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

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.

**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'.

**Attempt 1: 2023-06-10**

**Solution 1: DFS (10 min)**

class Solution {

public int numIslands(char[][] grid) {

int m = grid.length;

int n = grid[0].length;

int count = 0;

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

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

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

count++;

helper(i, j, grid);

}

}

}

return count;

}

int[] dx = {0,0,1,-1};

int[] dy = {1,-1,0,0};

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

// If out of boundary or the cell is water, stop, means reach the board of current land

if(x < 0 || x >= grid.length || y < 0 || y >= grid[0].length || grid[x][y] == '0') {

return;

}

// Critical logic to mark a land and all connected lands as water

// after visiting, work with "count++" when find a position is

// land in calling method to avoid counting same land position again

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

for(int k = 0; k < 4; k++) {

helper(x + dx[k], y + dy[k], grid);

}

}

}

Time Complexity: O(M\*N), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.

Space Complexity: O(M\*N)

**Refer to**

<https://leetcode.com/problems/number-of-islands/solutions/56340/python-simple-dfs-solution/>

**Iterate through each of the cell and if it is an island, do DFS to mark all adjacent islands, then increase the counter by 1.**

def numIslands(self, grid):

if not grid:

return 0

count = 0

for i in range(len(grid)):

for j in range(len(grid[0])):

if grid[i][j] == '1':

self.dfs(grid, i, j)

count += 1

return count

def dfs(self, grid, i, j):

if i<0 or j<0 or i>=len(grid) or j>=len(grid[0]) or grid[i][j] != '1':

return

grid[i][j] = '#'

self.dfs(grid, i+1, j)

self.dfs(grid, i-1, j)

self.dfs(grid, i, j+1)

self.dfs(grid, i, j-1)

**Complexity**

* Time: O(M\*N), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.
* Space: O(M\*N)

**Solution 2: Union Find + Path Compression + Weighted Union (10 min)**

class Solution {

int[] dx = {0,0,1,-1};

int[] dy = {1,-1,0,0};

public int numIslands(char[][] grid) {

int m = grid.length;

int n = grid[0].length;

UnionFind uf = new UnionFind(grid);

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

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

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

for(int k = 0; k < 4; k++) {

int new\_i = i + dx[k];

int new\_j = j + dy[k];

if(new\_i >= 0 && new\_i < m && new\_j >= 0 && new\_j < n && grid[new\_i][new\_j] == '1') {

uf.union(new\_i \* n + new\_j, i \* n + j);

}

}

}

}

}

return uf.getCount();

}

}

class UnionFind {

int count = 0;

int m = 0;

int n = 0;

int[] parents;

int[] rank;

public UnionFind(char[][] grid) {

this.m = grid.length;

this.n = grid[0].length;

parents = 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') {

int id = i \* n + j;

parents[id] = id;

rank[id] = 1;

count++;

}

}

}

}

// Path compression

public int find(int x) {

if(parents[x] == x) {

return x;

}

return parents[x] = find(parents[x]);

}

// While loop path compression

public int find2(int x) {

while(x != parents[x]) {

parents[x] = parents[parents[x]];

x = parents[x];

}

return x;

}

// Weighted Union

public void union(int x, int y) {

int src = find(x);

int dst = find(y);

if(src != dst) {

if(rank[src] > rank[dst]) {

parents[dst] = src;

rank[src] += rank[dst];

} else {

parents[src] = dst;

rank[dst] += rank[src];

}

count--;

}

}

// Standard union

public void union2(int x, int y) {

int src = find(x);

int dst = find(y);

if(src != dst) {

parents[src] = dst;

count--;

}

}

public int getCount() {

return count;

}

}

Time Complexity: O(MN \* α(MN)), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.

Explanation: Using both path compression and union by size ensures that the amortized time per UnionFind operation is only α(n), which is optimal, where α(n) is the inverse Ackermann function. This function has a value α(n) < 5 for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.

Reference: https://en.wikipedia.org/wiki/Disjoint-set\_data\_structure or https://www.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534 for more information.

Space Complexity: O(M\*N)

**Refer to**

<https://leetcode.com/problems/number-of-islands/solutions/56354/1d-union-find-java-solution-easily-generalized-to-other-problems/>

For any problem I work on, I will try to generalize some reusable template out for future use. We have limited time during interview and too much to worry about, so having some code template to use is very handy. For this problem, although it is easier and probably suggested to use BFS, but Union find also comes handy and can be easily extended to solve Island 2 and Surrounded regions.

I separate all the union find logic in a separate class and use 1d version to make the code clear. I also use a 2d array for the 4 direction visit. int[][] distance = {{1,0},{-1,0},{0,1},{0,-1}};

int[][] distance = {{1,0},{-1,0},{0,1},{0,-1}};

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0) {

return 0;

}

UnionFind uf = new UnionFind(grid);

int rows = grid.length;

int cols = grid[0].length;

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

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

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

for (int[] d : distance) {

int x = i + d[0];

int y = j + d[1];

if (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == '1') {

int id1 = i\*cols+j;

int id2 = x\*cols+y;

uf.union(id1, id2);

}

}

}

}

}

return uf.count;

}

Union Find:

class UnionFind {

int[] father;

int m, n;

int count = 0;

UnionFind(char[][] grid) {

m = grid.length;

n = grid[0].length;

father = new int[m\*n];

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

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

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

int id = i \* n + j;

father[id] = id;

count++;

}

}

}

}

public void union(int node1, int node2) {

int find1 = find(node1);

int find2 = find(node2);

if(find1 != find2) {

father[find1] = find2;

count--;

}

}

public int find (int node) {

if (father[node] == node) {

return node;

}

father[node] = find(father[node]);

return father[node];

}

}

**Refer to**

<https://leetcode.com/problems/number-of-islands/solutions/583745/python-3-solutions-dfs-bfs-union-find-concise-clean/>

**✔️ Solution 3a: Union-Find (Naive)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, u):

while u != self.parent[u]:

u = self.parent[u]

return u

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

self.parent[pu] = pv

return True

class Solution:

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

DIR = [0, 1, 0, -1, 0]

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

uf = UnionFind(m\*n)

component = 0

for r in range(m):

for c in range(n):

if grid[r][c] == "0": continue

component += 1

curId = r \* n + c

for i in range(4):

nr, nc = r + DIR[i], c + DIR[i+1]

if nr < 0 or nr == m or nc < 0 or nc == n or grid[nr][nc] == "0": continue

neiId = nr \* n + nc

if uf.union(curId, neiId):

component -= 1

return component

**Complexity**

* Time: O(MN^2), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.
* Space: O(M\*N)

**✔️ Solution 3b: Union-Find (Path Compression)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

def find(self, u):

if u != self.parent[u]:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

self.parent[pu] = pv

return True

class Solution:

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

DIR = [0, 1, 0, -1, 0]

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

uf = UnionFind(m\*n)

component = 0

for r in range(m):

for c in range(n):

if grid[r][c] == "0": continue

component += 1

curId = r \* n + c

for i in range(4):

nr, nc = r + DIR[i], c + DIR[i+1]

if nr < 0 or nr == m or nc < 0 or nc == n or grid[nr][nc] == "0": continue

neiId = nr \* n + nc

if uf.union(curId, neiId):

component -= 1

return component

**Complexity**

* Time: O(MN \* logMN), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.
* Space: O(M\*N)

**✔️ Solution 3c: Union-Find (Path Compression & Union by Size)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

self.size = [1] \* n

def find(self, u):

if u != self.parent[u]:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv: return False

if self.size[pu] < self.size[pv]: # Merge pu to pv

self.size[pv] += self.size[pu]

self.parent[pu] = pv

else:

self.size[pu] += self.size[pv]

self.parent[pv] = pu

return True

class Solution:

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

DIR = [0, 1, 0, -1, 0]

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

uf = UnionFind(m\*n)

component = 0

for r in range(m):

for c in range(n):

if grid[r][c] == "0": continue

component += 1

curId = r \* n + c

for i in range(4):

nr, nc = r + DIR[i], c + DIR[i+1]

if nr < 0 or nr == m or nc < 0 or nc == n or grid[nr][nc] == "0": continue

neiId = nr \* n + nc

if uf.union(curId, neiId):

component -= 1

return component

**Complexity**

* Time: O(MN \* α(MN)), where M <= 300 is number of rows, N <= 300 is number of columns in the matrix.
* Explanation: Using both **path compression** and **union by size** ensures that the **amortized time** per **UnionFind** operation is only α(n), which is optimal, where α(n) is the inverse Ackermann function. This function has a value α(n) < 5 for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.
* Reference: <https://en.wikipedia.org/wiki/Disjoint-set_data_structure> or <https://www.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534> for more information.
* Space: O(M\*N)