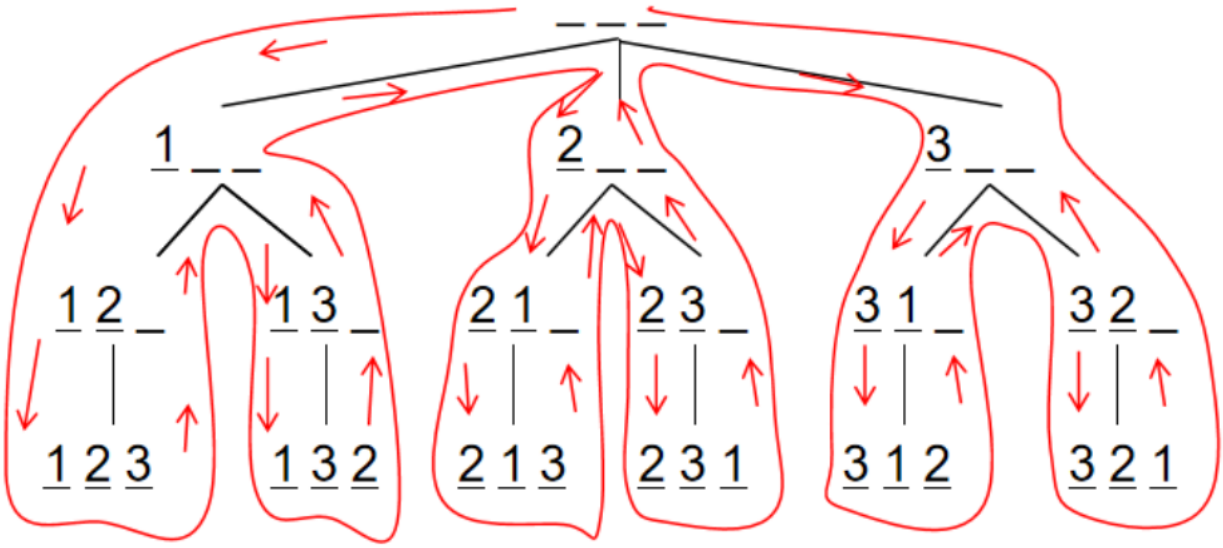
1：深度优先搜索（DFS)：

深度优先搜索（DFS）与宽度优先搜索（BFS)的区别：

搜索算法	数据结构	空间	最短性
DFS	栈	O（h)	不具有最短路性
BFS	队列	O（2 ^ h)	最短路

DFS 解决全排列问题，以 n = 3 为例，可以进行如下搜索：



假设有 3 个空位，从前往后填数字，每次填一个位置，填的数字不能和前面一样。最开始的时候，三个空位都是空的：_ _ _

首先填写第一个空位，第一个空位可以填 1，填写后为：1 _ _

填好第一个空位，填第二个空位，第二个空位可以填 2，填写后为：1 2 _

填好第二个空位，填第三个空位，第三个空位可以填 3，填写后为： 1 2 3

这时候，空位填完，无法继续填数，所以这是一种方案，输出

然后往后退一步，退到了状态：1 2 _ 。剩余第三个空位没有填数。第三个空位上除了填过的 3 ，没有其他数字可以填

因此再往后退一步，退到了状态：1 _ _。第二个空位上除了填过的 2，还可以填 3。第二个空位上填写 3，填写后为：1 3 _

填好第二个空位，填第三个空位，第三个空位可以填 2，填写后为： 1 3 2

这时候，空位填完，无法继续填数，所以这是一种方案，输出。

然后往后退一步，退到了状态：1 3 _ 。剩余第三个空位没有填数。第三个空位上除了填过的 2，没有其他数字可以填

因此再往后退一步，退到了状态：1 _ 。**第二个空位上除了填过的 2，3，没有其他数字可以填。因此再往后退一步，退到了状态：** _ _。第一个空位上除了填过的 1，还可以填

2。第一个空位上填写 2，填写后为：2 __ __
填好第一个空位，填第二个空位，第二个空位可以填 1，填写后为：2 1 __
填好第二个空位，填第三个空位，第三个空位可以填 3，填写后为：2 1 3
这时候，空位填完，无法继续填数，所以这是一种方案，输出。
然后往后退一步，退到了状态：2 1 __ 。剩余第三个空位没有填数。第三个空位上除了填过的 3，没有其他数字可以填。
因此再往后退一步，退到了状态：2 __ __。第二个空位上除了填过的 1，还可以填 3。第二个空位上填写 3，填写后为：2 3 __
填好第二个空位，填第三个空位，第三个空位可以填 1，填写后为：2 3 1
这时候，空位填完，无法继续填数，所以这是一种方案，输出。
然后往后退一步，退到了状态：2 3 __ 。剩余第三个空位没有填数。第三个空位上除了填过的 1，没有其他数字可以填。
因此再往后退一步，退到了状态：2 __ 。**第二个空位上除了填过的 1，3，没有其他数字可以填。因此再往后退一步，退到了状态：** __ __。第一个空位上除了填过的 1，2，还可以填 3。第一个空位上填写 3，填写后为：3 __ __
填好第一个空位，填第二个空位，第二个空位可以填 1，填写后为：3 1 __
填好第二个空位，填第三个空位，第三个空位可以填 2，填写后为：3 1 2
这时候，空位填完，无法继续填数，所以这是一种方案，输出
然后往后退一步，退到了状态：3 1 __ 。剩余第三个空位没有填数。第三个空位上除了填过的 2，没有其他数字可以填
因此再往后退一步，退到了状态：3 __ __。第二个空位上除了填过的 1，还可以填 2。第二个空位上填写 2，填写后为：3 2 __
填好第二个空位，填第三个空位，第三个空位可以填 1，填写后为：3 2 1
这时候，空位填完，无法继续填数，所以这是一种方案，输出
然后往后退一步，退到了状态：3 2 __ 。剩余第三个空位没有填数。第三个空位上除了填过的 1，2，没有其他数字可以填
因此再往后退一步，退到了状态：3 __ 。**第二个空位上除了填过的 1，2，没有其他数字可以填。因此再往后退一步，退到了状态：** __ __。第一个空位上除了填过的 1，2，3，没有其他数字可以填
此时深度优先搜索结束，输出了所有的方案
例题：
给定一个整数 n ，将数字1 - n 排成一排，将会有很多种的排列方法
现在，请你按照字典序将所有的排列方法输出

输入格式：

共一行，包含一个整数 n

输出格式：

按字典序输出所有排列方案，每个方案占一行

输入样例：

```
3
```

输出样例：

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

```
import java.util.Scanner;
public class Main{
    public static int N = 10;
    public static int[] path = new int[N];
    public static boolean[] st = new boolean[N];
    public static void dfs(int u, int n){
        if(u == n){
            for(int i = 0; i < n; i++){
                System.out.printf("%d ", path[i]);
            }
            System.out.println();
        }else{
            for(int i = 1; i <= n; i++){
                if(!st[i]){
                    path[u] = i;
                    st[i] = true;
                    dfs(u + 1, n);
                    st[i] = false;
                }
```

```
            }
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        dfs(0, n);
    }
}
```

DFS 解决 n --- 皇后问题：

每一行必定有一个皇后，对行进行深度遍历

对于第 r 行的第 i 个位置，判断每个点是否可以放皇后，如果可以，则放皇后，然后处理 r + 1 行

直到 r = n，程序指行完毕

核心思路：深度优先遍历

函数名：void dfs(int r): 深度优先遍历函数

参数 r：从第 r 行开始放棋子，处理第 r 行

递归结束判定：当 r == n 的时候，说明应该处理第 n 行了，也代表第 0 ~ n-1 行放好棋子，也就是整个棋盘放好了棋子，也就是得到了一种解，也就是递归结束

第 r 行，第 i 列能不能放棋子：用数组 dg udg cor 分别表示：点对应的两个斜线以及列上是否有皇后

dg[i + r] 表示 r 行 i 列处，所在的对角线上有没有棋子，udg[n - i + r]表示 r行i列处，所在的反对角线上有没有棋子，cor[i]表示第i列上有没有棋子。如果 r 行 i 列的对角线，反对角线上都没有棋子，即!cor[i] && !dg[i + r] && !udg[n - i + r]为真，则代表 r 行 i 列处可以放棋子

例题：

n --- 皇后问题是指将 n 个皇后放在 n * n 的国际象棋棋盘上，使得皇后不能相互攻击到，即任意两个皇后都不能处于同一行， 同一列或同一斜线上

现在给定整数 n，请你输出所有的满足条件的棋子摆法

输入格式：

共一行，包含整数 n

输出格式：

每个解决方案占 n 行，每行输出一个长度为 n 的字符串，用来表示完整的棋盘状态，其中，. 表示某一个位置的方格状态为空，Q 表示某一个位置上摆着皇后
每个方案输出完成后，输出一个空行
输入样例：

```
4
```

输出样例：

```
.Q..
...Q
Q...
..Q.

..Q.
Q...
...Q
.Q..
```

```
import java.util.Scanner;
public class Main{
    public static int N = 20;
    public static char[][] g = new char[N][N];
    public static boolean[] cal = new boolean[N];
    public static boolean[] dg = new boolean[N];
    public static boolean[] udg = new boolean[N];
    public static void dfs(int u, int n){
        if(u == n){
            for(int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                    System.out.print(g[i][j]);
                }
                System.out.println();
            }
            System.out.println();
            return;
```

```
        }else{
            for(int i = 0; i < n; i++){
                if(!cal[i] && !dg[i - u + n] && !udg[i + u]){
                g[u][i] = 'Q';
                cal[i] = dg[i - u + n] = udg[i + u] = true;
                dfs(u + 1, n);
                cal[i] = dg[i - u + n] = udg[i + u] = false;
                g[u][i] = '.';
            }
        }
    }
}
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                g[i][j] = '.';
            }
        }
        dfs(0, n);
    }
}
```

2：BFS ---- 广度优先搜索：（一般数边路权重相等时才可以用BFS）

基本框架：

```
queue<- 初始状态（将初始状态放入一个队列当中）
while(queue不空){
    t <- 队头（把队头拿出来）
    扩展 t
}
```

BFS 解决走迷宫问题（边的权值相同）

给定一个n*m的二维整数数组，用来表示一个迷宫，数组中只包含0或1，其中0表示可以走的路，1表示不可通过的墙壁

最初，有一个人位于左上角(1, 1)处，已知该人每次可以向上、下、左、右任意一个方向移动一个位置

请问，该人从左上角移动至右下角(n, m)处，至少需要移动多少次

数据保证(1, 1)处和(n, m)处的数字为0，且一定至少存在一条通路

输入格式

第一行包含两个整数n和m

接下来n行，每行包含m个整数（0或1），表示完整的二维数组迷宫

输出格式

输出一个整数，表示从左上角移动至右下角的最少移动次数

输入样例：

```
5 5
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

输出样例：

```
8
```

```
import java.io.*;
public class Main{
    public static int N = 110;
    public static int[][] g = new int[N][N];
    public static int[][] d = new int[N][N];
    public static PII[] q = new PII[N * N];
    public static int hh, tt;
    public static int n, m;
    public static int bfs(){
        hh = 0;
        tt = -1;
        d[0][0] = 0;
        q[++ tt] = new PII(0, 0);
        int[] dx = {-1, 0, 1, 0};
        int[] dy = {0, 1, 0, -1};
        while(hh <= tt){
```

```
            PII t = q[hh++];
            for(int i = 0; i < 4; i++){
            int x = t.first + dx[i];
            int y = t.second + dy[i];
            if(x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y] == -1){
                   d[x][y] = d[t.first][t.second] + 1;
                   q[++ tt] = new PII(x, y);
                }
            }
        }
        return d[n - 1][m - 1];
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
        String[] st = br.readLine().split(" ");
        n = Integer.parseInt(st[0]);
        m = Integer.parseInt(st[1]);
        for(int i = 0; i < n; i++){
            String[] s = br.readLine().split(" ");
            for(int j = 0; j < m; j++){
                g[i][j] = Integer.parseInt(s[j]);
                d[i][j] = -1;
            }
        }
        System.out.println(bfs());
        bw.flush();
        br.close();
        bw.close();
    }
}
class PII{
    int first, second;
    public PII(int first, int second){
        this.first = first;
        this.second = second;
    }
}
```

3：树与图的深度优先遍历：

树与图的存储方式：

```
//类似于单链表
public static int N = 100010;
public static int M = N * 2;
public static int[] h = new int[N];
public static int[] e = new int[M];
public static int[] ne = new int[M];
public static void add(int a, int b){
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}
```

树与图的深度遍历方式：

```
for(int i = h[u], i != -1; i = ne[i]){
    int j = e[i];
    if(!st[j]){
        dfs(j);
    }
}
```

例题：

给定一颗树，树中包含 n 个结点（编号1~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 之前存在一条边

输出格式

输出一个整数m，表示重心的所有的子树中最大的子树的结点数目

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
```

```
import java.util.Scanner;
public class Main{
    public static int N = 100010, M = N * 2;
    public static int idx,n;
    public static boolean[] st = new boolean[N];
    public static int[] h = new int[N];
    public static int[] e = new int[M];
    public static int[] ne = new int[M];
    public static int ans = N;
    public static void add(int a, int b){
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx;
        idx ++;
    }
    public static int dfs(int u){
        int res = 0;
        st[u] = true;
        int sum = 1;
        for(int i = h[u]; i != -1; i = ne[i]){
            int j = e[i];
            if(!st[j]){
                int s = dfs(j);
```

```
                res = Math.max(res, s);
                sum += s;
            }
        }
        res = Math.max(res, n - sum);
        ans = Math.min(res, ans);
        return sum;

    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        for(int i = 1; i <= n; i++){
            h[i] = -1;
        }
        for(int i = 0; i < n - 1; i++){
            int a = sc.nextInt();
            int b = sc.nextInt();
            add(a, b);
            add(b, a);
        }
        dfs(1);
        System.out.println(ans);
    }
}
```

4：树与图的广度优先遍历：

例题：

```
import java.util.Scanner;
public class Main{
    public static int N = 100010;
    public static int hh, tt;
    public static int n, m, idx;
    public static int[] h = new int[N];
    public static int[] e = new int[N];
    public static int[] ne = new int[N];
    public static int[] q = new int[N];
```

```java
    public static int[] d = new int[N];
    public static void add(int a, int b){
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx;
        idx ++;
    }
    public static int bfs(){
        hh = 0;
        tt = -1;
        d[1] = 0;
        q[++ tt] = 1;
        while(hh <= tt){
            int t = q[hh ++];
            for(int i = h[t]; i != -1; i = ne[i]){
                int s = e[i];
                if(d[s] == -1){
                    d[s] = d[t] + 1;
                    q[++ tt] = s;
                }
            }
        }
        return d[n];
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            h[i] = -1;
            d[i] = -1;
        }
        for(int i = 0; i < m; i++){
            int a = sc.nextInt();
            int b = sc.nextInt();
            add(a, b);
        }
        System.out.println(bfs());
    }
}
```

5：有向图的拓扑序列：

拓扑序列：若一个由图中所有的点构成的序列 A 满足：对于图中每条边（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

入度：指进入该结点的箭头数

出度：指出去该结点的箭头数

基本思路：

```
queue  <---- 所有入度为0的点；
while（queue不空）{
    t <---- 队头；
    枚举 t 的所有出边 t --> j；
    删掉 t ---> j，d[j] --；
    if(d[j] == 0){
        queue  <---- j;
    }
}
```

例题：

```
import java.util.Scanner;
public class Main{
    public static int N = 100010;
    public static int hh, tt, n, m, idx;
    public static int[] d = new int[N];
    public static int[] q = new int[N];
    public static int[] h = new int[N];
    public static int[] e = new int[N];
    public static int[] ne = new int[N];
    public static void add(int x, int y){
        e[idx] = y;
        ne[idx] = h[x];
        h[x] = idx;
        idx ++;
    }
    public static boolean bfs(){
        hh = 0;
```

```
        tt = -1;
        for(int i = 1; i <= n; i++){
            if(d[i] == 0){
                q[++ tt] = i;
            }
        }
        while(hh <= tt){
            int t = q[hh ++];
            for(int i = h[t]; i != -1; i = ne[i]){
                int s = e[i];
                d[s]--;
                if(d[s] == 0){
                    q[++ tt] = s;
                }
            }
        }
        return tt == n - 1;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            h[i] = -1;
        }
        for(int i = 0; i < m; i++){
            int x = sc.nextInt();
            int y = sc.nextInt();
            add(x, y);
            d[y] ++;
        }
        if(bfs()){
            for(int i = 0; i < n; i++){
                System.out.printf("%d ", q[i]);
            }
        }else{
            System.out.println("-1");
        }
    }
}
```