## 一、问题分析

### 1.1 处理的对象（数据）

处理的对象是每一个结点和每一条边。

### 1.2 实现的功能

对于每个顶点 $v$，输出以下四个值之一：

- 0，表示从顶点 1 到 $v$ 没有路径
- 1，表示从顶点 1 到 $v$ 只有一条路径
- 2，表示从顶点 1 到 $v$ 有超过一条路径，且路径数是有限的
- -1，表示从顶点 1 到 $v$ 有无穷多条路径

具体实现为从1开始进行拓扑排序， 过程中进行1、2情况的拓展；再从每一结点进行检查，过程中进行0、-1情况的辨别

### 1.3 结果显示

输出包括 t 行，第 i 行是第 i 个测试用例的答案，为一个 n 个整数序列，取值在-1到2之间。

### 1.4 样例求解过程

**样例输入**

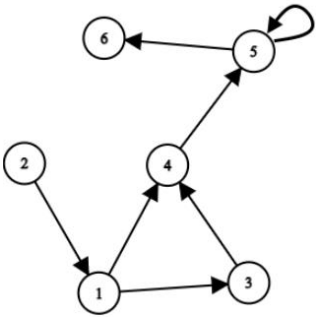
```
5

6 7
1 4
1 3
3 4
4 5
2 1
5 5
5 6

1 0

3 3
1 2
2 3
3 1

5 0

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

**样例输出**

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

**求解过程**

第一组数据，剩余略




**过程**

1) 从1开始拓扑排序。拓扑排序过程中依次为1、3、4入队，由于2、5、6前驱无法变为0，因此拓扑排序结束。

2) 检查每个点，主要由之前的拓扑排序是否到达进行区分。

5由于拓扑排序过程中减少过前驱但未减至0，说明能到达但有无数路径，5的后驱6也因5变为无数路径。

2由于拓扑过程中无法到达且检查过程中也没有因为前驱无数路径而无数路径，所以1到2没有路径。

**结果**

- 顶点 1 的结果为 1：从 1 到 1 只有一条路径（路径长度为 0）
- 顶点 2 的结果为 0：从 1 到 2 没有路径
- 顶点 3 的结果为 1：从 1 到 3 仅有一条路径（为边（1,3））
- 顶点 4 的结果为 2：从 1 到 4 有超过一条路径，但路径数量是有限的（两条路径：【（1,3），（3,4）】和【（1,4）】）
- 顶点 5 的结果为-1：从 1 到 5 的路径数是无穷的（环可以用于路径无穷多次）
- 顶点 6 的结果为-1：从 1 到 6 的路径数是无穷的（环可以用于路径无穷多次）

## 二、数据结构和算法设计

**题中的图是有向图，拓扑排序重要的是知道后驱结点编号和前驱结点个数，以及根据题目性质还有当前节点是否被访问过。**

**使用邻接表的数据结构可以方便得出后继结点编号，其他作为辅助信息储存。邻接表用STL中的set简单实现。**

**1. 抽象数据结构**

```
struct Node {
    set<int> behind; //后继节点
    int pre; //原前驱数量
    int mark; //减少过的前驱数量，以区分是否拓展过
    int ans; //结果
    Node():pre(0),mark(0),ans(0){
    }
};
class GraphTopo {
    private:
      int  flag;  //由于从1开始拓扑排序，假设没有从1到1的环。因此如果发现了这一环，标记过所有有限路径的结点都变为无限路径。
        int n,m;
        queue<int> tp; //拓扑排序的队列
        void sortExpand()；//排序第一步，拓扑排序的拓展
        void sortCheck(); //排序第二步，检查剩余结点的类型
    public:
        void initialTopo(); //初始化
        void sortTopo(); //排序
        void printTopo(); //输出
};
//ADT 只显示功能接口和重要数据
```

2.物理数据结构类型设计

```
struct Node {
    set<int> behind;
    int pre;
    int mark;
    int ans;
    Node():pre(0),mark(0),ans(0){
    }
}node[400000+5];
class GraphTopo {
    private:
```

```
int flag;
int n,m;
queue<int> tp;
void sortExpand(){
    node[1].pre=0;
    node[1].ans=1;
    tp.push(1);//从1开始拓扑遍历 ,注意1的前驱,假设数量为0
    while(!tp.empty()) {
        int top=tp.front();
        tp.pop();
        for(auto it:node[top].behind) {
            if(it==1) {
                flag=1;
                continue;
            }//特殊情况如果从1开始走走回到了1，形成了环，路上所有点改为-1
            if(node[top].ans==2)
                node[it].ans=2; //该点有有限条路径，则后继也有有限条路径（非最终结果）
            node[it].mark+=1;
            if(node[it].mark==node[it].pre) {
                tp.push(it);  //前驱个数为0时以此为拓展
                if(node[it].mark==1&&node[it].ans!=2)
                    node[it].ans=1;//该点前驱只有一个，且没有有限条路径的前驱（最终结果）
                else
                    node[it].ans=2;//有有限条路径（最终结果）
            }
        }
    }
}
void sortCheck(){
    //处理完成后仅剩无法到达和无限路径。无限路径由mark不为0的点开始拓展，剩余都是无法到达的点
    for(int i=2; i<=n; ++i) { //循环检查每个点
```

```
                    if(node[i].mark!=node[i].pre&&node[i].ans!=-1) { //排除有限路径的点和已结束标记的点
                        tp.push(i);
                        node[i].ans=-1;
                        while(!tp.empty()) {
                            int top=tp.front();
                            tp.pop();
                            for(auto it:node[top].behind) {
                                if(it==1) {
                                    flag=1;
                                    continue;
                                }
                                if(node[it].mark!=node[it].pre&&node[it].ans!=-1) { //排除有限路径的点和已结束标记的点
                                    node[it].ans=-1;
                                    tp.push(it); //以此往后都是无限路径
                                }
                            }
                        }
                    }
                }
            }
        public:
            GraphTopo(){
            }
            ~GraphTopo(){
            }
            void initialTopo() {
                flag=0;
                cin>>n>>m;
                for(int i=1; i<=n; ++i) {
                    node[i].behind.clear();
                    node[i].pre=0;
                    node[i].mark=0;
```

```
            node[i].ans=0;
        }
        for(int j=0; j<m; ++j) {
            int from,to;
            cin>>from>>to;
            node[from].behind.insert(to); //添加后继
            node[to].pre+=1; //前驱+1
        }
    }

    void sortTopo() {
        sortExpand();
        sortCheck();
    }

    void printTopo(){
        if(flag!=1){
            for(int i=1; i<=n; ++i) {
                cout<<node[i].ans<<' ';
            }
        }
        else{
            for(int i=1; i<=n; ++i) {
                if(node[i].ans!=0)
                    cout<<-1<<' ';
            }
        }
        cout<<endl;
    }
};
```

**3.算法思想的设计**

3.1 初始化：首先读取输入数据，包括节点数量n和边的数量m。然后创建节点数组node，每个节点包含一个集合behind表示其后继节点，以及三个整数pre、mark和ans，分别表示前驱节点数量、标记值和拓扑排序结果。

3.2 构建图：根据输入的边信息，将每条边的起始节点作为后继节点添加到对应节点的behind集合中，并将终点节点的前驱节点数量加一。

3.3 拓扑排序：通过以下两个步骤进行拓扑排序：

3.3.1 扩展阶段：从节点1开始，使用队列tp进行广度优先搜索。对于当前节点top，遍历其所有后继节点it，如果it为1，则说明存在环路，将flag标记为1；否则，更新it节点的标记值和前驱节点数量。如果it节点的前驱节点数量等于标记值，说明it节点没有前驱节点或所有前驱节点已经处理完毕，将其加入队列tp中，并根据情况更新其拓扑排序结果。

3.3.2 检查阶段：处理完成后，仅剩无法到达和无限路径的节点。从节点2开始，对于每个节点i，如果其前驱节点数量不等于标记值且拓扑排序结果不为-1，则将其标记为-1，并将其加入队列tp中。然后继续遍历tp中的节点，对于每个节点top，遍历其所有后继节点it，如果it的前驱节点数量不等于标记值且拓扑排序结果不为-1，则将其标记为-1，并将其加入队列tp中。重复此过程直到队列tp为空。

3.4 输出结果：根据flag的值判断是否存在环路。如果flag为0，则输出每个节点的拓扑排序结果；否则，flag等于1说明有从1到1的环路，经过的所有节点都是无穷路径，改原来的结果为-1.

3.5 循环处理：根据输入的测试用例数量t，重复执行上述步骤，对每个测试用例进行处理。

**4.关键功能的算法步骤**

从1开始拓扑排序，假设1的前驱为0个。

大情况1：从1开始走回到了1，形成了环，经过路上所有点改为-1，记录flag=1

大情况2：从1开始不走回1，flag=0，拓扑排序结果正常输出

拓扑排序过程中采用宽度优先搜索，保证从1开始有限路径的点和环的开始都可被标记到前驱数量减少。

在大情况2下，

遍历到的结点，有以下几种情况可以判断路径数量属于哪种类型：

1) 没有路径

拓展过程中不会前驱数量减少的点，可能为没有路径的情况

2) 唯一路径

前驱为唯一路径且本身原前驱数量为1，遍历到时减少为0

3) 有限路径

前驱为有限路径或者本身原前驱数量大于1，遍历到时减少为0

4) 无限路径

前驱数量有减少但是无法减少到0，以及其所有后驱

其中1、4情况需要再次检查所有未确定的结点，进行无限路径的拓展，类似拓扑排序的过程，采用宽度优先搜索，但不管前驱数量，全部进入队列进行遍历。

最后考虑大情况进行输出。

## 三、算法性能分析

**1.时间复杂度**

采用拓扑排序，为拓扑排序Kahn的时间复杂度，即O(V+E),V为节点个数，E为边数，意为理论上所有边所有点都会在输入、排序、检查中总共被处理常数次。

**2.空间复杂度**

存储V个结点和E条边，采用邻接表的数据结构，因此为O(V+E)

**3.总结**

时间复杂度O(V+E)

空间复杂度O(V+E)

**4.不足之处**

情况分类有点多，可能存在更简洁的判别方法。