**200. Number of Islands**

<https://leetcode.com/problems/number-of-islands/>

1. **Listen**

**Problem Statement:**

Given an **m** x **n** 2D binary grid **grid** which represents a 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.

**Input:**

**m** x **n** 2D matrix.

The matrix is a binary grid which represents a map of '1's (land) and '0's (water)

**Goal:**

Find the number of islands in the matrix

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.

**Return:**

return ***the number of islands***

1. **Example**

**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'.

**Test Cases:**

* 1 x 1 matrix
* 1 island surrounded by water
* 1 island adjacent to land
* no islands (all 0’s)
* one giant island (all 1’s)
* multiple islands

1. **Brute Force**

**Solution 1: DFS**

**Intuition**

We can run a graph DFS traversal on this matrix. We simply iterate over every cell, and if we encounter a ‘1’ (unvisited vertex), we recursively find all adjacent ‘1’s using DFS.

To mark a ‘1’ as visited (to avoid cycles), we change it to ‘0’ (which essentially marks it as visited as per our condition).

In this sense, we are ‘drowning all the islands’.

**Runtime**

There are two parts to the algorithm:

1. We iterate over the entire matrix – this takes O(N\*M)
2. We dfs for each occurrence of ‘1’ – can take up to O(N\*M)

Worst case, we iterate over the entire matrix twice – making it O(2(N\*M)), or just O(N\*M).

The DFS runtime is proportional to the total number of vertexes *and* edges of the graph visited.

In that case, there are **N\*M** (dimensions of the graph) vertexes and slightly less than 4\*N\*M edges. Therefore, the runtime is O(N\*M) (after dropping the constant).

**Why so**: because if we visit an unvisited vertex (1 instead of 0), we process each vertex that is horizontally or vertically adjacent to the current node.

In other words, each vertex has four edges (one to each adjacent vertex in each direction).

In the worst case, we will process all four edges for each vertex.

There are N\*M vertexes in the graph, and we will have to process up to 4 edges for each vertex.

**Space**

In the very worst case, if the entire grid consists of ‘1’ (there is no water, and the entire grid is one big island), then this will require all vertexes to be on the stack. Therefore, it will require O(N\*M) space.

**Implementation**

**DFS** finds a path between two vertices by exploring each possible path as far as possible before backtracking.

One of the most important parts of DFS is to be careful to ***avoid cycles***.

To do this, each time we visit a vertex *v*, we ***mark it visited***.

We recursively call depth-first search on all ***adjacent unvisited vertices***.

This continue to recursively travel down a path until there are ***no more* *unvisited adjacent vertices***.

When this happens, we backtrack up to the previously visited node, and if possible, visits another unvisited adjacent vertex.

1. **Optimize**

**Solution 2: BFS**

Similar to DFS, except we use a conventional BFS algorithm.

Same runtime and space as DFS.

1. **Walkthrough**
2. **Implement**
3. **Test**