剑指offer(14)——彻底解决搜索问题

题目695:

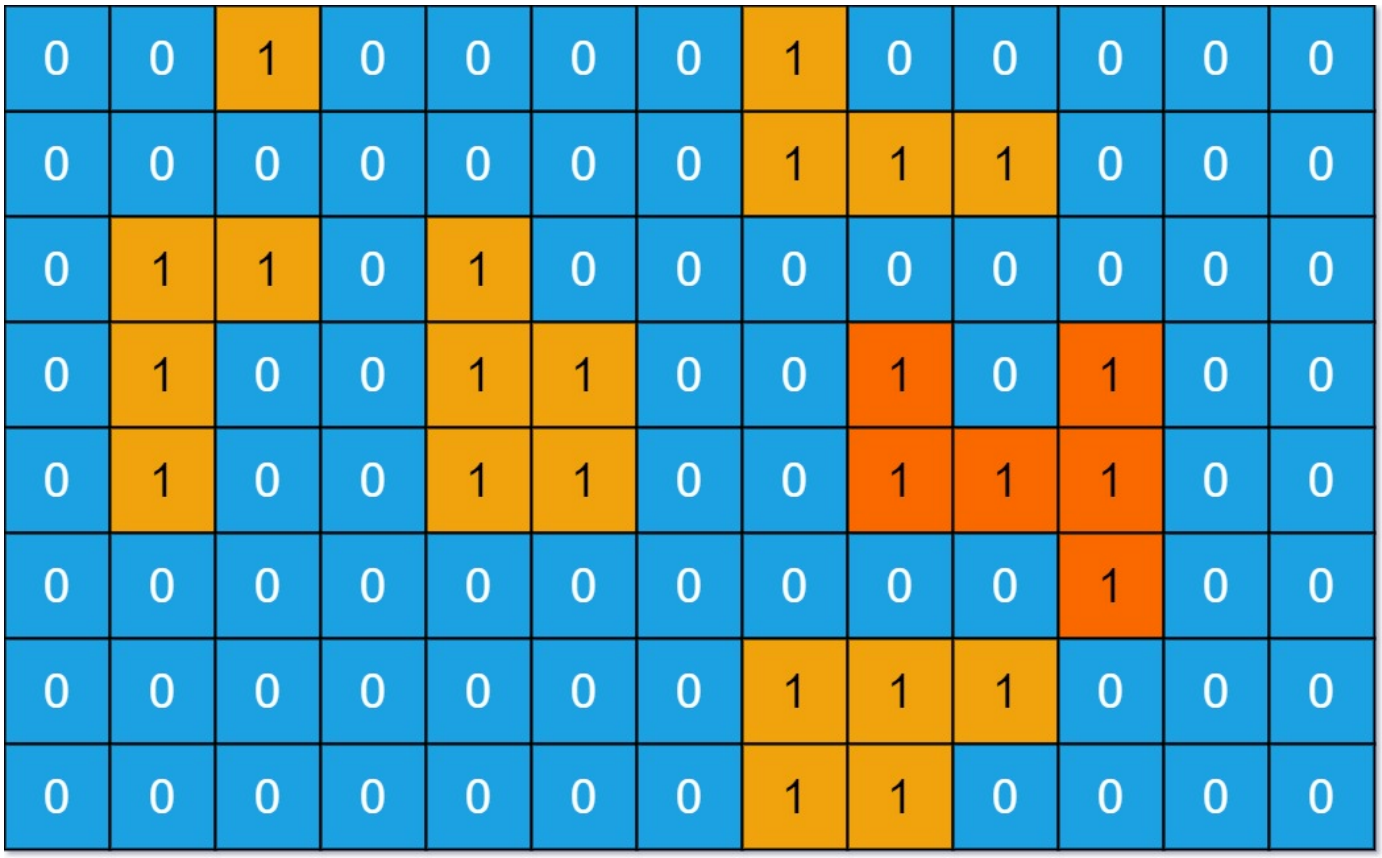
给你一个大小为 `m x n` 的二进制矩阵 `grid` 。

岛屿 是由一些相邻的 `1` (代表土地) 构成的组合，这里的「相邻」要求两个 `1` 必须在 **水平或者竖直的四个方向上** 相邻。你可以假设 `grid` 的四个边缘都被 `0` （代表水）包围着。

岛屿的面积是岛上值为 `1` 的单元格的数目。

计算并返回 `grid` 中最大的岛屿面积。如果没有岛屿，则返回面积为 `0` 。

示例 1：




```
输入：grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],
[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],
[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]
输出：6
解释：答案不应该是 11 ，因为岛屿只能包含水平或垂直这四个方向上的 1 。
```

示例 2：

```
输入：grid = [[0,0,0,0,0,0,0,0]]
输出：0
```

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 50`
- `grid[i][j]` 为 `0` 或 `1`

解法1:深度优先搜索

算法:

我们想知道网格中每个连通形状的面积，然后取最大值。

如果我们在一个土地上，以 444 个方向探索与之相连的每一个土地（以及与这些土地相连的土地），那么探索过的土地总数将是该连通形状的面积。

为了确保每个土地访问不超过一次，我们每次经过一块土地时，将这块土地的值置为 000。这样我们就不会多次访问同一土地。

```
class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int result = 0;
        for(int i = 0;i < grid.size(); i++){
            for(int j = 0;j < grid[0].size();j++){
                result = max(result, dfs(grid,i,j));
            }
        }
        return result;
    }

    int dfs(vector<vector<int>>& grid, int x, int y){
        if(x < 0 || y < 0 || x >= grid.size() || y >= grid[0].size() || grid[x][y] == 0){
            return 0;
        }
        grid[x][y] = 0;
        int cnt = 1;
        int pos_x[4] = {0,0,1,-1};
        int pos_y[4] = {1,-1,0,0};
        for(int i = 0;i < 4;i++){
            int cur_x = x + pos_x[i];
            int cur_y = y + pos_y[i];
            cnt += dfs(grid,cur_x,cur_y);
        }
        return cnt;
    }
};
```

这边感觉需要注意的就是那个max的使用，有时候会一下子想不上来

解法2:深度优先搜索 + 栈

算法:

方法一通过函数的调用来表示接下来想要遍历哪些土地，让下一层函数来访问这些土地。而方法二把接下来想要遍历的土地放在栈里，然后在取出这些土地的时候访问它们。

访问每一片土地时，我们将对围绕它四个方向进行探索，找到还未访问的土地，加入到栈 stack\textit{stack}stack 中；

另外，只要栈 stack\textit{stack}stack 不为空，就说明我们还有土地待访问，那么就从栈中取出一个元素并访问。

```
class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int ans = 0;
        for (int i = 0; i != grid.size(); ++i) {
            for (int j = 0; j != grid[0].size(); ++j) {
                int cur = 0;
                stack<int> stacki;
                stack<int> stackj;
                stacki.push(i);
                stackj.push(j);
                while (!stacki.empty()) {
                    int cur_i = stacki.top(), cur_j = stackj.top();
                    stacki.pop();
                    stackj.pop();
                    if (cur_i < 0 || cur_j < 0 || cur_i == grid.size() || cur_j ==
grid[0].size() || grid[cur_i][cur_j] != 1) {
                        continue;
                    }
                    ++cur;
                    grid[cur_i][cur_j] = 0;
                    int di[4] = {0, 0, 1, -1};
                    int dj[4] = {1, -1, 0, 0};
                    for (int index = 0; index != 4; ++index) {
                        int next_i = cur_i + di[index], next_j = cur_j + dj[index];
                        stacki.push(next_i);
                        stackj.push(next_j);
                    }
                }
                ans = max(ans, cur);
            }
        }
        return ans;
    }
};
```

解法3: 广度优先搜索

我们把方法二中的栈改为队列，每次从队首取出土地，并将接下来想要遍历的土地放在队尾，就实现了广度优先搜索算法。

```
class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int ans = 0;
        for (int i = 0; i != grid.size(); ++i) {
            for (int j = 0; j != grid[0].size(); ++j) {
                int cur = 0;
                queue<int> queuei;
                queue<int> queuej;
                queuei.push(i);
                queuej.push(j);
                while (!queuei.empty()) {
                    int cur_i = queuei.front(), cur_j = queuej.front();
                    queuei.pop();
                    queuej.pop();
                    if (cur_i < 0 || cur_j < 0 || cur_i == grid.size() || cur_j ==
grid[0].size() || grid[cur_i][cur_j] != 1) {
                        continue;
                    }
                    ++cur;
                    grid[cur_i][cur_j] = 0;
                    int di[4] = {0, 0, 1, -1};
                    int dj[4] = {1, -1, 0, 0};
                    for (int index = 0; index != 4; ++index) {
                        int next_i = cur_i + di[index], next_j = cur_j + dj[index];
                        queuei.push(next_i);
                        queuej.push(next_j);
                    }
                }
                ans = max(ans, cur);
            }
        }
        return ans;
    }
};
```

解法4: 极其简单的操作

```
class Solution {
    public int maxAreaOfIsland(int[][] grid) {
        int ans = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                // 发现陆地!
                if (grid[i][j] == 1) {
```

```
                    ans = Math.max(ans, dfs(grid, i, j));
                }
            }
        }
        return ans;
    }

    private int dfs(int[][] grid, int m, int n) {
        if (!(m >= 0 && m < grid.length && n >= 0 && n < grid[0].length)) return 0;
        // 这是海域或者是到达过的地方，不作数啦
        if (grid[m][n] == 0 || grid[m][n] == 2) return 0;
        grid[m][n] = 2;
        return 1 + dfs(grid, m + 1, n) +
                dfs(grid, m - 1, n) +
                dfs(grid, m, n + 1) +
                dfs(grid, m, n - 1);
    }
}
```

题目200: 岛屿数量

给你一个由 `'1'`（陆地）和 `'0'`（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且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。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

```
输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

示例 2：

```
输入：grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
输出：3
```

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` 的值为 `'0'` 或 `'1'`

我的解法：

就是dfs，然后统计个数。这边需要注意的就是每个块的表现形式，这边用的是char

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int result = 0;
        for(int i = 0;i < grid.size();i++){
            for(int j = 0;j < grid[0].size();j++){
                if(dfs(grid,i,j) > 0){
                    result ++;
                }
            }
        }
        return result;
    }

    int dfs(vector<vector<char>>& grid, int x,int y){
        if(!(x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size())){
            return 0;
        }
        if(grid[x][y] == '0' || grid[x][y] == '2'){
            return 0;
        }
        grid[x][y] = '2';
        return 1 + dfs(grid, x + 1, y) + dfs(grid, x - 1, y) + dfs(grid, x, y + 1) +
dfs(grid, x, y-1);
    }
};
```

解法2：广度优先搜索

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int nr = grid.size();
        if (!nr) return 0;
        int nc = grid[0].size();

        int num_islands = 0;
        for (int r = 0; r < nr; ++r) {
            for (int c = 0; c < nc; ++c) {
                if (grid[r][c] == '1') {
```

```
                    ++num_islands;
                    grid[r][c] = '0';
                    queue<pair<int, int>> neighbors;
                    neighbors.push({r, c});
                    while (!neighbors.empty()) {
                        auto rc = neighbors.front();
                        neighbors.pop();
                        int row = rc.first, col = rc.second;
                        if (row - 1 >= 0 && grid[row-1][col] == '1') {
                            neighbors.push({row-1, col});
                            grid[row-1][col] = '0';
                        }
                        if (row + 1 < nr && grid[row+1][col] == '1') {
                            neighbors.push({row+1, col});
                            grid[row+1][col] = '0';
                        }
                        if (col - 1 >= 0 && grid[row][col-1] == '1') {
                            neighbors.push({row, col-1});
                            grid[row][col-1] = '0';
                        }
                        if (col + 1 < nc && grid[row][col+1] == '1') {
                            neighbors.push({row, col+1});
                            grid[row][col+1] = '0';
                        }
                    }
                }
            }
        }

        return num_islands;
    }
};
```

解法3： 并查集

// todo：并查集还没有理解

算法：

为了求出岛屿的数量，我们可以扫描整个二维网格。如果一个位置为 111，则将其与相邻四个方向上的 111 在并查集中进行合并。

最终岛屿的数量就是并查集中连通分量的数目。

```
class UnionFind {
public:
    UnionFind(vector<vector<char>>& grid) {
        count = 0;
        int m = grid.size();
        int n = grid[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
```

```
                if (grid[i][j] == '1') {
                    parent.push_back(i * n + j);
                    ++count;
                }
                else {
                    parent.push_back(-1);
                }
                rank.push_back(0);
            }
        }
    }

    int find(int i) {
        if (parent[i] != i) {
            parent[i] = find(parent[i]);
        }
        return parent[i];
    }

    void unite(int x, int y) {
        int rootx = find(x);
        int rooty = find(y);
        if (rootx != rooty) {
            if (rank[rootx] < rank[rooty]) {
                swap(rootx, rooty);
            }
            parent[rooty] = rootx;
            if (rank[rootx] == rank[rooty]) rank[rootx] += 1;
            --count;
        }
    }

    int getCount() const {
        return count;
    }

private:
    vector<int> parent;
    vector<int> rank;
    int count;
};

class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int nr = grid.size();
        if (!nr) return 0;
        int nc = grid[0].size();

        UnionFind uf(grid);
        int num_islands = 0;
        for (int r = 0; r < nr; ++r) {
```

```
            for (int c = 0; c < nc; ++c) {
                if (grid[r][c] == '1') {
                    grid[r][c] = '0';
                    if (r - 1 >= 0 && grid[r-1][c] == '1') uf.unite(r * nc + c, (r-1) * nc
+ c);
                    if (r + 1 < nr && grid[r+1][c] == '1') uf.unite(r * nc + c, (r+1) * nc
+ c);
                    if (c - 1 >= 0 && grid[r][c-1] == '1') uf.unite(r * nc + c, r * nc + c
- 1);
                    if (c + 1 < nc && grid[r][c+1] == '1') uf.unite(r * nc + c, r * nc + c
+ 1);
                }
            }
        }

        return uf.getCount();
    }
};
```

题目129:

我的错误解法：

主要错就错在因为此题是需要回溯的，所以visited这个vector不能随便的修改

```
class Solution {
public:
    bool wordPuzzle(vector<vector<char>>& grid, string target) {
        for(int i = 0;i < grid.size();i++){
            for(int j = 0;j < grid[0].size();j++){
                vector<vector<int>> visited(grid.size(),vector<int>(grid[0].size(),0));
                if(dfs(grid,visited,target,0,i,j)){
                    return true;
                }
            }
        }
        return false;
    }

    bool dfs(vector<vector<char>>& grid, vector<vector<int>>& visited,string target, int
k,int x, int y){
        // A B C E
        // S F E S
        // A D E E
        if(k == target.size()){
            return true;
        }
        if(!(x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size())){
            return false;
        }
        if(grid[x][y] != target[k] || visited[x][y] == 1){
```

```
            return false;
        }
        visited[x][y] = 1;
        return dfs(grid, visited,target, k + 1, x + 1,y) || dfs(grid, visited,target, k + 1, x - 1,y)  || dfs(grid,visited, target, k + 1, x,y + 1) || dfs(grid,visited, target, k + 1, x,y - 1);
    }
};
```

大佬的解法：

需要注意的是这边的k是从1开始的。这边visited的回溯真的好巧妙啊！

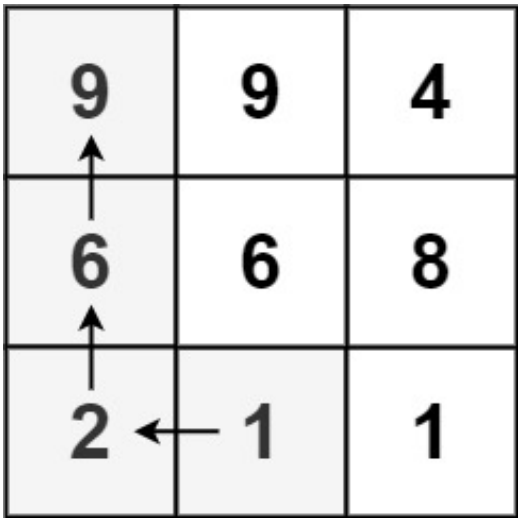
```
class Solution {
public:
    bool wordPuzzle(vector<vector<char>>& grid, string target) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<bool>> b(m,vector<bool>(n,true));
        vector<vector<int>> xy({{1,0}, {0,-1},{-1,0},{0,1}});
        for(int i = 0;i < m;i++){
            for(int j = 0;j < n;j++){
                if(grid[i][j] == target[0] && dfs(grid,target,xy,i,j,1,b)){
                    return true;
                }
            }
        }
        return false;
    }

     bool dfs(vector<vector<char>>& board, string& word, vector<vector<int>>& xy, int i, int j, int k, vector<vector<bool>>& b) {
         if(k == word.size()){
             return true;
         }
         b[i][j] = false;
         bool answer = false;
         for(int z = 0;z < 4;z++){
             int x = i + xy[z][0],y = j + xy[z][1];
            if(x >= 0 && x < board.size() && y >= 0 && y < board[0].size() && b[x][y] && board[x][y] == word[k] && dfs(board,word,xy,x,y,k + 1,b)){
                answer = true;
                break;
            }
         }
         b[i][j] = true;
         return answer;
     }

};
```

最有难度的一集：329 矩阵中的最长递增路径

给定一个 `m x n` 整数矩阵 `matrix` ，找出其中 **最长递增路径** 的长度。

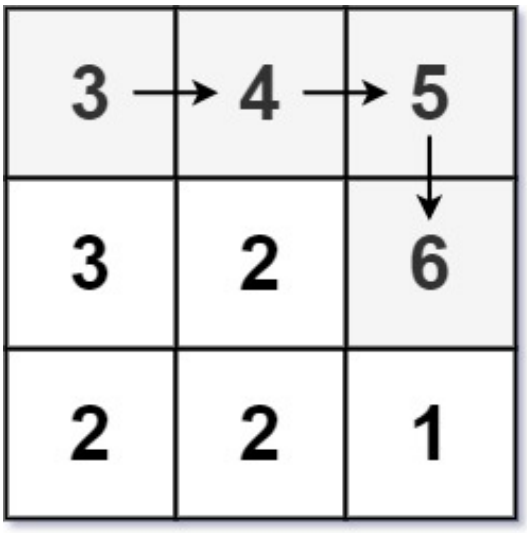
对于每个单元格，你可以往上，下，左，右四个方向移动。 你 **不能** 在 **对角线** 方向上移动或移动到 **边界外**（即不允许环绕）。

示例 1：




```
输入：matrix = [[9,9,4],[6,6,8],[2,1,1]]
输出：4
解释：最长递增路径为 [1, 2, 6, 9]。
```

示例 2：




```
输入：matrix = [[3,4,5],[3,2,6],[2,2,1]]
输出：4
解释：最长递增路径是 [3, 4, 5, 6]。注意不允许在对角线方向上移动。
```

示例 3：

```
输入：matrix = [[1]]
输出：1
```

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 200`
- `0 <= matrix[i][j] <= 231 - 1`

解法1: 记忆化深度优先搜索

将矩阵看成一个有向图，每个单元格对应图中的一个节点，如果相邻的两个单元格的值不相等，则在相邻的两个单元格之间存在一条从较小值指向较大值的有向边。问题转化成在有向图中寻找最长路径。

深度优先搜索是非常直观的方法。从一个单元格开始进行深度优先搜索，即可找到从该单元格开始的最长递增路径。对每个单元格分别进行深度优先搜索之后，即可得到矩阵中的最长递增路径的长度。

但是如果使用朴素深度优先搜索，时间复杂度是指数级，会超出时间限制，因此必须加以优化。

朴素深度优先搜索的时间复杂度过高的原因是进行了大量的重复计算，同一个单元格会被访问多次，每次访问都要重新计算。由于同一个单元格对应的最长递增路径的长度是固定不变的，因此可以使用记忆化的方法进行优化。用矩阵 memo\textit{memo}memo 作为缓存矩阵，已经计算过的单元格的结果存储到缓存矩阵中。

使用记忆化深度优先搜索，当访问到一个单元格 (i,j)(i,j)(i,j) 时，如果 memo[i][j]≠0\textit{memo}[i][j] \neq 0memo[i][j]
\
=0，说明该单元格的结果已经计算过，则直接从缓存中读取结果，如果 memo[i][j]=0\textit{memo}[i][j]=0memo[i][j]=0，说明该单元格的结果尚未被计算过，则进行搜索，并将计算得到的结果存入缓存中。

遍历完矩阵中的所有单元格之后，即可得到矩阵中的最长递增路径的长度。

```
class Solution {
public:
    static constexpr int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    int rows, columns;

    int longestIncreasingPath(vector< vector<int> > &matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        rows = matrix.size();
        columns = matrix[0].size();
        auto memo = vector< vector<int> > (rows, vector <int> (columns));
        int ans = 0;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < columns; ++j) {
                ans = max(ans, dfs(matrix, i, j, memo));
            }
```

```
        }
        return ans;
    }

    int dfs(vector< vector<int> > &matrix, int row, int column, vector< vector<int> >
&memo) {
        if (memo[row][column] != 0) {
            return memo[row][column];
        }
        ++memo[row][column];
        for (int i = 0; i < 4; ++i) {
            int newRow = row + dirs[i][0], newColumn = column + dirs[i][1];
            if (newRow >= 0 && newRow < rows && newColumn >= 0 && newColumn < columns &&
matrix[newRow][newColumn] > matrix[row][column]) {
                memo[row][column] = max(memo[row][column], dfs(matrix, newRow, newColumn,
memo) + 1);
            }
        }
        return memo[row][column];
    }
};
```

解法2: 拓扑排序

从方法一可以看到，每个单元格对应的最长递增路径的结果只和相邻单元格的结果有关，那么是否可以使用动态规划求解?

根据方法一的分析，动态规划的状态定义和状态转移方程都很容易得到。方法一中使用的缓存矩阵 $memo$ 即为状态值，状态转移方程如下：

$$
\begin{aligned}
&memo[i][j] = \max\{memo[x][y]\} + 1 \\
&\text{其中 } (x, y) \text{ 与 } (i, j) \text{ 在矩阵中相邻，并且 } matrix[x][y] > matrix[i][j]
\end{aligned}
$$

动态规划除了状态定义和状态转移方程，还需要考虑边界情况。这里的边界情况是什么呢?

如果一个单元格的值比它的所有相邻单元格的值都要大，那么这个单元格对应的最长递增路径是 1，这就是边界条件。这个边界条件并不直观，而是需要根据矩阵中的每个单元格的值找到作为边界条件的单元格。

仍然使用方法一的思想，将矩阵看成一个有向图，计算每个单元格对应的出度，即有多少条边从该单元格出发。对于作为边界条件的单元格，该单元格的值比所有的相邻单元格的值都要大，因此作为边界条件的单元格的出度都是 0。

基于出度的概念，可以使用拓扑排序求解。从所有出度为 0 的单元格开始广度优先搜索，每一轮搜索都会遍历当前层的所有单元格，更新其余单元格的出度，并将出度变为 0 的单元格加入下一层搜索。当搜索结束时，搜索的总层数即为矩阵中的最长递增路径的长度。

```
class Solution {
public:
    static constexpr int dirs[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    int rows, columns;
```

```
    int longestIncreasingPath(vector< vector<int> > &matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        rows = matrix.size();
        columns = matrix[0].size();
        auto outdegrees = vector< vector<int> > (rows, vector <int> (columns));
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < columns; ++j) {
                for (int k = 0; k < 4; ++k) {
                    int newRow = i + dirs[k][0], newColumn = j + dirs[k][1];
                    if (newRow >= 0 && newRow < rows && newColumn >= 0 && newColumn <
columns && matrix[newRow][newColumn] > matrix[i][j]) {
                        ++outdegrees[i][j];
                    }
                }
            }
        }
        queue < pair<int, int> > q;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < columns; ++j) {
                if (outdegrees[i][j] == 0) {
                    q.push({i, j});
                }
            }
        }
        int ans = 0;
        while (!q.empty()) {
            ++ans;
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                auto cell = q.front(); q.pop();
                int row = cell.first, column = cell.second;
                for (int k = 0; k < 4; ++k) {
                    int newRow = row + dirs[k][0], newColumn = column + dirs[k][1];
                    if (newRow >= 0 && newRow < rows && newColumn >= 0 && newColumn <
columns && matrix[newRow][newColumn] < matrix[row][column]) {
                        --outdegrees[newRow][newColumn];
                        if (outdegrees[newRow][newColumn] == 0) {
                            q.push({newRow, newColumn});
                        }
                    }
                }
            }
        }
        return ans;
    }
};
```