# Question

Given an m x n 2d grid map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

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

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

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

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

]

**Output:** 1

**Example 2:**

**Input:** grid = [

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

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

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

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

]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 300
* grid[i][j] is '0' or '1'.

# Solution

#### **Approach #1 DFS [Accepted]**

**Intuition**

Treat the 2d grid map as an undirected graph and there is an edge between two horizontally or vertically adjacent nodes of value '1'.

**Algorithm**

Linear scan the 2d grid map, if a node contains a '1', then it is a root node that triggers a Depth First Search. During DFS, every visited node should be set as '0' to mark as visited node. Count the number of root nodes that trigger DFS, this number would be the number of islands since each DFS starting at some root identifies an island.

1 / 8

|  |
| --- |
| **class Solution {**  **void dfs(char[][] grid, int r, int c) {**  **int nr = grid.length;**  **int nc = grid[0].length;**  **if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0') {**  **return;**  **}**  **grid[r][c] = '0';**  **dfs(grid, r - 1, c);**  **dfs(grid, r + 1, c);**  **dfs(grid, r, c - 1);**  **dfs(grid, r, c + 1);**  **}**  **public int numIslands(char[][] grid) {**  **if (grid == null || grid.length == 0) {**  **return 0;**  **}**  **int nr = grid.length;**  **int nc = grid[0].length;**  **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;**  **dfs(grid, r, c);**  **}**  **}**  **}**  **return num\_islands;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : O(M \times N)*O*(*M*×*N*) where M*M* is the number of rows and N*N* is the number of columns.
* Space complexity : worst case O(M \times N)*O*(*M*×*N*) in case that the grid map is filled with lands where DFS goes by M \times N*M*×*N* deep.

#### **Approach #2: BFS [Accepted]**

**Algorithm**

Linear scan the 2d grid map, if a node contains a '1', then it is a root node that triggers a Breadth First Search. Put it into a queue and set its value as '0' to mark as visited node. Iteratively search the neighbors of enqueued nodes until the queue becomes empty.

|  |
| --- |
| class Solution {  public int numIslands(char[][] grid) {  if (grid == null || grid.length == 0) {  return 0;  }  int nr = grid.length;  int nc = grid[0].length;  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'; // mark as visited  Queue<Integer> neighbors = new LinkedList<>();  neighbors.add(r \* nc + c);  while (!neighbors.isEmpty()) {  int id = neighbors.remove();  int row = id / nc;  int col = id % nc;  if (row - 1 >= 0 && grid[row-1][col] == '1') {  neighbors.add((row-1) \* nc + col);  grid[row-1][col] = '0';  }  if (row + 1 < nr && grid[row+1][col] == '1') {  neighbors.add((row+1) \* nc + col);  grid[row+1][col] = '0';  }  if (col - 1 >= 0 && grid[row][col-1] == '1') {  neighbors.add(row \* nc + col-1);  grid[row][col-1] = '0';  }  if (col + 1 < nc && grid[row][col+1] == '1') {  neighbors.add(row \* nc + col+1);  grid[row][col+1] = '0';  }  }  }  }  }  return num\_islands;  }  } |

**Complexity Analysis**

* Time complexity : O(M \times N)*O*(*M*×*N*) where M*M* is the number of rows and N*N* is the number of columns.
* Space complexity : O(min(M, N))*O*(*min*(*M*,*N*)) because in worst case where the grid is filled with lands, the size of queue can grow up to min(M,N*M*,*N*).

#### **Approach #3: Union Find (aka Disjoint Set) [Accepted]**

**Algorithm**

Traverse the 2d grid map and union adjacent lands horizontally or vertically, at the end, return the number of connected components maintained in the UnionFind data structure.

For details regarding to Union Find, you can refer to this [article](https://leetcode.com/articles/redundant-connection/).

1 / 6

|  |
| --- |
| **class Solution {**  **class UnionFind {**  **int count; // # of connected components**  **int[] parent;**  **int[] rank;**  **public UnionFind(char[][] grid) { // for problem 200**  **count = 0;**  **int m = grid.length;**  **int n = grid[0].length;**  **parent = new int[m \* n];**  **rank = new int[m \* n];**  **for (int i = 0; i < m; ++i) {**  **for (int j = 0; j < n; ++j) {**  **if (grid[i][j] == '1') {**  **parent[i \* n + j] = i \* n + j;**  **++count;**  **}**  **rank[i \* n + j] = 0;**  **}**  **}**  **}**  **public int find(int i) { // path compression**  **if (parent[i] != i) parent[i] = find(parent[i]);**  **return parent[i];**  **}**  **public void union(int x, int y) { // union with rank**  **int rootx = find(x);**  **int rooty = find(y);**  **if (rootx != rooty) {**  **if (rank[rootx] > rank[rooty]) {**  **parent[rooty] = rootx;**  **} else if (rank[rootx] < rank[rooty]) {**  **parent[rootx] = rooty;**  **} else {**  **parent[rooty] = rootx; rank[rootx] += 1;**  **}**  **--count;**  **}**  **}**  **public int getCount() {**  **return count;**  **}**  **}**  **public int numIslands(char[][] grid) {**  **if (grid == null || grid.length == 0) {**  **return 0;**  **}**  **int nr = grid.length;**  **int nc = grid[0].length;**  **int num\_islands = 0;**  **UnionFind uf = new UnionFind(grid);**  **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.union(r \* nc + c, (r-1) \* nc + c);**  **}**  **if (r + 1 < nr && grid[r+1][c] == '1') {**  **uf.union(r \* nc + c, (r+1) \* nc + c);**  **}**  **if (c - 1 >= 0 && grid[r][c-1] == '1') {**  **uf.union(r \* nc + c, r \* nc + c - 1);**  **}**  **if (c + 1 < nc && grid[r][c+1] == '1') {**  **uf.union(r \* nc + c, r \* nc + c + 1);**  **}**  **}**  **}**  **}**  **return uf.getCount();**  **}**  **}** |

**Complexity Analysis**

* Time complexity : O(M \times N)*O*(*M*×*N*) where M*M* is the number of rows and N*N* is the number of columns. Note that Union operation takes essentially constant time[[1]](https://leetcode.com/problems/number-of-islands/solution/#fn1) when UnionFind is implemented with both path compression and union by rank.
* Space complexity : O(M \times N)*O*(*M*×*N*) as required by UnionFind data structure.