Number of Islands II

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:

Given `m = 3, n = 3`, `positions = [[0,0], [0,1], [1,2], [2,1]]`.
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
```

We return the result as an array: `[1, 1, 2, 3]`

Challenge:

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

Solution 1

Union Find is an abstract data structure supporting `find` and `unite` on disjointed sets of objects, typically used to solve the network connectivity problem.

The two operations are defined like this:

`find(a,b)` : are `a` and `b` belong to the same set?

`unite(a,b)` : if `a` and `b` are not in the same set, unite the sets they belong to.

With this data structure, it is very fast for solving our problem. Every position is an new land, if the new land connect two islands `a` and `b`, we combine them to form a whole. The answer is then the number of the disjointed sets.

The following algorithm is derived from Princeton's lecture note on Union Find in Algorithms and Data Structures It is a well organized note with clear illustration describing from the naive QuickFind to the one with Weighting and Path compression. With Weighting and Path compression, The algorithm runs in `O((M+N) log* N)` where `M` is the number of operations (unite and find), `N` is the number of objects, `log*` is iterated logarithm while the naive runs in `O(MN)`.

For our problem, If there are `N` positions, then there are `O(N)` operations and `N` objects then total is `O(N log*N)`, when we don't consider the `O(mn)` for array initialization.

Note that `log*N` is almost constant (for `N` = 265536, `log*N` = 5) in this universe, so the algorithm is almost linear with `N`.

However, if the map is very big, then the initialization of the arrays can cost a lot of time when `mn` is much larger than `N`. In this case we should consider using a hashmap/dictionary for the underlying data structure to avoid this overhead.

Of course, we can put all the functionality into the Solution class which will make the code a lot shorter. But from a design point of view a separate class dedicated to the data sturcture is more readable and reusable.

I implemented the idea with 2D interface to better fit the problem.

Java

```java
public class Solution {

    private int[][] dir = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}};

    public List<Integer> numIslands2(int m, int n, int[][] positions) {
        UnionFind2D islands = new UnionFind2D(m, n);
        List<Integer> ans = new ArrayList<>();
        for (int[] position : positions) {
            int x = position[0], y = position[1];
            int p = islands.add(x, y);
            for (int[] d : dir) {
                int q = islands.getID(x + d[0], y + d[1]);
                if (q > 0 && !islands.find(p, q))
                    islands.unite(p, q);
            }
            ans.add(islands.size());
        }
        return ans;
    }
}

class UnionFind2D {
    private int[] id;
    private int[] sz;
    private int m, n, count;

    public UnionFind2D(int m, int n) {
        this.count = 0;
        this.n = n;
        this.m = m;
        this.id = new int[m * n + 1];
        this.sz = new int[m * n + 1];
    }

    public int index(int x, int y) { return x * n + y + 1; }

    public int size() { return this.count; }

    public int getID(int x, int y) {
        if (0 <= x && x < m && 0<= y && y < n)
            return id[index(x, y)];
        return 0;
    }

    public int add(int x, int y) {
        int i = index(x, y);
        id[i] = i; sz[i] = 1;
        ++count;
        return i;
    }

    public boolean find(int p, int q) {
        return root(p) == root(q);
    }

    public void unite(int p, int q) {
        int i = root(p), j = root(q);
        if (sz[i] < sz[j]) { //weighted quick union
            id[i] = j; sz[j] += sz[i];
        } else {
            id[j] = i; sz[i] += sz[j];
        }
        --count;
    }

    private int root(int i) {
        for (;i != id[i]; i = id[i])
            id[i] = id[id[i]]; //path compression
        return i;
    }
}
```

```
//Runtime: 20 ms
```

Python (using dict)

```
class Solution(object):
    def numIslands2(self, m, n, positions):
        ans = []
        islands = Union()
        for p in map(tuple, positions):
            islands.add(p)
            for dp in (0, 1), (0, -1), (1, 0), (-1, 0):
                q = (p[0] + dp[0], p[1] + dp[1])
                if q in islands.id:
                    islands.unite(p, q)
            ans += [islands.count]
        return ans

class Union(object):
    def __init__(self):
        self.id = {}
        self.sz = {}
        self.count = 0

    def add(self, p):
        self.id[p] = p
        self.sz[p] = 1
        self.count += 1

    def root(self, i):
        while i != self.id[i]:
            self.id[i] = self.id[self.id[i]]
            i = self.id[i]
        return i

    def unite(self, p, q):
        i, j = self.root(p), self.root(q)
        if i == j:
            return
        if self.sz[i] > self.sz[j]:
            i, j = j, i
        self.id[i] = j
        self.sz[j] += self.sz[i]
        self.count -= 1

#Runtime: 300 ms
```

written by dietpepsi original link here

Solution 2

This is a basic `union-find` problem. Given a graph with points being added, we can at least solve:

1. How many islands in total?
2. Which island is pointA belonging to?
3. Are pointA and pointB connected?

The idea is simple. To represent a list of islands, we use **trees**. i.e., a list of roots. This helps us find the identifier of an island faster. If `roots[c] = p` means the parent of node c is p, we can climb up the parent chain to find out the identifier of an island, i.e., which island this point belongs to:

```
Do root[root[roots[c]]]... until root[c] == c;
```

To transform the two dimension problem into the classic UF, perform a linear mapping:

```
int id = n * x + y;
```

Initially assume every cell are in non-island set `{-1}`. When point A is added, we create a new root, i.e., a new island. Then, check if any of its 4 neighbors belong to the same island. If not, `union` the neighbor by setting the root to be the same. Remember to skip non-island cells.

UNION operation is only changing the root parent so the running time is `O(1)`.

FIND operation is proportional to the depth of the tree. If N is the number of points added, the average running time is `O(logN)`, and a sequence of `4N` operations take `O(NlogN)`. If there is no balancing, the worse case could be `O(N^2)`.

Remember that one island could have different `roots[node]` value for each node. Because `roots[node]` is the parent of the node, not the highest root of the island. To find the actually root, we have to climb up the tree by calling **findIsland** function.

Here I've attached my solution. There can be at least two improvements: `union by rank` & `pass compression`. However I suggest first finish the basis, then discuss the improvements.

Cheers!

```java
int[][] dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};

public List<Integer> numIslands2(int m, int n, int[][] positions) {
    List<Integer> result = new ArrayList<>();
    if(m <= 0 || n <= 0) return result;

    int count = 0;                      // number of islands
    int[] roots = new int[m * n];       // one island = one tree
    Arrays.fill(roots, -1);

    for(int[] p : positions) {
        int root = n * p[0] + p[1];     // assume new point is isolated island
        roots[root] = root;             // add new island
        count++;

        for(int[] dir : dirs) {
            int x = p[0] + dir[0];
            int y = p[1] + dir[1];
            int nb = n * x + y;
            if(x < 0 || x >= m || y < 0 || y >= n || roots[nb] == -1) continue;

            int rootNb = findIsland(roots, nb);
            if(root != rootNb) {        // if neighbor is in another island
                roots[root] = rootNb;   // union two islands
                root = rootNb;          // current tree root = joined tree root
                count--;
            }
        }

        result.add(count);
    }
    return result;
}

public int findIsland(int[] roots, int id) {
    while(id != roots[id]) id = roots[id];
    return id;
}
```

Path Compression (Bonus)

If you have time, add one line to shorten the tree. The new runtime becomes: `19ms (95.94%)`.

```java
public int findIsland(int[] roots, int id) {
    while(id != roots[id]) {
        roots[id] = roots[roots[id]];   // only one line added
        id = roots[id];
    }
    return id;
}
```

written by yavinci original link here

Solution 3

The basic idea is the Union-Find approach. We assign a root number for each island, and use an array to record this number. For each input, we check its four neighbor cells. If the neighbor cell is an island, then we retrieve the root number of this island. If two neighbor cells belong to two different islands, then we union them and therefore the total number of islands will become one less.

```
public List<Integer> numIslands2(int m, int n, int[][] positions) {
    //use an array to hold root number of each island
    int[] roots = new int[m*n];
    //initialize the array with -1, so we know non negative number is a root numb
er
    Arrays.fill(roots, -1);

    int[] xOffset ={0, 0, 1, -1};
    int[] yOffset = {1, -1, 0, 0};

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

    for(int[] position : positions){
        //for each input cell, its initial root number is itself
        roots[position[0]*n + position[1]] = position[0]*n + position[1];
        //count variable is used to count the island in current matrix.
        //firstly, we assume current input is an isolated island
        int count = result.isEmpty()? 1 : result.get(result.size()-1) + 1;
        //check neighbor cells
        for(int i = 0; i < 4; i++){
            int newX = xOffset[i] + position[0];
            int newY = yOffset[i] + position[1];
            //if we found one neighbor is a part of island
            if(newX >= 0 && newX < m && newY >= 0 && newY < n && roots[newX * n +
newY] != -1){
                //get the root number of this island
                int root1 = find(newX * n + newY, roots);
                //get the root number of input island
                int root2 = roots[position[0]*n + position[1]];
                //if root1 and root2 are different, then we can connect two isola
ted island together,
                // so the num of island - 1
                if(root1 != root2) count--;
                //update root number accordingly
                roots[root1] = root2;
            }
        }
        result.add(count);
    }

    return result;
}

public int find(int target, int[] roots){
    //found root
    if(roots[target] == target) return target;
    //searching for root and update the cell accordingly
    roots[target] = find(roots[target], roots);
    //return root number
    return roots[target];
}
```

written by hpplayer original link here