import java.util.LinkedList;

import java.util.Queue;

public class NumberOfIslands {

public static final char[][] grid1 = {

{'1','1','1','1','0'},

{'1','1','0','1','0'},

{'1','1','0','0','0'},

{'0','0','0','0','0'}

};

public static final char[][] grid2 = {

{'1','1','0','0','0'},

{'1','1','0','0','0'},

{'0','0','1','0','0'},

{'0','0','0','1','1'}

};

public static void main(String[] args){

NumberOfIslands noi = new NumberOfIslands();

//int res1 = noi.numIslands1(grid2);

int res2 = noi.numIslands2(grid2);

//System.out.print(res1 + "\n");

System.out.print(res2);

}

private int row;

private int col;

*/\*Deep First Search*

*\* Time complexity : O(m \* n) Traverse the matrix (Two-dimensional array), so it's O(m\*n)*

*\* Space complexity : O(n) Uses stack when call the recursive function.*

*\* \*/*

private int numIslands1(char[][] grid) {

int res = 0;

row = grid.length; // get the row of matrix

if (row == 0)

return 0;

col = grid[0].length; // get the column

// Traverse the matrix

for (int i = 0; i < row; i++) {

for (int j = 0; j < col; j++) {

if (grid[i][j] == '1') {

dfs(grid, i, j); // dfs(Deep First Search)

// if the position is '1', then search '1' around it recursively

res++; // if the recursive function dfs returns, means there is no formed '1' around, so the result array plus 1

}

}

}

return res;

}

private void dfs(char[][] grid, int i, int j){

if(i < 0 || j < 0 || i >= row || j >= col || grid[i][j] == '0')

return;

grid[i][j] = '0'; // if the position have been visited, then set it by '0'

// recurse, find down, up, right and left side whether there is '1'

dfs(grid, i, j + 1);

dfs(grid, i, j - 1);

dfs(grid, i + 1, j);

dfs(grid, i - 1, j);

}

*/\*Broad First Search*

*\* Time complexity : O(m \* n) Traverse the matrix (Two-dimensional array), so it's O(m\*n)*

*\* Space complexity : O(m \* n) Uses queue to store value which can be m\*n in worst case.*

*\* \*/*

private int numIslands2(char[][] grid) {

int res = 0;

for(int i = 0; i < grid.length; i++) {

for (int j = 0; j < grid[0].length; j++) {

if (grid[i][j] == '1') {

bfs(grid, i, j); // bfs(Broad First Search)

res++;

}

}

}

return res;

}

private void bfs(char[][] grid, int x, int y){

grid[x][y] = '0';

int row = grid.length;

int col = grid[0].length;

Queue<Integer> queue = new LinkedList<>(); // use a queue to store value '1', then search around this position

int pos = x \* col + y; // get the number of value in matrix

queue.offer(pos);

while (!queue.isEmpty()){ // as long as queue is not empty, means

pos = queue.poll();

int i = pos / col; // get the index in matrix

int j = pos % col;

if(i > 0 && grid[i - 1][j] == '1'){ // search left

queue.offer((i - 1) \* col + j); // if find '1', then put the index of '1' into queue, and set this position to '0'

grid[i - 1][j] = '0';

}

if(i < row - 1 && grid[i + 1][j] == '1'){ // search right

queue.offer((i + 1) \* col + j);

grid[i + 1][j] = '0';

}

if(j > 0 && grid[i][j - 1] == '1'){ // search up

queue.offer(i \* col + j - 1);

grid[i][j - 1] = '0';

}

if(j < col - 1 && grid[i][j + 1] == '1'){ // search down

queue.offer(i \* col + j + 1);

grid[i][j + 1] = '0';

}

}

}

}