,10)3 段组成。只要顶点 $a$ 和顶点 $b$ 被第 $i$ 堵墙某一段阻挡，那么它们就被第 $i$ 堵墙阻挡了。

判断顶点 $a$ 和顶点 $b$ 是否被某段墙的某断阻挡，方法是：如果这段墙的两个端点同时位于顶点 $a$ 和顶点 $b$ 所确定的直线下方或上方，则它们没被这段墙阻挡；否则如果两个端点一个位于直线上方、另一个端点位于下方，则它们被这段墙阻挡了。

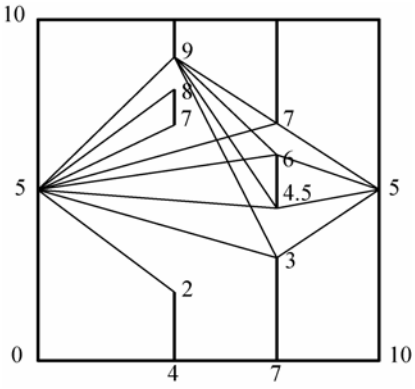



图 4.14　布满墙的房间：网络的构造

代码如下：

```
#define INF 2000000000
#define MAXN 100
struct POINT    //平面上点的坐标
{
    double x, y;
};
struct EDGE     //边
{
    int u, v;
};
int n;              //房间里墙的数目
double wX[20];      //每堵墙的 x 坐标(升序)
POINT p[MAXN];      //存储起点、每扇门的两个端点、终点的平面坐标
int pSize;          //点的数目(含起点、终点)
double pY[20][4]; //pY[i][0],pY[i][1],pY[i][2],pY[i][3]:第 i 堵墙的 4 个 y 坐标
double g[MAXN][MAXN];   //邻接矩阵
EDGE e[MAXN*MAXN];      //存储构造的每条边
int eSize;              //边的数目
int i, j;               //循环变量
double Dis( POINT a, POINT b ) //求平面上两个点之间的距离
{
    return sqrt( (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y) );
}
//判断点(x3,y3)是否位于点(x1,y1)和(x2,y2)所确定的直线的上方还是下方
//返回值>0 表示(x3,y3)位于直线上方，<0 表示位于下方
double Cross( double x1, double y1, double x2, double y2, double x3, double
y3 )
{
    return (x2-x1)*(y3-y1) - (x3-x1)*(y2-y1);
}
bool IsOk( POINT a, POINT b )  //在构造有向网时判断两个点之间能否连一条边
{
```

```
    if( a.x>=b.x )  return false;
    bool flag=true;             //是否能提前结束判断的状态变量
    int i=0;                    //循环变量
    while( wX[i]<=a.x && i<n )
        i++;
    while( wX[i]<b.x && i<n )   //判断点 a 和 b 之间是否被第 i 堵墙阻挡
    {
        if( Cross(a.x, a.y, b.x, b.y, wX[i], 0)
            *Cross(a.x,a.y,b.x,b.y,wX[i],pY[i][0])<0  //a 和 b 被第 1 段阻挡
            || Cross(a.x, a.y, b.x, b.y, wX[i], pY[i][1])
            *Cross(a.x,a.y,b.x,b.y,wX[i],pY[i][2])<0  //a 和 b 被第 2 段阻挡
            || Cross(a.x, a.y, b.x, b.y, wX[i], pY[i][3])
            *Cross(a.x, a.y, b.x, b.y, wX[i], 10)<0) //a 和 b 被第 3 段阻挡
        {
            flag=false;  break;
        }
        i++;
    }
    return flag;
}
double BellmanFord( int beg, int end ) //求起始顶点 beg 到终止顶点 end 的最短距离
{
    double d[MAXN]; //起点 beg 到其他每个顶点的最短距离
    int i, j;
    for( i=0; i<MAXN; i++ )
        d[i]=INF;
    d[beg]=0;
    bool ex=true;   //是否可以提前退出 bellman 循环的状态变量
    for( i=0; i<pSize && ex; i++ ) //bellman 算法
    {
        ex=false;
        for(j=0;j<eSize;j++) //判断每条边(u,v),能否使顶点 v 的最短路径距离缩短
        {
            if( d[e[j].u]<INF && d[e[j].v]>d[e[j].u] + g[e[j].u][e[j].v] )
            {
                d[e[j].v]=d[e[j].u]+g[e[j].u][e[j].v];
                ex=true;
            }
        }
    }
    return d[end];  //返回起点 beg 到终点 end 的最短路径距离
}
void Solve( )
{
    p[0].x=0;   //起点
    p[0].y=5;
    pSize=1;
```

```
    for( i=0; i<n; i++ )
    {
        scanf( "%lf", &wX[i] ); //读入每堵墙的 x 坐标
        for( j=0; j<4; j++ )
        {
            p[pSize].x=wX[i];
            scanf("%lf", &p[pSize].y); //读入墙上两扇门的 y 坐标
            pY[i][j]=p[pSize].y;
            pSize++;
        }
    }
    p[pSize].x=10;  //终点
    p[pSize].y=5;
    pSize++;
    for( i=0; i<pSize; i++ )    //初始化邻接矩阵 g
    {
        for( j=0; j<pSize; j++ )
            g[i][j]=INF;
    }
    eSize=0;    //边的数目
    for( i=0; i<pSize; i++ )
    {
        for( j=i+1; j<pSize; j++ )
        {
            if( IsOk( p[i], p[j] ) ) //判断第 i 个点和第 j 个点是否连线
            {
                g[i][j]=Dis( p[i], p[j] ); //邻接矩阵
                e[eSize].u=i;   //边
                e[eSize].v=j;
                eSize++;
            }
        }
    }
    //求第 0 个顶点到第 pSize-1 个顶点之间的最短距离
    printf( "%.2lf\n", BellmanFord( 0, pSize-1 ) );
}
int main( )
{
    while( scanf("%d", &n)!=EOF )
    {
        if( n==-1 )  break;
        Solve( );
    }
    return 0;
}
```

## 练　习

4.5　最小运输费用(Minimum Transport Cost), ZOJ1456

**题目描述：**

Spring 国家有 $N$ 个城市。两个城市之间要么有一条运输线路，要么没有。现在有一些货物需要从一个城市运往另一个城市。运输费用包含两部分：通过两个城市之间运输线路的费用，及通过一个城市时必须缴纳的税(起点城市和目标城市除外)。

编写程序，给定起点城市和目标城市，求具有最小费用的线路。

**输入描述：**

输入文件包含多个测试数据。每个测试数据的第 1 行为一个整数 $N$，表示城市的数目。$N$=0 表示输入结束。

每条线路的费用、通过每个城市必须缴纳的税、起点城市和目标城市等这些信息由输入给定，格式如下：

$a_{11}\ a_{12}\ \cdots\ a_{1N}$

$a_{21}\ a_{22}\ \cdots\ a_{2N}$

$\cdots$

$a_{N1}\ a_{N2}\ \cdots\ a_{NN}$

$b_1\quad b_2\quad \cdots\ b_N$

$c\ d$

$e\ f$

$\cdots$

$g\ h$

其中 $a_{ij}$ 为连接城市 $i$ 和城市 $j$ 的线路的费用，$a_{ij}$=−1 表示这两个城市之间没有直接线路。$b_i$ 代表通过城市 $i$ 需要缴纳的税。货物需要从城市 $c$ 运往城市 $d$，从城市 $e$ 运往城市 $f$，…，最后一行 $g$=$h$=−1 代表起点、目标城市序列结束，也代表测试数据结束。

**输出描述：**

对每个测试数据中的每对起点、目标城市序列，输出具有最小费用的路线及经过的城市，格式为：

From $c$ to $d$ :

Path: $c\rightarrow c_1\rightarrow\cdots\rightarrow c_k\rightarrow d$

Total cost : …

…

From $e$ to $f$ :

Path: $e\rightarrow e_1\rightarrow\cdots\rightarrow e_k\rightarrow f$

Total cost : …

注意：如果某对城市之间具有最小费用的线路不止一条，则输出字典序最小的那条。每个测试数据的输出之后，输出一个空行。

**样例输入：**

```
5
0 3 22 -1 4
3 0 5 -1 -1
22 5 0 9 20
-1 -1 9 0 4
4 -1 20 4 0
5 17 8 3 1
1 3
3 5
-1 -1
0
```

**样例输出：**

```
From 1 to 3 :
Path: 1-->5-->4-->3
Total cost : 21

From 3 to 5 :
Path: 3-->4-->5
Total cost : 16
```

4.6　最优连通子集, POJ1192

**题目描述：**

众所周知，可以通过直角坐标系把平面上的任何一个点 $P$ 用一个有序数对$(x, y)$来唯一表示，如果 $x$、$y$ 都是整数，则把点 $P$ 称为整点，否则点 $P$ 称为非整点。把平面上所有整点构成的集合记为 $W$。

定义 1：两个整点 $P_1(x_1, y_1)$, $P_2(x_2, y_2)$，若$|x_1-x_2| + |y_1-y_2|=1$，则称 $P_1$, $P_2$ 相邻，记作 $P_1 \sim P_2$，否则称 $P_1$, $P_2$ 不相邻。

定义 2：设点集 $S$ 是 $W$ 的一个有限子集，即 $S=\{ P_1, P_2, \cdots, P_n \}$ $(n \geqslant 1)$，其中 $P_i(1 \leqslant i \leqslant n)$属于 $W$，称 $S$ 为整点集。

定义 3：设 $S$ 是一个整点集，若点 $R$、$T$ 属于 $S$，且存在一个有限的点序列 $Q_1, Q_2, \cdots, Q_k$ 满足如下条件。

(1) $Q_i$ 属于 $S(1 \leqslant i \leqslant k)$。

(2) $Q_1=R$, $Q_k=T$;

(3) $Q_i \sim Q(i+1)(1 \leqslant i \leqslant k-1)$，即 $Q_i$ 与 $Q_{i+1}$ 相邻；

(4) 对于任何 $1 \leqslant i < j \leqslant k$，有 $Q_i \neq Q_j$。

则称点 $R$ 与点 $T$ 在整点集 $S$ 上连通，把点序列 $Q_1, Q_2, \cdots, Q_k$ 称为整点集 $S$ 中连接点 $R$ 与点 $T$ 的一条道路。

定义 4：若整点集 $V$ 满足：对于 $V$ 中的任何两个整点，$V$ 中有且仅有一条连接这两点的道路，则 $V$ 称为单整点集。

定义 5：对于平面上的每一个整点，可以赋予它一个整数，作为该点的权，于是把一个整点集中所有点的权的总和称为该整点集的权和。

本题希望对于给定的一个单整点集 $V$，求出一个 $V$ 的最优连通子集 $B$，满足如下条件。

(1) $B$ 是 $V$ 的子集。

(2) 对于 $B$ 中的任何两个整点，在 B 中连通。

(3) $B$ 是满足条件(1)和(2)的所有整点集中权和最大的。

**输入描述：**

第 1 行是一个整数 $N(2 \leqslant N \leqslant 1\,000)$，表示单整点集 $V$ 中点的个数。

以下 $N$ 行中，第 $i$ 行($1 \leqslant i \leqslant N$)有三个整数：$X_i, Y_i, C_i$，依次表示第 $i$ 个点的横坐标、纵坐标和权。同一行相邻两数之间用一个空格分隔。$-10^6 \leqslant X_i, Y_i \leqslant 10^6$；$-100 \leqslant C_i \leqslant 100$。

**输出描述：**

仅一个整数，表示所求最优连通集的权和。

**样例输入：**

```
5
0 0 -2
0 1 1
1 0 1
0 -1 1
-1 0 1
```

**样例输出：**

```
2
```

4.7　洞穴袭击(Cave Raider), ZOJ1791, POJ1613

**题目描述：**

Afkiyia 是一座大山，在山里有很多洞穴，这些洞穴通过地道连接。每条地道连接两个洞穴，两个洞穴之间可能有多条地道连接着。一个恐怖分子的首领藏在其中一个洞穴里。在地道和洞穴的接合处，有一扇门。有时，恐怖分子通过关闭两端的门，从而封锁整个地道，然后“清理”地道。当恐怖分子清理地道时，地道中的人(或其他生物)都将死亡。当清理完毕，门又会被开启，地道又可以用了。

现在，军人已经查明恐怖分子首领藏在哪个洞穴里，而且，他们已经知道了恐怖分子清理地道的安排表。军人准备深入到洞穴中，把恐怖分子首领抓出来。请帮助他以最短的时间到达首领所在的洞穴，并确保军人不会被关在某个地道中。

**输入描述：**

输入文件包含多个测试数据。每个测试数据的第 1 行为 4 个正整数：$n, m, s, t$，用空格隔开，其中 $n$ 表示洞穴的数目，这些洞穴的编号为 1, 2, …, $n$；$m$ 为地道的数目，编号为 1, 2, …, $m$；$s$ 为军人在时刻 0 所在的洞穴编号；$t$ 为恐怖分子首领所在的洞穴；$1 \leqslant s, t \leqslant n \leqslant 50$，$m \leqslant 500$。接下来 $m$ 行为 $m$ 条地道的信息：每行为一个整数序列，至多有 35 个整数，用空格隔开；前两个整数为该地道的两个端点所在的洞穴的编号，第 3 个整数为军人通过地道的时间，剩下的整数为一个递增的整数序列，每个整数都不超过 10 000，为该地道的一些关闭和开启时间。

例如，如果这一行为：10 14 5 6 7 8 9，则表示这条地道连接第 10、14 两个洞穴，通过这条地道花费 5 个单位时间，地道会在时刻 6 关闭、时刻 7 开启，时刻 8 又关闭、时刻 9 又开启。注意，这条地道从时刻 6 到时刻 7 会被清理，然后从时刻 8 到时刻 9 又被清理，然后就永远开着。

又如，如果这一行为：10 9 15 8 18 23，这表示这条地道连接第 10、9 两个洞穴，通过这条洞穴花费 15 个单位时间，地道在时刻 8 关闭，时刻 18 开启，时刻 23 关闭，然后永远关闭。

输入文件最后一行为一个 0，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，输出一行。如果军人能到达第 $t$ 个洞穴，则输出最短

时间；如果不能到达，则输出字符'*'。注意，开始时刻为时刻 0，因此，如果 $s=t$，即军人和恐怖分子首领在同一个洞穴，则输出 0。

**样例输入：**

```
2 2 1 2
1 2 5 4 10 14 20 24 30
1 2 6 2 10 22 30
```

**样例输出：**

```
16
```

## 4.3　Bellman-Ford 算法的改进——SPFA 算法

### 4.3.1　算法思想

Bellman-Ford 算法的时间复杂度比较高，为 $O(n^3)$或 $O(nm)$，原因在于 Bellman-Ford 算法要递推 $n$ 次，每次递推，扫描所有的边，在递推 $n$ 次的过程中很多判断是多余的。**SPFA** (Shortest Path Faster Algorithm)**算法**是 Bellman-Ford 算法的一种队列实现，减少了不必要的冗余判断。

SPFA 算法的大致流程是用一个队列来进行维护。初始时将源点加入队列。每次从队列中取出一个顶点，并对所有与它相邻的顶点进行松弛，若某个相邻的顶点松弛成功，则将其入队。重复这样的过程直到队列为空时算法结束。

SPFA 算法，简单地说就是队列优化的 Bellman-Ford 算法，是根据“每个顶点的最短距离不会更新次数太多”的特点来进行优化的。SPFA 算法可以在 $O(km)$的时间复杂度内求出源点到其他所有顶点的最短路径，并且可以处理负权值边。$k$ 为每个顶点入队列的平均次数，可以证明，对于通常的情况，$k$ 为 2 左右。

与 Bellman-Ford 算法类似，SPFA 算法也使用 dist[$i$]数组存储源点 $v_0$ 到顶点 $v_i$ 的最短路径长度，用 path[$i$]数组存储最终求得的源点 $v_0$ 到顶点 $v_i$ 的最短路径上顶点 $v_i$ 的前一个顶点的序号。初始时，dist[$v_0$]=0，其余元素值均为∞；path[$i$]均为 $v_0$；并将源点 $v_0$ 入队列。SPFA 算法的实现过程如下。

(1) 取出队列头顶点 $v$，扫描从顶点 $v$ 发出的每条边，设每条边的终点为 $u$，边<$v$, $u$>的权值为 $w$，如果 dist[$v$] + $w$ < dist[$u$]，则修改 dist[$u$]为 dist[$v$] + $w$，修改 path[$u$]为 $v$，若顶点 $u$ 不在当前队列中，还要将顶点 $u$ 入队列；如果 dist[$v$] + $w$ < dist[$u$]不成立，则对顶点 $u$ 不作任何处理。

(2) 重复执行步骤(1)直至队列为空。

SPFA 算法在形式上和广度优先搜索非常类似，不同的是广度优先搜索中一个顶点出了队列就不可能重新进入队列，但是 SPFA 中一个顶点可能在出队列之后再次被放入队列，也就是说一个顶点改进过其他的顶点之后，过了一段时间可能本身被改进，于是再次用来改进其他的顶点，这样反复迭代下去。

### 4.3.2　算法实现

**例 4.7**　利用 SPFA 算法求图 4.15(a)中顶点 0 到其他各顶点的最短路径长度，并输出对应的最短路径。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 $n$，然后输入每条边的

数据。每条边的数据格式为：$u$ $v$ $w$，分别表示这条边的起点、终点和边上的权值。顶点序号从 0 开始计起。最后一行为−1 −1 −1，表示输入数据的结束。

**分析：**

因为 SPFA 算法在执行过程中需要扫描队列头顶点发出的每条边，所以在实现 SPFA 算法时用邻接表(出边表)存储图便于算法的处理。本题将以图 4.15 所示的有向网为例分析 SPFA 算法的执行过程，其邻接表存储表示如图 4.15(b)所示。

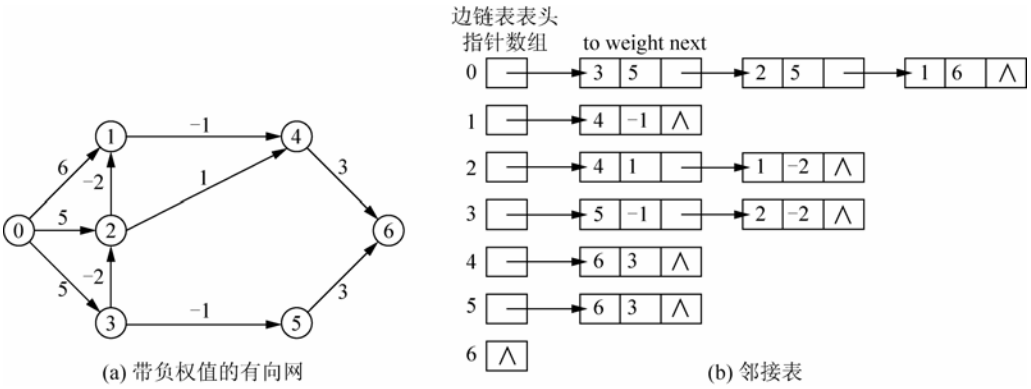



图 4.15　有向网及其邻接表存储表示

图 4.16 所示为 SPFA 算法的执行过程。dist 数组和 path 数组的初始值如图 4.16(a)所示；首先将源点 $v_0$ 入队列。图 4.16(b)描绘了第 1 次取出队列头顶点并修改 dist、path 数组的情形：将队列头顶点(即顶点 $v_0$)取出后，要判断它发出的每条有向边，看是否能松弛每条有向边的终点，因此在图 4.16(b)中一次修改了顶点 $v_3$、$v_2$、$v_1$ 的 dist、path 数组元素值，并将它们一一入队列。图 4.16(c)～图 4.16(h)所示为其余过程。在图 4.16(h)中，取出队列头顶点(即顶点 6)后，队列为空，所以下一次无法从队列中取出顶点。至此，SPFA 算法执行完毕。

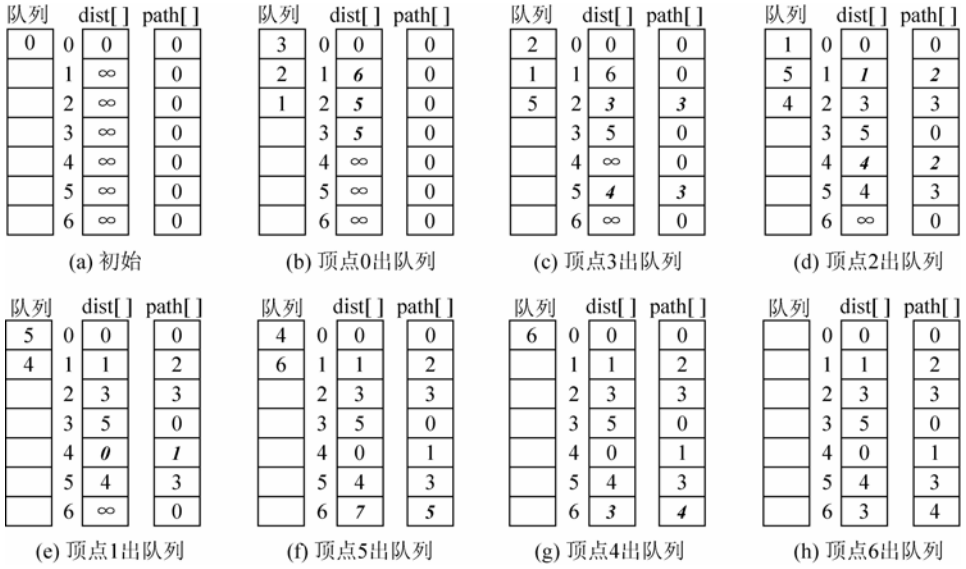



图 4.16　SPFA 算法的求解过程

代码如下：

```
#define INF 1000000  //无穷大
#define MAXN 10
using namespace std;
struct ArcNode
{
    int to;
    int weight;
    ArcNode *next;
};
queue<int> Q;           //队列中的结点为顶点序号
int n;                  //顶点个数
ArcNode* List[MAXN];    //每个顶点的边链表表头指针
int inq[MAXN];          //每个顶点是否在队列中的标志
int dist[MAXN], path[MAXN];
void SPFA( int src )
{
    int i, u;   //u 为队列头顶点序号
    ArcNode* temp;
    for( i=0; i<n; i++ )    //初始化
    {
        dist[i]=INF;  path[i]=src;  inq[i]=0;
    }
    dist[src]=0;  path[src]=src;  inq[src]++;
    Q.push( src );
    while( !Q.empty() )
    {
        u=Q.front( );  Q.pop( );  inq[u]--;
        temp=List[u];
        while( temp!=NULL )
        {
            int v=temp->to;
            if( dist[v]>dist[u]+temp->weight )
            {
                dist[v]=dist[u]+temp->weight;  path[v]=u;
                if( !inq[v] ){ Q.push(v);  inq[v]++; }
            }
            temp=temp->next;
        }
    }
}
int main( )
{
    int i, j;           //循环变量
    int u, v, w;        //边的起点和终点及权值
    scanf( "%d", &n );  //读入顶点个数 n
```

```
    memset( List, 0, sizeof(List) );
    ArcNode* temp;
    while( 1 )
    {
        scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
        if( u==-1 && v==-1 && w==-1 )  break;
        temp=new ArcNode;
        //构造邻接表
        temp->to=v;  temp->weight=w;  temp->next=NULL;
        if( List[u]==NULL )  List[u]=temp;
        else{ temp->next=List[u];  List[u]=temp; }
    }
    SPFA( 0 );  //求顶点 0 到其他顶点的最短路径
    for( j=0; j<n; j++ )    //释放边链表上各边结点所占用的存储空间
    {
        temp=List[j];
        while( temp!=NULL )
        {
            List[j]=temp->next;  delete temp;  temp=List[j];
        }
    }
    int shortest[MAXN];     //输出最短路径上的各个顶点时存放各个顶点的序号
    for( i=1; i<n; i++ )
    {
        printf( "%d\t", dist[i] ); //输出顶点 0 到顶点 i 的最短路径长度
        //以下代码用于输出顶点 0 到顶点 i 的最短路径
        memset( shortest, 0, sizeof(shortest) );
        int k=0;    //k 表示 shortest 数组中最后一个元素的下标
        shortest[k]=i;
        while( path[ shortest[k] ]!=0 )
        {
            k++; shortest[k]=path[ shortest[k-1] ];
        }
        k++; shortest[k]=0;
        for( j=k; j>0; j-- )
            printf( "%d->", shortest[j] );
        printf( "%d\n", shortest[0] );
    }
    return 0;
}
```

该程序的运行示例如下。

**输入：**

```
7
0 1 6
0 2 5
0 3 5
1 4 -1
```

**输出：**

```
1       0->3->2->1
3       0->3->2
5       0->3
0       0->3->2->1->4
4       0->3->5
```

```
2 1 -2                                    3       0->3->2->1->4->6
2 4 1
3 2 -2
3 5 -1
4 6 3
5 6 3
-1 -1 -1
```

### 4.3.3　关于 SPFA 算法的进一步讨论

1. SPFA 算法的时间复杂度分析

在 SPFA 算法中，如果每个顶点都入队列一遍，则将扫描有向图中每条边一次且仅一次，没有重复，其时间复杂度为 $O(m)$；如果每个顶点平均入队列 $k$ 次，则 SPFA 算法的时间复杂度为 $O(km)$。通常的情况，$k$ 为 2 左右。例如对图 4.16 所示的例子，顶点数 $n$ 为 7，各顶点入队列的顺序为：0, 3, 2, 1, 5, 4, 6，总次数为 7，$k$=1。

2. SPFA 算法中判断负权值回路的方法

在 SPFA 算法中，如果一个顶点入队列的次数超过 $n$，则表示有向网中存在负权值回路。具体判断方法请参考例 4.9。

3. 单源最短路径算法时间复杂度的下界

现已证明，对于单源最短路径问题的算法，其最小时间复杂度为 $O(m)$。这是很显然的，因为对任意一个单源最短路径算法来说，要是它不“查完”所有的边，那么这样的算法就不可能是正确的。所以不难相信，$O(m)$时间复杂度的单源最短路径算法是最好的算法。

### 4.3.4　例题解析

以下通过两道例题的分析，再详细介绍 SPFA 算法的基本思想及其实现方法。

**例 4.8**　奶牛派对(Silver Cow Party)

**题目来源：**

USACO 2007 February Silver, POJ3268

**题目描述：**

有 $N$(1≤i≤1 000)个农场，编号为 1～$N$，每个农场有一头奶牛。这些奶牛将参加在 $X$ (1≤$X$≤$N$)号农场举行的派对。这 $N$ 个农场之间有 $M$(1≤$M$≤100 000)条单向路，通过第 $i$ 条路将需要花费 $T_i$(1≤$T_i$≤100)单位时间。

每头奶牛必须走着去参加派对。派对开完以后，返回到它的农场。每头奶牛都很懒，所以总是选择一条具有最短时间的最优路径。每头奶牛的往返路线是不一样的，因为所有的路都是单向的。

对所有的奶牛来说，花费在去派对的路上和返回农场的最长时间是多少？

**输入描述：**

测试数据的格式为：第 1 行为 3 个整数 $N$、$M$ 和 $X$；第 2～$M$+1 行描绘了 $M$ 条单向路，其中第 $i$+1 描绘了第 $i$ 条路，为 3 个整数 $A_i$、$B_i$ 和 $T_i$，表示这条路是从 $A_i$ 农场通往 $B_i$ 农场，

所需时间为 $T_i$。

**输出描述：**

输出占一行，为所有奶牛必须花费的最大时间。

**样例输入：**

```
4 8 2
1 2 4
1 3 2
1 4 7
2 1 1
2 3 5
3 1 2
3 4 4
4 2 3
```

**样例输出：**

```
10
```

**分析：**

样例数据所描绘的有向网如图 4.17 所示。在该测试数据中，第 4 头奶牛去派对的路上，花费时间最少的路径为从农场 4→农场 2，时间为 3；返回农场花费时间最少的路径为从农场 2→农场 1→农场 3→农场 4，时间为 7；总时间为 10。其他奶牛到农场 2 并返回自己的农场所花费的最少时间都比 10 要少，所以答案是 10。

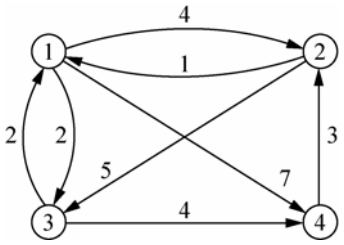



图 4.17　奶牛派对

本题两次运用 SPFA 算法求最短距离。

(1) 以农场 $X$ 为源点，各顶点的出边构成的邻接表，求源点到各农场的最短距离，该距离是从各农场到农场 $X$ 的最少时间；

(2) 以农场 $X$ 为源点，各顶点的入边作为出边，构成邻接表，求源点到各农场的最短距离，该距离是从农场 $X$ 返回各农场的最少时间。

将两个最少时间加起来，得到每头奶牛往返农场 $X$ 的最少时间，本题要求的是这些最少时间的最大值。

代码如下：

```
#define MAXN 101000
#define INT_MAX 2000000
struct NODE  //邻接表结构
{
    int v;
    int w;
    NODE *next;
```

```
}edge[MAXN], redge[MAXN], temp[MAXN * 2]; // edge与redge分别为正向图和反向图
int pos=0;
int ecost[MAXN];
int N, M, W, src, Q[MAXN]; //用数组Q模拟队列
bool visited[MAXN];         //标志数组，记录结点是否已访问
//SPFA算法，direction表示方向，0为正向，1为反向
void SPFA( int direction )
{
    int  h, t, u, i;
    NODE *ptr;
    h=0, t=1;
    memset( visited, 0, sizeof( visited ) );
    for( i=0; i<=N; ++i )
        ecost[i]=INT_MAX;
    Q[0]=src;
    ecost[src]=0;
    while( h != t )
    {
        u=Q[h];
        h++;
        visited[u]=false;
        if( direction!=0 )  ptr=edge[u].next;
        else  ptr=redge[u].next;
        while( ptr )
        {
            if( ecost[ptr->v]>ecost[u]+ptr->w )     //松弛操作
            {
                ecost[ptr->v]=ecost[u]+ptr->w;
                //顶点不在队列中则入队列
                if( !visited[ptr->v] )
                {
                    Q[t]=ptr->v;
                    t++;
                    visited[ptr->v]=true;
                }
            }
            ptr=ptr->next;
        }
    }
}
void Insert( const int& x, const int& y, const int& w ) //对邻接表进行插入
{
    NODE *ptr=&temp[pos++];
    ptr->v=y;
    ptr->w=w;
    ptr->next=edge[x].next;
```

```
    edge[x].next=ptr;
    ptr=&temp[pos++];
    ptr->v=x;
    ptr->w=w;
    ptr->next=redge[y].next;
    redge[y].next=ptr;
}
int main( )
{
    int i, x, y, w;
    int ans[MAXN], MaxTime;     //累加的最短时间以及最短时间的最大值
    while( scanf("%d %d %d", &N, &M, &src)!=EOF )
    {
        pos=0;
        for ( i=0; i<=N; ++i )
        {
            edge[i].next=NULL;
            redge[i].next=NULL;
        }
        for( i=0; i<M; ++i )
        {
            scanf( "%d %d %d", &x, &y, &w );
            Insert( x, y, w );
        }
        MaxTime=0;
        memset( ans, 0, sizeof(ans) );
        //正向求最短路
        SPFA( 0 );
        for( i=1; i<=N; ++i )
        {
            if ( i!=src )  ans[i]+=ecost[i];
        }
        //反向求最短路，并累加最短时间
        SPFA( 1 );
        for( i=1; i<=N; ++i )
        {
            if( i!=src )
            {
                ans[i]+=ecost[i];
                if( ans[i]>MaxTime )    //取最短时间的最大值
                    MaxTime=ans[i];
            }
        }
        printf("%d\n", MaxTime);
    }
    return 0;
}
```

**例 4.9**　昆虫洞(Wormholes)

**题目来源：**

USACO 2006 December Gold, POJ3259

**题目描述：**

当农场主 John 在开垦他的农场时，发现了许多奇怪的昆虫洞。这些昆虫洞是单向的，并且可以从入口到出口，并且使得时间倒退一段。John 的每个农场包含 $N(1\leqslant N\leqslant 500)$块地，编号从 1～$N$，这 $N$ 块地之间有 $M(1\leqslant M\leqslant 2\ 500)$条路、$W(1\leqslant W\leqslant 200)$个昆虫洞。

因为 John 是一个狂热的时间旅行迷，他想尝试着做这样一件事：从某块地出发，通过一些路径和昆虫洞，返回到出发点，并且时间早于出发时间，这样或许他可以遇到他自己。试帮助 John 判断是否可行。他将提供他的 $F(1\leqslant F\leqslant 5)$个农场的完整地图。每条路走完所需时间不超过 10 000 秒，每个昆虫洞倒退的时间也不超过 10 000 秒。

**输入描述：**

输入文件第 1 行为一个整数 $F$，表示 $F$ 个测试数据，每个测试数据描绘了一个农场。接下来是 $F$ 个农场的数据。

每个农场的第 1 行为 3 个整数：$N$、$M$ 和 $W$。第 2 行至第 $M$+1 行，每行为 3 个整数：$S$、$E$ 和 $T$，表示一条从 $S$ 到 $E$ 的双向路径，通过这条路径所需时间为 $T$ 秒，两块地之间至多有一条路。第 $M$+2～第 $M$+$W$+1 行，每行为 3 个整数：$S$、$E$ 和 $T$，表示从 $S$ 到 $E$ 的一条单向路，通过这条路将使得时间倒退 $T$ 秒。

**输出描述：**

对每个测试数据，如果 John 能实现自己的目标，输出"YES"；否则输出"NO"。

**样例输入：**

```
2
3 3 1
1 2 2
1 3 4
2 3 1
3 1 3
3 2 1
1 2 3
2 3 4
3 1 8
```

**样例输出：**

```
NO
YES
```

**分析：**

本题样例输入中两个测试数据所描述的有向网如图 4.18(a)和图 4.18(b)所示。在第 2 个测试数据中，John 沿着路径 1→2→3→1 从第 1 块土地出发、回到第 1 块土地，所需时间为 −1，这样就能碰到他自己了。本题实际上就是要求判断一个带负权的有向网中是否存在负权值回路，下面的代码用 SPFA 算法来实现。

用 SPFA 算法判断是否存在负权值回路的原理是：如果存在负权值回路，那么从源点到某个顶点的最短路径可以无限缩短，某些顶点入队列将超过 $N$ 次($N$ 为顶点个数)。因此，只需在 SPFA 算法中统计每个顶点入队列的次数，在取出队列头顶点时，都判断该顶点入

队列的次数是否已经超过 $N$ 次，如果是，说明存在负权值回路，则 SPFA 算法不需要再进行下去了。

另外，如果存在负权值回路，在退出 SPFA 算法时队列可能不为空，所以在进行下一个测试数据的处理前，要清空队列。

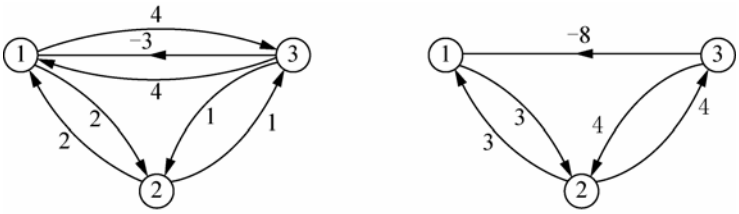



图 4.18　昆虫洞

代码如下：

```
#define INF 100000000   //无穷大
#define MAXN 600
using namespace std;
struct ArcNode
{
    int to;
    int weight;
    ArcNode *next;
};
queue<int> Q;           //队列中的结点为顶点序号
int N, M, W;            //农场个数，双向路径个数，单向路径个数
ArcNode* List[MAXN];    //每个顶点的边链表表头指针
int inq[MAXN];          //每个顶点是否在队列中的标志
int count1[MAXN];       //每个顶点入队列的次数
int dist[MAXN], path[MAXN];
bool SPFA( int src )
{
    int i, u;   //u 为队列头顶点序号
    ArcNode* temp;
    for( i=1; i<=N; i++ )   //初始化
    {
        dist[i]=INF;  path[i]=src;  inq[i]=0;
    }
    dist[src]=0;  path[src]=src;  inq[src]++;
    Q.push( src );  count1[src]++;
    while( !Q.empty() )
    {
        u=Q.front( );  Q.pop( );  inq[u]--;
        if( count1[u]>N )  return true;
        temp=List[u];
        while( temp!=NULL )
```

```
        {
            int v=temp->to;
            if( dist[v]>dist[u]+temp->weight )
            {
                dist[v]=dist[u]+temp->weight; path[v]=u;
                if( !inq[v] ){ Q.push(v);  inq[v]++; count1[v]++; }
            }
            temp=temp->next;
        }
    }
    return false;
}
int main( )
{
    int i, j, T;    //循环变量，T 表示测试数据个数
    int u, v, w;    //边的起点和终点及权值
    scanf( "%d", &T );
    for( i=1; i<=T; i++ )
    {
        scanf( "%d%d%d", &N, &M, &W );
        memset( List, 0, sizeof(List) );
        memset( inq, 0, sizeof(inq) );
        memset( count1, 0, sizeof(count1) );
        while( !Q.empty() )  Q.pop( );  //清空队列
        ArcNode* temp;
        for( j=1; j<=M; j++ )   //双向边的读入
        {
            scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
            temp=new ArcNode;
            temp->to=v;  temp->weight=w;  temp->next=NULL;  //构造邻接表
            if( List[u]==NULL )  List[u]=temp;
            else{ temp->next=List[u];  List[u]=temp; }
            temp=new ArcNode;
            temp->to=u;  temp->weight=w;  temp->next=NULL;  //构造邻接表
            if( List[v]==NULL )  List[v]=temp;
            else{ temp->next=List[v];  List[v]=temp; }
        }
        for( j=1; j<=W; j++ )   //双向边的读入
        {
            scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
            temp=new ArcNode;
            temp->to=v;  temp->weight=-w;  temp->next=NULL; //构造邻接表
            if( List[u]==NULL )  List[u]=temp;
            else{ temp->next=List[u];  List[u]=temp; }
        }
        if( SPFA(1) )  printf( "YES\n" );
        else  printf( "NO\n" );
        for( j=1; j<=N; j++ )   //释放边链表上各边结点所占用的存储空间
```

```
            {
                temp=List[j];
                while( temp!=NULL )
                {
                    List[j]=temp->next;  delete temp;  temp=List[j];
                }
            }
        }
        return 0;
    }
```

## 练　习

4.8　复活节假期(Easter Holidays), ZOJ3088

**题目描述：**

斯堪的纳维亚人经常在复活节假期去最大的滑雪胜地 Are 度假。Are 提供了极好的滑雪条件，有许多不同难度等级的雪橇和滑雪斜坡。然而有些雪橇速度比其他雪橇快，而有些雪橇坐的人比较多，需要排队。

Per 是一个滑雪初学者，他很害怕雪橇，尽管他想滑得尽可能快。现在，他发现他可以选择不同的雪橇和滑雪斜坡。他现在想这样安排他的滑雪行程。

(1) 从一架雪橇的起点出发并最终回到起点。

(2) 这个过程分为两个阶段：第 1 阶段，乘坐一架或多架雪橇上升；第 2 阶段，一直滑下来直到起点。

(3) 尽可能少地避免惊慌。这样，如果花在滑坡斜面上的时间与花在乘坐和等候雪橇的时间的比率越大就越好。

能帮 Per 找到一条惊慌最少的行程吗？

一个滑雪胜地包含了 $n$ 个地点、$m$ 个滑雪斜坡、$k$ 架雪橇，其中 $2 \leqslant n \leqslant 1\,000$、$1 \leqslant m \leqslant 1\,000$、$1 \leqslant k \leqslant 1\,000$。滑雪斜坡和雪橇总是从一个地点到另一个地点：滑雪斜坡是从高地点到低地点，而雪橇刚好相反(注意，雪橇不能下降)。

**输入描述：**

输入文件的第 1 行为一个整数 $T$，表示测试数据的个数，每个测试数据描述了一个滑雪胜地。每个测试数据的格式如下：第 1 行为 3 个整数 $n$、$m$ 和 $k$；接下来是 $m$ 行，描述了 $m$ 个滑雪斜坡，每行为 3 个整数，分别表示斜坡的起点和终点(地点的序号从 1～$n$)以及滑下斜坡所需时间(最大不超过 10 000)；最后是 $k$ 行，描述了 $k$ 个雪橇，每行也是 3 个整数，分别表示雪橇的起点和终点以及排队等候雪橇和乘坐雪橇到达终点所需时间(最大不超过 10 000)。假定任何两个地点之间的滑雪斜坡至多有一个，雪橇也至多有一个。

**输出描述：**

对每个测试数据，输出两行。第 1 行按被访问的顺序列出各地点，用空格隔开，第 1 个地点和最后一个地点必须一样。第 2 行为花在滑雪斜坡上的时间与乘坐雪橇和等候雪橇的时间的比率，精确到 1/1 000(如果有两种可能，则向远离 0 的方向舍入，例如 1.981 2 和 1.980 6 精确为 1.981，3.133 5 精确为 3.134，3.134 5 精确为 3.135)。如果有多个行程可以优先选择，且具有相等的最小惊慌，则输出任意一个。

**样例输入：**

```
1
5 4 3
1 3 12
2 3 6
3 4 9
5 4 9
4 5 12
5 1 12
4 2 18
```

**样例输出：**

```
4 5 1 3 4
0.875
```

4.9　攀岩(Cliff Climbing), ZOJ3103

**题目描述：**

17 点整，Jack 开始从敌营逃脱。在敌营和安全地之间有处悬崖。Jack 必须攀爬上这处几乎垂直的悬崖，在攀爬时，Jack 的脚踩在悬崖的凸出块上。有些凸出块很光滑，Jack 不得不小心通过，这需要花费 Jack 一定的时间；而有些凸出块太松了，不足以承受他的体重，所以他不得不绕过去。试编写程序，计算 Jack 攀爬完悬崖所需的最少时间。

图 4.19(a)给出了测试数据的一个例子。悬崖上布满了方块，Jack 从悬崖底下的地面上开始攀爬，将他的左脚或右脚踏在最底下一行中标有'S'的方块上。方块中的数字标明方块的“光滑等级”，标有数字 $t$ 的方块将花费他 $t$ 个单位时间安全地将他的脚踏在上面，$1 \leqslant t \leqslant 9$。他不能将脚踏在标有'X'的方块上。当他把左脚或右脚踏在最上一行中标有'T'的方块时，他才算成功的攀爬上悬崖了。

Jack 的攀爬还必须满足以下的限制：当他把左脚(或右脚)踏在一个方块上，他才能移动右脚(或左脚)；假设他左脚的位置为$(l_x, l_y)$，右脚的位置为$(r_x, r_y)$，必须满足 $l_x < r_x$、$|l_x - r_x| + |l_y - r_y| \leqslant 3$。这意味着，如图 4.19(b)所示，给定左脚的位置，他只能将右脚踏在有阴影的方块上；同样，如图 4.19(c)所示，给定右脚的位置，他只能将左脚踏在有阴影的方块上。

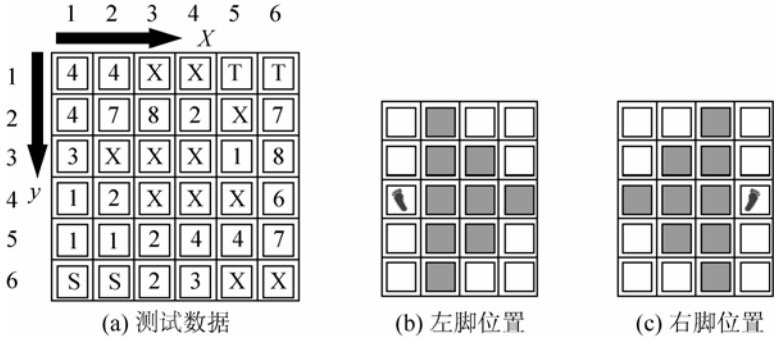



图 4.19　攀岩

**输入描述：**

输入文件中包含多个测试数据。输入文件中最后一行为两个 0，表示输入结束。每个测试数据的格式为：

*w h*

$s(1, 1)$ … $s(1, w)$

$s(2, 1)$ … $s(2, w)$

…

$s(h, 1)$ … $s(h, w)$

即，第 1 行为两个整数 $w$ 和 $h$，分别表示悬崖的宽度和高度，$2 \leqslant w \leqslant 30$，$5 \leqslant h \leqslant 60$；接下来有 $h$ 行，每行有 $w$ 个字符，字符含义如下。

'S'：起点。

'T'：终点。

'X'：Jack 不能将他的脚踏在这种方块上。

'1' - '9'：Jack 必须花费 $t$ 时间才能将他的脚踏在上面。

假定 Jack 将他的脚踏在'S'或'T'上是不需要时间的。

**输出描述：**

对每个测试数据，输出一行，如果 Jack 能成功攀爬上悬崖，输出攀爬完整个悬崖所需的最少时间；否则输出"-1"。

**样例输入：**

```
6 6
4 4 X X T T
4 7 8 2 X 7
3 X X X 1 8
1 2 X X X 6
1 1 2 4 4 7
S S 2 3 X X
```

**样例输出：**

```
12
```

## 4.4 所有顶点之间的最短路径——Floyd 算法

问题的提出：已知一个有向网(或无向网)，对每一对顶点 $v_i \neq v_j$，要求求出 $v_i$ 与 $v_j$ 之间的最短路径和最短路径长度。

解决该问题的方法有如下两种。

(1) 轮流以每个顶点为源点，重复执行 Dijkstra 算法(或 Bellman-Ford 算法)$n$ 次，就可求出每一对顶点之间的最短路径和最短路径长度，总的时间复杂度是 $O(n^3)$(或 $O(n^2+ne)$。

(2) 采用 Floyd(弗洛伊德)算法。Floyd 算法的时间复杂度也是 $O(n^3)$，但 Floyd 算法形式更直接。

### 4.4.1 算法思想

Floyd(弗洛伊德)算法的基本思想是：对一个顶点个数为 $n$ 的有向网(或无向网)，设置一个 $n \times n$ 的方阵 $A^{(k)}$，其中除对角线的矩阵元素都等于 0 外，其他元素 $A^{(k)}[i][j]$ $(i \neq j)$表示从顶点 $v_i$ 到顶点 $v_j$ 的有向路径长度，$k$ 表示运算步骤，$k$=−1, 0, 1, 2, …, $n$−1。

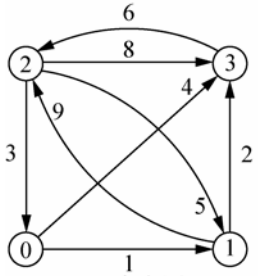
初始时：$A^{(-1)}$=Edge(图的邻接矩阵)，即初始时，以任意两个顶点之间的直接有向边的权值作为最短路径长度。

(1) 对于任意两个顶点 $v_i$ 和 $v_j$，若它们之间存在有向边，则以此边上的权值作为它们之间的最短路径长度。

(2) 若它们之间不存在有向边，则以 MAX 作为它们之间的最短路径。

以后逐步尝试在原路径中加入其他顶点作为中间顶点，如果增加中间顶点后，得到的路径比原来的最短路径长度减少了，则以此新路径代替原路径，修改矩阵元素，更新为新的更短的路径长度。

例如，在图 4.20 所示的有向网中，初始时，从顶点 $v_2$ 到顶点 $v_1$ 的最短路径距离为直接有向边<$v_2$, $v_1$>上的权值(=5)。加入中间顶点 $v_0$ 之后，边<$v_2$, $v_0$>和<$v_0$, $v_1$>上的权值之和(=4)小于原来的最短路径长度，则以此新路径<$v_2$, $v_0$, $v_1$>的长度作为从顶点 $v_2$ 到顶点 $v_1$ 的最短路径距离 A[2][1]。




(a) 有向网

$$\mathbf{Edge}=\begin{array}{c}\begin{matrix}0 & 1 & 2 & 3\end{matrix}\\ \begin{bmatrix}0 & 1 & \infty & 4\\ \infty & 0 & 9 & 2\\ 3 & 5 & 0 & 8\\ \infty & \infty & 6 & 0\end{bmatrix}\end{array}\begin{matrix}0\\1\\2\\3\end{matrix}$$

(b) 邻接矩阵

图 4.20　Floyd 算法：有向网及其邻接矩阵

将 $v_0$ 作为中间顶点可能还会改变其他顶点之间的距离。例如，路径<$v_2$, $v_0$, $v_3$>的长度(=7)小于原来的直接有向边<$v_2$, $v_3$>上的权值(=8)，矩阵元素 A[2][3]也要修改。

在下一步中又增加顶点 $v_1$ 作为中间顶点，对于图中的每一条有向边<$v_i$, $v_j$>，要比较从 $v_i$ 到 $v_1$ 的最短路径长度加上从 $v_1$ 到 $v_j$ 的最短路径长度是否小于原来从 $v_i$ 到 $v_j$ 的最短路径长度，即判断 A[$i$][1] + A[1][$j$] < A[$i$][$j$]是否成立。如果成立，则需要用 A[$i$][1] + A[1][$j$]的值代替 A[$i$][$j$]的值。这时，从 $v_i$ 到 $v_1$ 的最短路径长度，以及从 $v_1$ 到 $v_j$ 的最短路径长度已经由于 $v_0$ 作为中间顶点而修改过了，所以最新的 A[$i$][$j$]实际上是包含了顶点 $v_i$, $v_0$, $v_1$, $v_j$ 的路径的长度。

如图 4.20 所示，A[2][3]在引入中间顶点 $v_0$ 后，其值减为 7，再引入中间顶点 $v_1$ 后，其值又减到 6。当然，有时加入中间顶点后的路径较原路径更长，这时就维持原来相应的矩阵元素的值不变。依此类推，可得到 Floyd 算法。

Floyd 算法的描述如下。

定义一个 $n$ 阶方阵序列：$A^{(-1)}, A^{(0)}, A^{(1)}, \cdots, A^{(n-1)}$，其中：

$A^{(-1)}[i][j]$表示顶点 $v_i$ 到顶点 $v_j$ 的直接边的长度，$A^{(-1)}$就是邻接矩阵 **Edge**[$n$][$n$]。

$A^{(0)}[i][j]$表示从顶点 $v_i$ 到顶点 $v_j$，中间顶点(如果有，则)是 $v_0$ 的最短路径长度。

$A^{(1)}[i][j]$表示从顶点 $v_i$ 到顶点 $v_j$，中间顶点序号不大于 1 的最短路径长度。

……

$A^{(k)}[i][j]$表示从顶点 $v_i$ 到顶点 $v_j$ 的，中间顶点序号不大于 $k$ 的最短路径长度。

……

$A^{(n-1)}[i][j]$是最终求得的从顶点 $v_i$ 到顶点 $v_j$ 的最短路径长度。

采用递推方式计算 $A^{(k)}[i][j]$。

增加顶点 $v_k$ 作为中间顶点后，对于图中的每一对顶点 $v_i$ 和 $v_j$，要比较从 $v_i$ 到 $v_k$ 的最短路径长度加上从 $v_k$ 到 $v_j$ 的最短路径长度是否小于原来从 $v_i$ 到 $v_j$ 的最短路径长度，即比较 $A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$与 $A^{(k-1)}[i][j]$的大小，取较小者作为的 $A^{(k)}[i][j]$值。

因此，Floyd 算法的递推公式为：

$$A^{(-1)}[i][j]=\text{Edge}[i][j]$$

$$A^{(k)}[i][j]=\min\{ A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j] \}, k=0, 1, \cdots, n-1$$

### 4.4.2 算法实现

Floyd 算法在实现时，需要使用两个数组。

(1) 数组 A：使用同一个数组 A[$i$][$j$]来存放一系列的 $A^{(k)}[i][j]$，其中 $k$=−1, 0, 1, …, $n$−1。初始时，A[$i$][$j$]=Edge[$i$][$j$]，算法结束时 A[$i$][$j$]中存放的是从顶点 $v_i$ 到顶点 $v_j$ 的最短路径长度。

(2) path 数组：path[$i$][$j$]是从顶点 $v_i$ 到顶点 $v_j$ 的最短路径上顶点 $j$ 的前一顶点的序号。

Floyd 算法具体实现代码详见例 4.10。

**例 4.10** 利用 Floyd 算法求图 4.20(a)中各顶点间的最短路径长度，并输出对应的最短路径。

假设数据输入时采用如下的格式进行输入：首先输入顶点个数 $n$，然后输入每条边的数据。每条边的数据格式为：$u$ $v$ $w$，分别表示这条边的起点、终点和边上的权值。顶点序号从 0 开始计起。最后一行为−1 −1 −1，表示输入数据的结束。

**分析：**

如图 4.21 所示，初始时，数组 A 实际上就是邻接矩阵。path 数组的初始值：如果顶点 $v_i$ 到顶点 $v_j$ 有直接路径，则 path[$i$][$j$]初始为 $i$；如果顶点 $v_i$ 到顶点 $v_j$ 没有直接路径，则 path[$i$][$j$]初始为−1。在 Floyd 算法执行过程中，数组 A 和 path 各元素值的变化如图 4.21 所示。在该图中，如果数组元素的值有变化，则用粗体、斜体标明。

| | $A^{(-1)}$ | | | | $A^{(0)}$ | | | | $A^{(1)}$ | | | | $A^{(2)}$ | | | | $A^{(3)}$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | ∞ | **4** | 0 | 1 | ∞ | **4** | 0 | 1 | ***10*** | ***3*** | 0 | 1 | 10 | 3 | 0 | 1 | ***9*** | 3 |
| 1 | ∞ | 0 | 9 | 2 | ∞ | 0 | 9 | 2 | ∞ | 0 | 9 | 2 | ***12*** | 0 | 9 | 2 | ***11*** | 0 | ***8*** | 2 |
| 2 | 3 | 5 | 0 | 8 | 3 | ***4*** | 0 | ***7*** | 3 | **4** | 0 | ***6*** | 3 | **4** | 0 | 6 | 3 | **4** | 0 | 6 |
| 3 | ∞ | ∞ | 6 | 0 | ∞ | ∞ | 6 | 0 | ∞ | ∞ | 6 | 0 | ***9*** | ***10*** | 6 | 0 | 9 | 10 | 6 | 0 |
| | $path^{(-1)}$ | | | | $path^{(0)}$ | | | | $path^{(1)}$ | | | | $path^{(2)}$ | | | | $path^{(3)}$ | | | |
| | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 0 | −1 | 0 | −1 | 0 | −1 | 0 | −1 | 0 | −1 | 0 | ***1*** | ***1*** | −1 | 0 | 1 | 1 | −1 | 0 | ***3*** | 1 |
| 1 | −1 | −1 | 1 | 1 | −1 | −1 | 1 | 1 | −1 | −1 | 1 | 1 | ***2*** | −1 | 1 | 1 | ***2*** | −1 | ***3*** | 1 |
| 2 | 2 | 2 | −1 | 2 | 2 | ***0*** | −1 | ***0*** | 2 | 0 | −1 | ***1*** | 2 | 0 | −1 | 1 | 2 | 0 | −1 | 1 |
| 3 | −1 | −1 | 3 | −1 | −1 | −1 | 3 | −1 | −1 | −1 | 3 | −1 | ***2*** | ***0*** | 3 | −1 | 2 | 0 | 3 | −1 |

在递推 $A^{(k)}[i][j]$和 $path^{(k)}[i][j]$时，有更新的用粗体、斜体标明

图 4.21 Floyd 算法的求解过程中数组 A 和 path 的变化

以从 $A^{(-1)}$推导到 $A^{(0)}$解释 $A^{(k)}$的推导。从 $A^{(-1)}$推导到 $A^{(0)}$，实际上是将 $v_0$ 作为中间顶点。引入中间顶点 $v_0$ 后，因为 $A^{(-1)}[2][0] + A^{(-1)}[0][1]=4$，小于 $A^{(-1)}[2][1]$，所以要将 $A^{(0)}[2][1]$修改成 $A^{(-1)}[2][0] + A^{(-1)}[0][1]$，为 4；同样 $A^{(0)}[2][3]$的值也要更新成 7。

当 Floyd 算法运算完毕，如何根据 path 数组确定顶点 $v_i$ 到顶点 $v_j$ 的最短路径？方法与 Dijkstra 算法和 Bellman-Ford 算法类似。以顶点 $v_1$ 到顶点 $v_0$ 的最短路径加以解释。如图 4.21 所示，从 $path^{(3)}[1][0]=2$ 可知，最短路径上 $v_0$ 的前一个顶点是 $v_2$；从 $path^{(3)}[1][2]=3$ 可知，最短路径上 $v_2$ 的前一个顶点是 $v_3$；从 $path^{(3)}[1][3]=1$ 可知，最短路径上 $v_3$ 的前一个顶点是 $v_1$，就是最短路径的起点；因此，从顶点 1 到顶点 0 的最短路径为：$v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_0$，最短路径长度为 A[1][0]=11。

代码如下：

```
#define INF 1000000     //无穷大
#define MAXN 8
int n;                  //顶点个数
int Edge[MAXN][MAXN];   //邻接矩阵
int A[MAXN][MAXN], path[MAXN][MAXN];
void Floyd( )           //假定图的邻接矩阵和顶点个数已经读进来了
{
    int i, j, k;
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )
        {
            A[i][j]=Edge[i][j]; //对 a[ ][ ]初始化
            if( i!=j && A[i][j]<INF )  path[i][j]=i; //i 到 j 有路径
            else path[i][j]=-1; //从 i 到 j 没有直接路径
        }
    }
    //从 A(-1)递推到 A(0), A(1), ..., A(n-1),
    //或者理解成依次将 v0, v1, ..., v(n-1)作为中间顶点
    for( k=0; k<n; k++ )
    {
        for( i=0; i<n; i++ )
        {
            for( j=0; j<n; j++ )
            {
                if( k==i || k==j )  continue;
                if( A[i][k]+A[k][j]<A[i][j] )
                {
                    A[i][j]=A[i][k]+A[k][j] ;
                    path[i][j]=path[k][j];
                }
            }
        }
    }
}
```

```
int main( )
{
    int i, j;               //循环变量
    int u, v, w;            //边的起点和终点及权值
    scanf( "%d", &n );      //读入顶点个数 n
    for( i=0; i<n; i++ )    //设置邻接矩阵中每个元素的初始值为 INF
    {
        for( j=0; j<n; j++ )  Edge[i][j]=INF;
    }
    for( i=0; i<n; i++ )    //设置邻接矩阵中对角线上的元素值为 0
    {
        Edge[i][i]=0;
    }
    while( 1 )
    {
        scanf( "%d%d%d", &u, &v, &w ); //读入边的起点和终点
        if( u==-1 && v==-1 && w==-1 )  break;
        Edge[u][v]=w;                   //构造邻接矩阵
    }
    Floyd( );           //求各对顶点间的最短路径
    int shortest[MAXN]; //输出最短路径上的各个顶点时存放各个顶点的序号
    for( i=0; i<n; i++ )
    {
        for( j=0; j<n; j++ )
        {
            if( i==j )  continue;   //跳过
            //输出顶点 i 到顶点 j 的最短路径长度
            printf( "%d=>%d\t%d\t", i, j, A[i][j] );
            //以下代码用于输出顶点 0 到顶点 i 的最短路径
            memset( shortest, 0, sizeof(shortest) );
            int k=0;    //k 表示 shortest 数组中最后一个元素的下标
            shortest[k]=j;
            while( path[i][ shortest[k] ] != i )
            {
                k++; shortest[k]=path[i][ shortest[k-1] ];
            }
            k++; shortest[k]=i;
            for( int t=k; t>0; t-- )
                printf( "%d→", shortest[t] );
            printf( "%d\n", shortest[0] );
        }
    }
    return 0;
}
```

该程序的运行示例如下。

**输入：**

```
4
0 1 1
0 3 4
```

**输出：**

```
0=>1    1       0→1
0=>2    9       0→1→3→2
0=>3    3       0→1→3
```

```
1 2 9
1 3 2
2 0 3
2 1 5
2 3 8
3 2 6
-1 -1 -1
```

```
1=>0    11      1→3→2→0
1=>2    8       1→3→2
1=>3    2       1→3
2=>0    3       2→0
2=>1    4       2→0→1
2=>3    6       2→0→1→3
3=>0    9       3→2→0
3=>1    10      3→2→0→1
3=>2    6       3→2
```

### 4.4.3　关于 Floyd 算法的进一步分析

1. Floyd 算法时间复杂度分析

在例 4.10 的 Floyd 算法代码中，有一个三重嵌套的 for 循环，因此最内层的 if 语句总执行次数为 $n^3$，所以 Floyd 算法的时间复杂度为 $O(n^3)$。并且因为 Floyd 算法的思想是逐渐将顶点 $v_0, v_1, \cdots, v_u, \cdots, v_{n-1}$ 作为中间顶点，判断是否能减小任意一对 $v_i$ 和 $v_j$ 之间的最短路径长度，在这个过程需要用到顶点 $v_i$ 到顶点 $v_u$、及顶点 $v_u$ 到顶点 $v_j$ 的最短路径长度。这些信息都是保存在矩阵 A 中，而邻接矩阵只是用来初始化 A 的，所以改用邻接表存储图，并不能降低算法的时间复杂度。

2. Floyd 算法的适用范围

与 Bellman-Ford 算法类似，Floyd 算法允许图中有带负权值的边，但不允许有包含负权值回路。

3. Floyd 算法思想的应用

Floyd 算法中的递推公式：

$$A^{(k)}[i][j]=\min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k]+A^{(k-1)}[k][j]\}$$

可以灵活使用，比如可以把求较小值的 min 运算改成求较大值的 max 运算、“或”运算，把加法运算改成 min 运算、“与”运算等。详见例 4.11、例 4.12 和练习 4.10 等题目。

### 4.4.4　例题解析

以下通过两道例题的分析，再详细介绍 Floyd 算法的基本思想及其实现方法。

**例 4.11**　光纤网络(Fiber Network)

**题目来源：**

University of Ulm Local Contest 2001, ZOJ1967, POJ2570

**题目描述：**

一些公司决定搭建一个更快的网络，称为“光纤网”。他们已经在全世界建立了许多站点，这些站点的作用类似于路由器。不幸的是，这些公司在关于站点之间的接线问题上存在争论，这样“光纤网”项目就被迫终止了，留下的是每个公司自己在某些站点之间铺设的线路。

现在，当 Internet 服务供应商想从站点 *A* 传送数据到站点 *B*，就感到困惑了，到底哪个公司能够提供必要的连接。请帮助供应商回答他们的查询，查询所有可以提供从站点 *A* 到站定 *B* 的线路连接的公司。

**输入描述：**

输入文件包含多个测试数据。每个测试数据第 1 行为一个整数 *n*，代表网络中站点的个数，*n*=0 代表输入结束，否则 *n* 的范围为：1≤*n*≤200。站点的编号为 1, …, *n*。接下来列出了这些站点之间的连接。每对连接占一行，首先是两个整数 *A* 和 *B*，*A*=*B*=0 代表连接列表结束，否则 *A*、*B* 的范围为：1≤*A*, *B*≤*n*，表示站点 *A* 和站点 *B* 之间的单向连接；每行后面列出了拥有站点 *A* 到 *B* 之间连接的公司，公司用小写字母标识，多个公司的集合为包含小写字母的字符串。

连接列表之后，是供应商查询的列表。每个查询包含两个整数 *A* 和 *B*，*A*=*B*=0 代表查询列表结束，也代表整个测试数据结束，否则 *A*、*B* 的范围为：1≤*A*, *B*≤*n*，代表查询的起始和终止站点。假定任何一对连接和查询的两个站点都不相同。

**输出描述：**

对测试数据中的每个查询，输出一行，为满足以下条件的所有公司的标识：这些公司可以通过自己的线路为供应商提供从查询的起始站点到终止站点的数据通路。如果没有满足条件的公司，则仅输出字符"-"。每个测试数据的输出之后输出一个空行。

**样例输入：**

```
3
1 2 abc
2 3 ad
1 3 b
3 1 de
0 0
1 3
2 1
3 2
0 0
2
1 2 z
0 0
1 2
2 1
0 0
0
```

**样例输出：**

```
ab
d
-

z
-
```

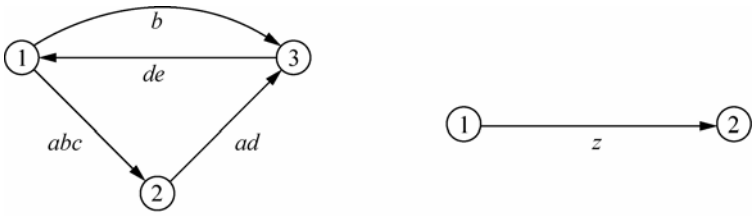
**分析：**

样例输入数据所描述的两个光纤网络如图 4.22(a)和 4.22(b)所示。

在第 1 个测试数据中，公司 *a* 和 *b* 都可以提供站点 1 到站点 3 之间的连接，其中公司 *b* 是直接连接站点 1 到站点 3，而公司 *a* 要通过中间站点 2 来连接站点 1 到站点 3；所以查询站点 1 到站点 3 之间的连接时，输出为 *ab*。与此类似，查询站点 2 到站点 1 之间的连接

时，输出 *d*。没有哪个公司能提供站点 3 到站点 2 之间的连接，所以输出"-"。

在第 2 个测试数据中，公司 *z* 能提供站点 1 到站点 2 之间的连接，所以查询站点 1 到站点 2 之间的连接时，输出为 *z*；而查询站点 2 到站点 1 之间的连接时，输出"-"。




(a) 测试数据1　　(b) 测试数据2

图 4.22　光纤网络：样例输入数据所描述的两个道路网络

本题并不是求解最短路径，但要用到 Floyd 算法求解最短路径的思想。定义一个 *n* 阶方阵序列：$m^{(0)}, m^{(1)}, \cdots, m^{(n)}$，其中：

$m^{(0)}[A][B]$表示提供直接连接站点 *A* 到站点 *B* 的所有公司组成的集合。

$m^{(1)}[A][B]$表示中间站点(如果有，则)取为站点 1，能提供从站点 *A* 到站点 *B* 之间连接的所有公司的集合。

…

$m^{(k)}[A][B]$表示中间站点序号不超过 *k*，能提供从站点 *A* 到站点 *B* 之间连接的所有公司的集合。

…

$m^{(n)}[A][B]$为最终所求，表示能够提供站点 *A* 到站点 *B* 之间连接的所有公司。

从 $m^{(k-1)}[A][B]$递推到 $m^{(k)}[A][B]$的方法如下。$m^{(k)}[A][B]$包括两部分：其中一部分为 $m^{(k-1)}[A][B]$；另一部分为 $m^{(k-1)}[A][k]$和 $A^{(k-1)}[k][B]$的交集，即既能提供站点 *A* 到站点 *k* 的连接，也能提供站点 *k* 到站点 *B* 的连接。

在本题中，要将 Floyd 算法的递推公式改成：

$A^{(0)}[i][j]$=提供站点 *i* 到站点 *j* 直接连接的所有公司组成的集合

$A^{(k)}[i][j]$=或运算{ $A^{(k-1)}[i][j]$, 与运算($A^{(k-1)}[i][k]$, $A^{(k-1)}[k][j]$), $k$=0, 1, …, $n$

另外，在本题中，需要很巧妙地处理公司集合。因为公司是用一个小写字母标识的，且每个公司的标识互不相同，也就是说最多只有 26 个公司，因此，可以用整数中的二进制位来表示每个公司，通过二进制位运算来实现集合的“或”运算和“与”运算。

例如，图 4.17(a)中，提供站点 1 到站点 2 的连接的公司集合为{ 'a', 'b', 'c' }，可以用“00000000000000000000000000000111”表示，提供站点 2 到站点 3 的连接的公司集合为{ 'a', 'd' }，用“00000000000000000000000000001001”表示，这两个整数进行二进制与运算后，得到“00000000000000000000000000000001”，表示通过中间站点 2，提供站点 1 到站点 3 的连接的公司集合为{ 'a' }。

这样上述递推公式可以改写成：

$$m[i][j] \mid= m[i][k] \ \&\ m[k][j]。$$

作了这样的处理后，还需要特别解决公司集合输入输出的问题。在读入提供站点 *A* 到站点 *B* 直接连接的公司字符串至字符数组 str 后，对其中的每个字符 str[*i*]，逻辑左移 str[*i*]−'a'位后添加到数组元素 m[*A*][*B*]中，其公式为：

m[A][B] |= 1 << (str[*i*] - 'a')。

在输出时，循环变量 ch 分别取值为'a'～'z'，如果“m[*A*][*B*] & (1 << ch-'a')”为 1，则表示 m[*A*][*B*]所代表的集合中包含公司 ch，则要输出。

代码如下：

```
#define MAXN 201
int main( )
{
    int m[MAXN][MAXN];  //Floyd 算法中的矩阵 A
    int n;              //站点个数
    int A, B;           //每对连接中的两个顶点，及每对查询中的每个顶点
    int i, j, k;        //循环变量
    char str[27];       //用来读入每对连接后的公司标识列表
    char ch;            //循环变量
    while( scanf("%d", &n) && n )
    {
        memset( m, 0, sizeof(m) ); //初始化矩阵 m
        //while 循环执行完毕后,m[A][B]为提供直接连接 A 和 B 的公司
        while( scanf( "%d %d", &A, &B ) )
        {
            if( A==0 && B==0 )  break;
            scanf( "%s", str );
            for( i=0; str[i]; ++i )
            {
                m[A][B]|=1<<(str[i]-'a');
            }
        }
        for( k=1; k<=n; ++k )   //Floyd 算法
        {
            for( i=1; i<=n; ++i )
            {
                for( j=1; j<=n; ++j )
                {
                    m[i][j]|=m[i][k] & m[k][j];
                }
            }
        }
        while( scanf( "%d %d", &A, &B ) )  //查询
        {
            if( A==0 && B==0 )  break;
            for( ch='a';ch<='z';++ch )      //输出
            {
                if( m[A][B] & (1<<ch-'a') )
```

```
                    putchar( ch );
                }
                if( !m[A][B] )  putchar( '-' );
                putchar( '\n' );
                }
                putchar( '\n' );
        }
        return 0;
}
```

例 4.12　重型运输(Heavy Cargo)

**题目来源：**

University of Ulm Local Contest 1998, ZOJ1952, POJ2263

**题目描述：**

Big Johnsson 运输汽车制造公司是专门生产大型汽车的厂商。它们最新型号的运输车，Godzilla V12，运载量是如此之大，以至于它所能装载的重量从不取决于它本身，而是取决于所经过道路的承载限制。

给定起点和终点城市，试计算 Godzilla V12 能够装载的最大重量，使得从起点城市到终点城市所经的路径不会超过道路的承载限制。

**输入描述：**

输入文件包含多个测试数据，每个测试数据的第一行为两个整数：城市的个数 $n$($2 \leqslant n \leqslant 200$)，组成道路网络的道路的条数 $r$($1 \leqslant r \leqslant 19\,900$)。

接下来有 $r$ 行，每行描述了一条直接连接两个城市的道路，格式为：所连接的两个城市的名字，道路的承载限制。城市的名称不会超过 30 个字符，并且不会包含空格字符。重量限制是 0 到 10 000 之间的整数。道路是双向的。

每个测试数据的最后一行是两个城市的名称：起点城市和终点城市。

输入文件的最后一行是两个 0，为 $n$ 和 $r$ 的取值，代表输入的结束。

**输出描述：**

对每个测试数据，输出以下 3 行。

第 1 行格式为："Scenario #x"，其中 $x$ 是测试数据的序号。

第 2 行为："y tons"，其中 $y$ 为可能装载的最大重量。

第 3 行为空行。

**样例输入：**

```
4 3
Karlsruhe Stuttgart 100
Stuttgart Ulm 80
Ulm Muenchen 120
Karlsruhe Muenchen
5 5
Karlsruhe Stuttgart 100
Stuttgart Ulm 80
Ulm Muenchen 120
```

**样例输出：**

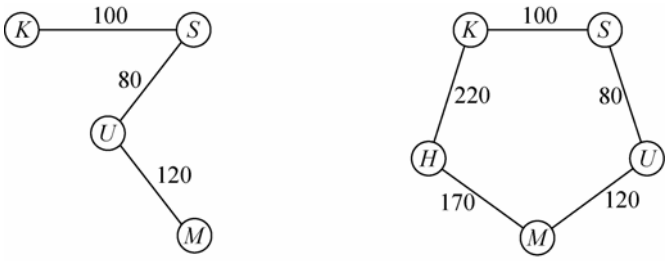
```
Scenario #1
80 tons

Scenario #2
170 tons
```

```
Karlsruhe Hamburg 220
Hamburg Muenchen 170
Muenchen Karlsruhe
0 0
```

**分析：**

样例输入数据所描述的两个道路网络如图 4.23(a)和图 4.23(b)所示。在图 4.23 中，用城市名称中第 1 个字母来表示道路网络中的顶点。第 1 个测试数据要求从 Karlsruhe 到 Muenchen 所经的所有路径中承载限制的最大值，答案为 80；第 2 个测试数据要求从 Muenchen 到 Karlsruhe 所经的所有路径中承载限制的最大值，答案为 170。




(a) 第1个测试数据　　(b) 第2个测试数据

图 4.23　重型运输：样例输入数据所描述的两个道路网络

本题并不是求解最短路径，而是要求通行能力(可称为容量)最大的路，这种路可以称为最大容量路，可用 Floyd 算法求最短路径的思想来求解。

以样例输入中第 2 个测试数据(对应的道路网络如图 4.23(b)所示)为例解释。设城市 $i$ 到城市 $j$ 的承载重量记为$[i, j]$。初始时 $K$ 到 $M$ 没有直接路径，因此 $K$ 到 $M$ 的承载重量为 0，即$[K, M]$=0。加入中间结点 $H$ 后，$K$ 到 $M$ 的承载重量改为：

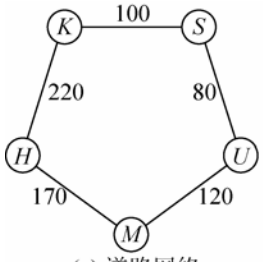
$$\text{MAX}\{\ [K, M], \text{MIN}\{\ [K, H], [H, M]\ \}\ \},$$

其值为 170。这就是 Floyd 算法的思想。在本题中，要将 Floyd 算法的递推公式改成：

$A^{(-1)}[i][j]=\text{Edge}[i][j]$

$A^{(k)}[i][j]=\max\{\ A^{(k-1}\ [i][j], \min(A^{(k-1)}[i][k], A^{(k-1)}[k][j]), k=0, 1, \cdots, n-1$

样例输入中第 2 个测试数据的求解过程如图 4.24(c)所示，最终 $A^{(4)}[0][3]=170$ 即为所求。




(a) 道路网络

$$\begin{bmatrix} \infty & 100 & 0 & 0 & 220 \\ 100 & \infty & 80 & 0 & 0 \\ 0 & 80 & \infty & 120 & 0 \\ 0 & 0 & 120 & \infty & 170 \\ 220 & 0 & 0 & 170 & \infty \end{bmatrix}$$

(b) 邻接矩阵

在递推$A^{(k)}[i][j]$时，有更新的用粗体、斜体标明

图 4.24　重型运输：Floyd 算法执行过程

| | $A^{(-1)}$ | | | | | $A^{(0)}$ | | | | | $A^{(1)}$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |
| 0 | ∞ | 100 | 0 | 0 | 220 | ∞ | 100 | 0 | 0 | 220 | ∞ | 100 | ***80*** | 0 | 220 |
| 1 | 100 | ∞ | 80 | 0 | 0 | 100 | ∞ | 80 | 0 | ***100*** | 100 | ∞ | 80 | 0 | 100 |
| 2 | 0 | 80 | ∞ | 120 | 0 | 0 | 80 | ∞ | 120 | 0 | ***80*** | 80 | ∞ | 120 | ***80*** |
| 3 | 0 | 0 | 120 | ∞ | 170 | 0 | 0 | 120 | ∞ | 170 | 0 | 0 | 120 | ∞ | 170 |
| 4 | 220 | 0 | 0 | 170 | ∞ | 220 | ***100*** | 0 | 170 | ∞ | 220 | 100 | ***80*** | 170 | ∞ |
| | $A^{(2)}$ | | | | | $A^{(-1)}$ | | | | | $A^{(-1)}$ | | | | |
| | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |
| 0 | ∞ | 100 | 80 | ***80*** | 220 | ∞ | 100 | 80 | 80 | 220 | ∞ | 100 | ***120*** | ***170*** | 220 |
| 1 | 100 | ∞ | 80 | ***80*** | 100 | 100 | ∞ | 80 | 80 | 120 | 100 | ∞ | ***100*** | ***100*** | 100 |
| 2 | 80 | 80 | ∞ | 120 | 80 | 80 | 80 | ∞ | 120 | ***120*** | ***120*** | ***100*** | ∞ | 120 | 120 |
| 3 | ***80*** | ***80*** | 120 | ∞ | 170 | 80 | 80 | 120 | ∞ | 170 | ***170*** | ***100*** | 120 | ∞ | 170 |
| 4 | 220 | 100 | 80 | 170 | ∞ | 220 | 100 | ***120*** | 170 | ∞ | 220 | 100 | 120 | 170 | ∞ |

(c) Flogd算法执行过程中数组A的变化

图 4.24　重型运输：Floyd 算法执行过程(续)

代码如下：

```
#define MAXCITIES 256
#define INF 1000000000
#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))
int kase=0;                          //测试数据的序号
int n, r;                            //城市的个数和道路的个数
int w[MAXCITIES][MAXCITIES];         //floyd 算法中的 A 矩阵
char city[MAXCITIES][30];            //城市名
char start[30], dest[30];            //起点城市和终点城市
int numcities;                       //城市名在 city 数组中的序号
//把陆续读进来的城市名存储到 city 数组中，index 函数的功能是给定一个城市名，
//返回它在 city 数组中的下标，如果不存在，则把该城市名追加到 city 数组中
int index( char* s )
{
    int i;
    for( i=0; i<numcities; i++ )
    {
        if( !strcmp(city[i],s) )  return i;
    }
    strcpy( city[i], s );
    numcities++;
    return i;
}
int read_case( )    //读入测试数据
{
    int i, j, k, limit;
    scanf( "%d%d", &n, &r );
    if( n==0 )  return 0;
```

```
    for( i=0; i<n; i++ )    //初始化邻接矩阵
    {
        for( j=0; j<n; j++ )  w[i][j]=0;
    }
    for( i=0; i<n; i++ )  w[i][i]=INF;
    //读入道路网络
    numcities=0;
    for( k=0; k<r; k++ )
    {
        scanf( "%s%s%d", start, dest, &limit );
        i=index(start);
        j=index(dest);
        w[i][j]=w[j][i]=limit;  //Floyd 算法中矩阵 A 的初始值就是邻接矩阵
    }
    //读入起点城市和终点城市
    scanf( "%s%s", start, dest);
    return 1;
}
void solve_case( )
{
    int i,j,k;
    //Floyd-Warshall 算法
    for( k=0; k<n; k++ )
    {
        for( i=0; i<n; i++ )
        {
            for( j=0; j<n; j++ )
            {
                w[i][j]=MAX( w[i][j], MIN( w[i][k], w[k][j] ) );
            }
        }
    }
    i=index( start );
    j=index( dest );
    printf( "Scenario #%d\n", ++kase );
    printf( "%d tons\n\n", w[i][j] );
}
int main( )
{
    while ( read_case( ) )
        solve_case( );
    return 0;
}
```

## 练　习

4.10　青蛙(Frogger), ZOJ1942, POJ2253

**题目描述：**

青蛙 Freddy 坐在湖中的一块石头上。突然，他看到青蛙 Fiona 在另外一块石头上。他

想过去问候它。但是由于湖水太脏了，他不想游过去，而是想跳过去。

不幸的是，Fiona 所在的石头离 Freddy 太远了，超出了他能跳跃的范围。因此，Freddy 考虑将其他石头作为跳板，通过连续跳几次，到达 Fiona 所在的石头。

为了完成给定的连续跳跃序列，青蛙能跳跃的最大距离，很显然必须至少跟跳跃序列中最长的一次跳跃一样长。

两块石头之间的青蛙距离(也称为最小最大距离)，被定义成两块石头之间所有路径中的最大跳跃距离的最小值。

给定湖中 Freddy 所在的石头、Fiona 所在的石头，以及其他石头的坐标，试计算 Freddy 和 Fiona 之间的青蛙距离。

**输入描述：**

输入文件包含多个测试数据。测试数据的第 1 行为一个整数 $n$，$2 \leqslant n \leqslant 200$，表示石头的数目。接下来有 $n$ 行，每行为两个整数：$x_i$ 和 $y_i$，$0 \leqslant x_i$、$y_i \leqslant 1\,000$，代表第 $i$ 块石头的坐标。第 1 块石头为 Freddy 所在的石头，第 2 块石头为 Fiona 所在的石头。其他 $n-2$ 块石头没被占用。每个测试数据之后有一个空行。输入文件最后一行为 $n=0$，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，输出一行为："Scenario #x"，另一行为："Frog Distance=y"，其中 $x$ 为测试数据序号，测试数据序号从 1 开始计起，以为求得的青蛙距离，为一个实数，精确到小数点后 3 位有效数字。在每个测试数据(包括最后一个测试数据)的输出之后输出一个空行。

**样例输入：**

```
3
17 4
19 4
18 5

0
```

**样例输出：**

```
Scenario #1
Frog Distance=1.414
```

提示：思路与例 4.12 有点类似。

### 4.11　旅行费用(Travelling Fee), ZOJ2027

**题目描述：**

暑假快到了，Samball 想去旅行，现在可以制订一个计划了。选定旅行目的地后，接下来就是选择旅行路线了。由于他并没有多少钱，所以他想找一条最省钱的路线。Samball 得知旅游公司在暑假会推出一个折扣方案：选定一条线路后，在这条线路上连接两个城市间机票费用最贵的费用将被免去，这可是个好消息。给定出发地和目的地，以及所有机票的费用，请计算最小的费用。假定 Samball 选定的航线没有回路，并且出发地总是可以到达目的地的。

**输入描述：**

输入文件中包含多个测试数据。每个测试数据的第 1 行为出发地城市和目的地城市的名字。接下来一行为一个整数 $m$，$m \leqslant 100$，表示航线的数目。接下来有 $m$ 行，每行描述了一条航线，格式为：航线的起点城市，终点城市，费用。城市名称是由大写字母组成的字符串，且不超过 10 个字符。航线费用为[0, 1 000]范围内的整数。

测试数据一直到文件尾。

**输出描述：**

对输入文件中的每个测试数据，输出求得的最小费用。

**样例输入：**

```
HANGZHOU BEIJING
2
HANGZHOU SHANGHAI 100
SHANGHAI BEIJING 200
```

**样例输出：**

```
100
```

### 4.12 离芝加哥 106 英里(106 miles to Chicago), ZOJ2797, POJ2472

**题目描述：**

在电影“布鲁斯兄弟”中，如果 Elwood 和 Jack 不支付 5 000 美元的税金给芝加哥 Cook 县的税收办公室，那么抚养他们长大的孤儿院将会被卖给一个教育委员会。在挣够了 5 000 美元之后，他们必须找到一条通往芝加哥的路。然而，这看起来很简单，但其实很难，因为他们正被警察通缉。而且，离芝加哥还有 106 英里，天已经黑了，而他们还戴着墨镜。

由于他们在完成一件神圣的使命，请帮助他们找到一条通往芝加哥的最安全的路。在本题中，最安全的路被定义成一条 Elwood 和 Jack 最不可能被警察抓到的路。

**输入描述：**

输入文件包含多个测试数据。每个测试数据第 1 行为两个整数 $n$ 和 $m$，$2 \leqslant n \leqslant 100$，$1 \leqslant m \leqslant n \times (n-1)/2$，其中 $n$ 为交叉路口的数目，$m$ 为街道的数目。接下来有 $m$ 行，每行描述了一条街道，每条街道占一行，用 3 个整数描述：$a$、$b$ 和 $p$，$1 \leqslant a$、$b \leqslant n$，$a \neq b$，$1 \leqslant p \leqslant 100$，其中 $a$ 和 $b$ 为这条街道的两个交叉路口，$p$ 为布鲁斯兄弟通过这条街道时不被抓到的可能性(百分比)。每条街道都是双向的。假定每两个交叉路口之间最多有一条街道。

输入文件的最后一行为一个 0，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，计算从交叉路口 1 到交叉路口 $n$ 最安全路径的(布鲁斯兄弟不被抓到的)概率。可以假定从交叉路口 1 到交叉路口 $n$ 至少有一条路径。

输出概率时，以百分比的形式输出，且精确到小数点后 6 位有效数字。输出的百分比值只要与裁判的输出相差不超过 $10^{-6}$，都被认为是正确的。每个测试数据的输出占一行，格式如样例输出所示。

**样例输入：**

```
5 7
5 2 100
3 5 80
2 3 70
2 1 50
3 4 90
4 1 85
3 1 70
0
```

**样例输出：**

```
61.200000 percent
```

**注解：**

样例数据中，最安全的路为 1→4→3→5。

4.13　股票经纪人之间的谣言(Stockbroker Grapevine), ZOJ1082, POJ1125

**题目描述：**

股票经纪人因对传闻敏感而为世人所知。你被雇佣来开发一种在股票经纪人之间散布虚假信息，从而使你的雇主在股票市场上处于战术有利位置的方法。为了达到最好的效果，必须将虚假传闻以最快的速度散布。

不幸的是，股票经纪人只相信来自他们“信赖源”的信息。这意味着当你散布一个虚假传闻时必须考虑他们之间的联系圈子。一个股票经纪人将虚假传闻告诉给他的圈子里的每个经纪人需要花费一定的时间。试编写程序：选定一个经纪人，首先将传闻散布给他(首先把传闻传给他可以达到最小时间)，计算经纪人圈中所有人都收到传闻这个过程所需的时间。

**输入描述：**

输入文件包含多个测试数据，每个测试数据描述了一个经纪人圈子。每个测试数据第 1 行为一个整数 $n$，$1 \leqslant n \leqslant 100$，表示经纪人的数目，这些经纪人的编号从 1～$n$。接下来有 $n$ 行，每行描述了一个经纪人：首先是一个整数 $m$，$0 \leqslant m \leqslant n-1$，表示该经纪人与其他 $m$ 个经纪人有联系；然后是 $m$ 对整数 $a$ 和 $t$，$1 \leqslant a \leqslant n$，$1 \leqslant t \leqslant 10$，表示该经纪人可以把传闻传给 $a$ 经纪人，所花费的时间为 $t$ 分钟。

输入文件最后一行 $n$=0，代表输入结束。

**输出描述：**

对输入文件中的每个测试数据，输出占一行：首先是选定的第 1 个经纪人编号(首选选定这个经纪人进行传闻散布，可以达到最少时间)，以及所有经纪人都收到传闻所需时间，单位为分钟(整数)。

注意：输入文件中可能包含这样的测试数据：某些经纪人收不到传闻，即经纪人网络是不连通的。对于这样的测试数据，程序只需要输出"disjoint"即可。另外，如果经纪人 $A$ 可以将传闻传给 $B$，$B$ 也可以传给 $A$ 的话，两者所需时间不一定相等。

**样例输入：**

```
3
2 2 4 3 5
2 1 2 3 6
2 1 2 2 2
```

**样例输出：**

```
3 2
```

4.14　Risk 游戏(Risk), ZOJ1221, POJ1603

**题目描述：**

Risk 是一个棋盘游戏，几个对家同时征服世界。棋盘为一个世界地图，整个世界被分割成几个假想的国家。轮到一个玩家走棋时，他的军队开始征服其他国家，但是驻扎在一个国家的军队只能攻击与这个国家接壤的国家，征服了这些国家后，他的军队就可以驻扎进去了。

在玩家征服世界过程中，经常需要将他的军队从一个起始国家 $A$ 调到到一个目标国家

*B*。通常，这个过程要求花费时间最小，这样玩家需要选择一些中间国家，以使得从 *A* 到 *B* 的路线上需要征服的国家数目最少。假设整个世界由 20 个国家组成，每个国家与其他一些(1～19)国家接壤。试编写程序，计算从起始国家 *A* 到目标国家 *B* 至少需要征服多少个国家。不需要输出这些国家的序列，只需要输出国家的数目(包含目标国家)。例如，如果起始国家 *A* 和目标国家 *B* 接壤，则程序要输出 1。

图 4.25 给出了样例输入中的测试数据所描述的世界格局。

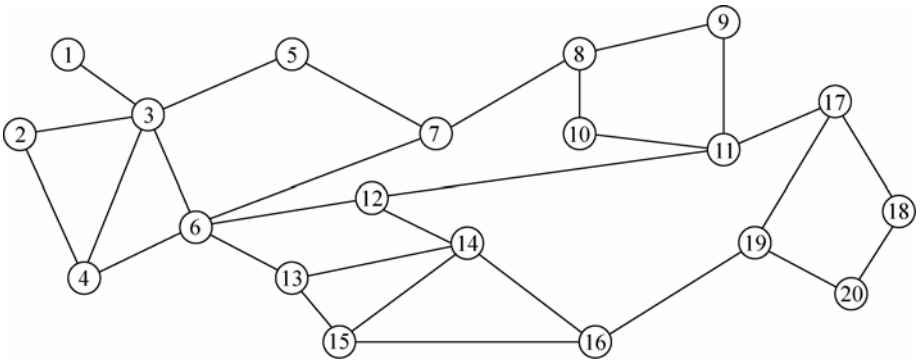



图 4.25　Risk 游戏：样例输入数据所描述的世界格局

**输入描述：**

输入文件中包含多个测试数据，每个测试数据描述了一个世界格局。每个测试数据的第 1～19 行描述了国家之间的接壤关系。为了避免重复列出国家之间的接壤关系，规定当国家 *I* 与国家 *J* 接壤时有 *I*<*J*。这样，第 *I* 行，*I*<20，首先是一个整数 *X*，代表与国家 *I* 接壤的国家中，序号大于 *I* 的有 *X* 个，然后是 *X* 个不同的整数 *J*，*I*<*J*≤20，每个整数代表与国家 *I* 接壤的国家。第 20 行为一个整数 *N*，1≤*N*≤100，表示接下来有 *N* 对国家。接下来有 *N* 行，每行只有两个整数 *A* 和 *B*，1≤*A*、*B*≤20，且 *A*!=*B*，代表征服路线上的起始国家和终止国家。输入数据保证从 *A* 到 *B* 至少有一条路线。

测试数据一直到文件尾。

**输出描述：**

对输入文件中的每个测试数据，首先输出一行"Test Set #T"，其中 *T* 为测试数据的序号，序号从 1 开始计起。接下来有 *N* 行，每行为测试数据中每对国家 *A* 和 *B* 的计算结果：需要征服的国家数目的最小值。每行的格式为：*A* to *B*: min，其中 min 为求得的最小值。每个测试数据的输出之后，输出一个空行。

**样例输入：**

```
1 3
2 3 4
3 4 5 6
1 6
1 7
2 12 13
1 8
2 9 10
1 11
```

**样例输出：**

```
Test Set #1
1 to 20: 7
2 to 9: 5
```

```
1 11
2 12 17
1 14
2 14 15
2 15 16
1 16
1 19
2 18 19
1 20
1 20
2
1 20
2 9
```

4.15　消防站(Fire Station), ZOJ1857, POJ2607

**题目描述：**

某个城市的消防任务由一些消防站承担。有些居民抱怨离他们家最近的消防站距离太远了，所以市政府决定再修建一个新的消防站。试选择消防站的位置，以减小离这些居民家最近的消防站的距离。

城市最多由 500 个交叉路口，这些路口由不同长度的道路段连接。对每个交叉路口，在此汇合的道路段不超过 20 个。居民房屋和消防站的位置假定都在路口，而且在每个路口最少有一栋房屋，每个路口也可以有多个消防站。

**输入描述：**

输入文件的第 1 行为两个正整数：$f$ 和 $i$，$f$ 表示已经存在的消防站数目，$f \leqslant 100$；$i$ 表示交叉路口的数目，$i \leqslant 500$，交叉路口用 $1 \sim i$ 的序号标明。接下来有 $f$ 行，每行给出了一个消防站的路口序号。接下来有若干行，每行为 3 个正整数，描述了连接两个路口的道路段，格式为：$A$ $B$ $L$，$A$ 和 $B$ 为该道路所连接的两个路口，$L$ 表示道路段的长度。所有的道路都是双向的，每对路口都是连通的。

测试数据之间用空行隔开。

**输出描述：**

对输入文件中的每个测试数据，输出一个整数 $n$，$n$ 的含义是新的消防站所在的交叉路口序号，选择 $n$ 可以使得所有交叉路口到最近的一个消防站的距离中最大值减小，且 $n$ 是满足条件的交叉路口序号中序号最小的。

**样例输入：**

```
1 6
2
1 2 10
2 3 10
3 4 10
4 5 10
5 6 10
6 1 10
```

**样例输出：**

```
5
```

# 4.5 差分约束系统

## 4.5.1 差分约束系统与最短路径

### 1. 差分约束系统

假设有这样一组不等式：

$$\begin{cases} X_1 - X_2 \leqslant 0 \\ X_1 - X_5 \leqslant -1 \\ X_2 - X_5 \leqslant 1 \\ X_3 - X_1 \leqslant 5 \\ X_4 - X_1 \leqslant 4 \\ X_4 - X_3 \leqslant -1 \\ X_5 - X_3 \leqslant -3 \\ X_5 - X_4 \leqslant -3 \end{cases}$$ 不等式组(1)

在不等式组(1)中，每个不等式都是两个未知数的差小于等于某个常数(大于等于也可以，因为左右乘以−1 就可以化成小于等于)。这样的不等式组就称作**差分约束系统**(System Of Difference Constraints)。

这个不等式组要么无解，要么就有无数组解。因为如果有一组解$\{X_1, X_2, \cdots, X_n\}$的话，那么对于任何一个常数 $k$，$\{X_1 + k, X_2 + k, \cdots, X_n + k\}$肯定也是一组解，因为任何两个数同时加一个数之后，它们的差是不变的，那么这个差分约束系统中的所有不等式都不会被破坏。

### 2. 差分约束系统与最短路径

差分约束系统的求解要利用单源最短路径问题中的**三角形不等式**(Triangle Inequality)。即对于有向网(或无向网)中的任何一条边<$u$, $v$>，都有：

$$d(v) \leqslant d(u) + Edge[u][v]$$

式中，$d(u)$和 $d(v)$是求得的从源点分别到顶点 $u$ 和顶点 $v$ 的最短路径的长度；$Edge[u][v]$是边<$u$, $v$>的权值。

这是很显然的：如图 4.26 所示，如果存在顶点 $u$ 到顶点 $v$ 的有向边(或无向边)，那么从源点到顶点 $v$ 的最短路径长度小于等于从源点到顶点 $u$ 的最短路径长度加上边<$u$, $v$>的权值。

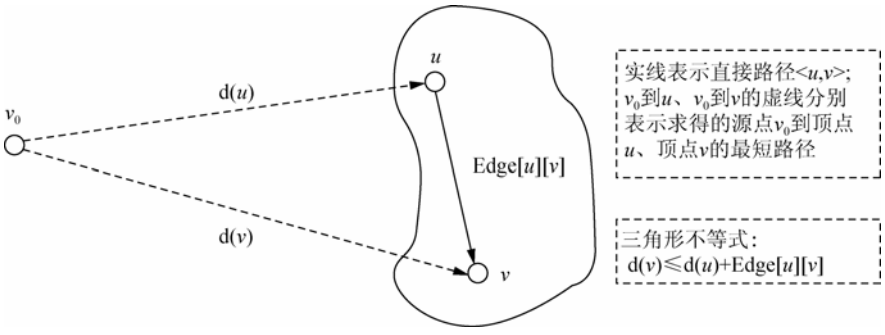



图 4.26 单源最短路径中的三角形不等式

显然以上不等式就是 d($v$)−d($u$)≤Edge[$u$][$v$]。这个形式正好和差分约束系统中的不等式形式相同。于是就可以把一个差分约束系统转化成一个有向网(或无向网)。

3. 有向网的构造

构造方法如下。

(1) 每个不等式中的每个未知数 $X_i$ 对应图中的一个顶点 $V_i$；

(2) 把所有不等式都化成图中的一条边。对于不等式 $X_i-X_j\leqslant c$，把它化成三角形不等式：$X_i\leqslant X_j+c$，就可以化成边$<V_j, V_i>$，权值为 $c$。

最后，在这张图上求一次单源最短路径，这些三角形不等式就都全部都满足了，因为它是最短路径问题的基本性质。

进一步：增加源点。所谓单源最短路径，当然要有一个源点，然后再求这个源点到其他所有顶点的最短路径。那么源点在哪呢？不妨自己造一个。以上面的不等式组为例，就再新加一个未知数 $X_0$。然后对原来的每个未知数都对 $X_0$ 随便加一个不等式(这个不等式当然也要和其他不等式形式相同，即两个未知数的差小于等于某个常数)。索性就全都写成 $X_n-X_0\leqslant 0$，于是这个差分约束系统中就多出了下列不等式：

$$\begin{cases} X_1 - X_0\leqslant 0 \\ X_2 - X_0\leqslant 0 \\ X_3 - X_0\leqslant 0 \\ X_4 - X_0\leqslant 0 \\ X_5 - X_0\leqslant 0 \end{cases}$$ 不等式组(2)

对于这 5 个不等式，也在图中建出相应的边。

构造好以后，得到的有向网如图 4.27 所示。图中的每一条边都代表差分约束系统中的一个不等式。现在以 $V_0$ 为源点，求单源最短路径。由于存在负权值边，所以必须用 Bellman-Ford 算法求解。最终得到的 $V_0$ 到 $V_n$ 的最短路径长度就是{$X_n$}的一个解。

在图 4.27 中，源点 $V_0$ 到其他各顶点的最短距离分别是{ −5, −3, 0, −1, −4 }，因此满足以上不等式的一组解是{ $x_1, x_2, x_3, x_4, x_5$ }={ −5, −3, 0, −1, −4 }。

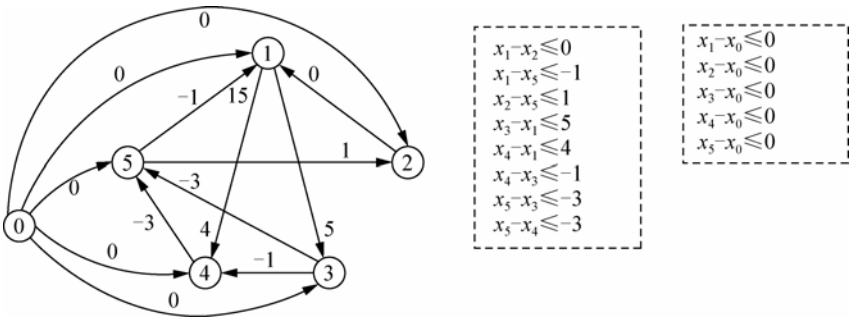



图 4.27　差分约束系统：有向网的构造

当然把每个数都加上 10 也是一组解：{ 5, 7, 10, 9, 6 }。但是这组解只满足不等式组(1)，也就是原先的差分约束系统；而不满足不等式组(2)，也就是后来加上去的那些不等式。当然这是无关紧要的，因为 $X_0$ 本来就是个新增的变量，是后来加上去的，满不满足与 $X_0$ 有关的不等式并不影响原不等式组的解。

关于源点 $V_0$ 的取值。其实，对于前面求得的一组解来说，它代表的这组解其实是{ 0, −5, −3, 0, −1, −4 }，也就是说 $X_0$ 的值也在这组解当中。但是 $X_0$ 的值是无可争议的，既然是以它作为源点求的最短路径，那么源点到它的最短路径长度当然是 0 了。因此，实际上解的这个差分约束系统无形中又存在一个条件：

$$X_0=0$$

4. 差分约束系统无解的情形

前面所描述的差分约束系统也有可能出现无解的情况，也就是从源点到某一个顶点不存在最短路径。在 4.2.3 节介绍到，如果有向网中存在负权值回路，则求出来的最短路径是没有意义的(从而不等式组也就无解)，因为可以重复走这个回路，使得最短路径无穷小。为什么有向网中存在负权值回路，则对应的差分约束系统就无解呢？举例分析，假设构造得到的有向网中存在如图 4.28(a)所示的回路，且回路权值总和为−3，小于 0，对应的不等式组如图 4.28(b)所示。将这个不等式组相加后得到 0≤−3，这是矛盾的，因此不等式组无解。

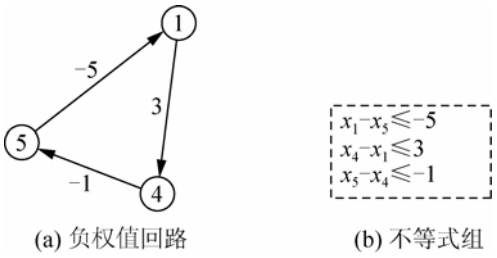



图 4.28　差分约束系统：负权值回路代表无解

判断差分约束系统是否有解的方法详见 4.2.3 节中的分析及例 4.13 中的代码。

### 4.5.2　例题解析

以下通过两道例题的分析，详细介绍差分约束系统的求解思想及其实现方法。

**例 4.13**　火烧连营(Burn the Linked Camp)

**题目来源：**

ZOJ Monthly, October 2006, ZOJ2770

**题目描述：**

大家都知道，三国时期，蜀国刘备被吴国大都督陆逊打败了。刘备失败的原因是刘备的错误决策。他把军队分成几十个大营，每个大营驻扎一队部队，又用树木编成栅栏，把大营连成一片，称为连营。

现在回到那个时代。陆逊派了很多密探，获得了他的敌人——刘备军队的信息。通过密探，他知道刘备的军队已经分成几十个大营，这些大营连成一片(一字排开)，这些大营从左到右用 1, 2, … , $n$ 编号。第 $i$ 个大营最多能容纳 $C_i$ 个士兵。而且通过观察刘备军队的动静，陆逊可以估计到从第 $i$ 个大营到第 $j$ 个大营至少有多少士兵。最后，陆逊必须估计出刘备最少有多少士兵，这样他才知道要派多少士兵去烧刘备的大营。

**输入描述：**

输入文件中有多个测试数据。每个测试数据的第 1 行有两个整数 $n(0<n\leqslant 1\,000)$和

$m(0 \leqslant m \leqslant 10\,000)$。第 2 行有 $n$ 个整数 $C_1, C_2, \cdots, C_n$。接下来有 $m$ 行，每行有 3 个整数 $i, j, k(0<i \leqslant j \leqslant n, 0 \leqslant k<2^{31})$，表示从第 $i$ 个大营到第 $j$ 个大营至少有 $k$ 个士兵。

**输出描述：**

对每个测试数据，输出一个整数，占一行，为陆逊估计出刘备军队至少有多少士兵。然而，陆逊的估计可能不是很精确，如果不能很精确地估计出来，输出"Bad Estimations"。

**样例输入：**

```
3 2
1000 2000 1000
1 2 1100
2 3 1300
3 1
100 200 300
2 3 600
```

**样例输出：**

```
1300
Bad Estimations
```

**分析：**

以样例输入中第 1 个测试数据为例解释差分约束系统的构造及求解。其数学模型为：设 3 个军营的人数分别为 $A_1$、$A_2$、$A_3$，容量为 $C_1$、$C_2$、$C_3$，前 $n$ 个军营的总人数为 $S_n$，则有以下不等式组。

(1) 根据第 $i$ 个大营到第 $j$ 个大营士兵总数至少有 $k$ 个，得不等式组(1)。

$$\begin{cases} S_2 - S_0 \geqslant 1\,100，\text{等价于：} S_0 - S_2 \leqslant -1\,100 \\ S_3 - S_1 \geqslant 1\,300，\text{等价于：} S_1 - S_3 \leqslant -1\,300 \end{cases}$$ 不等式组(1)

(2) 又根据实际情况，第 $i$ 个大营到第 $j$ 个大营的士兵总数不超过这些兵营容量之和，设 $d[i]$为前 $i$ 个大营容量总和，得不等式组(2)。

$$\begin{cases} S_2 - S_0 \leqslant d[2] - d[0] = 3\,000 \\ S_3 - S_1 \leqslant d[3] - d[1] = 3\,000 \end{cases}$$ 不等式组(2)

(3) 每个兵营实际人数不超过容量，得不等式组(3)：

$$\begin{cases} A_1 \leqslant 1\,000，\text{等价于：} S_1 - S_0 \leqslant 1\,000 \\ A_2 \leqslant 2\,000，\text{等价于：} S_2 - S_1 \leqslant 2\,000 \\ A_3 \leqslant 1\,000，\text{等价于：} S_3 - S_2 \leqslant 1\,000 \end{cases}$$ 不等式组(3)

(4) 另外由 $A_i>=0$，又得到不等式组(4)：

$$\begin{cases} S_0 - S_1 \leqslant 0 \\ S_1 - S_2 \leqslant 0 \\ S_2 - S_3 \leqslant 0 \end{cases}$$ 不等式组(4)

本题要求的是 $A_1 + A_2 + A_3$ 的最小值，即 $S_3 - S_0$ 的最小值。

**有向网的构造**：首先每个 $S_i$ 对应到有向网中的一个顶点 $V_i$；然后对上述 4 个不等式组中的每一个不等式：$S_i - S_j \leqslant c$，转化成从 $S_j$ 到 $S_i$ 的一条有向边，权值为 $c$。由不等式组(1)和(2)可知，存在$<S_j, S_{i-1}>$和$<S_{i-1}, S_j>$双向边，只是权值不一样；由不等式组(3)和(4)可知，存在$<S_i, S_{i+1}>$和$<S_{i+1}, S_i>$双向边，只是权值不一样。构造好的有向网如图 4.29 所示。

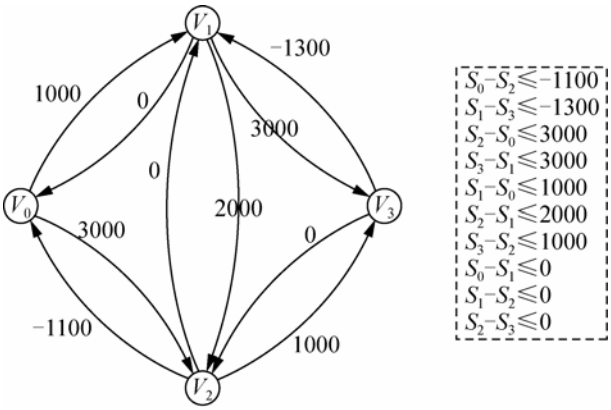



图 4.29　火烧连营：测试数据 1

构造好网络之后，最终要求什么？要求的是 $S_3 - S_0$ 的最小值，即要求不等式：

$$S_3 - S_0 \geqslant M$$

中的约束 $M$，并且 $M$ 取其最大值，转化成：

$$S_0 - S_3 \leqslant -M$$

即求 $S_3$ 到 $S_0$ 的最短路径(最小值)，长度为$-M$，求得$-M$ 为−1 300，即 $M$ 为 1 300($M$ 的最大值)。

对样例输入中的第 2 个测试数据，对应差分系统中的不等式及构造的有向网如图 4.30 所示。

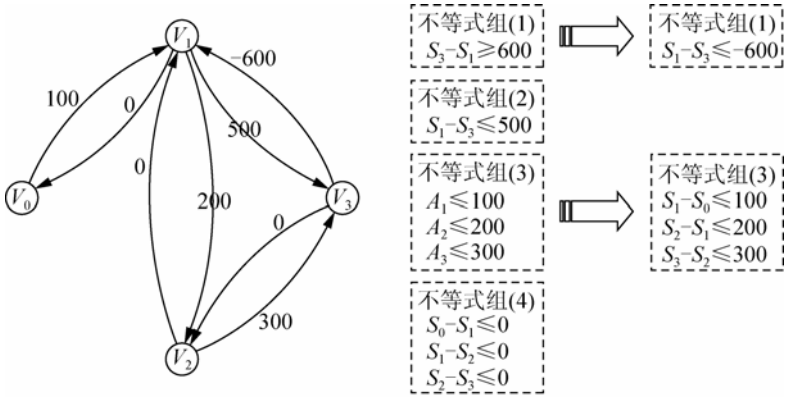



图 4.30　火烧连营：测试数据 2

为什么这个测试数据的输出为"Bad Estimations"？在 Bellman-Ford 算法中，执行完本身的二重循环之后，还应该检查每条边<$u$, $v$>，判断一下：加入这条边是否会使得顶点 $v$ 的最短路径值再缩短，即判断：dist[$u$] + $w$($u$, $v$) < dist[$v$]是否成立，如果成立，则说明存在从源点可达的负权值回路。这时应该输出"Bad Estimations"。

另外，在下面的代码中，dist 数组中除源点外，其余顶点的 dist[ ]值初始为∞，这样 Bellman-Ford 算法要多循环一次，详见 4.2.3 节中的讨论。

代码如下：

```
#define INF 9999999
#define NMAX 1001
```

```
#define EMAX 23000
int n;          //一共有 n 个大营
int m;          //已知从第 i 个大营到第 j 个大营至少有多少士兵，这些信息有 m 条
int c[NMAX];    //第 i 个大营最多有 c[i]个士兵
int dist[NMAX]; //从源点到各个顶点最短路径长度(注意：源点为 Sn 而不是 S0)
int d[NMAX];//d[0]=0,d[1]=c[1],d[2]=c[1]+c[2],...，即 d[i]为前 i 个大营容量总和
int ei;         //边的序号
struct eg
{
    int u, v, w;    //边：起点、终点、权值
}edges[EMAX];
void init( )    //初始化函数
{
    ei=0;
    int i;
    //除源点 Sn 外，其他顶点的最短距离初始为 INF，
    //则 bellman-ford 算法的第 1 重循环要执行 n-1 次
    for( i=0; i<=n; i++ )  dist[i]=INF;
    d[0]=0;
    dist[n]=0;//以下 bellman-ford 算法是以 Sn 为源点的，所以 dist[n]为 0
}
//如果存在源点可达的带负权值边的回路，则返回 false
bool bellman_ford( )    //bellman-ford 算法
{
    int i, k, t;
    //bellman-ford 算法的第 1 重循环要执行 N-1 次,N 是网络中顶点个数
    //在本题中,顶点个数是 n+1
    for( i=0; i<n; i++ )
    {
    //假设第 k 条边的起点是 u,终点是 v,以下循环考虑第 k 条边是否会使得源点 v0 到 v 的最短
    //距离缩短，即判断 dist[edges[k].u]+edges[k].w<dist[edges[k].v] 是否成立
        for(k=0; k<ei; k++)
        {
            t=dist[edges[k].u]+edges[k].w;
            if( dist[edges[k].u]!=INF && t<dist[edges[k].v] )
            {
                dist[edges[k].v]=t;
            }
        }
    }
    //以下是检查，若还有更新则说明存在无限循环的负值回路
    for(k=0; k<ei; k++)
    {
        if( dist[edges[k].u]!=INF && dist[edges[k].u]+edges[k].w<dist
            [edges[k].v] )
            return false;
    }
    return true;
}
int main( )
{
```

```
    while( scanf("%d %d", &n, &m)!=EOF )//输入数据一直到文件尾
    {
        init( );
        int i, u, v, w;
        for( i=1; i<=n; i++ )       //构造不等式组(3)和(4)
        {
            scanf("%d", &c[i]);     //读入第 i 个兵营最多有 ci 个士兵
            edges[ei].u=i-1;
            edges[ei].v=i;
            edges[ei].w=c[i];       //构造边<i-1,i>, 权值为 Ci
            ei++;
            edges[ei].u=i;
            edges[ei].v=i-1;
            edges[ei].w=0;          //构造边<i,i-1>, 权值为 0
            ei++;
            d[i]=c[i]+d[i-1];
        }
        for( i=0; i<m; i++ )        //构造不等式组(1)和(2)
        {
            scanf("%d %d %d", &u, &v, &w);
            edges[ei].u=v;
            edges[ei].v=u-1;
            edges[ei].w=-w;         //构造边<v,u-1>,权值为-w
            ei++;
            edges[ei].u=u-1;
            edges[ei].v=v;
            edges[ei].w=d[v]-d[u-1];  //构造边<u-1,v>,权值为 d[v]-d[u-1]
            ei++;
        }
        if( !bellman_ford( ) )  printf("Bad Estimations\n");
        else  printf("%d\n", dist[n]-dist[0]);
    }
    return 0;
}
```

**例 4.14**　区间(Intervals)

**题目来源：**

Southwestern Europe 2002, ZOJ1508, POJ1201

**题目描述：**

给定 $n$ 个整数闭区间$[a_i, b_i]$和 $n$ 个整数 $c_1, c_2, \cdots, c_n$。编程实现以下 3 点。

(1) 以标准输入方式读入闭区间的个数，每个区间的端点和整数 $c_1, c_2, \cdots, c_n$；

(2) 求一个最小的整数集合 $Z$，满足$|Z \cap [a_i, b_i]| \geqslant c_i$，即 $Z$ 里边的数中范围在闭区间$[a_i, b_i]$的个数不小于 $c_i$ 个，$i$=1, 2, …, $n$。

(3) 以标准输出方式输出答案。

**输入描述：**

输入文件包含多个测试数据，每个测试数据的第 1 行为一个整数 $n$(1≤$n$≤50 000)，表示区间的个数。接下来 $n$ 行描述了这 $n$ 个区间：第 $i$+1 行包含了 3 个整数 $a_i, b_i, c_i$，用空格

隔开，$0 \leqslant a_i \leqslant b_i \leqslant 50\,000$，$1 \leqslant c_i \leqslant b_i - a_i + 1$。

输入数据一直到文件尾。

**输出描述：**

对输入文件中的每个测试数据，输出一个整数，为最小的整数集合 $Z$ 的元素个数$|Z|$，整数集合 $Z$ 满足：$Z$ 里边的数中范围在闭区间$[a_i, b_i]$的个数不小于 $c_i$ 个，$i$=1, 2, …, $n$。

**样例输入：**

```
5
3 7 3
8 10 3
6 8 1
1 3 1
10 11 1
```

**样例输出：**

```
6
```

**分析：**

该题目可建模成一个差分约束系统。以样例输入中的测试数据为例进行分析。

设 $S[i]$是集合 $Z$ 中小于等于 $i$ 的元素个数，即 $S[i]=|\{s \mid s \in Z, s \leqslant i\}|$。则有以下不等式组。

(1) $Z$ 集合中范围在$[a_i, b_i]$的整数个数即 $S[b_i] - S[a_i-1]$至少为 $c_i$，得不等式组(1) (即约束条件(1) )。

$S[b_i] - S[a_{i-1}] \geqslant c_i$，转换成：$S[a_{i-1}] - S[b_i] \leqslant -c_i$。

$S_2 - S_7 \leqslant -3$

$S_7 - S_{10} \leqslant -3$

$S_5 - S_8 \leqslant -1$

$S_0 - S_3 \leqslant -1$

$S_9 - S_{11} \leqslant -1$

根据实际情况，还有两个约束条件。

(2) $S[i] - S[i-1] \leqslant 1$。

(3) $S[i] - S[i-1] \geqslant 0$，即 $S[i-1] - S[i] \leqslant 0$。

最终要求的是什么？设所有区间右端点的最大值为 $mx$，如该测试数据中 $mx$=11，所有区间左端点的最小值为 $mn$，如该测试数据中 $mn$=1，$mn - 1$=0，最终要求的是 $S[mx] - S[mn-1]$的最小值，即求 $S_{11}-S_0 \geqslant M$ 中的 $M$，转换成 $S_0-S_{11} \leqslant -M$，即要求源点 $S_{11}$ 到 $S_0$ 的最短路径长度，长度为$-M$。

假设最终求得的各顶点到源点 $S_{11}$ 的最短路径长度保存在数组 dist 中，那么$-M$=dist[0] − dist[11]，即 $M$=dist[11] − dist[0]，即为所求。

与例 4.13 直接根据约束条件构造网络图求解最短路径的方法不同的是，由于第(2)、(3)个约束条件中的不等式有 $2\times(mx-mn+1)$个，再加上约束条件(1)，构造的边数最多可达 3×50 000 条，所以将所有的约束条件转换成图中的边，不是个好方法。

更好的方法如下。

(1) 先仅仅用约束条件(1)构造网络图，各顶点到源点的最短距离初始为 0，这是因为 $S_i - S_{mx} \leqslant 0$，所以源点到各顶点的最短距离肯定是小于 0 的。注意本题中源点是 $S[mx]$。

(2) 即刻用 Bellman-Ford 算法求各顶点到源点的最短路径(注意 Bellman-Ford 算法的思

想)，在每次循环中，约束条件(1)判断完后再加上约束条件(2)和(3)的判断。

① 约束条件(2)的判断。

$S[i] \leqslant S[i-1] + 1$ 等效于 $S[i] - S[mx] \leqslant S[i-1] - S[mx] + 1$。

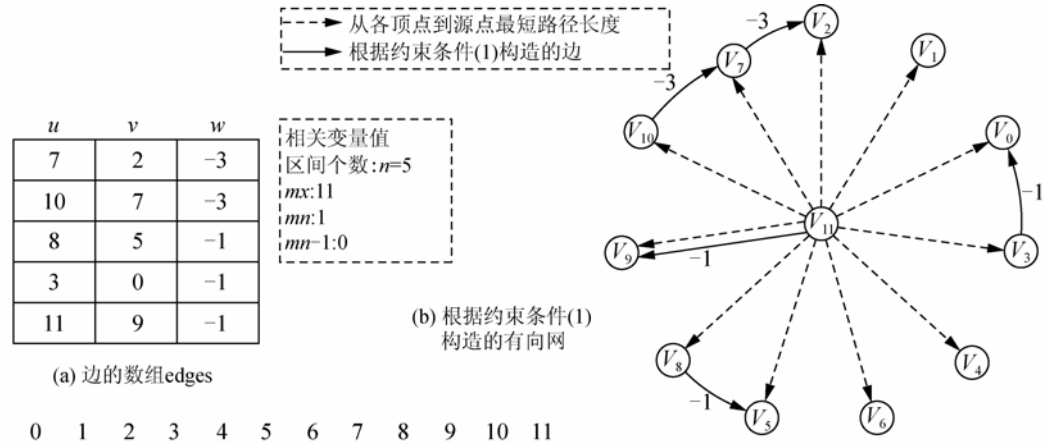
假设 dist[$i$]为源点 $mx$ 到顶点 $S_i$ 的最短路径，那么 $S[_i] - S[mx]$就是 dist[$i$]，$S[i-1] - S[mx] + 1$ 就是 dist[$i-1$] + 1，即如果顶点 $S_i$ 到源点的最短路径长度大于 $S_i-1$ 到源点的最短路径长度加 1，则修改 dist[$i$]为 dist[$i-1$] + 1。

② 约束条件(3)的判断。

$S[i-1] \leqslant S[i]$等效于 $S[i-1] - S[mx] \leqslant S[i] - S[mx]$。

$S[i] - S[mx]$就是 dist[$i$]，$S[i-1] - S[mx]$就是 dist[$i-1$]，即如果顶点 $S_{i-1}$ 到源点的最短路径长度大于 $S_i$ 到源点的最短路径，则修改 dist[$i-1$]为 dist[$i$]。

样例输入中测试数据对应的有向网的构造及差分约束系统的求解如图 4.31 所示。其中图 4.31(b)所示为仅根据约束条件(1)构造的有向网。




| u | v | w |
|---|---|---|
| 7 | 2 | −3 |
| 10 | 7 | −3 |
| 8 | 5 | −1 |
| 3 | 0 | −1 |
| 11 | 9 | −1 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 初始 |
| ***−1*** | 0 | ***−3*** | 0 | 0 | ***−1*** | 0 | ***−3*** | 0 | ***−1*** | 0 | 0 | Bellman循环：每条边都判断一下，是否会修改最短路径 |
| −1 | 0 | −3 | ***−2*** | ***−1*** | −1 | 0 | −3 | ***−2*** | −1 | 0 | 0 | 条件(2)判断，将a[$i$]修改为a[$i$-1]+1 (1) |
| ***−3*** | ***−3*** | −3 | ***−3*** | ***−3*** | ***−3*** | ***−3*** | −3 | −2 | −1 | 0 | 0 | 条件(3)判断，将a[$i$-1]修改为a[i] |
| ***−4*** | −3 | ***−6*** | −3 | −3 | −3 | −3 | −3 | −2 | −1 | 0 | 0 | Bellman循环 |
| −4 | −3 | −6 | ***−5*** | ***−4*** | −3 | −3 | −3 | −2 | −1 | 0 | 0 | 条件(2)判断 (2) |
| ***−6*** | ***−6*** | −6 | −5 | −4 | −3 | −3 | −3 | −2 | −1 | 0 | 0 | 条件(3)判断 |
| −6 | −6 | −6 | −5 | −4 | −3 | −3 | −3 | −2 | −1 | 0 | 0 | Bellman循环 |
| −6 | −6 | −6 | −5 | −4 | −3 | −3 | −3 | −2 | −1 | 0 | 0 | 条件(2)判断 (3) |
| −6 | −6 | −6 | −5 | −4 | −3 | −3 | −3 | −2 | −1 | 0 | 0 | 条件(3)判断 |

(c) dist数组的变化

图 4.31　区间：有向网的构造及差分约束系统的求解

在图 4.31(c)所示为 Bellman-Ford 算法执行过程当中 dist 数组各元素值的变化(在该图中，如果 dist 数组元素的值有变化，则用粗体、斜体标明)，其过程如下。

(1) dist 数组各元素的初始值均为 0。

(2) 第 1 次执行 Bellman-Ford 算法中的循环时，首先根据约束条件(1)修改 dist 数组。例如，因为存在边<7, 2>，权为−3，且 dist[7] + (−3) < dist[2]，所以要将 dist[2]修改成−3。然后根据约束条件(2)修改 dist 数组，例如，当 *i*=3 时，因为 dist[2] + 1 < dist[3]，所以要将 dist[3]修改成 dist[2] + 1=-2。最后根据约束条件(3)修改 dist[ ]数组，例如，当 *i*=2 时，因为 dist[1] > dist[2]，所以要将 dist[1]修改成 dist[2]=−3。

(3) 第 2 次执行 Bellman-Ford 算法中的循环的过程同上。

(4) 第 3 次执行完 Bellman-Ford 算法中的循环后，dist 数组每个元素的值都没有发生变化，所以 Bellman-Ford 算法中后续的循环没有必要执行下去了。

通过上述分析可知，Bellman-Ford 算法不一定要执行 *N*−2 次循环，*N* 为有向网(无向网)中的顶点个数，在图 4.31(b)中，*N*=12。其实只要在某次循环过程中，考虑每条边后，都没能改变当前源点到所有顶点的最短路径长度，那么 Bellman-Ford 算法就可以提前结束了。

代码如下：

```
#define inf 99999
#define EMAX 50002
struct e
{
    int u, v, w;    //边：起点、终点、权值
}edges[EMAX];
int n;              //区间的个数
int dist[EMAX];     //求得的从源点到各顶点的最短路径
int mn;             //所有区间左端点的最小值
int mx;             //所有区间右端点的最大值
void init( )        //初始化函数
{
    int i;
    for( i=0; i<EMAX; i++ ) //将源点到各顶点的最短路径长度初始为 0
        dist[i]=0;
    //这是因为 Si-Smx<=0，所以源点到各顶点的最短距离肯定是小于 0 的
    //Si:Z 中小于等于 i 的元素个数，即 S[i]=|{s|s∈Z,s<=i}|
    mx=1;  mn=inf;
}
bool bellman_ford( )
{
    int i, t; //循环变量和临时变量
    int f=1;//标志变量，为提前结束 Bellman-Ford 算法的标志变量
    //只要某次循环过程中,没能改变源点到各顶点的最短距离，则可以提前结束
    while( f )
    {
        f=0;
        //Bellman-Ford 算法本身的循环，考虑每条边是否能改变源点到各顶点的最短距离
        for( i=0; i<n; i++ )
        {
            t=dist[edges[i].u]+edges[i].w;
```

```
                if( dist[edges[i].v]>t )
                {
                    dist[edges[i].v]=t;  f=1;
                }
            }
            //根据约束条件 s[i]<=s[i-1]+1 进一步修改 s[i]值
            for( i=mn; i<=mx; i++ )
            {
                t=dist[i-1]+1;
                if( dist[i]>t )
                {
                    dist[i]=t;  f=1;
                }
            }
            //根据约束条件 s[i-1]<=s[i], 进一步修改 s[i-1]值
            for( i=mx; i>=mn; i-- )
            {
                t=dist[i];
                if( dist[i-1]>t )
                {
                    dist[i-1]=t;  f=1;
                }
            }
        }
        return true;
    }
    int main( )
    {
        while( scanf("%d", &n)!=EOF )
        {
            init( );
            int i;
            int u, v, w;    //区间的两个端点、ci
            for( i=0; i<n; i++ )
            {
                scanf( "%d %d %d", &u, &v, &w );
                //构造边<v,u-1,-w>
                edges[i].u=v, edges[i].v=u-1, edges[i].w=-w;
                if( mn>u )  mn=u;   //求得 mn 为所有区间左端点的最小值
                if( mx<v )  mx=v;   //求得 mx 为所有区间右端点的最大值
            }
            bellman_ford( );
            printf( "%d\n", dist[mx]-dist[mn-1] );
        }
        return 0;
    }
```

## 练　习

4.16　国王(King), ZOJ1260, POJ1364

**题目描述：**

从前有一个王国，皇后怀孕了。她祈祷到：如果我的孩子是儿子，我希望他是一个健

康的国王。9 个月后，她的孩子出生了，的确，她生了一个漂亮的儿子。

但不幸的是，正如皇室家庭经常发生的那样，皇后的儿子智力迟钝。经过多年的学习后，他只能做整数的加法，以及比较加法的结果比给定的一个整数是大还是小。另外，用来求和的数必须排列成一个序列，他只能对序列中连续的整数进行求和。

老国王对他的儿子非常不满意。但他决定为他的儿子准备一切，使得在他去世后，他的儿子还能统治王国。考虑到他儿子的能力，他规定国王需要决断的所有问题必须表示成有限的整数序列，并且国王需要决断的问题只是判断这个序列的和与给定的一个约束的大小关系。作了这样的规定，至少还有一些希望：他的儿子能做出一些决策。

老国王去世后，新国王开始统治王国。但很快，人们开始不满意他的决策，决定废除他。人们试图通过证明新国王的决策是错误的，从而名正言顺地废除新国王。

因此，试图篡位的人们给新国王出了一些题目，让国王做出决策。问题是从序列 $S=\{a_1, a_2, \cdots, a_n\}$ 中取出一个子序列 $S_i=\{aS_i, aS_{i+1}, \cdots, aS_{i+ni}\}$。国王有一分钟的思考时间，然后必须做出判断：他对每个子序列 $S_i$ 中的整数进行求和，即 $aS_i + aS_{i+1} + \cdots + aS_{i+ni}$，然后对每个子序列的和设定一个约束 $k_i$，即 $aS_i + aS_{i+1} + \cdots + aS_{i+ni} < k_i$，或 $aS_i + aS_{i+1} + \cdots + aS_{i+ni} > k_i$。

过了一会，他意识到他的判断是错误的。他不能取消他设定的约束，但他努力挽救自己：通过伪造篡位者给他的整数序列。他命令他的幕僚找出这样的一个序列 $S$，满足他设定的这些约束。请帮助幕僚，编写程序，判断这样的序列是否存在。

**输入描述：**

输入文件中包含多块输入。除最后一块输入外，每块输入对应一组问题及国王的决策。每块输入的第 1 行为两个整数：$n$ 和 $m$，其中 $0<n\leqslant 100$ 表示序列 $S$ 的长度，$0<m\leqslant 100$ 为子序列 $S_i$ 的个数。接下来有 $m$ 行为国王的决策，每个决策的格式为：$s_i$ $n_i$ $o_i$ $k_i$，其中 $o_i$ 代表关系运算符">"(用"gt"表示)或"<"(用"lt"表示)，$s_i$、$n_i$ 和 $k_i$ 的含义如题目描述中所述。最后一块输入只有一行，为 0，表示输入结束。

**输出描述：**

对输入文件中的每块输入，输出占一行字符串：当满足约束的序列 $S$ 不存在时，输出"successful conspiracy"；否则输出"lamentable kingdom"。对最后一块输入，没有输出内容。

**样例输入：**

```
4 2
1 2 gt 0
2 2 lt 2
1 2
1 0 gt 0
1 0 lt 0
0
```

**样例输出：**

```
lamentable kingdom
successful conspiracy
```

4.17　出纳员的雇佣(Cashier Employment), ZOJ1420, POJ1275

**题目描述：**

德黑兰的一家每天 24 小时营业的超市，需要一批出纳员来满足它的需求。超市经理雇佣你来帮他解决一个问题——超市在每天的不同时段需要不同数目的出纳员(例如，午夜只需一小批，而下午则需要很多)来为顾客提供优质服务，他希望雇佣最少数目的出纳员。

超市经历已经提供一天里每一小时需要出纳员的最少数量——$R(0)$, $R(1)$, …, $R(23)$。$R(0)$表示从午夜到凌晨 1∶00 所需出纳员的最少数目；$R(1)$表示凌晨 1:00 到 2:00 之间需要的；等等。每一天，这些数据都是相同的。有 $N$ 人申请这项工作，每个申请者 $i$ 在每天 24 小时当中，从一个特定的时刻开始连续工作恰好 8 小时。定义 $t_i(0 \leqslant t_i \leqslant 23)$为上面提到的开始时刻，也就是说，如果第 $i$ 个申请者被录用，他(或她)将从 $t_i$ 时刻开始连续工作 8 小时。

试编写一个程序，输入 $R(i)$，$i$=0 … 23，以及 $t_i$，$i$=1 … $N$，它们都是非负整数，计算为满足上述限制需要雇佣的最少出纳员数目。在每一时刻可以有比对应 $R(i)$更多的出纳员在工作。

**输入描述：**

输入文件的第 1 行为一个整数 $T$，表示输入文件中测试数据的数目(至多 20 个)。每个测试数据第 1 行为 24 个整数，表示 $R(0), R(1), \cdots, R(23)$，$R(i)$最大可以取到 1 000。接下来一行是一个整数 $N$，表示申请者的数目，$0 \leqslant N \leqslant 1\,000$。接下来有 $N$ 行，每行为一个整数 $t_i$，$0 \leqslant t_i \leqslant 23$，测试数据之间没有空行。

**输出描述：**

对输入文件中的每个测试数据，输出占一行，为需要雇佣的出纳员的最少数目。如果某个测试数据没有解，则输出"No Solution"。

**样例输入：**

```
1
1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
5
0
23
22
1
10
```

**样例输出：**

```
1
```

4.18　进度表问题(Schedule Problem), ZOJ1455

**题目描述：**

一个项目被分成几个部分，每部分必须在连续的天数完成。也就是说，如果某部分需要 3 天才能完成，则必须花费连续的 3 天来完成它。对项目的这些部分工作中，有 4 种类型的约束：FAS, FAF, SAF 和 SAS。两部分工作之间存在一个 FAS 约束的含义是：第一部分工作必须在第 2 部分工作开始之后完成；FAF 约束的含义是：第 1 部分工作必须在第 2 部分工作完成之后完成；SAF 的含义是：第 1 部分工作必须在第 2 部分工作完成之后开始；SAS 的含义是：第 1 部分工作必须在第 2 部分工作开始之后开始。假定参与项目的人数足够多，也就是说可以同时作任意多的部分工作。试编写程序，对给定的项目设计一个进度表，使得项目完成时间最短。

**输入描述：**

输入文件中包含多个测试数据，每个测试数据描述了一个项目。每个项目包含如下行：第 1 行为一个整数 $N$，表示该项目被分成 $N$ 部分，$N$=0 代表数据结束。接下来有 $N$ 行，第 $i$ 行为第 $i$ 个部分完成所需的时间。接下来有若干行，每行描述了两个部分之间的约束关系。

每个项目的最后一行为'#'，代表该项目的输入结束。

**输出描述：**

每个测试数据的输出占若干行：第 1 行输出项目的序号，接下来有 $N$ 行，每行为某部分的序号及它的开始时间，时间为非负整数，且被安排成最先完成的工作的开始时间为 0；如果该问题没有解，则在第 1 行后只输出一行，为字符串"impossible"。

在每个测试数据的输出之后，输出一个空行。

**样例输入：**

```
3
2
3
4
SAF 2 1
FAF 3 2
#
0
```

**样例输出：**

```
Case 1:
1 0
2 2
3 1
```

### 4.19 母牛的排列(Layout), POJ3169

**题目描述：**

像人类一样，母牛在排队等候食物时喜欢跟自己的朋友站在一起。FJ 有 $N(2 \leqslant N \leqslant 1\,000)$ 头母牛，编号为 1～$N$，排成一条直线，等候食物。这 $N$ 头母牛按照它们的编号顺序排列在一行。由于它们的坚持，两头或多头母牛可能排列到同一个位置(也就是说，如果认为每头母牛位于同一行的某个坐标位置，那么多个母牛的坐标位置可能相同)。

一些母牛互相喜欢，希望相互之间的距离在某个距离之内。而有些母牛相互排斥，希望相互之间的距离在某个距离之外。给定一个列表，列表中有 $ML$ 个约束，$1 \leqslant ML \leqslant 10\,000$，描述了相互喜欢的母牛，及他们能够分隔开的最大距离；随后是另一个列表，列表中有 $MD$ 个约束，$1 \leqslant MD \leqslant 10\,000$，描述了互相排斥的母牛，及他们必须分隔开的最小距离。

试(如果存在的话)计算，满足上述距离限制条件下，第 1 头母牛和第 $N$ 头母牛之间距离的最大值。

**输入描述：**

第 1 行：3 个用空格隔开的整数 $N$、$ML$ 和 $MD$。

第 2～$ML$+1 行：每行为 3 个用空格隔开的正整数 $A$、$B$ 和 $D$，$1 \leqslant A < B \leqslant N$，表示 $A$ 和 $B$ 之间能分隔开的最大距离为 $D$，$1 \leqslant D \leqslant 1\,000\,000$。

第 $ML$+2～$ML$+$MD$+1 行，每行为 3 个用空格隔开的正整数 $A$、$B$ 和 $D$，$1 \leqslant A < B \leqslant N$，表示 $A$ 和 $B$ 之间必须隔开至少 $D$ 距离，$1 \leqslant D \leqslant 1\,000\,000$。

**输出描述：**

输出一行，为一个整数：如果不存在满足条件的排列，输出-1；如果第 1 头母牛和第 $N$ 头母牛之间的距离可以任意，输出-2；否则输出第 1 头母牛和第 $N$ 头母牛之间的最大距离。

**样例输入：**

```
4 2 1
1 3 10
2 4 20
2 3 3
```

**样例输出：**

```
27
```