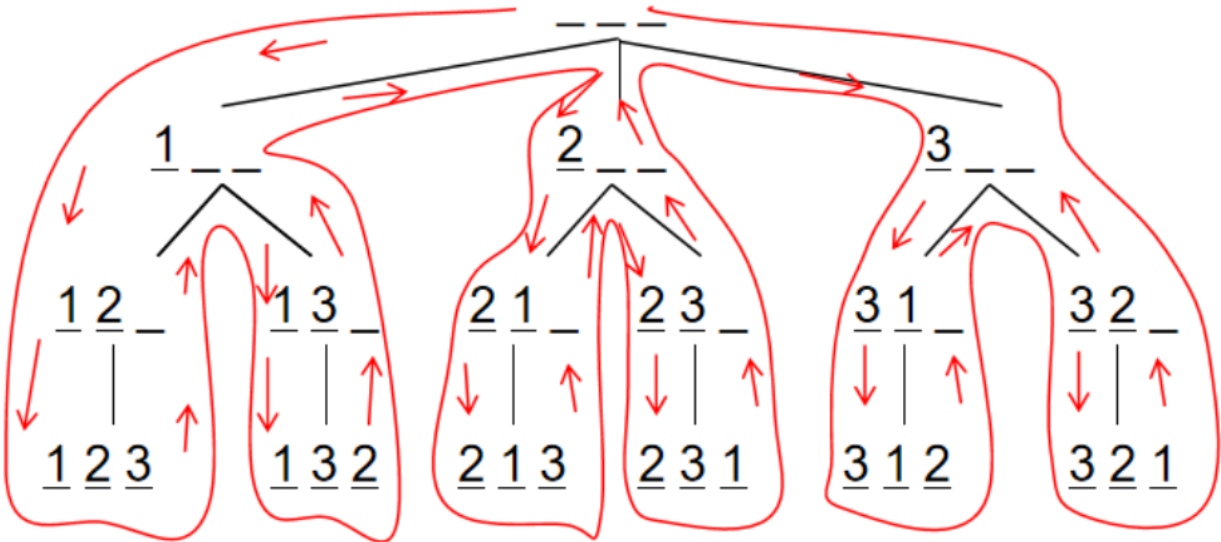


1: 深度优先搜索 (DFS):

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:

```
1 3
```

输出样例:

```
1 1 2 3
2 1 3 2
3 2 1 3
4 2 3 1
5 3 1 2
6 3 2 1
```

```
1 import java.util.Scanner;
2 public class Main{
3     public static int N = 10;
4     public static int[] path = new int[N];
5     public static boolean[] st = new boolean[N];
6     public static void dfs(int u, int n){
7         if(u == n){
8             for(int i = 0; i < n; i++){
9                 System.out.printf("%d ", path[i]);
10            }
11            System.out.println();
12        }else{
13            for(int i = 1; i <= n; i++){
14                if(!st[i]){
15                    path[u] = i;
16                    st[i] = true;
17                    dfs(u + 1, n);
18                    st[i] = false;
19                }
20            }
21        }
22    }
23 }
```

```

20         }
21     }
22 }
23 public static void main(String[] args){
24     Scanner sc = new Scanner(System.in);
25     int n = sc.nextInt();
26     dfs(0, n);
27 }
28 }

```

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 $0 \sim n-1$ 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：

```
1 4
```

输出样例：

```
1 .Q..  
2 ...Q  
3 Q...  
4 ..Q.  
5  
6 ..Q.  
7 Q...  
8 ...Q  
9 .Q..
```

```
1 import java.util.Scanner;  
2 public class Main{  
3     public static int N = 20;  
4     public static char[][] g = new char[N][N];  
5     public static boolean[] cal = new boolean[N];  
6     public static boolean[] dg = new boolean[N];  
7     public static boolean[] udg = new boolean[N];  
8     public static void dfs(int u, int n){  
9         if(u == n){  
10             for(int i = 0; i < n; i++){  
11                 for(int j = 0; j < n; j++){  
12                     System.out.print(g[i][j]);  
13                 }  
14                 System.out.println();  
15             }  
16             System.out.println();  
17             return;  
18         }  
19         for(int i = 0; i < n; i++){  
20             if(!cal[i] && !dg[u+i] && !udg[u-i]){  
21                 g[u][i] = 'Q';  
22                 cal[i] = dg[u+i] = udg[u-i] = true;  
23                 dfs(u+1, n);  
24                 g[u][i] = '.';  
25                 cal[i] = dg[u+i] = udg[u-i] = false;  
26             }  
27         }  
28     }  
29     Scanner sc = new Scanner(System.in);  
30     int n = sc.nextInt();  
31     dfs(0, n);  
32 }
```

```

18         }else{
19             for(int i = 0; i < n; i++){
20                 if(!cal[i] && !dg[i - u + n] && !udg[i + u]){
21                     g[u][i] = 'Q';
22                     cal[i] = dg[i - u + n] = udg[i + u] = true;
23                     dfs(u + 1, n);
24                     cal[i] = dg[i - u + n] = udg[i + u] = false;
25                     g[u][i] = '.';
26                 }
27             }
28         }
29     }
30     public static void main(String[] args){
31         Scanner sc = new Scanner(System.in);
32         int n = sc.nextInt();
33         for(int i = 0; i < n; i++){
34             for(int j = 0; j < n; j++){
35                 g[i][j] = '.';
36             }
37         }
38         dfs(0, n);
39     }
40 }

```

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：

```

1 queue<- 初始状态（将初始状态放入一个队列当中）
2 while(queue不空){
3     t <- 队头（把队头拿出来）
4     扩展 t
5 }

```

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：

```
1 5 5
2 0 1 0 0 0
3 0 1 0 1 0
4 0 0 0 0 0
5 0 1 1 1 0
6 0 0 0 1 0
```

输出样例：

```
1 8
```

```
1 import java.io.*;
2 public class Main{
3     public static int N = 110;
4     public static int[][] g = new int[N][N];
5     public static int[][] d = new int[N][N];
6     public static PII[] q = new PII[N * N];
7     public static int hh, tt;
8     public static int n, m;
9     public static int bfs(){
10         hh = 0;
11         tt = -1;
12         d[0][0] = 0;
13         q[++ tt] = new PII(0, 0);
14         int[] dx = {-1, 0, 1, 0};
15         int[] dy = {0, 1, 0, -1};
16         while(hh <= tt){
```

```

17         PII t = q[hh++];
18         for(int i = 0; i < 4; i++){
19             int x = t.first + dx[i];
20             int y = t.second + dy[i];
21             if(x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y] == -1){
22                 d[x][y] = d[t.first][t.second] + 1;
23                 q[++ tt] = new PII(x, y);
24             }
25         }
26     }
27     return d[n - 1][m - 1];
28 }
29 public static void main(String[] args) throws IOException{
30     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
31     BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
32     String[] st = br.readLine().split(" ");
33     n = Integer.parseInt(st[0]);
34     m = Integer.parseInt(st[1]);
35     for(int i = 0; i < n; i++){
36         String[] s = br.readLine().split(" ");
37         for(int j = 0; j < m; j++){
38             g[i][j] = Integer.parseInt(s[j]);
39             d[i][j] = -1;
40         }
41     }
42     System.out.println(bfs());
43     bw.flush();
44     br.close();
45     bw.close();
46 }
47 }
48 class PII{
49     int first, second;
50     public PII(int first, int second){
51         this.first = first;
52         this.second = second;
53     }
54 }

```

3: 树与图的深度优先遍历:

树与图的存储方式：

```
1 //类似于单链表
2 public static int N = 100010;
3 public static int M = N * 2;
4 public static int[] h = new int[N];
5 public static int[] e = new int[M];
6 public static int[] ne = new int[M];
7 public static void add(int a, int b){
8     e[idx] = b;
9     ne[idx] = h[a];
10    h[a] = idx;
11    idx++;
12 }
```

树与图的深度遍历方式：

```
1 for(int i = h[u], i != -1; i = ne[i]){
2     int j = e[i];
3     if(!st[j]){
4         dfs(j);
5     }
6 }
```

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：

```
1  9
2  1 2
3  1 7
4  1 4
5  2 8
6  2 5
7  4 3
8  3 9
9  4 6
```

输出样例：

```
1  4
```

```
1  import java.util.Scanner;
2  public class Main{
3      public static int N = 100010, M = N * 2;
4      public static int idx,n;
5      public static boolean[] st = new boolean[N];
6      public static int[] h = new int[N];
7      public static int[] e = new int[M];
8      public static int[] ne = new int[M];
9      public static int ans = N;
10     public static void add(int a, int b){
11         e[idx] = b;
12         ne[idx] = h[a];
13         h[a] = idx;
14         idx ++;
15     }
16     public static int dfs(int u){
17         int res = 0;
18         st[u] = true;
19         int sum = 1;
20         for(int i = h[u]; i != -1; i = ne[i]){
21             int j = e[i];
22             if(!st[j]){
23                 int s = dfs(j);
```

```

24         res = Math.max(res, s);
25         sum += s;
26     }
27 }
28 res = Math.max(res, n - sum);
29 ans = Math.min(res, ans);
30 return sum;
31
32 }
33 public static void main(String[] args){
34     Scanner sc = new Scanner(System.in);
35     n = sc.nextInt();
36     for(int i = 1; i <= n; i++){
37         h[i] = -1;
38     }
39     for(int i = 0; i < n - 1; i++){
40         int a = sc.nextInt();
41         int b = sc.nextInt();
42         add(a, b);
43         add(b, a);
44     }
45     dfs(1);
46     System.out.println(ans);
47 }
48 }

```

4: 树与图的广度优先遍历:

例题:

```

1  import java.util.Scanner;
2  public class Main{
3      public static int N = 100010;
4      public static int hh, tt;
5      public static int n, m, idx;
6      public static int[] h = new int[N];
7      public static int[] e = new int[N];
8      public static int[] ne = new int[N];
9      public static int[] q = new int[N];

```

```
10     public static int[] d = new int[N];
11     public static void add(int a, int b){
12         e[idx] = b;
13         ne[idx] = h[a];
14         h[a] = idx;
15         idx ++;
16     }
17     public static int bfs(){
18         hh = 0;
19         tt = -1;
20         d[1] = 0;
21         q[++ tt] = 1;
22         while(hh <= tt){
23             int t = q[hh ++];
24             for(int i = h[t]; i != -1; i = ne[i]){
25                 int s = e[i];
26                 if(d[s] == -1){
27                     d[s] = d[t] + 1;
28                     q[++ tt] = s;
29                 }
30             }
31         }
32         return d[n];
33     }
34     public static void main(String[] args){
35         Scanner sc = new Scanner(System.in);
36         n = sc.nextInt();
37         m = sc.nextInt();
38         for(int i = 1; i <= n; i++){
39             h[i] = -1;
40             d[i] = -1;
41         }
42         for(int i = 0; i < m; i++){
43             int a = sc.nextInt();
44             int b = sc.nextInt();
45             add(a, b);
46         }
47         System.out.println(bfs());
48     }
49 }
```

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:

```
1 queue <----- 所有入度为0的点;
2 while (queue不空) {
3     t <----- 队头;
4     枚举 t 的所有出边 t --> j;
5     删掉 t ---> j, d[j] --;
6     if(d[j] == 0){
7         queue <----- j;
8     }
9 }
```

例题:

```
1 import java.util.Scanner;
2 public class Main{
3     public static int N = 100010;
4     public static int hh, tt, n, m, idx;
5     public static int[] d = new int[N];
6     public static int[] q = new int[N];
7     public static int[] h = new int[N];
8     public static int[] e = new int[N];
9     public static int[] ne = new int[N];
10    public static void add(int x, int y){
11        e[idx] = y;
12        ne[idx] = h[x];
13        h[x] = idx;
14        idx ++;
15    }
16    public static boolean bfs(){
17        hh = 0;
```

```

18     tt = -1;
19     for(int i = 1; i <= n; i++){
20         if(d[i] == 0){
21             q[++ tt] = i;
22         }
23     }
24     while(hh <= tt){
25         int t = q[hh ++];
26         for(int i = h[t]; i != -1; i = ne[i]){
27             int s = e[i];
28             d[s]--;
29             if(d[s] == 0){
30                 q[++ tt] = s;
31             }
32         }
33     }
34     return tt == n - 1;
35 }
36 public static void main(String[] args){
37     Scanner sc = new Scanner(System.in);
38     n = sc.nextInt();
39     m = sc.nextInt();
40     for(int i = 1; i <= n; i++){
41         h[i] = -1;
42     }
43     for(int i = 0; i < m; i++){
44         int x = sc.nextInt();
45         int y = sc.nextInt();
46         add(x, y);
47         d[y] ++;
48     }
49     if(bfs()){
50         for(int i = 0; i < n; i++){
51             System.out.printf("%d ", q[i]);
52         }
53     }else{
54         System.out.println("-1");
55     }
56 }
57 }

```

