Поиск в глубину

Олейникова Т., Матвеев И.

Задача «Предок»

Напишите программу, которая для двух вершин дерева определяет, является ли одна из них предком другой.

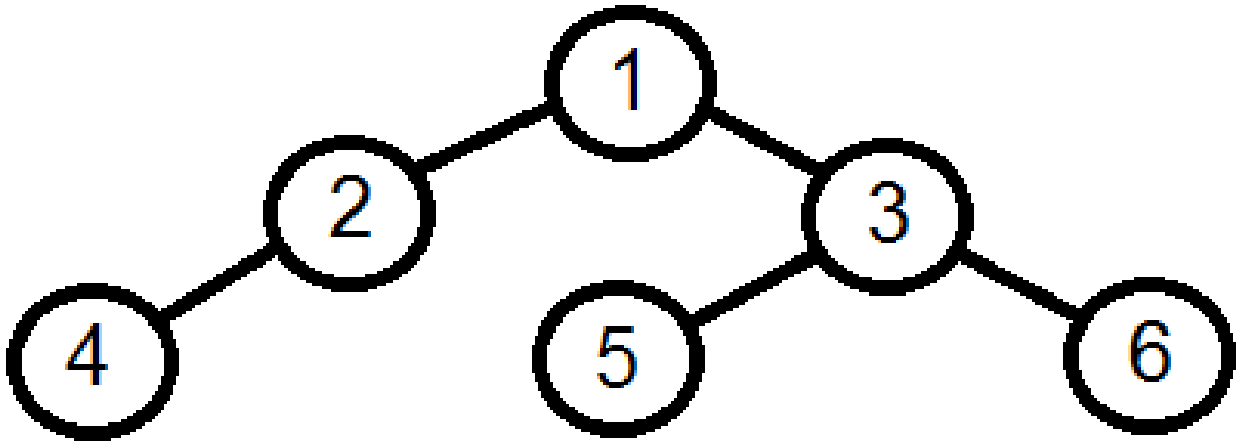
Формат входного файла

Первая строка входного файла содержит натуральное число n ($1 \leqslant n \leqslant 100\,000$) — количество вершин в дереве. Во второй строке находится n чисел, i-ое из которых определяет номер непосредственного родителя вершины с номером i. Если это число равно нулю, то вершина является корнем дерева.

В третьей строке находится число m ($1 \leqslant m \leqslant 100\,000$) — количество запросов. Каждая из следующих m строк содержит два различных числа a и b ($1 \leqslant a, b \leqslant n$).

Формат выходного файла

Для каждого из m запросов выведите на отдельной строке число 1, если вершина a является одним из предков вершины b, и 0 в противном случае.



Пример

ancestor.in	ancestor.out
6 0 1 1 2 3 3 5 4 1 1 4 3 6 2 6 6 5	0 1 1 0 0

```java
import java.util.*;

public class Ancestor
{
    FastScanner in;
    PrintWriter out;

    int count = 0;
    int head;

    class Node
    {
        LinkedList<Integer> Childs = new LinkedList<>();
        int in;
        int out;
    }
    Node node[];
```

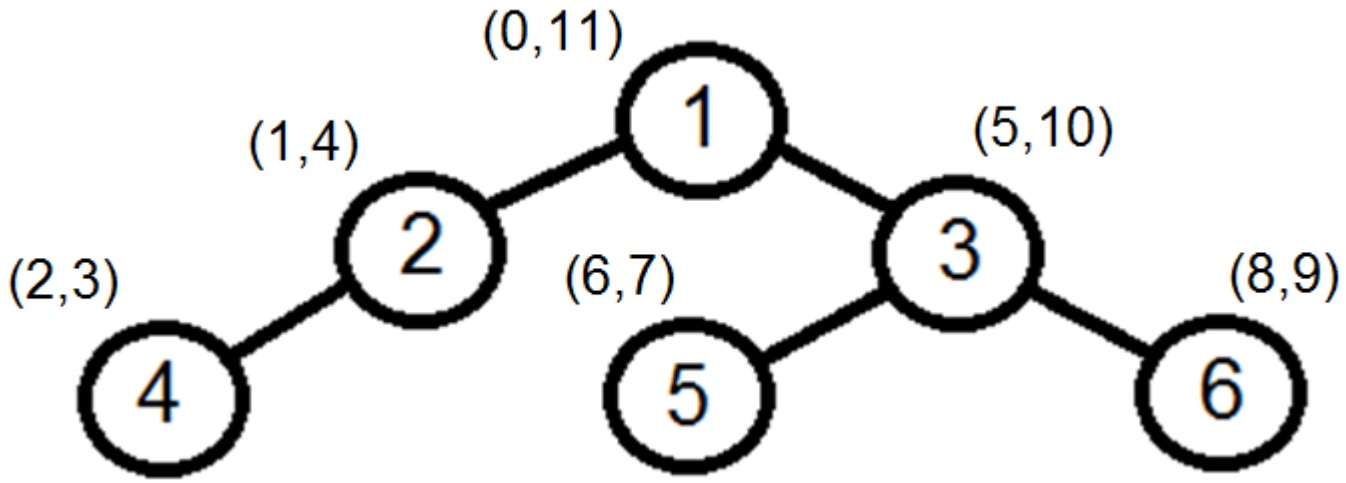
```
public void solve() throws IOException
{
    int N = in.nextInt();
    node = new Node[N];

    for (int NodeIndex = 0; NodeIndex < N; NodeIndex++)
        node[NodeIndex] = new Node();

    for (int NodeIndex = 0; NodeIndex < N; NodeIndex++)
    {
        int Parent = (int) in.nextLong();
        if (Parent == 0)
            head = NodeIndex;
        else
            node[Parent-1].Childs.add(NodeIndex);
    }

    DFSVisit(head);
```

```java
public void DFSVisit(int NodeIndex)
{
    this.node[NodeIndex].in = this.count++;
    for (Integer child : this.node[NodeIndex].Childs)
       DFSVisit(child);
    this.node[NodeIndex].out = this.count++;
 }
```



```
    DFSVisit(head);

    int M = in.nextInt();
    for (int j=0; j<M; j++)
    {
        int ParentIndex = in.nextInt() - 1;
        int NodeIndex   = in.nextInt() - 1;

    if (NodeIndex!=head)
        {
            if (node[ParentIndex].in < node[NodeIndex].in &&
                    node[ParentIndex].out > node[NodeIndex].out)
                out.println(1);
            else
                out.println(0);
        }

        else
            out.println(0);
    }
}
```

Задача «Поиск цикла»

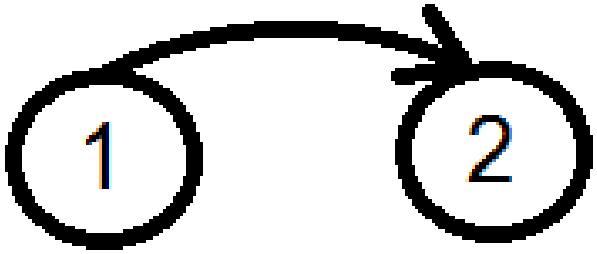
Дан ориентированный невзвешенный граф без кратных рёбер. Необходимо определить, есть ли в нём циклы, и если есть, то вывести любой из них.

Формат входного файла

В первой строке входного файла находятся два натуральных числа N и M ($1 \leqslant N \leqslant 100\,000$, $M \leqslant 100\,000$) — количества вершин и рёбер в графе соответственно. Далее в M строках перечислены рёбра графа. Каждое ребро задаётся парой чисел — номерами начальной и конечной вершин.

Формат выходного файла

Если в графе нет цикла, то вывести «`NO`», иначе — «`YES`» и затем перечислить все вершины в порядке обхода цикла.



Примеры

cycle2.in	cycle2.out
2 2 1 2 2 1	YES 1 2
2 1 1 2	NO

```java
import java.util.*;
import java.awt.Point;
import java.io.*;
import java.math.BigInteger;

public class Cycles {

    FastScanner in;
    PrintWriter out;

       int N;
       int M;

       ArrayList<Integer> Chain;
       LinkedList<Integer> WhiteNode  = new LinkedList<Integer>();

       class Node
       {

           ArrayList<Integer> LinkedNodes;

       }
       HashMap<Integer, Node> Nodes = new HashMap<Integer, Node>();
```

```java
public void solve() throws IOException
{
    int NodeStart, NodeEnd;
    N =  in.nextInt();
    M =  in.nextInt();
    for(int i= 0;i<M;i++)
    {
        NodeStart    = in.nextInt();
        NodeEnd      = in.nextInt();
        Node CurrentNode = Nodes.get(NodeStart);
        if (CurrentNode == null)
        {
            CurrentNode = new Node();
            CurrentNode.LinkedNodes = new ArrayList<Integer>();
            Nodes.put(NodeStart, CurrentNode);
            WhiteNode.add(NodeStart);
        }
        CurrentNode.LinkedNodes.add(NodeEnd);

        CurrentNode = Nodes.get(NodeEnd);
        if (CurrentNode == null)
        {
            CurrentNode = new Node();
            CurrentNode.LinkedNodes = new ArrayList<Integer>();
            Nodes.put(NodeEnd, CurrentNode);
            WhiteNode.add(NodeEnd);
        }
    }
    DFS();
}
```

```
public void DFS()
{
      boolean res = false;
      Chain = new ArrayList<Integer>(N);

      while(!WhiteNode.isEmpty())
      {
          int i = WhiteNode.getFirst();
          Chain.clear();
          Chain.add(i);
          res = DFSVisit(i);
          WhiteNode.remove(WhiteNode.indexOf(i));

          if (res == true)
             break;
      }

      if (res == false)
      {
          out.println("NO");
      }
}
```

```
public boolean DFSVisit(int u)
{
     boolean res = false;

     Node CurrentNode = Nodes.get(u);
     for(int ln: CurrentNode.LinkedNodes)
     {
         if (WhiteNode.contains(ln) == true)
         {
             if (this.Chain.contains(ln) == true)
             {
                 PrintCycle(ln);
                 res = true;
                 break;
             }
             else
             {
                     this.Chain.add(ln);

                     res = DFSVisit(ln);
                     int i = WhiteNode.indexOf(ln);
                     if(i != -1)
                     {
                         WhiteNode.remove(i);
                     }
                     if (res == true)
                         break;
                     else
                         this.Chain.remove(this.Chain.size()-1);

             }
         }
     }
     return res;
 }
```

```
public void PrintCycle(int v)
  {
     int node;
     String CycleNodes = "";
     boolean Start = false;
     for (int i = 0; i < this.Chain.size(); i++ )
     {
         node = this.Chain.get(i);
         if (node == v)
             Start = true;
         if (Start == true)
             CycleNodes = CycleNodes + node +" ";
     }
     out.println("YES");
     out.print(CycleNodes);
  }
```

Поиск мостов

Постановка задачи:

- Дан неориентированный граф, требуется найти все мосты в нем.

Формат входного файла

Первая строка входного файла содержит два натуральных числа n и m — количество вершин и ребер графа соответственно ($n \leqslant 20\,000$, $m \leqslant 200\,000$).

Следующие m строк содержат описание ребер по одному на строке. Ребро номер i описывается двумя натуральными числами b_i, e_i — номерами концов ребра ($1 \leqslant b_i, e_i \leqslant n$).

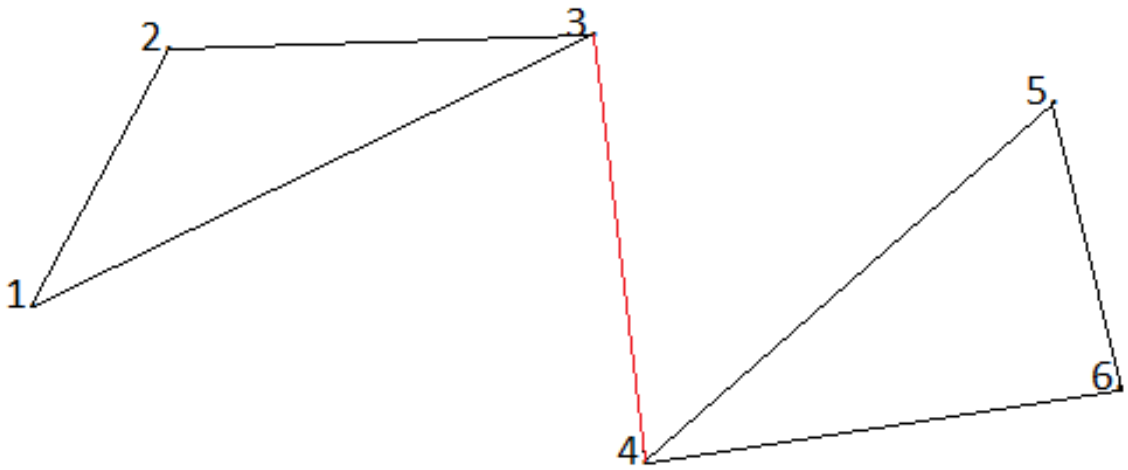
Формат выходного файла

Первая строка выходного файла должна содержать одно натуральное число b — количество мостов в заданном графе. На следующей строке выведите b целых чисел — номера ребер, которые являются мостами, в возрастающем порядке. Ребра нумеруются с единицы в том порядке, в котором они заданы во входном файле.

Пример:

Пример

bridges.in	bridges.out
6 7 1 2 2 3 3 4 1 3 4 5 4 6 5 6	1 3


1
2
3
4
5
6

```
void solve(){
        Node nodes[] = new Node[in.nextInt()+1]; // создаем массив для хранения объектов класса Node
        for (int i = 0; i < nodes.length; i++){ //
            nodes[i] = new Node(i); // заполняем массив для хранения вершин вершинами
        }
        Link links[] = new Link[2*in.nextInt()]; //создаем массив для хранения ребер
        int linkIdx = 1; //нумеруем ребро с 1
        for (int i = 0; i < links.length;){ //
            int f = in.nextInt(); // считывание вершин из которых состоит ребро
            int s = in.nextInt();
            links[i] = new Link(linkIdx,nodes[f],nodes[s]); //создание объекта класса ребро и задание ему номера
            nodes[f].addLink(links[i]); //связываю вершину с ребром
            i++;
            links[i] = new Link(linkIdx,nodes[s],nodes[f]); //
            nodes[s].addLink(links[i]); //
            linkIdx++;
            i++;
        }
        for (int i = 1; i < nodes.length; i++){ //начинаем обход по вершинам
            if (!nodes[i].used){ //запускаем поиск в глубину начиная с первой
                nodes[i].dfs(null); // запускаем для первой вершины поиск в глубину
            }
        }
```

```
class Link implements Comparable<Link>{ // реализуем интерфейс comparable для того чтоб иметь возможность сравнивать ребра друг с другом
        int idx; // номер ребра
        Node from; // номер вершины откуда началось ребро
        Node to; // номер вершины куда пришло ребро

        public int compareTo(Link second){ // метод в котором происходит сравнение
            return this.idx-second.idx; // происходит сравнение номеров ребер
        }

        Link(int idx, Node from, Node to){
            this.from = from;
            this.to = to;
            this.idx = idx;
        }

}
```

```
class Node{
    int idx; // номер вершины
    ArrayList<Link> links = new ArrayList<Link>(); // в links хранятся ребра частью которого является вершина
    boolean used; // флаг
    static int time = 0; // общее количество шагов которые мы пройдем
    static ArrayList<Link> bridges = new ArrayList<Link>(); // хранит все ребра которые являются мостами
    int tIn; // то количество шагов которые мы прошли до вершины
    int fUp; // то количство шагов которые мы прошли до сына этой вершины

    Node(int idx){ // номер вершины
        this.idx = idx;
    }

    void addLink(Link link){ // добавляет в линкс те ребра частью которых является данная вершина, установка связи для вершин с ребрами
        this.links.add(link); // links харнит те ребра частью которых является данная вершина
    }
```

- Collections.*sort(Node.bridges); // сортируем по возрастанию*
- out.println(Node.*bridges.size()); // выводим количество мостов*
- **for (int i = 0; i < Node.*bridges.size(); i++){***
- out.println((Node.*bridges.get(i).idx)); // выводим номера этих мостов*