58、可以组成网络的服务器，考点或实现——深度优先搜索 DFS

题目描述

在一个机房中，服务器的位置标识在 n*m 的整数矩阵网格中，1 表示单元格上有服务器，0 表示没有。如果两台服务器位于同一行或者同一列中紧邻的位置，则认为它们之间可以组成一个局域网。

请你统计机房中最大的局域网包含的服务器个数。

输入描述

第一行输入两个正整数，n和m，0<n,m<=100

之后为n*m的二维数组，代表服务器信息

输出描述

最大局域网包含的服务器个数。

用例

输入	2 2 1 0 1 1
输出	3
说明	[0][0]、[1][0]、[1][1]三台服务器相互连接，可以组成局域网

题目解析

本题可以用并查集求解。

题目描述中说：

“如果两台服务器位于同一行或者同一列中紧邻的位置，则认为它们之间可以组成一个局域网。”

其实这就是并查集的判断两个顶点是否可以合并的条件。

但是我们需要注意题目描述说是：同一行或者同一列中**紧邻**的位置，其实就是处于上下左右的位置。

所有服务器合并检查完后，属于同一个父级的服务器，即为同一局域网的服务器。然后我们再比较出服务器数量最多的局域网即可。

补充：

使用并查集的话，可能会重复判断已经合并的两个顶点，并且最后我们还需要根据ufs.fa数组再遍历一遍，统计每个局域网的服务器数量，这其实都挺冗余的。

本题其实采用dfs才是更好的选择，我们找到一个服务器后，就再去其上下左右找下一个服务器，当找到新服务器，再递归去找其上下左右，按此逻辑，就像拔地瓜藤一样，一下子把所有地瓜都拔出来。而这就是深度优先搜索，即dfs。

为了避免重复统计，我们将统计过的服务器位置的值从1变为0。

JavaScript算法源码

并查集解法

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const lines = [];
let n, m;
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 1) {
    [n, m] = lines[0].split(" ").map(Number);
  }

  if (n && lines.length === n + 1) {
    lines.shift();
    const grid = lines.map((line) => line.split(" ").map(Number));

    console.log(countServers(grid));

    lines.length = 0;
  }
```

```
});

/**
 * @param {number[][]} grid
 * @return {number}
 */
var countServers = function (grid) {
  const m = grid.length;
  const n = grid[0].length;

  const ufs = new UnionFindSet(m * n);

  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      if (grid[i][j] === 1) {
        const x = i * n + j;
        if (i - 1 >= 0 && grid[i - 1][j] === 1) ufs.union(x, x - n);
        if (i + 1 < m && grid[i + 1][j] === 1) ufs.union(x, x + n);
        if (j - 1 >= 0 && grid[i][j - 1] === 1) ufs.union(x, x - 1);
        if (j + 1 < n && grid[i][j + 1] === 1) ufs.union(x, x + 1);
      }
    }
  }

  const count = {};

  ufs.fa.forEach((f) => {
    count[f] ? count[f]++ : (count[f] = 1);
  });

  return Object.values(count).sort((a, b) => b - a)[0];
};

class UnionFindSet {
  constructor(n) {
    this.fa = new Array(n).fill(0).map((_, idx) => idx);
  }

  find(x) {
    if (x !== this.fa[x]) {
      return (this.fa[x] = this.find(this.fa[x]));
    }
    return x;
  }

  union(x, y) {
    const x_fa = this.find(x);
    const y_fa = this.find(y);
    if (x_fa !== y_fa) {
      this.fa[y_fa] = x_fa;
    }
  }
}
```

深度优先搜索DFS解法

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const lines = [];
let n, m;
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 1) {
    [n, m] = lines[0].split(" ").map(Number);
  }

  if (n && lines.length === n + 1) {
    lines.shift();
    const grid = lines.map((line) => line.split(" ").map(Number));

    console.log(countServers(grid));

    lines.length = 0;
  }
});

/**
 * @param {number[][]} grid
 * @return {number}
 */
var countServers = function (grid) {
  const n = grid.length;
  const m = grid[0].length;

  function dfs(i, j, count) {
    if (i < 0 || i >= n || j < 0 || j >= m || !grid[i][j]) return count;

    count++;
    grid[i][j] = 0;

    count = dfs(i - 1, j, count);
    count = dfs(i + 1, j, count);
    count = dfs(i, j - 1, count);
    count = dfs(i, j + 1, count);

    return count;
  }

```

```
  let max = 0;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < m; j++) {
      max = Math.max(max, dfs(i, j, 0));
    }
  }

  return max;
};
```

Java算法源码

深度优先搜索DFS解法

```
import java.util.Scanner;

public class Main {
  static int n;
  static int m;
  static int[][] matrix;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    n = sc.nextInt();
    m = sc.nextInt();

    matrix = new int[n][m];
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        matrix[i][j] = sc.nextInt();
      }
    }

    System.out.println(getResult());
  }

  public static int getResult() {
    int ans = 0;
    for (int i = 0; i < n; i++) {
```

```
      for (int j = 0; j < m; j++) {
        ans = Math.max(ans, dfs(i, j, 0));
      }
    }

    return ans;
  }

  public static int dfs(int i, int j, int count) {
    if (i < 0 || i >= n || j < 0 || j >= m || matrix[i][j] == 0) {
      return count;
    }

    count++;
    matrix[i][j] = 0;

    count = dfs(i - 1, j, count);
    count = dfs(i + 1, j, count);
    count = dfs(i, j - 1, count);
    count = dfs(i, j + 1, count);

    return count;
  }
}
```

Python算法源码

深度优先搜索DFS解法

```
# 输入获取
n, m = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(n)]


def dfs(i, j, count):
    if i < 0 or i >= n or j < 0 or j >= m or matrix[i][j] == 0:
        return count

    count += 1
    matrix[i][j] = 0

    count = dfs(i - 1, j, count)
    count = dfs(i + 1, j, count)
    count = dfs(i, j - 1, count)
    count = dfs(i, j + 1, count)

    return count


# 算法入口
def getResult():
```

```
    ans = 0

    for i in range(n):
        for j in range(m):
            ans = max(ans, dfs(i, j, 0))

    return ans


# 算法调用
print(getResult())
```