<https://leetcode.ca/all/305.html>

A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. 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:

Input: m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]]

Output: [1,1,2,3]

Explanation:

Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

0 0 0

0 0 0

0 0 0

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

1 0 0

0 0 0 Number of islands = 1

0 0 0

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

1 1 0

0 0 0 Number of islands = 1

0 0 0

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

1 1 0

0 0 1 Number of islands = 2

0 0 0

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

1 1 0

0 0 1 Number of islands = 3

0 1 0

Follow up:

Can you do it in time complexity O(k log mn), where k is the length of the positions?

**Attempt 1: 2023-03-03**

**Solution 1: Union Find (10 min)**

import java.util.\*;

public class Solution {

class UnionFind {

int[] parent;

public UnionFind(int n) {

parent = new int[n];

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

parent[i] = i;

}

}

public int find(int x) {

if(x == parent[x]) {

return x;

}

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

}

public void union(int a, int b) {

int root\_a = find(a);

int root\_b = find(b);

if(root\_a != root\_b) {

parent[root\_a] = root\_b;

}

}

}

public List<Integer> numIslands2(int m, int n, int[][] positions) {

List<Integer> result = new ArrayList<>();

// An array to maintain the state of the land and water.

int[] isLand = new int[m \* n];

UnionFind u = new UnionFind(m \* n);

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

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

int count = 0;

for(int[] pos : positions) {

// Increment island count.

count++;

int x = pos[0];

int y = pos[1];

// Flatten the 2D position to 1D to use in union-find.

int index = x \* m + y;

// The code ensures that land cells are not counted more than once by

// checking if the cell is already land before performing the union

// operations. If a position is already land, it continues to the next

// iteration without altering the island count.

if(isLand[index] != 1) {

isLand[index] = 1;

// Explore all 4 adjacent directions.

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

int new\_x = x + dx[k];

int new\_y = y + dy[k];

int new\_index = new\_x \* m + new\_y;

// If adjacent cell is within bounds, is land, and is not already unioned with the current cell.

// Union the two cells.

u.union(new\_index, index);

// Decrement island count as we connected two islands.

count--;

}

}

}

result.add(count);

}

return result;

}

public static void main(String[] args) {

Solution so = new Solution();

int[][] positions = new int[][]{{0,0},{0,1},{1,2},{2,1}};

int target = 10;

List<Integer> result = so.numIslands2(3, 3, positions);

System.out.println(result);

}

}

Time Complexity: O(k \* α(m \* n)), where k is the number of positions and α denotes the Inverse

Space Complexity: O(m \* n)

**Refer to**

<https://algo.monster/liteproblems/305>

**Problem Description**

The task is to perform a series of land addition operations on a given 2D binary grid, which initially represents a map with all cells being water (denoted by 0s). We receive a list of positions that specify where land (1) should be added. Each operation may potentially connect with adjacent land to form or expand islands. An island is defined as a region of connected lands (horizontally or vertically), and each corner of the grid is considered surrounded by water. The problem requires us to return an array that reflects the number of distinct islands after each land addition operation.

**Intuition**

To efficiently track the changes and the number of islands as land cells are added, we can use the Union-Find algorithm. This algorithm is useful for keeping track of elements that are split into one or more disjoint sets and for merging these sets together.

Each cell in the 2D grid can be represented by a unique index in a 1D parent array (for instance, by mapping a cell (i, j) to an index i \* n + j in the parent array). Initially, every cell is its own parent, indicating no connection to other cells.

The idea is to initially treat each new piece of land as a new island (incrementing the island count). Then, for each new land addition, we look at its neighboring cells (up, down, left, right) to check if any of them contains land. If a neighbor is also land, we find the root parent of both the current cell and the neighbor. If both have the same root parent, they are part of the same island; if they have different root parents, it means we have connected two previously separate islands, so we unite them under one parent and decrement our island counter since we're merging two islands into one.

This process repeats for each piece of land added, with the overall number of islands being adjusted accordingly. After each operation, the current number of islands is stored in the result array which is returned at the end.

The solution given above implements this approach. It keeps track of the parent array and the grid representation, updating both as land cells are added and connected. For each added land cell, it either creates a new island or merges existing islands, adjusting the total count and recording the state after each operation.

**Solution Approach**

The solution implements the Union-Find algorithm, which is efficient in handling dynamic connectivity queries. In this context, Union-Find helps to keep track of which pieces of land are connected to each other, thereby forming islands.

The main components of the algorithm are:

**Parent array (p):** A 1D array where the index represents a cell on the 2D grid, and the value represents the "parent" of that cell. Initially, each cell is its own parent.

**Grid representation (grid):** A 2D array reflecting the current state of the grid where land and water cells are represented by 1s and 0s respectively.

**Steps Involved:**

**Initialization:** Set up the parent array with each cell as its own parent and initialize the 2D grid array with zeros.

**Find function (find):** A function to determine the root parent of a cell. This is used to find the ultimate parent (or representative) of an element. If two elements have the same root parent, they belong to the same set (or island).

def find(x):

if p[x] != x:

p[x] = find(p[x])

return p[x]

**Adding land:** Iterate over each position in the positions list, turning the corresponding water cell into land in the grid array.

**Connecting islands:** For each new land cell, increment the island count as it starts as a new island (cur += 1). Then, explore its four adjacent cells. If an adjacent cell is land, apply the find function to get the root parents for both the new land cell and its neighbor. If the root parents are different, it means that the new land addition has connected two separate islands and they should be united.

for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

if check(i + x, j + y) and find(i \* n + j) != find((i + x) \* n + j + y):

p[find(i \* n + j)] = find((i + x) \* n + j + y)

cur -= 1

**Updating the result:** After each land addition and potential island connections, the result (res) is appended with the current number of islands (cur).

The code ensures that land cells are not counted more than once by checking if the cell is already land before performing the union operations. If a position is already land, it continues to the next iteration without altering the island count.

By using Union-Find, the solution maintains an accurate count of islands after each operation and avoids recomputing the number of islands from scratch, resulting in a more efficient algorithm for the problem at hand.

**Example Walkthrough**

Let's go through a small example using the provided solution approach.

Consider a 3x3 grid, which is all water initially, and the positions to add land are [(0, 0), (0, 1), (1, 1), (2, 2)].

**Initialization:** The grid looks like this:

0 0 0

0 0 0

0 0 0

And we have a parent array of p for each cell, each initialized to their own index.

**Adding land (0, 0):** We start by turning the cell (0, 0) into land.

1 0 0

0 0 0

0 0 0

Island count cur is incremented to 1. Since there are no adjacent land cells, we move on to the next operation.

**Adding land (0, 1):** Then we add land to cell (0, 1).

1 1 0

0 0 0

0 0 0

cur becomes 2. We check adjacent cells and find that cell (0, 0) is land. Since cell (0, 1) is a new addition, their parents are different, and we perform a union. The find operation will update the parent of (0, 1) to be the same as (0, 0), and cur is decremented to 1.

**Adding land (1, 1):** Next, we add land at position (1, 1).

1 1 0

0 1 0

0 0 0

cur is increased to 2. We look at its neighbors and connect it with (0, 1). After the union with both (0, 1) and (1, 0), its parent will be updated, and cur will be decremented back to 1.

**Adding land (2, 2):** Finally, we add land to cell (2, 2).

1 1 0

0 1 0

0 0 1

cur increases to 2, and since the new land does not connect with existing land, the island count remains at 2. There are no union operations to perform here.

At each land addition, we append the current island count to the result array res, so after all operations, res would be [1, 1, 1, 2].

This illustrates the solution approach using the Union-Find algorithm to efficiently keep track of the number of islands after each land addition operation.

**Java Solution**

import java.util.ArrayList;

import java.util.List;

class Solution {

    private int[] parent; // Parent array to represent the disjoint set (union-find structure)

    // Function to calculate number of islands after each addLand operation.

    public List<Integer> numIslands2(int m, int n, int[][] positions) {

        parent = new int[m \* n]; // Initialize union-find array, each node is its own parent at start.

        Arrays.fill(parent, -1); // Fill with -1 to indicate water (uninhabited cell).

        int[][] grid = new int[m][n]; // Grid to maintain the state of the land and water.

        int count = 0; // Island count.

        List<Integer> answer = new ArrayList<>();

        int[] directions = {-1, 0, 1, 0, -1}; // Directions for exploring adjacent cells.

        // Iterate over the positions where we need to add land.

        for (int[] pos : positions) {

            int i = pos[0], j = pos[1];

            int index = i \* n + j; // Flatten the 2D position to 1D to use in union-find.

            // If land is already present at the position, skip and record current island count.

            if (grid[i][j] == 1) {

                answer.add(count);

                continue;

            }

            grid[i][j] = 1; // Mark the cell as land.

            parent[index] = index; // Set itself as its parent since it's a new island.

            count++; // Increment island count.

            // Explore all 4 adjacent directions.

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

                int x = i + directions[k];

                int y = j + directions[k + 1];

                int adjacentIndex = x \* n + y;

                // If adjacent cell is within bounds, is land, and is not already unioned with the current cell.

                    union(adjacentIndex, index); // Union the two cells.

                    count--; // Decrement island count as we connected two islands.

                }

            }

            answer.add(count); // Record the current count of islands.

        }

        return answer;

    }

    // Find operation using path compression.

    private int find(int x) {

        if (parent[x] != x) {

            parent[x] = find(parent[x]);

        }

        return parent[x];

    }

    // Union operation to join two elements (here: cells of the grid).

    private void union(int x, int y) {

        int rootX = find(x);

        int rootY = find(y);

        if (rootX != rootY) {

            parent[rootY] = rootX; // Make one root point to the other.

        }

    }

}

**Time and Space Complexity**

The given code defines a class Solution with a method numIslands2 that maintains a dynamic list of the number of islands as new lands are added one by one. The algorithm uses Union-Find to group adjacent lands into a single island.

**Time Complexity**

The time complexity of this algorithm is primarily determined by the number of positions to process and the efficiency of the Union-Find operations. For each position added, the code checks its four adjacent cells (constant time) and potentially performs Union-Find operations.

Each find operation, in the worst case, is O(m \* n), but with path compression (which is applied here), the amortized time complexity becomes near O(1).

Each cell is initially processed once and then may partake in up to 4 union operations if all its neighbors are lands.

Therefore, the overall time complexity can be seen as O(k \* α(m \* n)), where k is the number of positions and α denotes the Inverse Ackermann function, which grows very slowly and is practically considered a constant for all reasonable values of m and n.

**Space Complexity**

The space complexity is determined by the storage required for the grid and the parent array p.

The grid requires O(m \* n) space to represent the entire grid.

The Union-Find structure p also requires O(m \* n) space.

So the total space complexity of the algorithm is O(m \* n), as both the grid and Union-Find structures are linear with respect to the size of the grid.