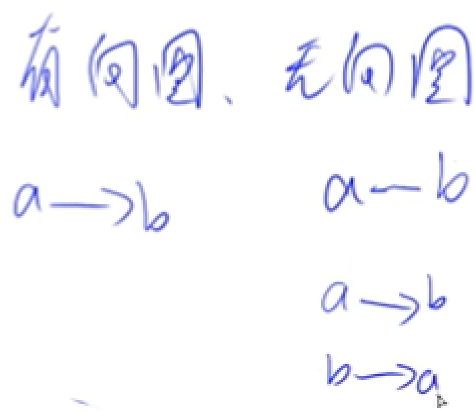
算法基础（十）：图（存储与遍历

其实树是一种特殊的图，**树是无环连通图**，树和图的存储可以统一起来

图分为两种，有向图和无向图，如下：



对于无向图，他是一种特殊的有向图，我们对于无向图的每一条边都建立两个方向边，所以我们下面只需要考虑有向图如何存储即可

图的存储

图的存储有两种方式

1. 邻接矩阵法

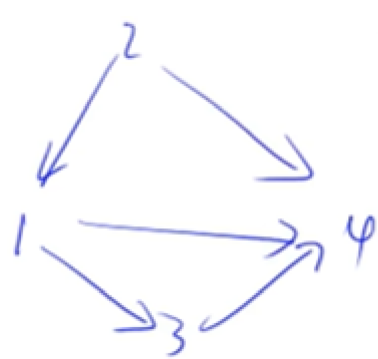
就是一个二维数组，`g[a][b]`存储`a -> b`的边的信息，如果有权重那就存放权重，没有权重就存放一个`bool`变量表示有这么一条边，邻接矩阵不能存储重边，用于存储稠密图

2. 邻接表法

邻接表就是一堆单链表，如果图中有`n`个点那就有`n`个单链表，每一个结点对应一个单链表

这个单链表存的就是从这个点可以走到的下一个点

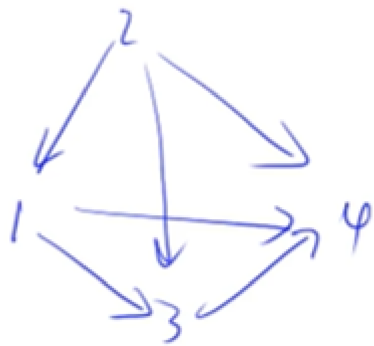
以下图为例：



有四个结点就存在四个单链表：

```
1: -> 3 -> 4 -> -1;
2: -> 1 -> 4 -> -1;
3: -> 4 -> -1;
4: -> -1;
//-1表示指向空集
```

如果插入一条边的话就直接在对应的链表头部插入一个结点，比如插入边`2 -> 3`



链表变化如下：

```
1: -> 3 -> 4 -> -1;
2: -> 3 -> 1 -> 4 -> -1;
3: -> 4 -> -1;
4: -> -1;
//-1表示指向空集
```

存储的代码实现：

```
int h[N], e[N], ne[N], idx;
//h[N]是每个单链表的头指针，指向每个单链表的第一个结点
//e[N]是链表的结点元素
//ne[N]是链表的结点的指针，其指向对应单链表的下一个结点
//e[N]和ne[N]通过下标联系起来，下标相同的e[]和ne[]属于同一个结点，下标就是结点的地址
//需要注意的是，e[]和ne[]被切分成了多个链表
//idx表示用到了哪个空闲结点，含义与链表中的idx的含义一样
```

初始化：

```
memset(h, -1, sizeof h);
//将所有邻接表的头指针指向-1，即空集
```

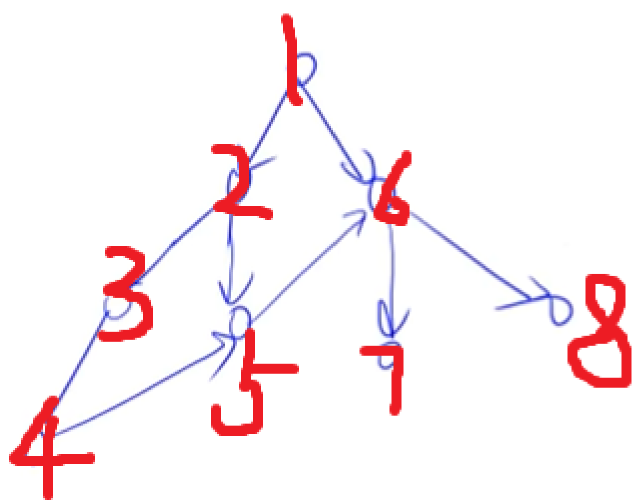
插入一条边：

```
void add(int a, int b)
{
    e[idx] = b;//b作为结点元素
    ne[idx] = h[a];//b对应的结点插入到a结点的单链表中，我们用对应的图结点元素来索引对应的头指针
    h[a] = idx;//插入到头部，头指针指向b结点对应的地址
    idx ++;//空闲指针指向下一个空闲结点的地址
}
```

图的遍历

深度优先遍历

跟前面的深度优先搜索一样，沿着某个分支一直走下去，走到底了之后回溯，然后搜索下一个分支



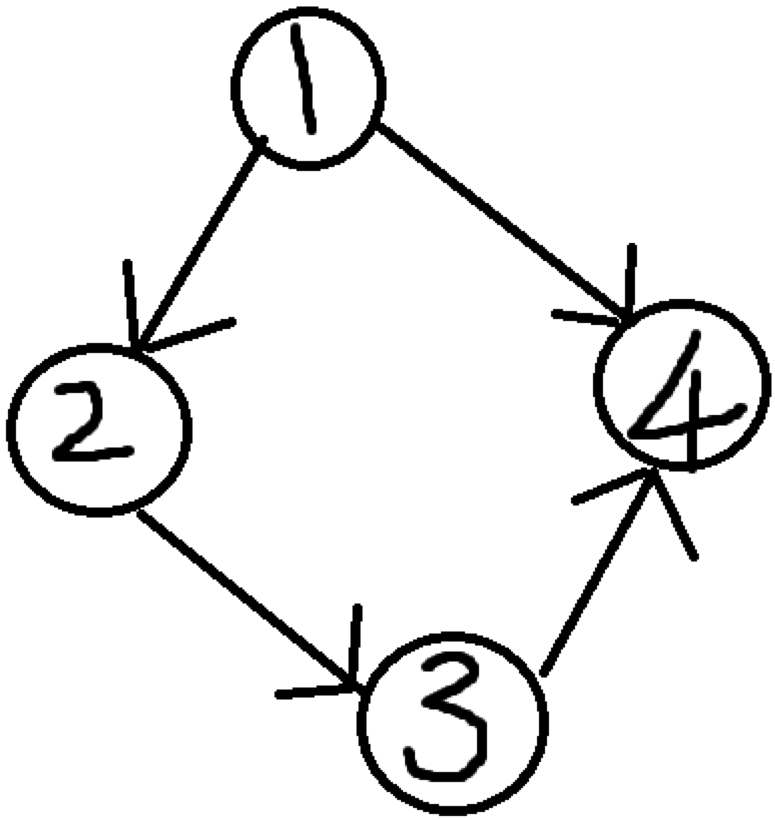
图深度优先遍历是一种特殊的深度优先搜索

代码实现：

```
//由于遍历只遍历一次，用bool数组来记录哪些点已经被遍历过
bool st[N];

void dfs(int u)
{
    st[u] = true;//标记一下这个点已经被搜过了

    //遍历这个结点的所有出边，类似于深度优先搜索中的遍历所有的分支
    for(i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];//记录链表里的结点对应图中的编号是多少
        if(!st[j]) dfs(j);//如果没有被搜过就一直搜下去

```

```
            //因为只是单纯的遍历，所以搜完后回溯不需要再回到什么初始状态
        }
}
```

以下图为例来分析其递归过程：



存储结构如下：

```
1: -> 2 -> 4 -> -1
2: -> 3 -> -1
3: -> 4 -> -1
4: -> -1
```

1. 首先是`dfs(1)`
2. 将`st[1]`设置为`true`表示第一个结点已经被遍历过了
3. 然后进入`dfs(1)`的`for`循环，来遍历`1`的每一个出边
    1. 结点`2`没有被遍历，进入`dfs(2)`
    2. 将`st[2]`设置为`true`表示第二个结点被遍历
    3. 然后进入`dfs(2)`的`for`循环，来遍历`2`的每一个出边
        1. 结点`3`没有被遍历，进入`dfs(3)`
        2. 将`st[3]`设置为`true`表示第三个结点被遍历
        3. 然后进入`dfs(3)`的`for`循环，来遍历`3`的每一个出边

            1. 结点`4`没有被遍历，进入`dfs(4)`
            2. 将`st[4]`设置为`true`表示第四个结点被遍历
            3. 然后进入`dfs(4)`的`for`循环，来遍历`4`的每一个出边
            4. 发现结点`4`没有出边，于是返回到`dfs(3)`的`for`循环
        4. 结点`3`的出边遍历完毕，返回到`dfs(2)`的`for`循环
    4. 结点`2`的出边遍历完毕，返回到`dfs(1)`的`for`循环
4. 结点`1`遍历下一个出边，因为结点`4`的`st[4]`为`true`表示结点四被遍历过了，不再进入递归
5. 结点`1`的出边遍历完全，返回到最开始调用`dfs(1)`的地方，即主函数

由于所有的点只被遍历一次指`st[u] = true`，并且对于每一个点`for`循环其所有的边，且每一个边也只遍历一次，所以深度优先遍历的时间复杂度是与点数和边数成线性关系的为`O(n + m)`

需要注意的是再全排列中深度优先搜索的时间复杂度的`O( n!)`这里的`n`指的是数字的个数，并不是结点的个数，`n`个数字形成的状态结点的个数是`n!`个

样例分析：

> 给定一颗树，树中包含 n 个结点（编号 $1 \sim n$）和 $n-1$ 条无向边。
>
> 请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。
>
> 重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。
>
> **输入格式**
>
> 第一行包含整数 n，表示树的结点数。
>
> 接下来 $n-1$ 行，每行包含两个整数 a 和 b，表示点 a 和点 b 之间存在一条边。
>
> **输出格式**
>
> 输出一个整数 m，表示将重心删除后，剩余各个连通块中点数的最大值。
>
> **数据范围**
>
> $1 \le n \le 10^5$
>
> **输入样例**
>
> ```
> 9
> 1 2
> 1 7
> 1 4
> 2 8
> 2 5
> 4 3
> 3 9
> 4 6
> ```
>
> **输出样例：**
>
> ```
> 4
> ```

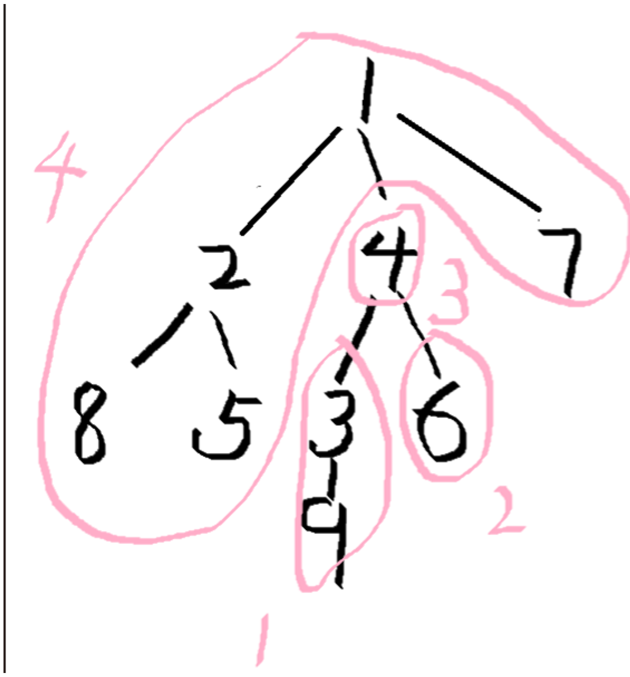
```
#include<iostream>
#include<cstring>
```

```
#include<algorithm>

using namespace std;
//N代表总的结点个数
//题目中的图的边的个数不会大于N，这里M = N * 2是为存储无向图，边要变成两倍
const int N = 100010, M = N * 2;

//用邻接链表的方式来存储图的点和边
int h[N], e[M], ne[M], idx;

//题目输入的结点的个数
int n;

//由于遍历只遍历一次，用bool数组来记录哪些点已经被遍历过
bool st[N];

//全局答案存的就是删除一个结点后形成的最大连通块
//然后删掉不同结点形成的最大连通块中最小结点个数
int ans = N;

//插入一条边
void add(int a, int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx ++;
}

int dfs(int u)
{
    st[u] = true;//标记一下这个点已经被搜过了

    int sum = 1;//sum存放以当前u为根结点的树的结点个数
    int res = 0;//res存放删掉这个结点后最大连通块的结点个数

    //遍历这个结点的所有出边，类似于深度优先搜索中的遍历所有的分支
    for(int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];//记录链表里的结点对应图中的编号是多少
        if(!st[j])//如果没有被搜过就一直搜下去
        {
            int s = dfs(j);//s存放对应分支这颗子树的结点个数
            res = max(res, s);//找出子树连通量的最大值
            //当前u为根结点的树的结点个数需要加上所有的子树的结点个数
            sum += s;
        }
        //因为只是单纯的遍历，所以搜完后回溯不需要再回到什么初始状态
    }
    //res存放删掉这个结点后最大连通块的结点个数
    //所以res当然还包括删掉以当前u为根结点的树的结点个数后
    //剩下连通块的结点个数
```

```
    res = max(res, n - sum);

    ans = min(res, ans);//找出删掉不同结点形成的最大连通块中最小的那个作为答案

    return sum;
}

int main()
{
    //对所有的邻接表进行初始化
    memset(h, -1, sizeof h);
    cin >> n;//树的结点个数

    // 题目接下来会输入，n-1行数据
    for(int i = 0; i < n - 1; i ++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        add(b, a);//无向图
    }
    dfs(1);//从第一个点开始深度优先搜索

    cout << ans << endl;
    return 0;
}

```

首先分析题意，我们需要找到重心，重心是指，将某个结点删除后，会留下一堆的连通分量，这写连通分量中必然有一个最大的连通分量，将不同的结点删除后就会有不同的最大的连通分量，而将重心删除后，剩下的最大的这个连通分量在所有的最大的连通分量中最小，题目就是要我们输出这个最小值

基本思路是，按照深度优先遍历每一个结点，找到对应的最大连通分量，然后比较得到最小值，其中在删除一个结点的时候联通量分为下图这个情况：



以4为例，先是计算1的节点个数，比较以4为根的其他子树的节点个数(比较res)，再计算2区域的节点个数，比较以4为根的其他子树的节点个数。1+2+3计算该三个区域的节点的个数，跳出for循环，与4区域比较连通量。最后与以删除其他节点的最大连通量比较，取最小。ans=min(ans,res);

连通量分为，以`4`为根结点的两颗子树`1`区域和`2`区域，以及删掉以`4`为根结点的这颗子树剩下的连通区域`4`

以上图中的`4`为例来进行递归分析：

`dfs(1)`进入第二次`for`循环调用`dfs(4)`

1. 进入`dfs(4)`
2. 将`st[4]`设置为`true`，表示`4`被遍历过了
3. `sum = 1`，最开始以`4`为根结点的这颗子树的大小先设置为`1`
4. `res = 0`，删掉这个`4`结点后最大连通量先设置为`0`
5. `dfs(4)`的第一次`for`循环
    1. 进入`dfs(3)`
    2. 将`st[3]`设置为`true`，表示`3`被遍历过了
    3. `sum = 1`，最开始以`3`为根结点的这颗子树的大小先设置为`1`
    4. `res = 0`，删掉这个`3`结点后最大连通量先设置为`0`
    5. `dfs(3)`的第一次`for`循环
        1. 进入`dfs(9)`
        2. 将`st[9]`设置为`true`，表示`3`被遍历过了
        3. `sum = 1`，最开始以`9`为根结点的这颗子树的大小先设置为`1`
        4. `res = 0`，删掉这个`9`结点后最大连通量先设置为`0`
        5. `dfs(9)`因为是双向边，所以进入`for`循环，遍历到`3`但是`3`遍历过了，所以进入一次后退出了`for`循环
        6. 从而得到，以`9`为根结点的这颗子树的大小为`1`
        7. `res = max(res, n - sum)`，从而得到删掉这个`9`结点后最大连通量为`8`

        8. 更新一下`ans`
        9. 返回以`9`为根结点的这颗子树的大小`1`
    6. 得到了一个子树的大小，也就得到了连通量，所以更新一下`res`
    7. `sum += s`加上上次递归返回的子树的大小，更新一下本身的树的大小
    8. `dfs(3)`不进入第二次`for`循环
    9. 由于遍历了所有的子树，加上了所有的子树的大小，所以得到了以`3`为根结点的这颗子树的大小为`2`
    10. 由于遍历了所有的子树，也就找到了所有子树连通量的最大值`res = 1`
    11. 更新一下`res = max(res, n - sum)`，从而得到删掉这个`3`结点后最大连通量为`7`
    12. 更新一下`ans`
    13. 返回以`3`为根结点的这颗子树的大小`2`
6. `dfs(4)`的第二次`for`循环
    1. 进入`dfs(6)`
    2. 将`st[6]`设置为`true`，表示`6`被遍历过了
    3. `sum = 1`，最开始以`6`为根结点的这颗子树的大小先设置为`1`
    4. `res = 0`，删掉这个`6`结点后最大连通量先设置为`0`
    5. `dfs(6)`因为是双向边，所以进入`for`循环，遍历到`4`但是`4`遍历过了，所以进入一次后退出了`for`循环
    6. 从而得到，以`6`为根结点的这颗子树的大小为`1`
    7. `res = max(res, n - sum)`，从而得到删掉这个`6`结点后最大连通量为`8`
    8. 更新一下`ans`
    9. 返回以`6`为根结点的这颗子树的大小`1`
7. 得到了一个子树的大小，也就得到了连通量，所以更新一下`res`
8. `sum += s`加上上次递归返回的子树的大小，更新一下本身的树的大小
9. 由于遍历了所有的子树，加上了所有的子树的大小，所以得到了以`4`为根结点的这颗子树的大小为`4`
10. 由于遍历了所有的子树，也就找到了所有子树连通量的最大值`res = 2`
11. 更新一下`res = max(res, n - sum)`，从而得到删掉这个`4`结点后最大连通量为`5`
12. 更新一下`ans`
13. 返回以`4`为根结点的这颗子树的大小`4`

接着`dfs(1)`进入第三次`for`循环进入`dfs(7)`重复上述过程

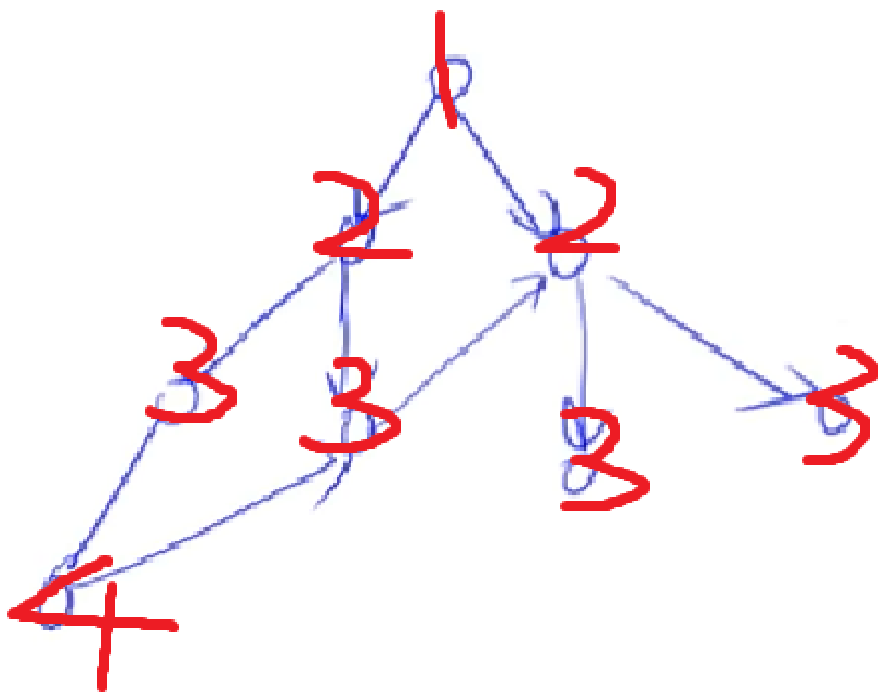
我们仍需要注意的是

1. 我们的目的是遍历所有的结点，由于是一个无向连通图，所以我们从任意一个结点开始都可以深度遍历所有的结点，都可以得到答案

2. 由于题目中说了是一颗树，所以我们不用考虑出现环，但是我们需要注意的是这颗树是无向的连通图，当从`4`结点递归到`6`的时候，如不标记`4`已经被遍历的话，会再从`6`遍历到`4`，从而进入无限递归无法退出，所以必须得用`if(!st[j])`这条语句

宽度优先遍历

宽度优先 遍历也与宽度优先搜索差不多，一层层往下搜索，可以参考走迷宫的例子进行理解



给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环。

所有边的长度都是 1，点的编号为 $1 \sim n$。

请你求出 1 号点到 n 号点的最短距离，如果从 1 号点无法走到 n 号点，输出 -1。

输入格式

第一行包含两个整数 n 和 m。

接下来 m 行，每行包含两个整数 a 和 b，表示存在一条从 a 走到 b 的长度为 1 的边。

输出格式

输出一个整数，表示 1 号点到 n 号点的最短距离。

数据范围

$1 \le n, m \le 10^5$

输入样例：

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

输出样例：

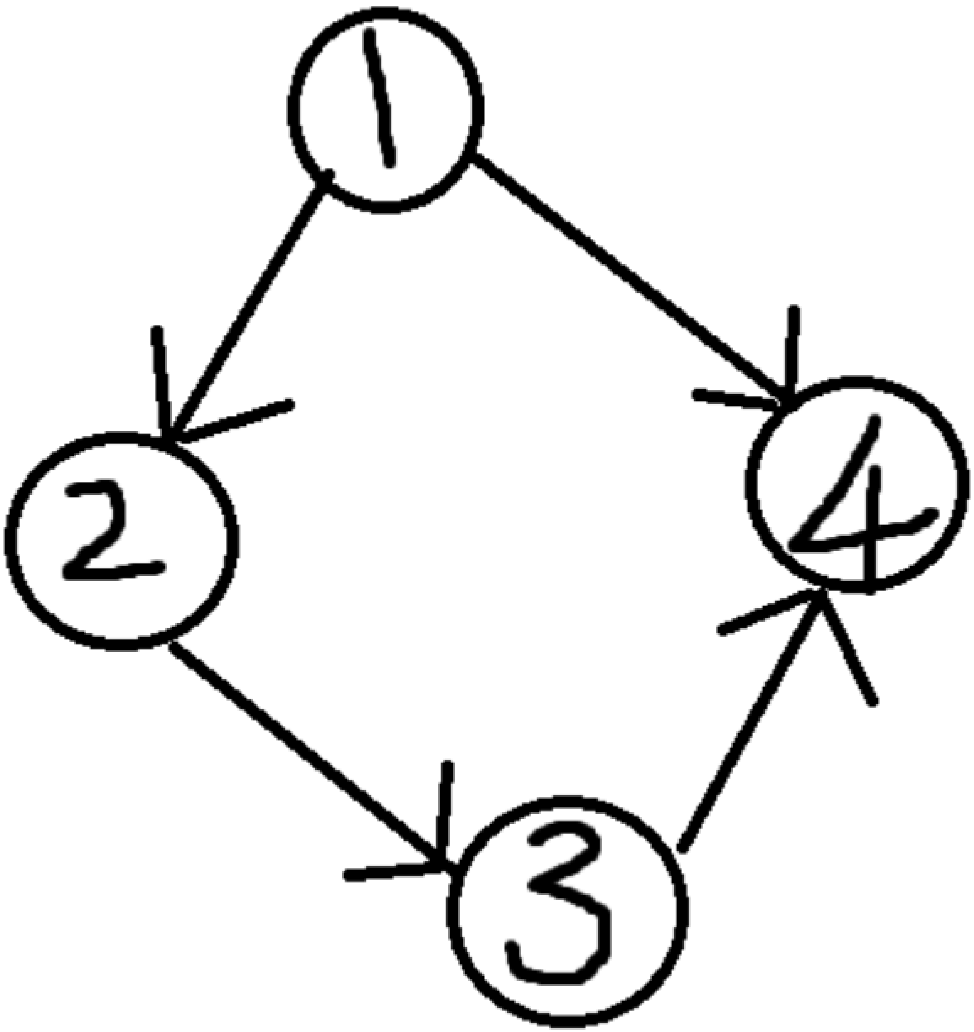
```
1
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 100010;

int n, m;
//定义图的结构
int h[N], e[N], ne[N], idx;
//定义距离数组和队列
int d[N], q[N];

//加入一条边
void add(int a, int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
```

```cpp
    idx ++;
}

int bfs()
{
    //定义队列的头部和尾部
    int hh, tt;
    hh = tt = 0;
    q[0] = 1;
    //将所有的点的距离初始化为-1
    memset(d, -1, sizeof d);
    //第一个点的初始距离为0
    d[1] = 0;

    while(hh <= tt)
    {
        int t = q[hh ++];

        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(d[j] == -1)
            {
                d[j] = d[t] + 1;
                q[++ tt] = j;
            }
        }
    }

    return d[n];
}


int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);

    for(int i = 0; i < m; i ++)
    {
        int a, b;
        cin >> a >> b;

        add(a, b);
    }

    cout << bfs() << endl;

    return 0;
}

```

这个代码跟`bfs`走迷宫问题几乎一摸一样，在这里只是简单分析一下`if(d[j] == -1)`这条语句的作用

假如有下面这个图



第一次，`1`进入队列

第二次，遍历到`2` 和`4`，将`2`和`4`加入队列，`1`出队列，`d[2] = d[4] = 1`

第三次，遍历到`3`，将`3`加入队列，`2`出队列，`d[3] = 2`
第四次，`4`出队列，队列中只剩下`3`

第五次，遍历到`4`，这时`4`是从`3`遍历的，如果此时允许遍历的话，那么`d[4] = 3`很明显大于最开始`d[4]`的值

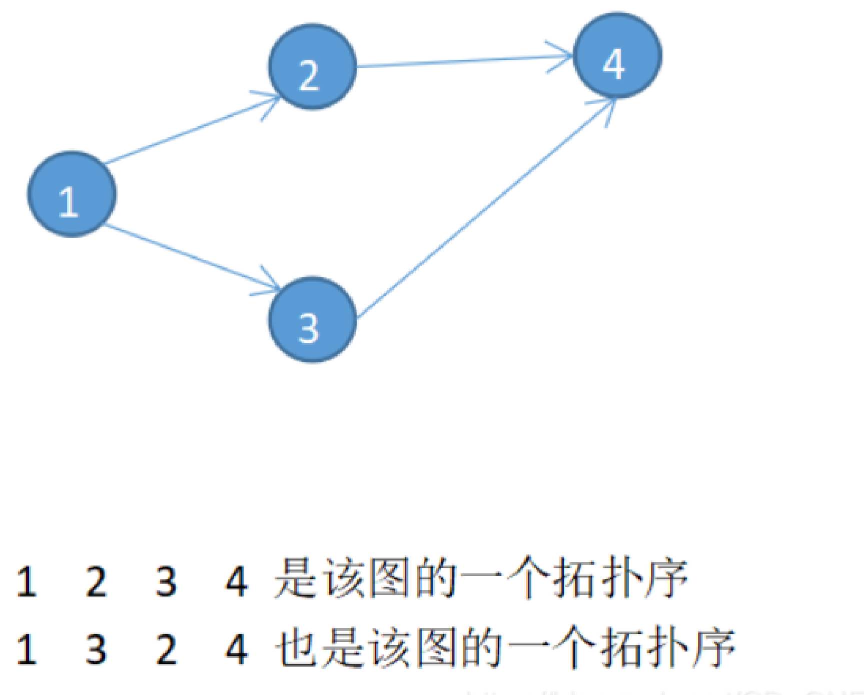
所以用`if(d[j] == -1)`这条语句来保证所有的结点都是第一次遍历的，只有第一次遍历得到的距离才是最短的

拓扑序列

基本知识

拓扑序列是对于**有向图**而言的，有向图的拓扑序是其顶点的线性排序，使得对于从顶点`u`到顶点`v`的每个有向边`u v`， `u` 在序列中都在`v` 之前。

如下图：



对于上图， 存在`4`条边：`（1,3）（1,2）（2,4）（2,3）`
该图的拓扑序必须要满足以下两点：

1. 每个顶点只出现一次。
2. 对于图中的任何一条边，起点必须在终点之前。

另外有几个概念：

1. 入度，指的是指向这个顶点的边的个数，比如上图中`1`的入度就是`0`
2. 出度，指的是这个顶点的出边的个数，比如上图中`1`的出度就是`2`
3. 有向无环图一定是存在一个拓扑序列的，并且有向无环图一定有入度为`0`的顶点
4. 重边，**两个完全相同的边**，称作（一组）重边。在无向图中 `(u,v)` 和`(v,u)` 算一组重边，而在有向图中， `u -> v` 和 `v -> u` 不为重边
5. 自环 (Loop)： 边 e 的两个端点相同，则 e 称为一个自环

基本求法：

拓扑序是按照点的先后顺序排列的，也就是说入度为0的点一定是排在前面的，我们直接对一个图`BFS`一遍， `BFS`过程中更新每个点的入度，如果一个点的入度为`0`，那么就将其加入拓扑序，并且删除其与后继结点的所有边。

代码实现

给定一个 n 个点 m 条边的有向图，点的编号是 1 到 n，图中可能存在重边和自环。

请输出任意一个该有向图的拓扑序列，如果拓扑序列不存在，则输出 -1。

若一个由图中所有点构成的序列 A 满足：对于图中的每条边 (x, y)，x 在 A 中都出现在 y 之前，则称 A 是该图的一个拓扑序列。

输入格式

第一行包含两个整数 n 和 m。

接下来 m 行，每行包含两个整数 x 和 y，表示存在一条从点 x 到点 y 的有向边 (x, y)。

输出格式

共一行，如果存在拓扑序列，则输出任意一个合法的拓扑序列即可。

否则输出 -1。

数据范围

$1 \le n, m \le 10^5$

输入样例：

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

输出样例：

```
1 2 3
```

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 100010;

int h[N], e[N], ne[N], idx;
int q[N];
int d[N];//d[N]用来存放每个结点的入度
int n, m;

void add(int a, int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx ++;
}


bool topsort()
{
    //一开时将队列的设置为一个空队列
```

```
    int hh = 0, tt = -1;


    //先从前往后遍历所有的点
    //将所有入度为0的点插入到队列里面去
    for(int i = 1; i <= n; i ++)
    {
        if(!d[i]) q[++ tt] = i;
    }

    //当队列不空的时候
    while(hh <= tt)
    {
        int t = q[hh ++];
        //拓展队头的元素，遍历每一个出边
        for(int i = h[t]; i != -1; i = ne[i])
        {
            //找到出边对应的这个点
            int j = e[i];

            //即对于一个队头，每扩展一个元素，将这个出边删除
            //让这个点的入度--
            d[j] --;
             //当这个元素的入度为0的时候就将这个元素加入队尾
             //说明这个元素是下一轮的拓扑序列
             //由于是队列，此时队列中的元素始终满足拓扑序列
             //类似于bfs，将结点一层层加入队列然后输出
            if(d[j] == 0)
            {
                q[++ tt] = j;
            }


        }

        //在for循环中只要删除出边后存在入度为0的结点，都会加入队列
        //所以队列就不会为空，一直到最后一个元素加入队列，然后输出
        //如果存在环的话形成环的元素始终不会加入到队列中
    }


    //如果所有的点都进入队列了，说明存在一个拓扑序列
    //说明队列尾一共增加了n次
    //tt从-1加了n次后的结果就是n - 1
    //说明只要tt为n - 1就存在一个拓扑序列
    return tt == n - 1;
}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
```

```
    memset(d, 0, sizeof d);
    for(int i = 0; i < m; i ++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        //每插入一条边就更新一下入度
        //一条a->b的边，则b的入度++
        d[b] ++;

    }

    if(topsort())
    {
        for(int i = 0; i < n; i ++) cout <<" "<<q[i];
    }
    else puts("-1");


    return 0;
}
```