POJ1797

题目描述（POJ1797）：Hugo 需要将巨型起重机从工厂运输到他的客户所在的地方，经过的所有街道都必须能承受起重机的重量。他已经有了所有街道及其承重的城市规划。不幸的是，他不知道如何找到街道的最大承重能力，以将起重机可以有多重告诉他的客户。

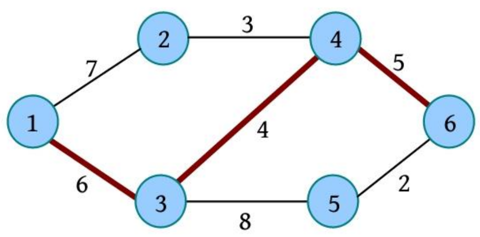
街道（具有重量限制）之间的交叉点编号为 1 ~ n。找到从 1 号（Hugo 的地方）到 n 号（客户的地方）可以运输的最大重量。假设至少有一条路径，所有街道都是双向的。

输入：第 1 行包含测试用例数量。每个测试用例的第 1 行都包含 n（1≤n≤1000）和 m，分别表示街道交叉口的数量和街道的数量。以下 m 行，每行都包含 3 个整数（正数且不大于 10^6），分别表示街道的开始、结束和承重。在每对交叉点之间最多有一条街道。

输出：对每个测试用例，输出都以包含“Scenario #i:”的行开头，其中 i 是从 1 开始的测试用例编号。然后单行输出可以运输给客户的最大承重。在测试用例之间有一个空行。

输入样例	输出样例
1 3 3 1 2 3 1 3 4 2 3 5	Scenario #1: 4

题解：本题要求找到一条通路，是最小边权最大的通路，该通路的最小边权即最大承重。如下图所示，从节节点 1 到节点 6 有 3 条通路，其中 1-2-4-6 的最小边权为 3；1-3-4-6 的最小边权为 4；1-3-5-6 的最小边权为 2；最小边权最大的通路为 1-3-4-6，该通路的最大承重为 4，超过 4 则无法承受。



1. 算法设计

（1）将所有街道都采用链式前向星存储，每个街道都是双向的。

（2）将 Dijkstra 算法的更新条件变形一下，改为最小值最大的更新。

```
if(dis[v]<min(dis[x],e[i].w))//求最小值最大的路径
    dis[v]=min(dis[x],e[i].w);
```

2. 算法实现

```
int dis[maxn];//dis[v]表示从源点出发到当前节点 v 所有路径上最小边权的最大值
void solve(int u){//Dijkstra 算法的变形，求最小值最大的路径
    priority_queue<pair<int,int> >q;
    memset(vis,0,sizeof(vis));
    memset(dis,0,sizeof(dis));
    dis[u]=inf;
    q.push(make_pair(dis[u],u));//最大值优先
    while(!q.empty()){
        int x=q.top().second;
        q.pop();
        if(vis[x])
            continue;
        vis[x]=1;
        if(vis[n])
            return;
        for(int i=head[x];~i;i=e[i].next){
            int v=e[i].to;
            if(vis[v])
                continue;
            if(dis[v]<min(dis[x],e[i].w)){//求最小值最大的路径
                dis[v]=min(dis[x],e[i].w);
                q.push(make_pair(dis[v],v));
            }
        }
    }
}
```

POJ1860

题目描述（POJ1860）：有几个货币兑换点，每个点只能兑换两种特定货币。可以有几个专门针对同一种货币的兑换点。每个兑换点都有自己的汇率，货币 A 到货币 B 的汇率是 1A 兑换 B 的数量。此外，每个交换点都有一些佣金，即必须为交换操作支付的金额。佣金始终以源货币收取。

例如，如果想在兑换点用 100 美元兑换俄罗斯卢布，而汇率为 29.75，佣金为 0.39，则将获得(100 - 0.39)×29.75=2963.3975RUR。

可以处理 N 种不同的货币。货币编号为 1～N。对每个交换点都用 6 个数字来描述：整数 A 和 B（交换的货币类型），以及 R_{AB}、C_{AB}、R_{BA}和 C_{BA}（分别表示交换 A 到 B 和 B 到 A 时的汇率和佣金）。

尼克有一些货币 S，并想知道他是否能在一些交易所操作之后增加他的资本。当然，他最终想要换回货币 S。在进进行操作时所有金额都必须是非负数。

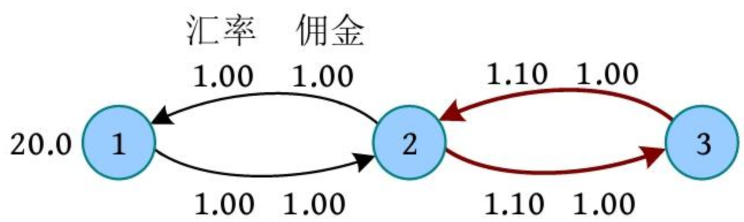
输入：输入的第 1 行包含 4 个数字：N 表示货币类型的数量，M 表示交换点的数量，S 表示尼克拥有的货币类型，V 表示他拥有的货币数量。以下 M 行，每行都包含 6 个数字，表示相应交换点的描述。数字由一个或多个空格分隔。1≤S≤N≤100，1≤M≤100，V 是实数，0≤V≤10^3。汇率和佣金在小数点后至多有两位，10^{-2} ≤汇率≤10^2，0≤佣金≤10^2。

输出：如果尼克可以增加他的财富，则输出“YES”，在其他情况下输出“NO”。

输入样例	输出样例
3 2 1 20.0 1 2 1.00 1.00 1.00 1.00 2 3 1.10 1.00 1.10 1.00	YES

题解：本题从当前货币出发，走一个回路，赚到一些钱。因为走过的边是双向的，因此能走过去就一定能够走回来。只需判断在图中是否有正环，即使这个正环不包含 S 也没关系，走一次正环就会多赚一些钱。

输入样例 1，如下图所示，包含一个正环 2 -3-2，每走一次就赚一些钱。



计算过程如下。

- 1-2：(20-1.00)×1.00=19.00。
- 2-3、3-2：(19-1.00)×1.10=19.80、(19.8-1.00)×1.10=20.68。
- 2-3、3-2：(20.68-1.00)×1.10=21.648、(21.648-1.00)×1.10=22.7128。
- 2-1：(22.7128-1.00)×1.00=21.7128。

1. 算法设计

（1）Bellman-Ford 算法，判断正环。用边松弛 n-1 次后，再执行一次，如果还可以松弛，则说明有环（是正环还是负环，主要取决于松弛条件）。注意：对双向边，边数是 2m 或使用边数计数器 cnt。

```
if(dis[e[j].b]<(dis[e[j].a]-e[j].c)*e[j].r)//松弛，a、b 为边的节点，r、c 为汇率和佣金
    dis[e[j].b]=(dis[e[j].a]-e[j].c)*e[j].r;
```

（2）SPFA 算法，判断正环。松弛时，若对一个节点访问 n 次，则存在环。

（3）DFS 深度优先搜索，判断正环。若在松弛时访问到已遍历的节点，则存在环。

2. 算法实现

```
bool bellman_ford(){//判正环
    memset(dis,0,sizeof(dis));
    dis[s]=v;
    for(int i=1;i<n;i++){//执行 n-1 次
        bool flag=0;
        for(int j=0;j<cnt;j++)//注意：边数是 2m 或 cnt
            if(dis[e[j].b]<(dis[e[j].a]-e[j].c)*e[j].r){
                dis[e[j].b]=(dis[e[j].a]-e[j].c)*e[j].r;
                flag=true;
            }
        if(!flag)
            return false;
    }
    for(int j=0;j<cnt;j++)//再执行 1 次，还能松弛，说明有环
        if(dis[e[j].b]<(dis[e[j].a]-e[j].c)*e[j].r)
            return true;
    return false;
}
```

POJ3259

题目描述（POJ3259）：在探索许多农场时，约翰发现了一些令人惊奇的虫洞。虫洞是非常奇特的，因为它是一条单向路径，可以将人穿越到虫洞之前的某个时间！约翰想从某个地点开始，穿过一些路径和虫洞，并在他出发前的一一段时间返回起点，也许他将能够见到自己。

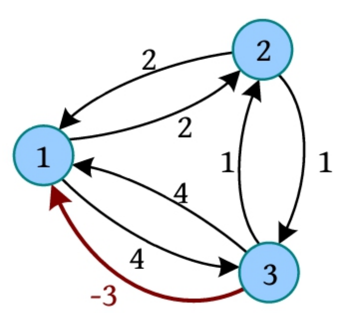
输入：第 1 行是单个整数 F（1≤F≤5），表示农场的数量。每个农场的第 1 行有 3 个整数 N、M、W，表示编号为 1~N 的 N（1≤N≤500）块田、M（1≤M≤2500）条路径和 W（1≤W≤200）个虫洞。第 2~M+1 行，每行都包含 3 个数字 S、E、T，表示穿过 S 与 E 之间的路径（双向）需要 T 秒。两块田都可能有多个路径。第 M+2~M+W+1 行，每行都包含 3 个数字 S、E、T，表示对从 S 到 E 的单向路径，旅行者将穿越 T 秒。没有路径需要超过 10000 秒的旅行时间，没有虫洞可以穿越超过 10000 秒。

输出：对于每个农场，如果约翰可以达到目标，则输出“YES”，否则输出“NO”。

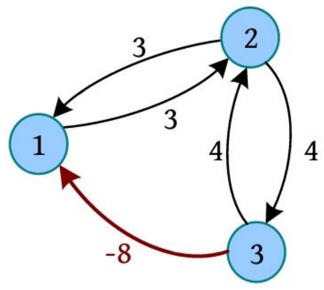
输入样例	输出样例
2 3 3 1 1 2 2 1 3 4 2 3 1 3 1 3 3 2 1 1 2 3 2 3 4 3 1 8	NO YES

提示：对于农场 1，约翰无法及时返回；对于农场 2，约翰可以在 1→2→3→1 的周期内及时返回，在他离开前 1 秒返回他的起始位置。他可以从周期内的任何地方开始实现这一目标。

题解：根据输入样例 1，如下图所示，约翰无法在他出发之前的时间返回。



根据输入样例 2，如下图所示。约翰可以在 1→2→3→1 的周期内及时返回，在他离开前 1 秒返回他的起始位置。他可以从周期内的任何地方开始实现这一目标。因为存在一个负环（边权之和为负）1→2→3→1，边权之和为-1。



1. 算法设计

本题其实就是判断是否有负环，使用 SPFA 判断负环即可。注意：普通道路是双向的，虫洞是单向的，而且时间为负值。

2. 算法实现

```
bool spfa(int u){
    queue<int>q;
    memset(vis,0,sizeof(vis));
    memset(sum,0,sizeof(sum));
    vis[u]=1;
    dis[u]=0;
    sum[u]++;
    q.push(u);
    while(!q.empty()){
        int x=q.front();
        q.pop();
        vis[x]=0;
        for(int i=head[x];~i;i=e[i].next){
            if(dis[e[i].to]>dis[x]+e[i].c){
                dis[e[i].to]=dis[x]+e[i].c;
                if(!vis[e[i].to]){
                    if(++sum[e[i].to]>=n)
                        return false;
                    vis[e[i].to]=1;
                    q.push(e[i].to);
                }
            }
        }
    }
    return true;
}

bool solve(){
    memset(dis,0x3f,sizeof(dis));
    for(int i=1;i<=n;i++)
        if(dis[i]==inf)//如果已经到达该点，没找到负环，则不需要再从该点找
            if(!spfa(i))
                return 1;
    return 0;
}
```

POJ3268

题目描述（POJ3268）：母牛从 N 个农场中的任一个去参加盛大的母牛聚会，聚会地点在 X 号农场。共有 M 条单行道分别连接两个农场，且通过路 i 需要花 Ti 时间。每头母牛都必须参加宴会，并且在宴会结束时回到自己的领地，但是每头母牛都会选择时间最少的方案。来时的路和去时的路可能不一样，因为路是单向的。求所有的母牛中参加聚会来回的最长时间。

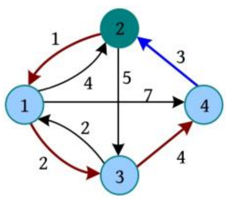
输入：第 1 行包含 3 个整数 N、M 和 X。在第 2～M+1 行中，第 i+1 描述道路 i，有 3 个整数：A_i、B_i和 T_i，表示从 A_i号农场到 B_i号农场需要 T_i时间。其中，$1 \le N \le 1000$，$1 \le X \le N$，$1 \le M \le 100\ 000$，$1 \le T_i \le 100$。

输出：单行输出母牛必须花费的时间最大值。

输入样例	输出样例
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	10

提示：母牛从 4 号农场进入聚会地点（2 号农场），再通过 1 号农场和 3 号农场返回，共计 10 个时间。

题解：根据输入样例，有 4 个农场、8 条路，聚会地点在 2 号农场。母牛从 4 号农场出发，走一个回路 4-2-1-3-4，共计 10 个时间，该时间是所有母牛中来回时间最长的，如下图所示。



1. 算法设计

因为母牛来回走的都是最短路径，所以先求每个节点从出发到聚会地点来回的最短路径之和，然后求最大值即可。

（1）从 i 号农场到聚会地点 X，相当于在反向图中从 X 到 i。

（2）从聚会地点 X 返回到 i 号农场，相当于在正向图中从 X 到 i。

（3）创建正向图和反向图，都把 X 作为源点，分别调用 SPFA 算法求正向图、反向图中源点到其他各个点的最短时间 dis[i]和 rdis[i]，求最大和值。

2. 算法实现

因为正向图、反向图均要调用 SPFA 算法，因此将图的存储结构 e[ ]、head[ ]及最短距离 dis[ ]作为参数，调用时传参即可。

```
void spfa(node *e,int *head,int u,int *dis){
    queue<int>q;
    memset(vis,0,sizeof(vis));
    memset(dis,0x3f,maxn*sizeof(int));//数组作参数，不能用 sizeof(dis)测量
    vis[u]=1;
    dis[u]=0;
    q.push(u);
    while(!q.empty()){
        int x=q.front();
        q.pop();
        vis[x]=0;
        for(int i=head[x];~i;i=e[i].next){
            if(dis[e[i].to]>dis[x]+e[i].w){
                dis[e[i].to]=dis[x]+e[i].w;
                if(!vis[e[i].to]){
                    vis[e[i].to]=1;
                    q.push(e[i].to);
                }
            }
        }
    }
}
```