https://leetcode.com/problems/number-of-islands/m (https://leetcode.com/problems/number-of-islands/m)

In [ ]:

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:

        visited = set()
        count = 0
        for i in range(0,len(grid)):
            for j in range(0, len(grid[0])):
                if self.explore(grid, i, j, visited):
                    count +=1

        return count

    def explore(self, grid, i,j, visited):
        # boundary
        if i < 0 or j < 0 or i >= len(grid) or j >= len(grid[0]):
            return False

        # Invalid State
        if grid[i][j] == "0":
            return False

        if (i,j) in visited:
            return False

        visited.add( (i,j) )

        # Recursion: all possible paths from here
        self.explore(grid, i+1, j, visited) # down
        self.explore(grid,i, j+1, visited) # right
        self.explore(grid,i, j-1, visited) # left
        self.explore(grid,i-1, j, visited) # up
        return True

# SC: O(M*N) + O(m*n) : O(m*n)
# TC: O(m*n)
```

In [ ]:

```
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        Set<String> visited = new HashSet<>();

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (explore(grid, i, j, visited)) {
                    count++;
                }
            }
        }

        return count;
    }

    private boolean explore(char[][] grid, int i, int j, Set<String> visited)
        // Boundary check
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length) {
            return false;
        }

        // Invalid state
        if (grid[i][j] == '0') {
            return false;
        }

        String key = i + "," + j;
        if (visited.contains(key)) {
            return false;
        }

        visited.add(key);

        // Recursion
        explore(grid, i + 1, j, visited); // down
        explore(grid, i, j + 1, visited); // right
        explore(grid, i, j - 1, visited); // left
        explore(grid, i - 1, j, visited); // up

        return true;
    }
}
```

In [ ]:

```
public class Solution {
    public int numIslands(char[][] grid) {
    int count = 0;

    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            if (grid[i][j] == '1') {
                count++;
                backtrack(grid, i, j);
            }
        }
    }
    return count;
}

private void backtrack(char[][] grid, int i, int j) {
    if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j
        return;

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

    backtrack(grid, i + 1, j);
    backtrack(grid, i - 1, j);
    backtrack(grid, i, j + 1);
    backtrack(grid, i, j - 1);
    }
}

// SC: Worst: O(m*n) Best: O(1)
// TC: O(M*N)
```

In [ ]:

```
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        boolean[][] visited = new boolean[grid.length][grid[0].length];

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    traverse(grid, i, j, visited);
                    count++;
                }
            }
        }
        return count;
    }

    private void traverse(char[][] grid, int i, int j, boolean[][] visited) {
        if (i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[
            return;
        }

        visited[i][j] = true;
        traverse(grid, i - 1, j, visited);
        traverse(grid, i, j + 1, visited);
        traverse(grid, i, j - 1, visited);
        traverse(grid, i + 1, j, visited);
    }
}
```

In [ ]:

In [ ]:

https://leetcode.com/problems/max-area-of-island/submissions
(https://leetcode.com/problems/max-area-of-island/submissions)

In [ ]:

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:

        visited = set()
        max_area = 0
        for i in range(0,len(grid)):
            for j in range(0, len(grid[0])):
                area = self.explore(grid, i, j, visited)
                if area > max_area:
                    max_area = area

        return max_area

    def explore(self, grid, i,j, visited):
        # boundary
        if i < 0 or j < 0 or i >= len(grid) or j >= len(grid[0]):
            return 0

        # Invalid State
        if grid[i][j] == 0:
            return 0

        if (i,j) in visited:
            return 0

        visited.add( (i,j) )

        # Recursion: all possible paths from here
        return 1 + self.explore(grid, i+1, j, visited) + self.explore(grid,i,
        self.explore(grid,i, j-1, visited) + self.explore(grid,i-1, j, visited

# SC: O(M*N) + O(m*n) : O(m*n)
# TC: O(m*n)
```