LeetCode 200 — Number of Islands

🔗 https://leetcode.com/problems/number-of-islands/

1. Problem Title & Link

Title: LeetCode 200: Number of Islands
Link: https://leetcode.com/problems/number-of-islands/

2. Problem Statement (Short Summary)

Given a 2D grid of '1' (land) and '0' (water), return the **number of islands**.
An island is a group of '1's connected **horizontally or vertically** (4-directional).
You may modify the grid in-place.

3. Examples (Input → Output)

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

Example 3

Input: grid = []
Output: 0

4. Constraints

- m == grid.length

- n == grid[i].length
- 0 <= m, n <= 300
- grid[i][j] is '0' or '1'
- Must run in O(m * n) ideally
- In-place modification is allowed (common solution)

5. Core Concept (Pattern / Topic)

⭐ **Graph traversal → Flood fill**

Implementations: DFS, BFS, or Union-Find (disjoint set)

6. Thought Process (Step-by-Step Explanation)

Brute idea

Scan every cell; when you find a '1', start a flood-fill (DFS/BFS) marking all connected '1' → increment island count. This visits each cell at most once → O(m*n).

Why this works

Flood-fill explores the entire connected component (island). Counting how many times you initiate a flood-fill equals number of connected components (islands).

7. Visual / Intuition Diagram (ASCII)

Grid:

1 1 0

1 0 0

0 0 1

Scan left→right, top→bottom:

- At (0,0): flood-fill marks connected land: (0,0),(0,1),(1,0) → island #1
- Continue, find (2,2) → island #2

8. Pseudocode (Language Independent)

```
count = 0
for i in 0..m-1:
  for j in 0..n-1:
    if grid[i][j] == '1':
      count += 1
      dfs_mark(i, j)  # mark entire island as '0' or visited
return count

function dfs_mark(i, j):
  if i out of bounds or j out of bounds or grid[i][j] == '0':
```

```
    return
  grid[i][j] = '0'
  for each (di,dj) in [(1,0),(-1,0),(0,1),(0,-1)]:
    dfs_mark(i+di, j+dj)
```

9. Code Implementation

✅ **Python (DFS in-place)**

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid or not grid[0]:
            return 0

        m, n = len(grid), len(grid[0])

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or grid[i][j] == '0':
                return
            grid[i][j] = '0'  # mark visited
            dfs(i+1, j)
            dfs(i-1, j)
            dfs(i, j+1)
            dfs(i, j-1)

        count = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    count += 1
                    dfs(i, j)
        return count
```

✅ **Python (BFS)**

```
from collections import deque

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid or not grid[0]:
            return 0
        m, n = len(grid), len(grid[0])
        count = 0
        for i in range(m):
```

```
            for j in range(n):
                if grid[i][j] == '1':
                    count += 1
                    grid[i][j] = '0'
                    q = deque([(i, j)])
                    while q:
                        x, y = q.popleft()
                        for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
                            nx, ny = x + dx, y + dy
                            if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] ==
'1':
                                grid[nx][ny] = '0'
                                q.append((nx, ny))
        return count
```

✅ **Java (DFS)**

```
class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;
        int m = grid.length, n = grid[0].length, count = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    count++;
                    dfs(grid, i, j, m, n);
                }
            }
        }
        return count;
    }

    private void dfs(char[][] grid, int i, int j, int m, int n) {
        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') return;
        grid[i][j] = '0';
        dfs(grid, i+1, j, m, n);
        dfs(grid, i-1, j, m, n);
        dfs(grid, i, j+1, m, n);
        dfs(grid, i, j-1, m, n);
    }
}
```

10. Time & Space Complexity

- **Time:** O(m * n) — each cell is visited at most once.

- **Space:**
  - DFS recursive: O(h) recursion depth; worst-case O(m*n) (if grid all land and recursion stacks all cells, but practically O(min(m,n)) for thin shapes).
  - BFS queue: O(min(m*n, perimeter-of-largest-island)) worst case O(mn).
  - If using Union-Find: additional O(m*n) space.

11. Common Mistakes / Edge Cases

- ❌ Forgetting to mark visited → infinite loop / repeated counts
- ❌ Using diagonal neighbors (should be only 4-directional)
- ❌ Modifying original grid when not allowed (but LeetCode allows it) — if not allowed, use visited set
- ❌ Recursion depth overflow for very large grids (use iterative BFS or increase recursion limit)
- Edge cases: empty grid, all '0', all '1', single row/column

12. Detailed Dry Run (Step-by-Step Table)

Grid:

1 1 0

1 0 0

0 0 1

Scan order (i,j):

- (0,0): '1' → count=1 → DFS marks (0,0),(0,1),(1,0)
  Grid becomes:
  0 0 0
- 0 0 0
- 0 0 1
-
- Continue scanning: all zeros until (2,2)
- (2,2): '1' → count=2 → mark it → grid all zeros
  Return 2.

Table (key steps):

Step	Cell	Action	Count
start	(0,0)	find '1' → dfs mark island	1
after dfs	marked (0,0),(0,1),(1,0)	—	1
continue	(2,2)	find '1' → dfs mark	2
end	—	return count	2

13. Common Use Cases (Real-Life / Interview)

- Counting connected components in binary images (blob detection)
- Map / geography analysis: counting islands or forest patches
- Graph connectivity in grid-based games or simulations
- Grouping contiguous user activity regions (heatmaps)

14. Common Traps (Important!)

- Using diagonal connectivity accidentally
- Stack overflow in recursive DFS on large full-land grids
- Mistaking index bounds (row vs column order)
- Using expensive visited structures instead of marking in-place when allowed

15. Builds To (Related LeetCode Problems)

- LC 695 — Max Area of Island (similar flood fill)
- LC 463 — Island Perimeter
- LC 827 — Making A Large Island (Union-Find)
- LC 289 — Game of Life (grid simulation)
- LC 994 — Rotting Oranges (BFS level-order on grid)

16. Alternate Approaches + Comparison

- **DFS (in-place)** — O(m*n) time, low auxiliary memory, simple — most common.
- **BFS (in-place)** — O(m*n) time, iterative (no recursion limit), queue memory.
- **Union-Find** — build DSU for every land cell, union neighbors, then count distinct roots; good when many queries or dynamic changes, requires O(m*n) extra space and slightly more code.
- **Bitset / bitmap optimizations** — for memory-tight solutions, advanced.

17. Why This Solution Works (Short Intuition)

Each time we encounter an unvisited '1' we trigger a flood-fill that marks the entire connected component (island). Counting flood-fills equals counting connected components (islands). Every cell is processed once → optimal.

18. Variations / Follow-Up Questions

- How to modify to allow **diagonal** connectivity? (use 8 directions)
- Return sizes of all islands (collect counts in flood-fill).
- Find the **largest island** area (LC 695).
- Dynamic grid with flipping water→land queries (use Union-Find).
- Count islands in a streaming grid or very large grid using external memory techniques.