

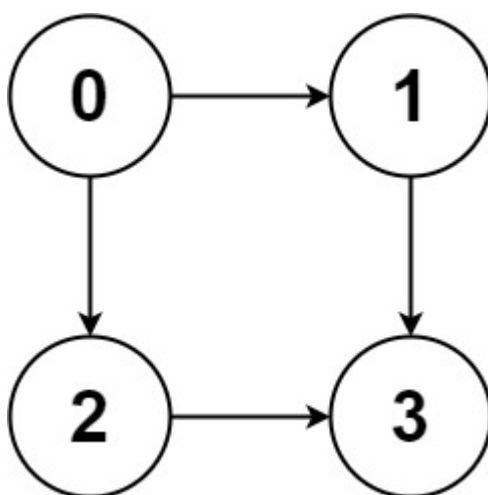
## 797 所有可能到达的路径

### 题目

给你一个有  $n$  个节点的 **有向无环图** (DAG)，请你找出所有从节点  $0$  到节点  $n-1$  的路径并输出（**不要求按特定顺序**）

$\text{graph}[i]$  是一个从节点  $i$  可以访问的所有节点的列表（即从节点  $i$  到节点  $\text{graph}[i][j]$  存在一条有向边）。

示例 1:

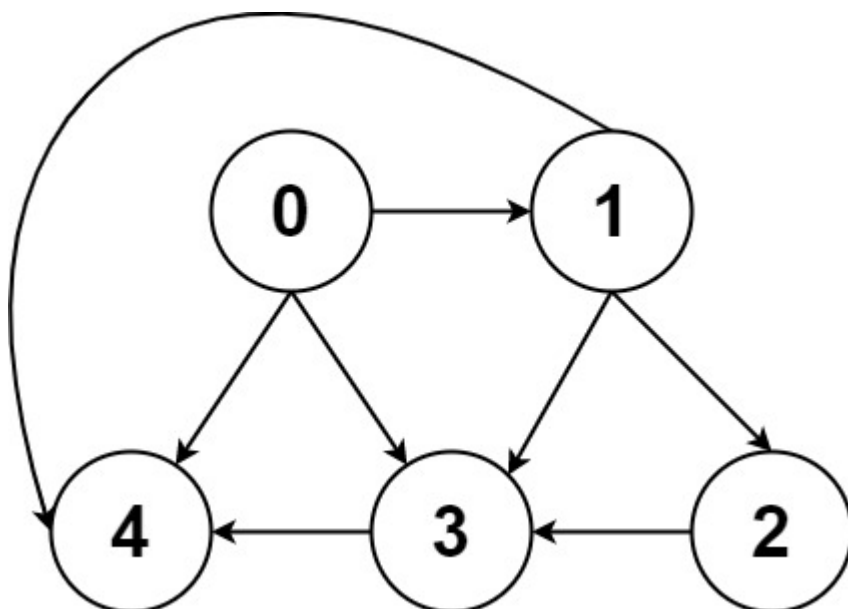


输入:  $\text{graph} = [[1,2],[3],[3],[]]$

输出:  $[[0,1,3],[0,2,3]]$

解释: 有两条路径  $0 \rightarrow 1 \rightarrow 3$  和  $0 \rightarrow 2 \rightarrow 3$

示例 2:



```
输入: graph = [[4,3,1],[3,2,4],[3],[4],[]]
输出: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]
```

提示:

- `n == graph.length`
- `2 <= n <= 15`
- `0 <= graph[i][j] < n`
- `graph[i][j] != i` (即不存在自环)
- `graph[i]` 中的所有元素 **互不相同**
- 保证输入为 **有向无环图 (DAG)**

## 题目大意

从节点0到节点n-1输出所有可以到达的路径, 注意给的图不一定是强连通图

## 解题思路

- 此题是dfs的模板题, 用来练手的, 一定要掌握

```
void dfs(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择: 本节点所连接的其他节点) {
        处理节点;
        dfs(图, 选择的节点); // 递归
        回溯, 撤销处理结果
    }
}
```

## 代码

```
/*
 * @Author: Jean_Leung
 * @Date: 2024-10-27 20:53:59
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-27 21:42:57
 * @FilePath: \code\graph_leetcode797.cpp
 * @Description:
 *
 */
```

```

* Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
*/

#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> result;
vector<int> path; // 用来存储经过的路径,只有满足一定条件才能放进res.push(path)
// graph输入的图
// x当前遍历的节点
// n图遍历的终点
void dfs(vector<vector<int>> &graph, int x, int n) {
    // 终止条件
    // 回顾回溯的框架
    // 当前遍历的节点如果为n的时候
    // 那么就到达终止条件
    if (x == n) {
        result.push_back(path);
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (graph[x][i] == 1) {
            // 找到x指向的节点,就是节点i
            path.push_back(i);
            dfs(graph, i, n);
            path.pop_back(); // 回溯
        }
    }
}

// acm模式
int main() {
    // n 代表有n个节点
    // m代表有m条有向边,
    // s代表 有向边起点
    // t代表 有向边终点
    int n, m, s, t;
    cin >> n >> m;
    // 初始化图,有n个节点,都初始化为,边都初始化为0
    vector<vector<int>> graph(n + 1, vector<int>(n + 1, 0));
    // 需要初始化边
    while (m--) {
        cin >> s >> t;
        graph[s][t] = 1;
    }
    // 默认从1开始
    path.push_back(1);
    dfs(graph, 1, n);
    // 输出
    if (result.size() == 0) {
        cout << -1 << endl;
    }
}

```

```

    }
    for (const vector<int> &pa : result) {
        for (int i = 0; i < pa.size() - 1; i++) {
            cout << pa[i] << " ";
        }
        cout << pa[pa.size() - 1] << endl;
    }
}

// leetcode提交
class Solution {
public:
    vector<vector<int>> result;
    vector<int> path;
    vector<vector<int>> allPathsSourceTarget(vector<vector<int>> &graph) {
        result.clear();
        if (graph.size() == 0) {
            return result;
        }
        path.push_back(0);
        dfs(graph, 0);
        return result;
    }

    // graph输入的图
    // x当前遍历的节点
    void dfs(vector<vector<int>> &graph, int x) {
        // 终止条件
        // 回顾回溯的框架
        // 当前遍历的节点如果为n的时候
        // 那么就到达终止条件
        if (x == graph.size() - 1) {
            result.push_back(path);
            return;
        }
        for (int i = 0; i < graph[x].size(); i++) {
            // 找到x指向的节点,就是节点i
            path.push_back(graph[x][i]);
            dfs(graph, graph[x][i]);
            path.pop_back(); // 回溯
        }
    }
};

```

## DFS和BFS模板

### DFS

---

## 路径模板(找寻路径问题)

## 染色问题模板(岛屿问题)

## BFS

---

## 染色问题模板(岛屿问题)

# 岛屿数量

## 题目

---

### 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，你需要计算岛屿的数量。岛屿由水平方向或垂直方向上相邻的陆地连接而成，并且四周都是水域。你可以假设矩阵外均被水包围。

### 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。

后续 N 行，每行包含 M 个数字，数字为 1 或者 0。

### 输出描述

输出一个整数，表示岛屿的数量。如果不存在岛屿，则输出 0。

### 输入示例

```
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

### 输出示例

```
3
```

### 提示信息

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

根据测试案例中所展示，岛屿数量共有 3 个，所以输出 3。

数据范围：

$1 \leq N, M \leq 50$

## 题目大意

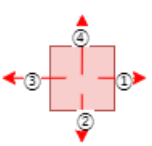
---



## 解题思路

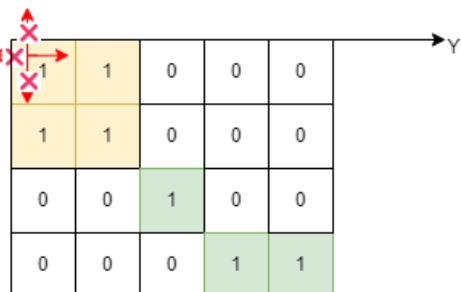
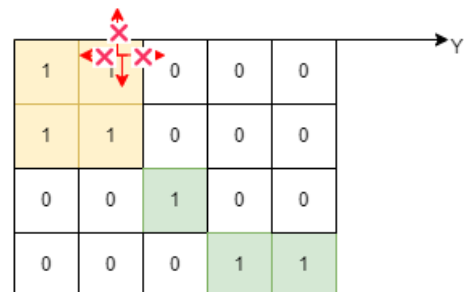
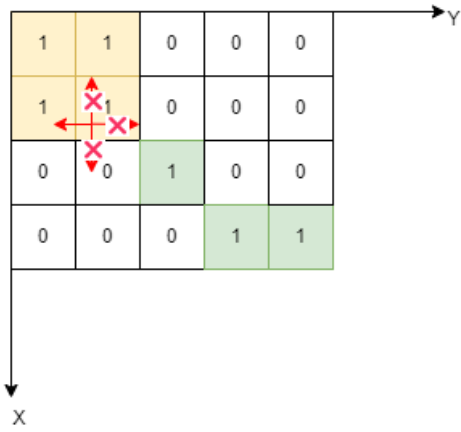
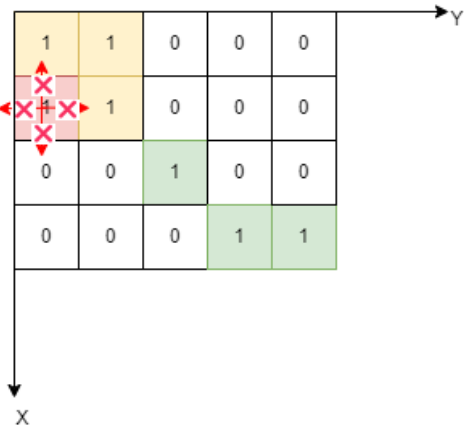
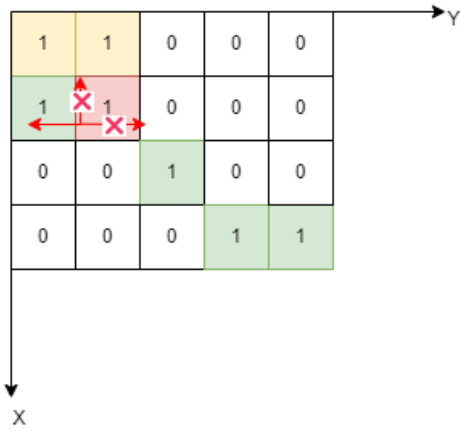
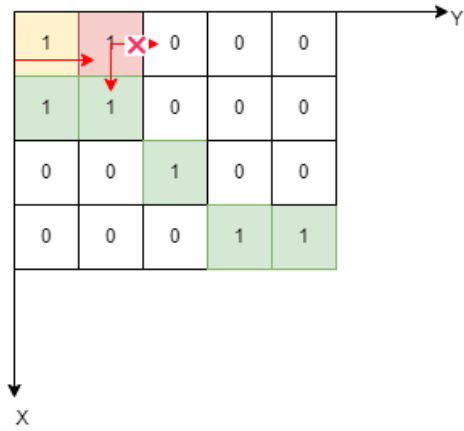
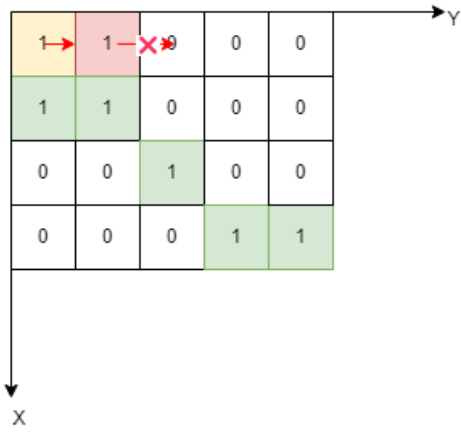
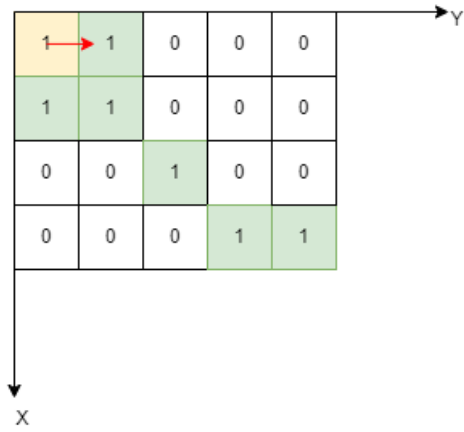
---

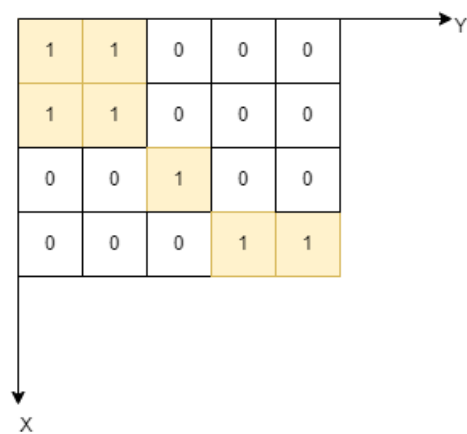
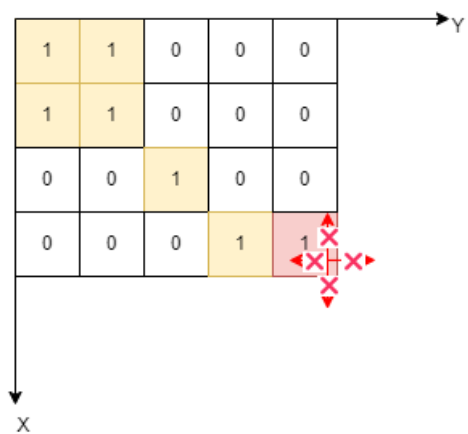
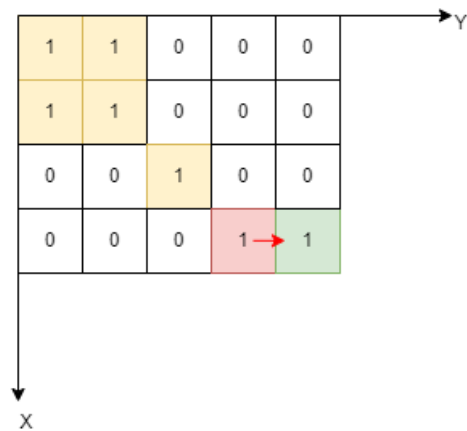
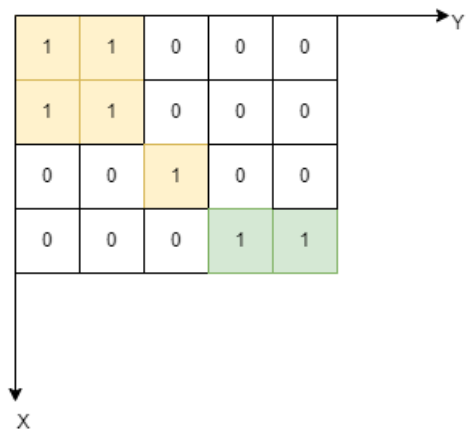
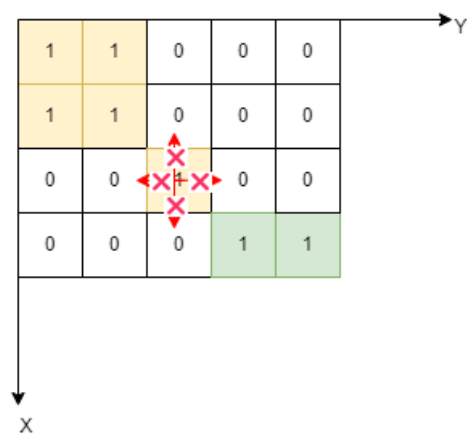
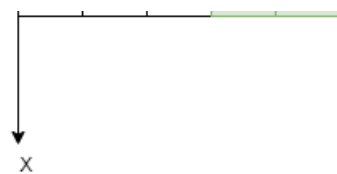
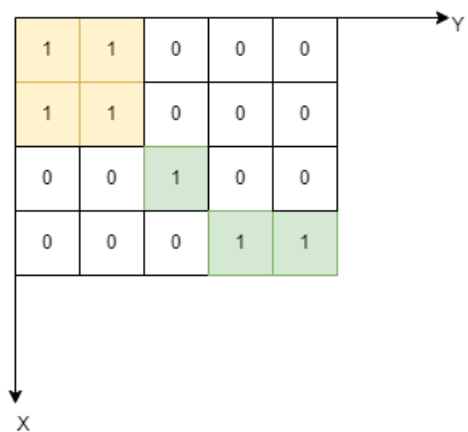
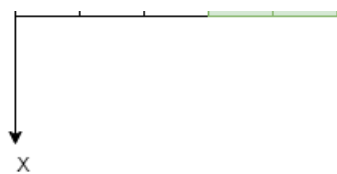
- DFS

```
int dir[4][2] = {0,1,1,0,-1,0,0,-1}
```



 :代表进入了DFS,且是当前层递归的节点  
 :代表是已经访问过的







# 代码

- DFS代码

```
// 版本一
#include <iostream>
#include <vector>
using namespace std;

// x方向, y方向
int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1}; // 代表遍历的四个方向

void dfs(const vector<vector<int>> &grid, vector<vector<bool>> &visited, int x,
        int y) {
    // x代表x方向, y代表y反向
    // 代表深搜的思路, 沿着给定初始x, y坐标的周围四个方向进行深搜
    for (int i = 0; i < 4; i++) {
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        // 越界处理
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
        // 如果下个位置为未访问且为陆地, 则继续深搜
        if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
            visited[next_x][next_y] = true;
            dfs(grid, visited, next_x, next_y);
        }
    }
}

int main() {
    int n, m;
    // n行数
    // m列数
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    int res = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j] && grid[i][j] == 1) {
                visited[i][j] = true;
                res++;
                dfs(grid, visited, i, j);
            }
        }
    }
}
```

```

    }
}
std::cout << res << std::endl;
}

```

- BFS代码

```

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};

void bfs(const vector<vector<int>> &grid, vector<vector<bool>> &visited, int x,
        int y) {
    // bfs遍历
    // 节点队列, 需要将图周围所有的节点加入队列中
    queue<pair<int, int>> que;
    que.push({x, y});
    visited[x][y] = true;
    while (!que.empty()) {
        pair<int, int> cur = que.front();
        que.pop();
        int cur_x = cur.first;
        int cur_y = cur.second;
        for (int i = 0; i < 4; i++) {
            int next_x = cur_x + dir[i][0];
            int next_y = cur_y + dir[i][1];
            if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
                next_y >= grid[0].size()) {
                continue;
            }
            if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
                que.push({next_x, next_y});
                visited[next_x][next_y] = true;
            }
        }
    }
}

int main() {
    int n, m;
    // n行数
    // m列数
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    vector<vector<bool>> visited(n, vector<bool>(m, false));
}

```

```
int res = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (!visited[i][j] && grid[i][j] == 1) {
            res++;
            bfs(grid, visited, i, j);
        }
    }
}
std::cout << res << std::endl;
}
```

# 岛屿最大面积

## 题目

### 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，计算岛屿的最大面积。岛屿面积的计算方式为组成岛屿的陆地的总数。岛屿由水平方向或垂直方向上相邻的陆地连接而成，并且四周都是水域。你可以假设矩阵外均被水包围。

### 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。后续 N 行，每行包含 M 个数字，数字为 1 或者 0，表示岛屿的单元格。

### 输出描述

输出一个整数，表示岛屿的最大面积。如果不存在岛屿，则输出 0。

### 输入示例

```
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

### 输出示例

```
4
```

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

样例输入中，岛屿的最大面积为 4。

数据范围：

$1 \leq M, N \leq 50$ 。

## 题目大意

---

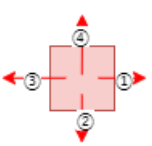
## 解题思路

---

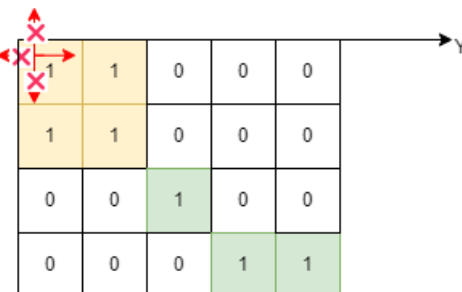
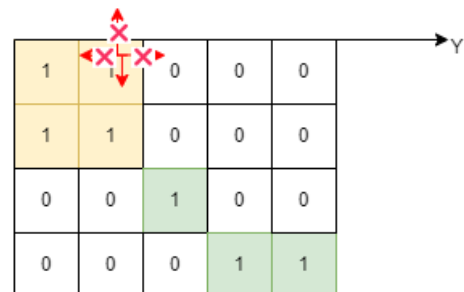
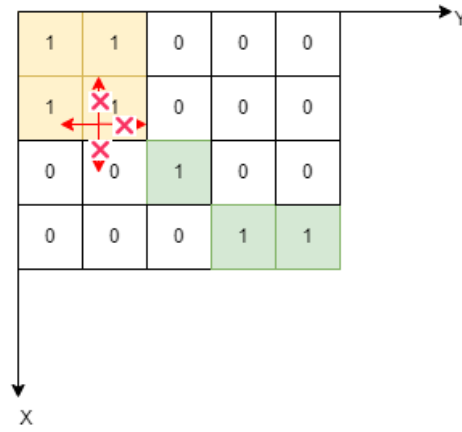
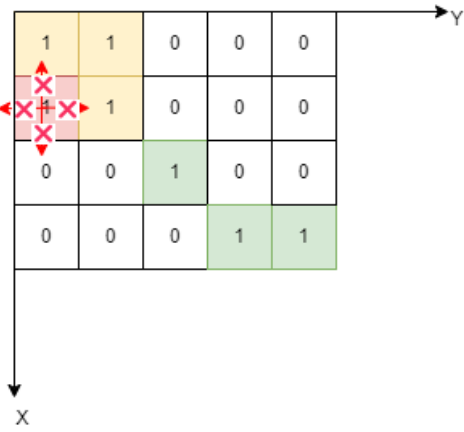
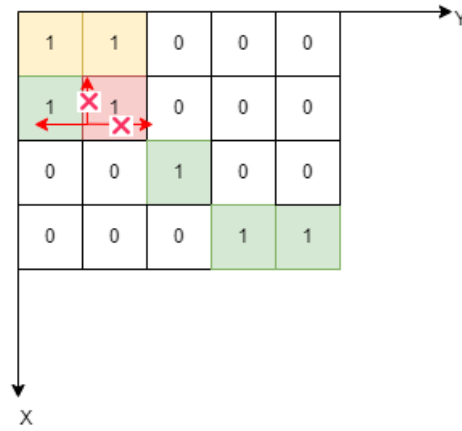
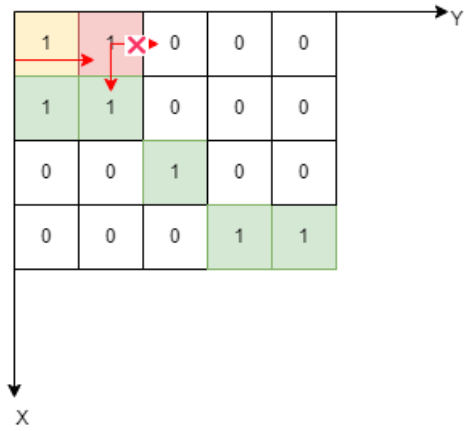
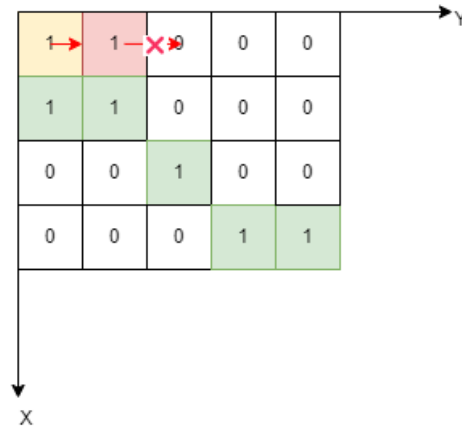
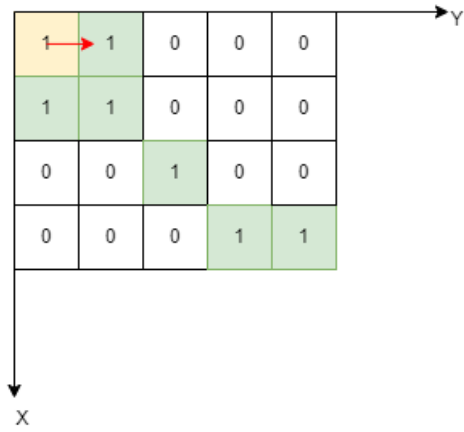
dfs或者BFS

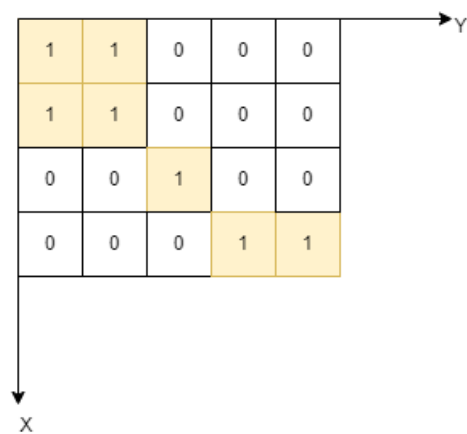
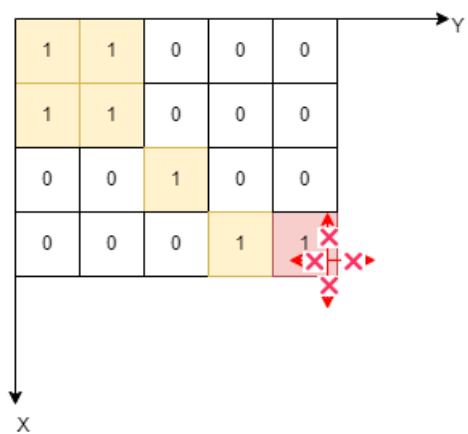
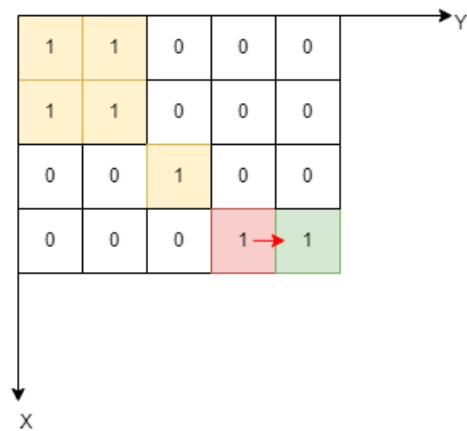
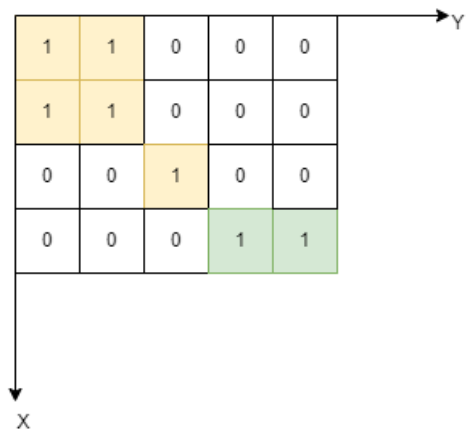
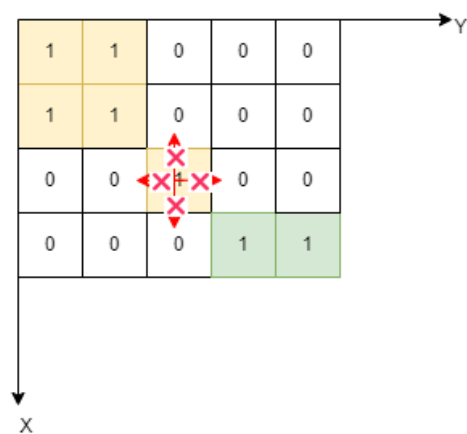
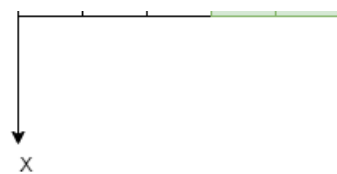
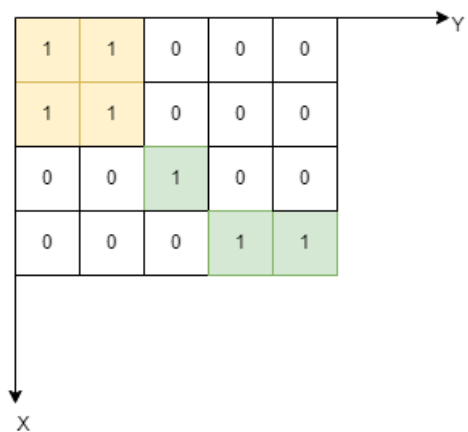
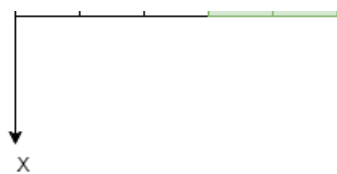
- DFS

```
int dir[4][2] = {0,1,1,0,-1,0,0,-1}
```



:代表进入了DFS,且是当前层递归的节点  
 :代表是已经访问过的





# 代码

- DFS

```
/*
 * @Author: Jean_Leung
 * @Date: 2024-10-28 14:17:57
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-28 14:41:25
 * @FilePath: \code\graph_kamacoding_dfs02.cpp
 * @Description:
 *
 * Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
 */
#include <iostream>
#include <vector>

using namespace std;

// 代表遍历方向
int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int area; // 岛屿面积
void dfs(const vector<vector<int>> &grid, vector<vector<bool>> visited, int x,
        int y) {
    for (int i = 0; i < 4; i++) {
        int next_x = grid[x][y] + dir[i][0];
        int next_y = grid[x][y] + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
        if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
            visited[next_x][next_y] = true;
            area++;
            dfs(grid, visited, next_x, next_y);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    int res = 0; // 答案

    vector<vector<bool>> visited(n, vector<bool>(m, false));
```

```

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            visited[i][j] = true;
            area = 1;
            dfs(grid, visited, i, j);
            res = max(area, res);
        }
    }
    std::cout << res << std::endl;
}

```

- BFS

```

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int area = 0; // 岛屿面积
void bfs(const vector<vector<int>> &grid, vector<vector<bool>> &visited, int x,
        int y) {
    // bfs遍历
    // 节点队列, 需要将图周围所有的节点加入队列中
    queue<pair<int, int>> que;
    que.push({x, y});
    visited[x][y] = true;
    while (!que.empty()) {
        pair<int, int> cur = que.front();
        que.pop();
        int cur_x = cur.first;
        int cur_y = cur.second;
        for (int i = 0; i < 4; i++) {
            int next_x = cur_x + dir[i][0];
            int next_y = cur_y + dir[i][1];
            if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
                next_y >= grid[0].size()) {
                continue;
            }
            if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
                area++;
                que.push({next_x, next_y});
                visited[next_x][next_y] = true;
            }
        }
    }
}

int main() {
    int n, m;
    // n行数
    // m列数
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
}

```



```

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    int res = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j] && grid[i][j] == 1) {
                area = 0;
                bfs(grid, visited, i, j);
                res = max(res, area);
            }
        }
    }
    std::cout << res << std::endl;
}

```

# 孤岛的总面积

## 题目

- 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，岛屿指的是由水平或垂直方向上相邻的陆地单元格组成的区域，且完全被水域单元格包围。孤岛是那些位于矩阵内部、所有单元格都不接触边缘的岛屿。

现在你需要计算所有孤岛的总面积，岛屿面积的计算方式为组成岛屿的陆地的总数。

- 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。之后 N 行，每行包含 M 个数字，数字为 1 或者 0。

- 输出描述

输出一个整数，表示所有孤岛的总面积，如果不存在孤岛，则输出 0。

- 输入示例

```

4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1

```

- 输出示例

```

1

```

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

在矩阵中心部分的岛屿，因为没有任何一个单元格接触到矩阵边缘，所以该岛屿属于孤岛，总面积为 1。

数据范围：

$1 \leq M, N \leq 50$

## 题目大意

找到上下左右都无水域的孤岛,然后统计该孤岛的最大面积

## 解题思路

注意题目中每座岛屿只能由**水平方向**和**或竖直方向**上相邻的陆地连接形成。

也就是说斜角度链接是不算了，例如示例二，是三个岛屿，如图：

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

["1","1","0","0","0"],  
["1","1","0","0","0"],  
["0","0","1","0","0"],  
["0","0","0","1","1"]



代码随想录

这道题目也是 dfs bfs基础类题目，就是搜索每个岛屿上“1”的数量，然后取一个最大的。

## 代码

- DFS

```
/*
 * @Author: Jean_Leung
 * @Date: 2024-10-28 14:17:57
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-28 14:41:25
 * @FilePath: \code\graph_kamacoding_dfs02.cpp
 * @Description:
 *
 * Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
 */
#include <iostream>
#include <vector>

using namespace std;

// 代表遍历方向
int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int area; // 岛屿面积
void dfs(const vector<vector<int>> &grid, vector<vector<bool>> visited, int x,
         int y) {
    for (int i = 0; i < 4; i++) {
        int next_x = grid[x][y] + dir[i][0];
        int next_y = grid[x][y] + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
    }
}
```

```

    }
    if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
        visited[next_x][next_y] = true;
        area++;
        dfs(grid, visited, next_x, next_y);
    }
}
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    int res = 0; // 答案

    vector<vector<bool>> visited(n, vector<bool>(m, false));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            visited[i][j] = true;
            area = 1;
            dfs(grid, visited, i, j);
            res = max(area, res);
        }
    }
    std::cout << res << std::endl;
}

```

- BFS

```

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int area = 0; // 岛屿面积
void bfs(const vector<vector<int>> &grid, vector<vector<bool>> &visited, int x,
        int y) {
    // bfs遍历
    // 节点队列, 需要将图周围所有的节点加入队列中
    queue<pair<int, int>> que;
    que.push({x, y});
    visited[x][y] = true;
    while (!que.empty()) {
        pair<int, int> cur = que.front();
        que.pop();
        int cur_x = cur.first;
        int cur_y = cur.second;
    }
}

```

```

        for (int i = 0; i < 4; i++) {
            int next_x = cur_x + dir[i][0];
            int next_y = cur_y + dir[i][1];
            if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
                next_y >= grid[0].size()) {
                continue;
            }
            if (!visited[next_x][next_y] && grid[next_x][next_y] == 1) {
                area++;
                que.push({next_x, next_y});
                visited[next_x][next_y] = true;
            }
        }
    }
}

int main() {
    int n, m;
    // n行数
    // m列数
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    int res = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j] && grid[i][j] == 1) {
                area = 0;
                bfs(grid, visited, i, j);
                res = max(res, area);
            }
        }
    }
    std::cout << res << std::endl;
}

```

# 沉没孤岛

## 题目

- 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，岛屿指的是由水平或垂直方向上相邻的陆地单元格组成的区域，且完全被水域单元格包围。孤岛是那些位于矩阵内部、所有单元格都不接触边缘的岛屿。

现在你需要将所有孤岛“沉没”，即将孤岛中的所有陆地单元格（1）转变为水域单元格（0）。

- 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。

之后 N 行，每行包含 M 个数字，数字为 1 或者 0，表示岛屿的单元格。

- 输出描述

输出将孤岛“沉没”之后的岛屿矩阵。注意：每个元素后面都有一个空格

- 输入示例

```
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

- 输出示例

```
1 1 0 0 0
1 1 0 0 0
0 0 0 0 0
0 0 0 1 1
```

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	0	1	1

数据范围：

$1 \leq M, N \leq 50$ 。

## 题目大意

需要将所有孤岛沉没

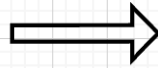
## 解题思路

步骤一：深搜或者广搜将地图周边的 1（陆地）全部改成 2（特殊标记）

步骤二：将水域中间 1（陆地）全部改成 水域（0）

步骤三：将之前标记的 2 改为 1（陆地）

0	1	0	0	0	0	0	0
1	1	1	0	0	0	1	1
0	1	1	1	0	1	1	1
0	0	0	0	1	0	0	0
0	1	0	0	1	0	0	0
0	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0

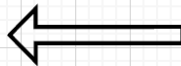


0	2	0	0	0	0	0	0
2	2	2	0	0	0	2	2
0	2	2	2	0	2	2	2
0	0	0	0	1	0	0	0
0	1	0	0	1	0	0	0
0	0	2	0	0	0	0	0
0	2	2	0	0	2	0	0



**D**  
代码随想录

0	1	0	0	0	0	0	0
1	1	1	0	0	0	1	1
0	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0



0	2	0	0	0	0	0	0
2	2	2	0	0	0	2	2
0	2	2	2	0	2	2	2
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	2	0	0	0	0	0
0	2	2	0	0	2	0	0

## 代码

```

/*
 * @Author: Jean_Leung
 * @Date: 2024-10-30 10:41:46
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-30 11:09:30
 * @FilePath: \code\graph_kamacoding_dfs03.cpp
 * @Description:
 *
 * Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
 */

#include <iostream>
#include <vector>

```



```

using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int count = 0; // 孤岛面积
// 需要将不靠近边界的陆地变成海洋
// 也就是grid[i][0] 和grid[i][grid[0].size()- 1]开始将陆地变成海洋,也就是两边
// 再从上下边将陆地变成海洋
void dfs(vector<vector<int>> &grid, int x, int y) {
    grid[x][y] = 0;
    count++;
    // 图论深搜
    for (int i = 0; i < 4; i++) {
        // 判断下一个位置
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        // 如果下一个访问过, 或者越界需要continue
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
        // 假如这里不是海洋, 那么就需要重新遍历
        if (grid[next_x][next_y] == 0) {
            continue;
        }
        // 否则直接dfs
        // 需要判断next_x和next_y不碰到边界
        // if (grid[next_x][next_y] == 1 && !visited[next_x][next_y]) {
        //     dfs(grid, visited, next_x, next_y);
        // }
        dfs(grid, next_x, next_y);
    }
}

int main() {
    // 行列数
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    // 需要先消除边的陆地, 将其变成海洋
    for (int i = 0; i < n; i++) {
        if (grid[i][0] == 1) {
            dfs(grid, i, 0);
        }
        if (grid[i][m - 1] == 1) {
            dfs(grid, i, m - 1);
        }
    }
}

```

```

// 上下边
for (int j = 0; j < m; j++) {
    if (grid[0][j] == 1) {
        dfs(grid, 0, j);
    }
    if (grid[n - 1][j] == 1) {
        dfs(grid, n - 1, j);
    }
}
count = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (grid[i][j] == 1) {
            dfs(grid, i, j);
        }
    }
}
std::cout << count << std::endl;
}

```

```

/*
 * @Author: Jean_Leung
 * @Date: 2024-10-30 11:30:52
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-30 11:30:59
 * @FilePath: \code\graph_kamacoding_dfs04.cpp
 * @Description:
 *
 * Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
 */

#include <iostream>
#include <vector>

using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};

void dfs(vector<vector<int>> &grid, int x, int y) {
    // 将靠边的陆地改为2
    grid[x][y] = 2;
    for (int i = 0; i < 4; i++) {
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
        if (grid[next_x][next_y] == 0 || grid[next_x][next_y] == 2) {
            continue;
        }
        dfs(grid, next_x, next_y);
    }
}

```

```

}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    for (int i = 0; i < n; i++) {
        if (grid[i][0] == 1) {
            dfs(grid, i, 0);
        }
        if (grid[i][m - 1] == 1) {
            dfs(grid, i, m - 1);
        }
    }
    for (int j = 0; j < m; j++) {
        if (grid[0][j] == 1) {
            dfs(grid, 0, j);
        }
        if (grid[n - 1][j] == 1) {
            dfs(grid, n - 1, j);
        }
    }
    // 将孤岛变成0, 将2变成1
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (grid[i][j] == 1) {
                grid[i][j] = 0;
            }
            if (grid[i][j] == 2) {
                grid[i][j] = 1;
            }
        }
    }
    cout << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
}

```

# 水流问题

## 题目

- 题目描述

现有一个  $N \times M$  的矩阵，每个单元格包含一个数值，这个数值代表该位置的相对高度。矩阵的左边界和上边界被认为是第一组边界，而矩阵的右边界和下边界被视为第二组边界。

矩阵模拟了一个地形，当雨水落在上面时，水会根据地形的倾斜向低处流动，但只能从较高或等高的地点流向较低或等高并且相邻（上下左右方向）的地点。我们的目标是确定那些单元格，从这些单元格出发的水可以达到第一组边界和第二组边界。

- 输入描述

第一行包含两个整数  $N$  和  $M$ ，分别表示矩阵的行数和列数。

后续  $N$  行，每行包含  $M$  个整数，表示矩阵中的每个单元格的高度。

- 输出描述

输出共有多行，每行输出两个整数，用一个空格隔开，表示可达第一组边界和第二组边界的单元格的坐标，输出顺序任意。

- 输入示例

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

- 输出示例

```
0 4
1 3
2 2
3 0
3 1
3 2
4 0
4 1
```

## 第一组边界

1	3	1	2	4
1	2	1	3	2
2	4	7	2	1
4	5	6	1	1
1	4	1	2	1

## 第二组边界

图中的蓝色方块上的雨水既能流向第一组边界，也能流向第二组边界。所以最终答案为所有蓝色方块的坐标。

数据范围：

$1 \leq M, N \leq 100$ 。

## 题目大意

找到图中所有的起点，水流从该起点出发的时候，能同时流到第一组边界和第二组边界

## 解题思路

### • 方法一

一个比较直白的想法，其实就是 遍历每个点，然后看这个点 能不能同时到达第一组边界和第二组边界。

至于遍历方式，可以用dfs，也可以用bfs，以下用dfs来举例。

这种思路很直白，但很明显，以上代码超时了。来看看时间复杂度。

遍历每一个节点，是  $m * n$ ，遍历每一个节点的时候，都要做深搜，深搜的时间复杂度是：  $m * n$

那么整体时间复杂度 就是  $O(m^2 * n^2)$ ，这是一个四次方的时间复杂度

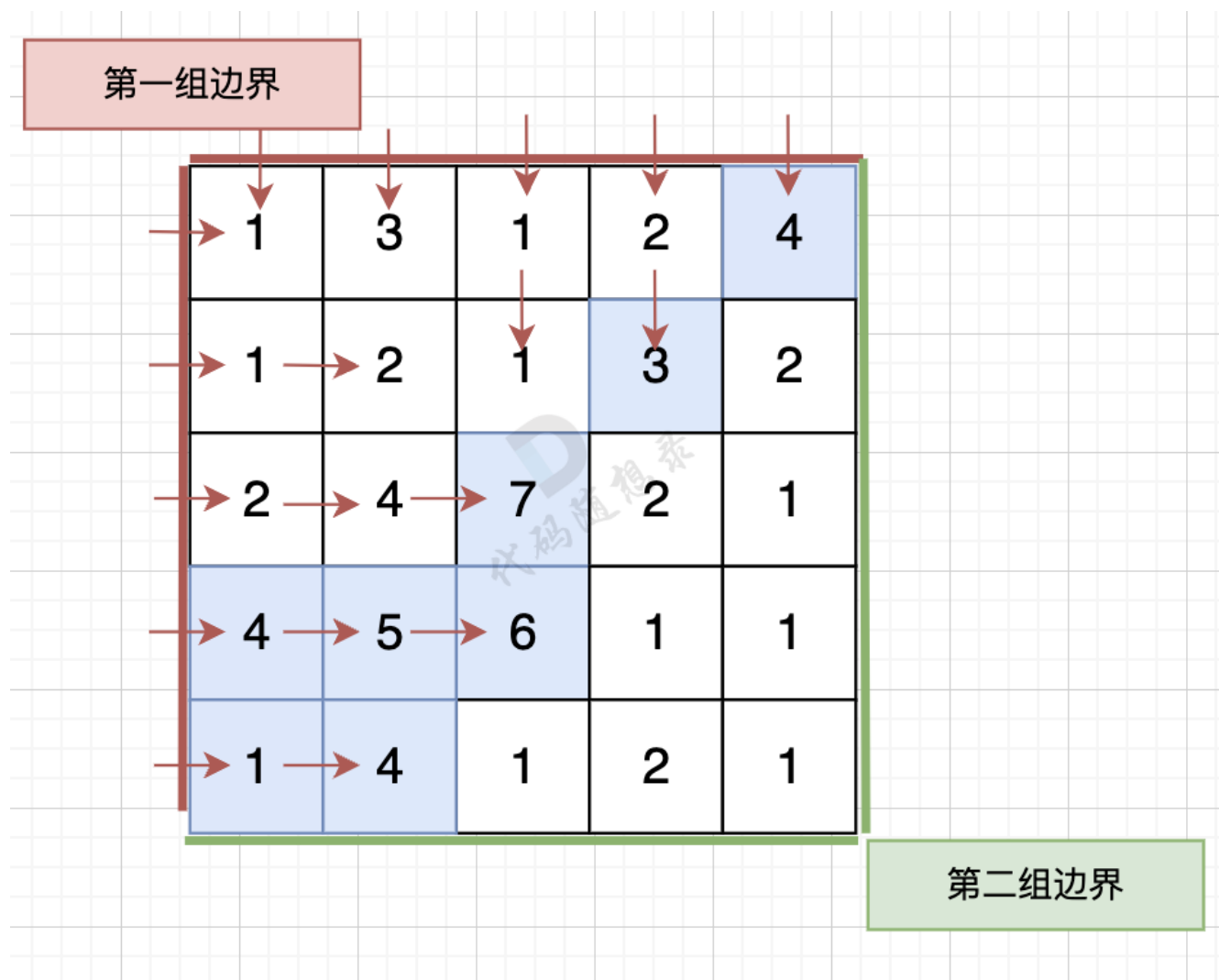
- 方法二

那么我们可以反过来想，从第一组边界上的节点 逆流而上，将遍历过的节点都标记上。

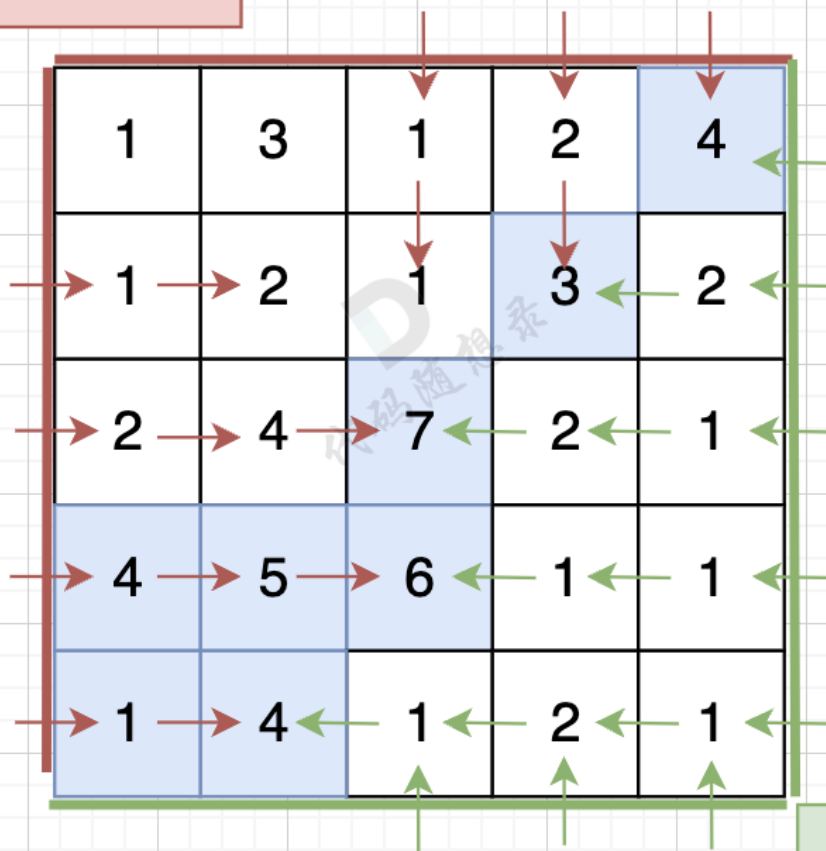
同样从第二组边界的边上节点 逆流而上，将遍历过的节点也标记上。

然后**两方都标记过的节点就是既可以流太平洋也可以流大西洋的节点。**

从第一组边界边上节点出发，如图：



### 第一组边界



### 第二组边界

能同时 逆行达到同一个位置，那么这个位置就是起点

## 代码

- DFS

```
/*
 * @Author: Jean_Leung
 * @Date: 2024-10-30 17:20:07
 * @LastEditors: Jean_Leung
 * @LastEditTime: 2024-10-30 17:57:58
 * @FilePath: \code\graph_kamacoding_water01.cpp
 * @Description:
 *
 * Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
 */

#include <iostream>
#include <vector>

using namespace std;
```

```

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};

void dfs(const vector<vector<int>> &grid, vector<vector<bool>> &visited, int x,
        int y) {
    // 如果当前遍历过
    if (visited[x][y]) {
        return;
    }
    visited[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[x].size()) {
            continue;
        }
        // 如果当前的{x,y}位置的值比{next_x,next_y}的值大, 或者相等
        // 就可以继续遍历
        if (grid[x][y] < grid[next_x][next_y]) {
            continue;
        }
        dfs(grid, visited, next_x, next_y);
    }
    return;
}

// 需要判断每个节点是否能到达第一边界和第二边界
bool isArrival(const vector<vector<int>> &grid, int x, int y) {
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    dfs(grid, visited, x, y);
    bool isFirst = false;
    bool isSecond = false;
    // 先判断第一边界的上界
    for (int j = 0; j < m; j++) {
        if (visited[0][j]) {
            isFirst = true;
            break;
        }
    }
    // 再判断第一边界的左界
    for (int i = 0; i < n; i++) {
        if (visited[i][0]) {
            isFirst = true;
            break;
        }
    }
    // 判断第二边界的右界
    for (int i = 0; i < n; i++) {
        if (visited[i][m - 1]) {
            isSecond = true;
        }
    }
}

```



```

        break;
    }
}
// 判断第二边界的下界
for (int j = 0; j < m; j++) {
    if (visited[n - 1][j]) {
        isSecond = true;
        break;
    }
}
return isFirst && isSecond;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    // vector<vector<bool>>> visited(n, vector<bool>(m, false));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (isArraival(grid, i, j)) {
                cout << i << " " << j << endl;
            }
        }
    }
}

```

- 逆流优化

```

#include <iostream>
#include <vector>

using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};

void dfs(const vector<vector<int>>> &grid, vector<vector<bool>>> &visited, int x,
        int y) {
    // 如果当前遍历过
    if (visited[x][y]) {
        return;
    }
    visited[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[x].size()) {
            continue;
        }
    }
}

```

```

    }
    // 如果当前的{x,y}位置的值比{next_x,next_y}的值大, 或者相等
    // 就可以继续遍历
    if (grid[x][y] > grid[next_x][next_y]) {
        continue;
    }
    dfs(grid, visited, next_x, next_y);
}
return;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    vector<vector<bool>> first_border(n, vector<bool>(m, false));
    vector<vector<bool>> second_border(n, vector<bool>(m, false));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    // 需要从左右边界开始遍历
    for (int i = 0; i < n; i++) {
        dfs(grid, first_border, i, 0);
        dfs(grid, second_border, i, m - 1);
    }
    // 从上下边界开始
    for (int j = 0; j < m; j++) {
        dfs(grid, first_border, 0, j);
        dfs(grid, second_border, n - 1, j);
    }
    // 只要同时满足first_border 和 second_border同时访问过
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (first_border[i][j] && second_border[i][j]) {
                std::cout << i << " " << j << std::endl;
            }
        }
    }
}

```

## 建造最大岛屿

### 题目

- 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，你最多可以将矩阵中的一格水变为一块陆地，在执行了此操作之后，矩阵中最大的岛屿面积是多少。

岛屿面积的计算方式为组成岛屿的陆地的总数。岛屿是被水包围，并且通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设矩阵外均被水包围。

- 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。之后 N 行，每行包含 M 个数字，数字为 1 或者 0，表示岛屿的单元格。

- 输出描述

输出一个整数，表示最大的岛屿面积。

- 输入示例

```
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

- 输出示例

```
6
```

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

1	1	0	0	0
1	1	0	0	0
0	1	1	0	0
0	0	0	1	1

1	1	0	0	0
1	1	1	0	0
0	0	1	0	0
0	0	0	1	1

数据范围：

$1 \leq M, N \leq 50$ 。

## 题目大意

找到一个水域，然后填海，将其岛屿连接的面积达到最大

## 解题思路

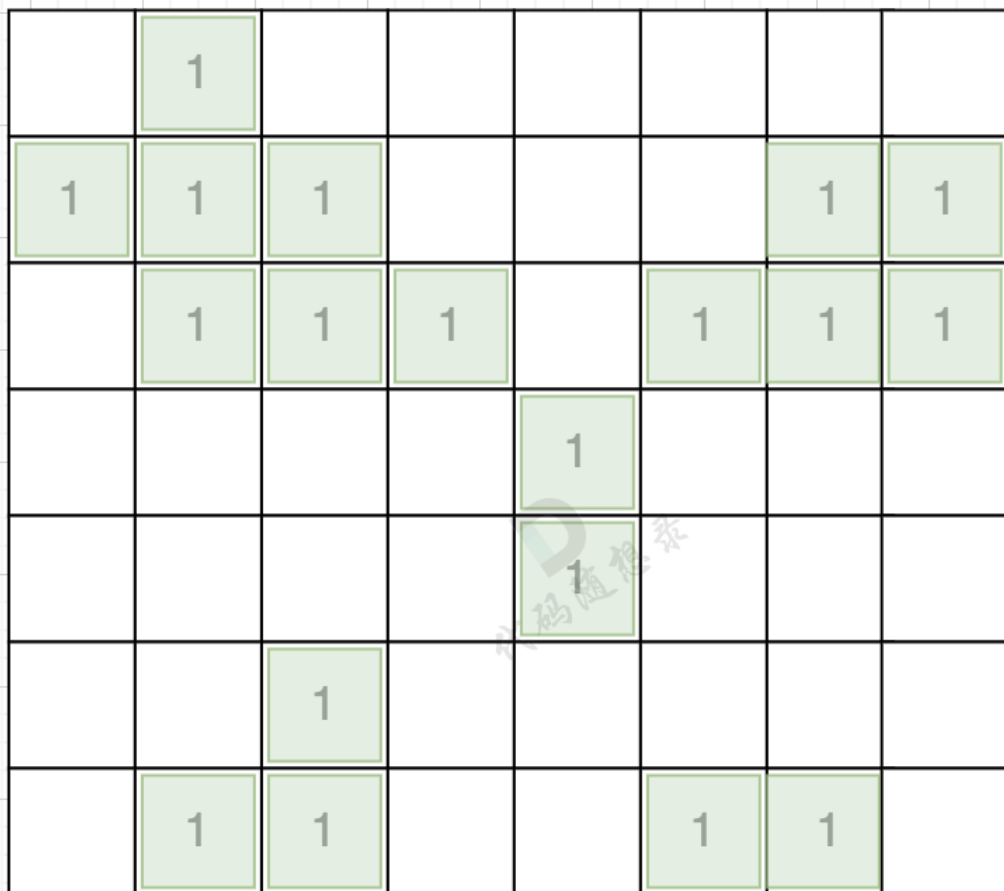
其实每次深搜遍历计算最大岛屿面积，我们都做了很多重复的工作。

只要用一次深搜把每个岛屿的面积记录下来就好。

第一步：一次遍历地图，得出各个岛屿的面积，并做编号记录。可以使用map记录，key为岛屿编号，value为岛屿面积

第二步：再遍历地图，遍历0的方格（因为要将0变成1），并统计该1（由0变成的1）周边岛屿面积，将其相邻面积相加在一起，遍历所有0之后，就可以得出选一个0变成1之后的最大面积。

拿如下地图的岛屿情况来举例：（1为陆地）



代码随想录

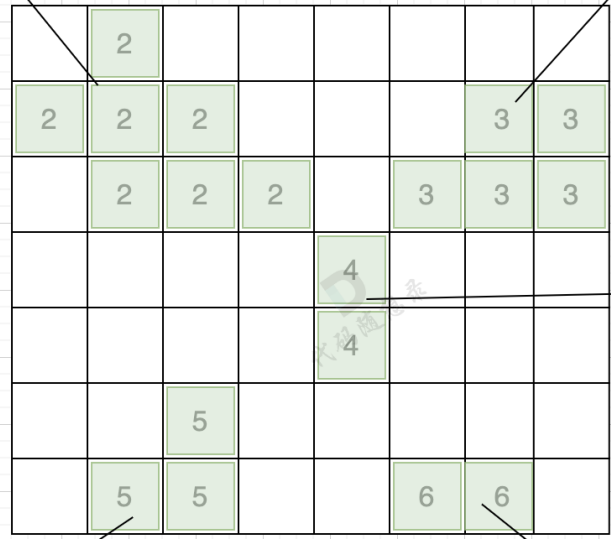
第一步，则遍历题目，并将岛屿到编号和面积上的统计，过程如图所示：

岛屿编号2，面积：7

岛屿编号3，面积：5



代码随想录



岛屿编号4，面积：2

岛屿编号5，面积：3

岛屿编号6，面积：2

这个过程时间复杂度  $n * n$ 。可能有录友想：分明是两个for循环下面套这一个dfs，时间复杂度怎么回事  $n * n$ 呢？

其实大家可以仔细看一下代码， $n * n$ 这个方格地图中，每个节点我们就遍历一次，并不会重复遍历。

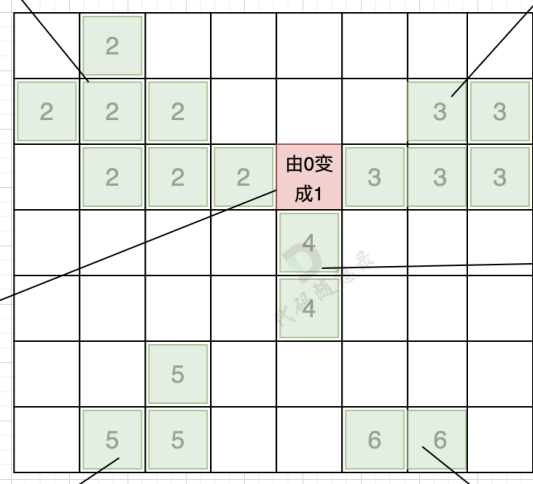
第二步过程如图所示：

岛屿编号2，面积：7

岛屿编号3，面积：5



代码随想录



岛屿编号4，面积：2

该格由0变1之后，  
岛屿面积为：  
岛屿编号2 + 岛屿编号  
3 + 岛屿编号4 + 1  
= 7 + 5 + 2 + 1 = 15

岛屿编号5，面积：3

岛屿编号6，面积：2

也就是遍历每一个0的方格，并统计其相邻岛屿面积，最后取一个最大值。

这个过程的时间复杂度也为  $n * n$ 。

所以整个解法的时间复杂度，为  $n * n + n * n$  也就是  $n^2$ 。

当然这里还有一个优化的点，就是 可以不用 visited数组，因为有mark来标记，所以遍历过的grid[i][j]是不等于1的。

## 代码

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
int count = 0;
void dfs(vector<vector<int>> &grid, vector<vector<bool>> &visited, int x, int y,
        int mark) {
    // 如果访问过就退出循环
    if (visited[x][y] || grid[x][y] == 0) {
        return;
    }
    // 将对应的岛屿标记编号
    grid[x][y] = mark;
    // 是否访问过
    visited[x][y] = true;
    count++;
    for (int i = 0; i < 4; i++) {
        int next_x = x + dir[i][0];
        int next_y = y + dir[i][1];
        if (next_x < 0 || next_x >= grid.size() || next_y < 0 ||
            next_y >= grid[0].size()) {
            continue;
        }
        // if (grid[next_x][next_y] == 0) {
        //     continue;
        // }
        dfs(grid, visited, next_x, next_y, mark);
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    int mark = 2; // 编号，从2号开始编号，因为题目的1代表是陆地，0是海洋
```

```

// {编号: 岛屿面积}
unordered_map<int, int> gridNum;
// 标记是否全是陆地
bool is_allisland = true;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (grid[i][j] == 0) {
            is_allisland = false;
        }
        if (!visited[i][j] && grid[i][j] == 1) {
            // 每次岛屿面积都要初始化为0
            count = 0;
            dfs(grid, visited, i, j, mark);
            gridNum[mark] = count;
            mark++;
        }
    }
}

// for (int i = 0; i < n; i++) {
//     for (int j = 0; j < m; j++) {
//         cout << grid[i][j] << " ";
//     }
//     cout << endl;
// }
// 如果全是陆地, 那么返回整块陆地面积
if (is_allisland) {
    cout << n * m << endl;
    return 0;
}
// 使用unordered_set
// 因为可以去重
// 表示访问过的岛屿
int result = 0;
unordered_set<int> visited_island;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        count = 1; // 表示填海的面积
        // 表示已经访问过
        visited_island.clear(); // 需要清空
        if (grid[i][j] == 0) {
            // 也要从该陆地四周开始寻找填的岛屿
            for (int k = 0; k < 4; k++) {
                int next_i = i + dir[k][0];
                int next_j = j + dir[k][1];
                if (next_i < 0 || next_i >= n || next_j < 0 ||
                    next_j >= m) {
                    continue;
                }
                // 去重重重复添加的岛屿
                // 如果找到重复的岛屿, 那么就要继续寻找
                if (visited_island.count(grid[next_i][next_j])) {
                    continue;
                }
            }
        }
    }
}

```



```

        }
        count += gridNum[grid[next_i][next_j]];
        // 将其添加到visited_island里面，表示已经连接的岛屿
        visited_island.insert(grid[next_i][next_j]);
    }
}
// 找到最大result
result = max(result, count);
}
}
cout << result << endl;
}

```

# 字符串接龙

## 题目

- 题目描述

字典 strList 中从字符串 beginStr 和 endStr 的转换序列是一个按下述规格形成的序列：

1. 序列中第一个字符串是 beginStr。
2. 序列中最后一个字符串是 endStr。
3. 每次转换只能改变一个字符。
4. 转换过程中的中间字符串必须是字典 strList 中的字符串，且strList里的每个字符串只用使用一次。

给你两个字符串 beginStr 和 endStr 和一个字典 strList，找到从 beginStr 到 endStr 的最短转换序列中的字符串数目。如果不存在这样的转换序列，返回 0。

- 输入描述

第一行包含一个整数 N，表示字典 strList 中的字符串数量。第二行包含两个字符串，用空格隔开，分别代表 beginStr 和 endStr。后续 N 行，每行一个字符串，代表 strList 中的字符串。

- 输出描述

第一行包含一个整数 N，表示字典 strList 中的字符串数量。第二行包含两个字符串，用空格隔开，分别代表 beginStr 和 endStr。后续 N 行，每行一个字符串，代表 strList 中的字符串。

- 输入示例

```

6
abc def
efc
dbc
ebc
dec
dfc
yhn

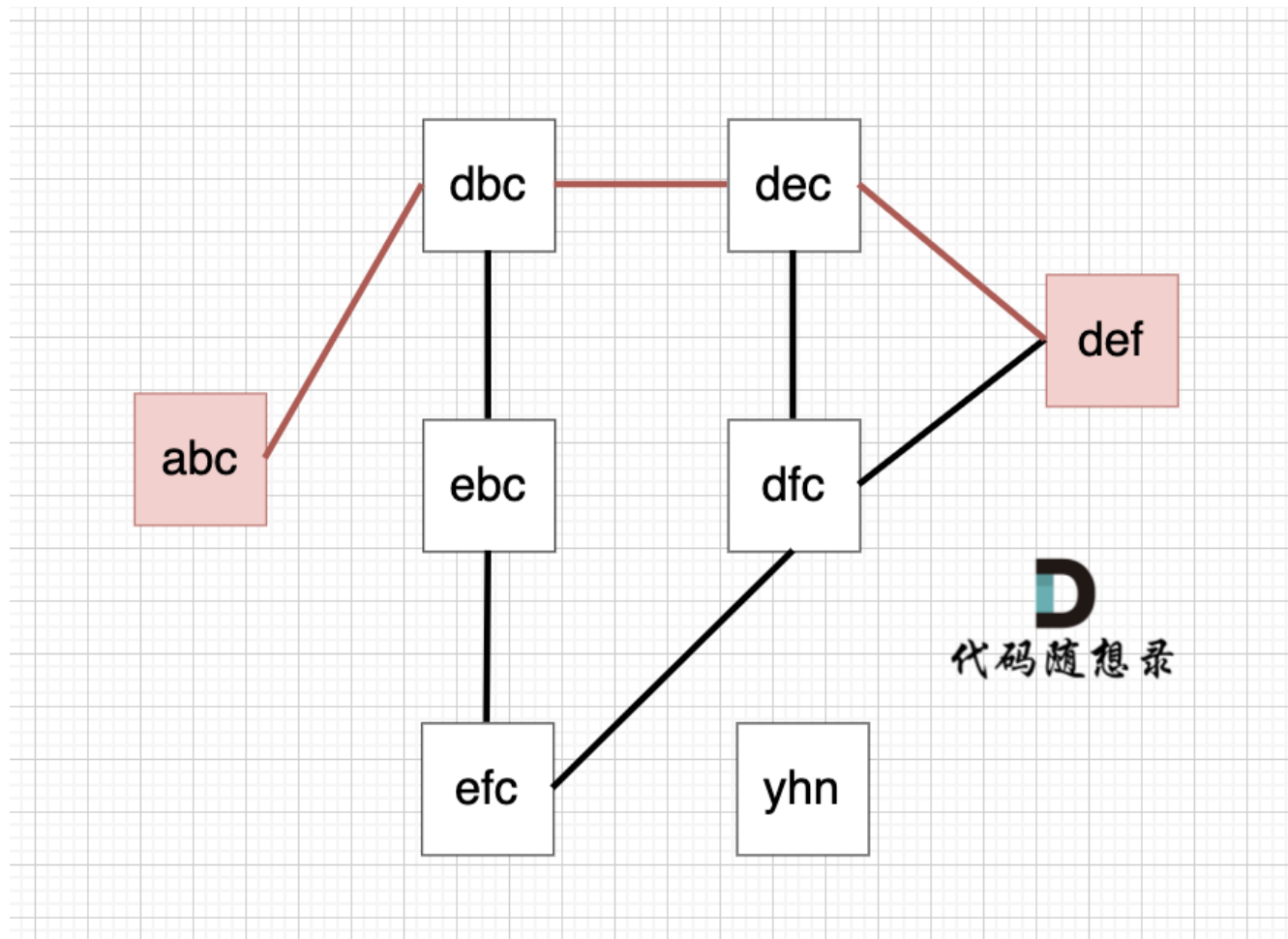
```

- 输出示例

4

- 提示信息

从 startStr 到 endStr，在 strList 中最短的路径为 abc -> dbc -> dec -> def，所以输出结果为 4，如图：



数据范围：

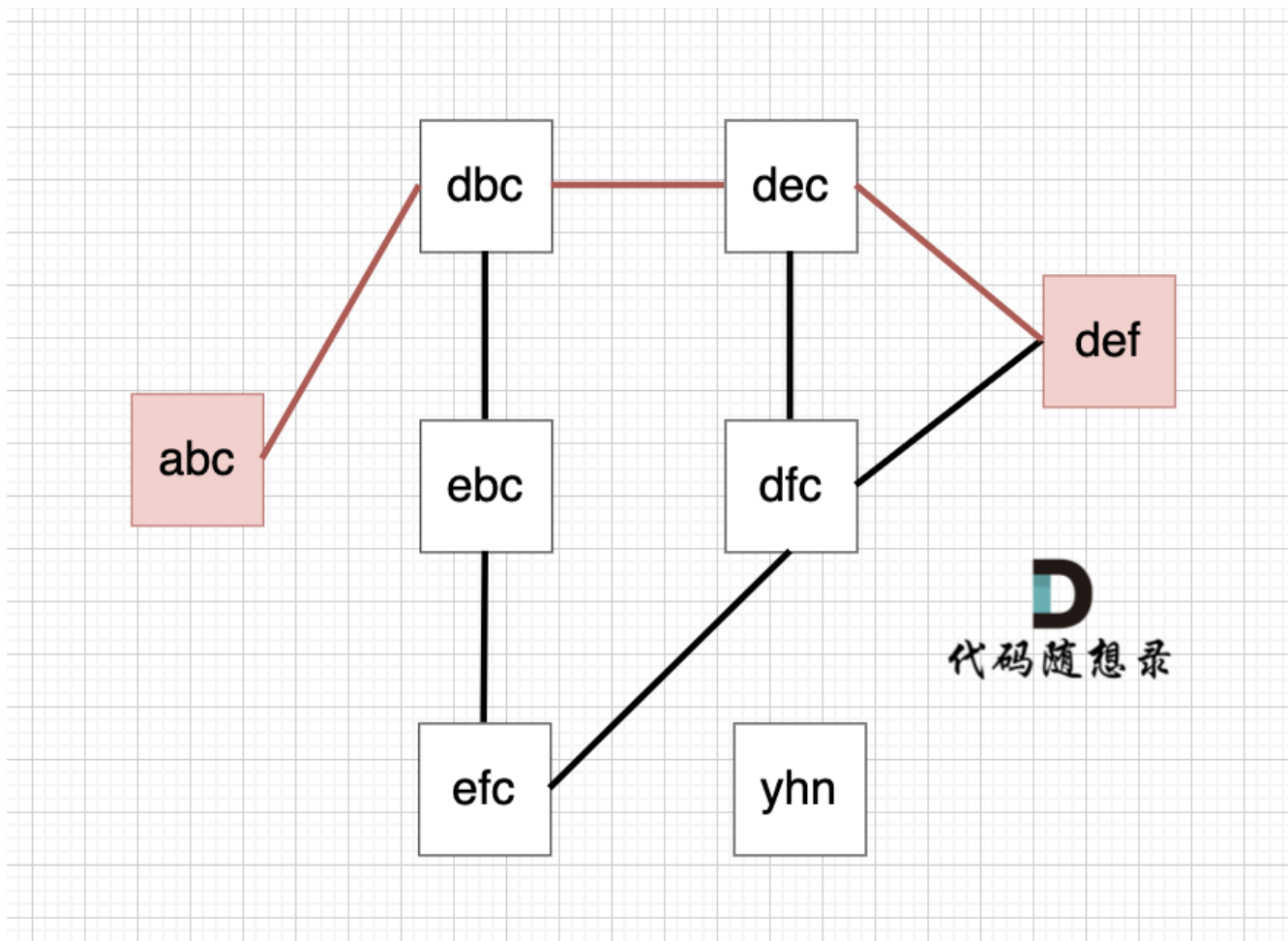
$2 \leq N \leq 500$

## 题目大意

给出start字符串，每一次只能变换一个字母，然后通过一群字符串中找到最短变换到end字符串的路径长度

## 解题思路

以示例1为例，从这个图中可以看出 abc 到 def的路线 不止一条，但最短的一条路径上是4个节点。



本题只要求出最短路径的长度就可以了，不用找出具体的路径。

所以这道题要解决两个问题：

- 图中的线是如何连在一起的
- 起点和终点的最短路径长度

首先题目中并没有给出点与点之间的连线，而是要我们自己去连，条件是字符只能差一个。

所以判断点与点之间的关系，需要判断是不是差一个字符，**如果差一个字符，那就是有链接。**

然后就是求起点和终点的最短路径长度，**这里无向图求最短路，广搜最为合适，广搜只要搜到了终点，那么一定是最短的路径。**因为广搜就是以起点中心向四周扩散的搜索。

**本题如果用深搜，会比较麻烦，要在到达终点的不同路径中选则一条最短路。**而广搜只要达到终点，一定是最短路。

另外需要有一个注意点：

- 本题是一个无向图，需要用标记位，标记着节点是否走过，否则就会死循环！
- 使用set来检查字符串是否出现在字符串集合里更快一些

## 代码

```
/*  
 * @Author: Jean_Leung
```

```

* @Date: 2024-10-30 20:15:48
* @LastEditors: Jean_Leung
* @LastEditTime: 2024-10-31 17:02:01
* @FilePath: \code\graph_kamacoding_string.cpp
* @Description:
*
* Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
*/

#include <iostream>
#include <queue>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;
int main() {
    // 需要根据输入的字符串进行无向图构造
    // 如果和当前字符串只差一个字符
    // 那么就需要将当前字符串与只差一个字符的字符串相连
    // 起始字符串, 结尾字符串, 字符串组
    string begin_str, end_str, str;
    int n;
    // 用unordered_set来储存, 达到去重的效果
    unordered_set<string> str_set;
    cin >> n;
    cin >> begin_str >> end_str;
    for (int i = 0; i < n; i++) {
        cin >> str;
        str_set.insert(str);
    }
    // {记录的字符串, 路径长度}
    // 记录strSet里的字符串是否被访问过, 同时记录路径长度
    unordered_map<string, int> visited_map;
    // 这道题适合用BFS来做, 因为无向图的最短路径适合用BFS
    // BFS是一圈一圈的进行搜索, 肯定是最短路径
    queue<string> que;
    que.push(begin_str);
    // 加入visited里面, 表示访问过
    visited_map.insert(pair<string, int>(begin_str, 1));
    while (!que.empty()) {
        string word = que.front();
        que.pop();
        int path = visited_map[word];
        // 要在这个字符连接的字符开始寻找
        // 开始在这个str中, 挨个字符去替换
        for (int i = 0; i < word.size(); i++) {
            // 用一个新字母去替代字母
            string new_word = word;
            for (int j = 0; j < 26; j++) {
                new_word[i] = j + 'a';
                if (new_word == end_str) {

```

```

        cout << path + 1 << endl;
        return 0;
    }
    // 字符串集合里出现new_word,且new_word没有访问过
    if (str_set.find(new_word) != str_set.end() &&
        visited_map.find(new_word) == visited_map.end()) {
        // 添加访问信息
        visited_map.insert(pair<string, int>(new_word, path + 1));
        que.push(new_word);
    }
}
}
}
cout << 0 << endl;
}

// 但是我认为可能更优解是利用这些字符串先构造无向图

```

## 有向图的完全可达性

### 题目

- 题目描述

给定一个有向图，包含  $N$  个节点，节点编号分别为  $1, 2, \dots, N$ 。现从 1 号节点开始，如果可以从 1 号节点的边可以到达任何节点，则输出 1，否则输出 -1。

- 输入描述

第一行包含两个正整数，表示节点数量  $N$  和边的数量  $K$ 。后续  $K$  行，每行两个正整数  $s$  和  $t$ ，表示从  $s$  节点有一条边单向连接到  $t$  节点。

- 输出描述

如果可以从 1 号节点的边可以到达任何节点，则输出 1，否则输出 -1。

- 输入示例

```

4 4
1 2
2 1
1 3
2 4

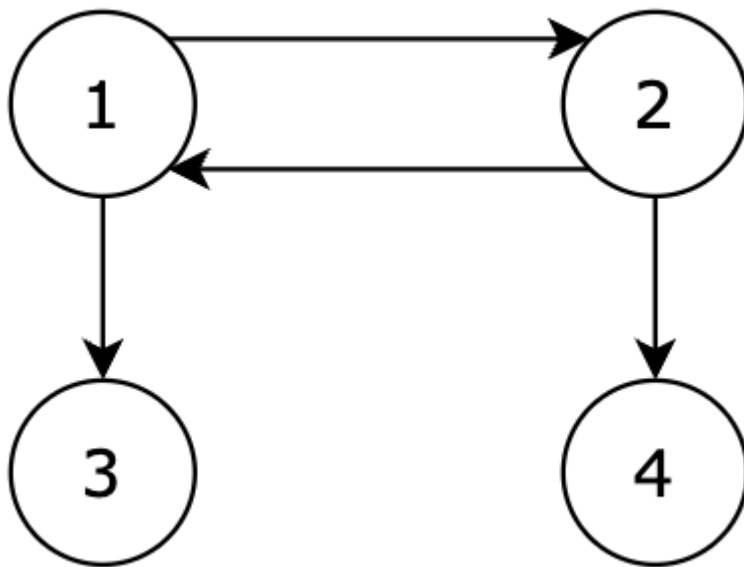
```

- 输出示例

```

1

```



从 1 号节点可以到达任意节点，输出 1。

**数据范围：**

$1 \leq N \leq 100$ ;  $1 \leq K \leq 2000$ 。

## 题目大意

如果1节点能到达任何节点，输出1，否则输出-1

## 解题思路

深搜三部曲：

### 1. 确认递归函数，参数

需要传入地图，需要知道当前我们拿到的key，以至于去下一个房间。

同时还需要一个数组，用来记录我们都走过了哪些房间，这样好知道最后有没有把所有房间都遍历的，可以定义一个一维数组。

### 2. 确认终止条件

遍历的时候，什么时候终止呢？

这里有一个很重要的逻辑，就是在递归中，**我们是处理当前访问的节点，还是处理下一个要访问的节点。**

这决定 终止条件怎么写。

首先明确，本题中什么叫做处理，就是 visited数组来记录访问过的节点，该节点默认 数组里元素都是false，把元素标记为true就是处理 本节点了。

如果我们是处理当前访问的节点，当前访问的节点如果是 true，说明是访问过的节点，那就终止本层递归，如果不是true，我们就把它赋值为true，因为这是我们处理本层递归的节点。

### 3. 处理目前搜索节点出发的路径

其实在上面，深搜三部曲 第二部，就已经讲了，因为终止条件的两种写法，直接决定了两种不一样的递归写法。

因为这道题不用进行路径添加，只需要标记就行，不需要进行回溯

## 代码

```
#include <iostream>
#include <list>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

// key代表当前遍历的节点
// visited代表是否有遍历过
void dfs(const vector<list<int>> &graph, int key, vector<bool> &visited) {
    // 这道题就是dfs的方法
    // 这道题是有向图，不是第一题的无向图
    // 回溯三部曲
    // 终止条件
    if (visited[key]) {
        return;
    }
    // 寻找节点
    visited[key] = true;
    // 需要用链表存储？
    // 这道题储存用的是邻接表，
    // 并不是用邻接矩阵
    // list<int> 是 C++ STL
    // 中的一个`双向链表容器`，适合于频繁的插入和删除操作。以下是 list<int>
    list<int> keys = graph[key];
    for (int key : keys) {
        // 继续深搜
        dfs(graph, key, visited);
    }
}

int main() {
    // n 代表点数
    // m代表边数
    // s代表开始指向方向
    int n, m, s, t;
    cin >> n >> m;
```

```

// 节点编号从1到n, 所以申请 n+1 这么大的数组
vector<list<int>> graph(n + 1);
// 先将vector的数组进行定义先
// 然后在初始化边表
while (m--) {
    cin >> s >> t;
    // 点表为s, s->t, 所以将t放置在边表中
    graph[s].push_back(t);
}
vector<bool> visited(n + 1, false);
dfs(graph, 1, visited);
// 检查是否都能走
for (int i = 1; i <= n; i++) {
    if (visited[i] == false) {
        cout << -1 << endl;
        return 0;
    }
}
cout << 1 << endl;
}

```

# 岛屿的周长

## 题目

- 题目描述

给定一个由 1（陆地）和 0（水）组成的矩阵，岛屿是被水包围，并且通过水平方向或垂直方向上相邻的陆地连接而成的。

你可以假设矩阵外均被水包围。在矩阵中恰好拥有一个岛屿，假设组成岛屿的陆地边长都为 1，请计算岛屿的周长。岛屿内部没有水域。

- 输入描述

第一行包含两个整数 N, M，表示矩阵的行数和列数。之后 N 行，每行包含 M 个数字，数字为 1 或者 0，表示岛屿的单元格。

- 输出描述

输出一个整数，表示岛屿的周长。

- 输入示例

```

5 5
0 0 0 0 0
0 1 0 1 0
0 1 1 1 0
0 1 1 1 0
0 0 0 0 0

```

- 输出示例



0	0	0	0	0
0	1	0	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

岛屿的周长为 14。

数据范围：

$1 \leq M, N \leq 50$ 。

## 题目大意

就是求出岛屿的周长

## 解题思路

- 解法一

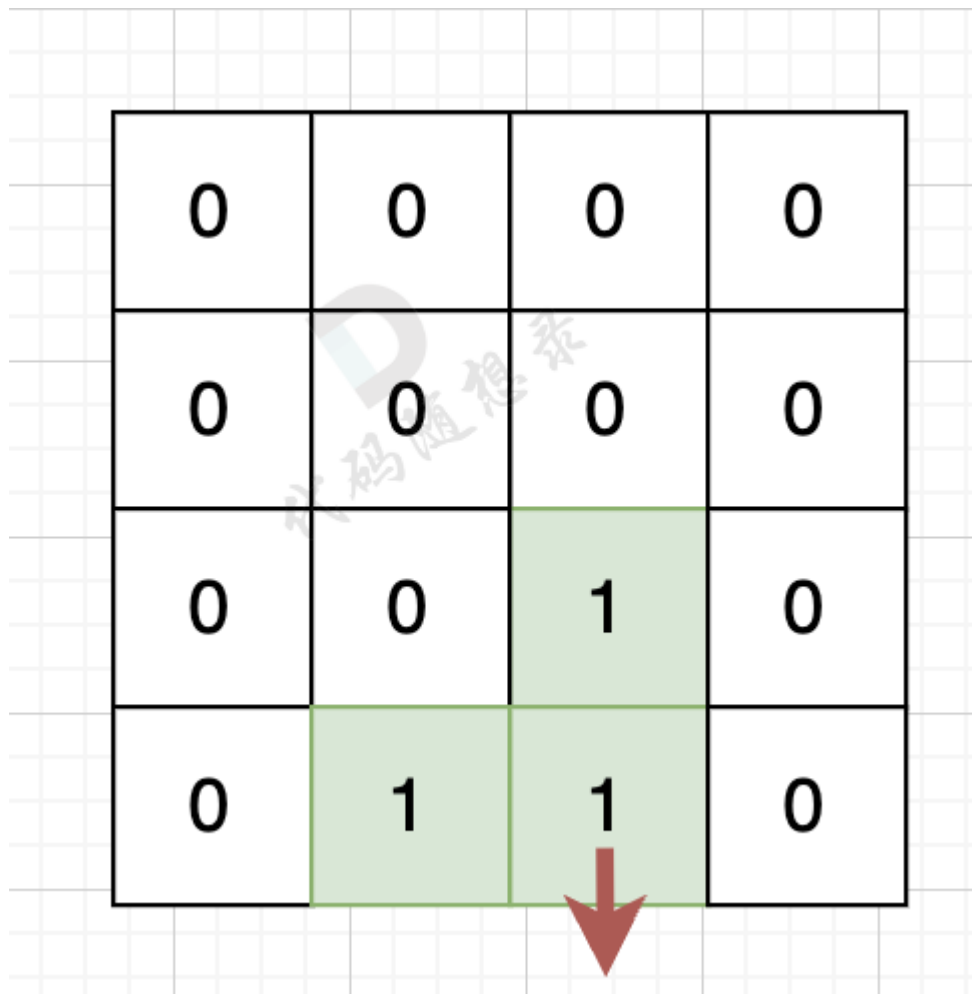
遍历每一个空格，遇到岛屿则计算其上下左右的空格情况。

如果该陆地上下左右的空格是有水域，则说明是一条边，如图：

0	0	0	0
0	0	0	0
0	0	1	0
0	1	1	0

陆地的右边空格是水域，则说明找到一条边。

如果该陆地上下左右的空格出界了，则说明是一条边，如图：



该陆地的下边空格出界了，则说明找到一条边。

因为越界就说明这个边是靠边的，肯定有一条边

## 代码

```
#include <iostream>
#include <vector>

using namespace std;

int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> grid[i][j];
        }
    }
    // 代表岛屿周长
```

```
int result = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (grid[i][j] == 1) {
            for (int k = 0; k < 4; k++) {
                int x = i + dir[k][0];
                int y = j + dir[k][1];
                if (x < 0 || x >= grid.size() || y < 0 ||
                    y >= grid[0].size() || grid[x][y]) {
                    result++;
                }
            }
        }
    }
}
cout << result << endl;
}
```

## 并查集理论基础