MEDIUM

Number of Islands - II

Intuition

The question is similar to the Number of Island but we need to answer the queries here.

1	1	1	1	1
1	1	1	1	0
0	0	1	0	0
0	0	1	0	0

0	0	1
0	0	1
1	1	2
1	0	1
0	1	1
0	3	2
1	3	2
0	4	2
3	2	3
2	2	3
1	2	1
0	2	1

(row, col) = row*col

We can either use a visited array so that we do not use a cell twice

Initially :

Row	col	visited	island
0	0	1	1
0	0	1	
1	1	1	1+1 = 2
1	0	1	3-1 = 2-1 = 1 // surrounded by 1 and 2
0	1	1	2-1 = 1 // already connected with the rest
0	3	1	2
1	3	1	3-1 = 2
0	4	1	3-1 = 2

3	2	1	3
2	2	1	4-1 = 3
1	2	1	3-1 = 2-1 = 1
0	2	1	2-1 = 1 // already connected to the rest

Now we can return the result vector having the connections

Approach

- Create a disjoint set having n*m nodes
- Creating a visited vector to mark if the island has been visited or not
- Create a result vector to store the result of the queries
- Creating a counter variable to count the number of islands
- Traverse for all queries :
    - Checking if the node is already visited :
        - Add the count of islands to the result vector
        - Continue for the current iteration
    - Mark the node as visited
    - Increment the counter variable
    - Traverse for adjacent elements :
        - Check for validity of adjacent elements :
            - Check if the element is an island :
                - Calculate the node number as row*m + col
                - Calculate the adjacent node number as adjacent_row*m+adjacent_column
                - Check if parents are not same :
                    - Decrement the counter variable
                    - Perform union for disjoint set
        - Add the counter value to the result vector
    - Return the result vector

Function Code

```
class DisjointSet
{
    public:
    vector<int> parent;
    vector<int> rank;

    // creating a constructor
    DisjointSet(int n)
    {
        rank.resize(n,0);
```

```
        parent.resize(n);
        for(int i=0;i<n;i++)
        {
            parent[i] = i;
        }
    }
    // creating a find parent function
    int findParent(int node)
    {
        if(parent[node]==node)
        {
            return node;
        }
        return parent[node] = findParent(parent[node]);
    }
    // creating a union function
    void unionbyrank(int u,int v)
    {
        int upu = findParent(u);
        int upv = findParent(v);
        // checking if parents are same
        if(upu==upv)
        {
            return;
        }
        if(rank[upu]<rank[upv])
        {
            rank[upv]+=1;
            parent[upu] = upv;
        }
        else if(rank[upu]==rank[upv])
        {
            parent[upu] = upv;
        }
        else
        {
            rank[upu]+=1;
            parent[upv] = upu;
        }
    }
};
class Solution {
  public:
```

```
    vector<int> numOfIslands(int n, int m, vector<vector<int>> &operators)
{
        // creating a disjoint set of number of elements possible
        DisjointSet ds(n*m);
        // creating a visited vector
        vector<vector<int>> visited(n,vector<int>(m,0));
        // creating an island count
        int count = 0;
        // creating a result vector
        vector<int> ans;
        // traversing through all given queries and finding answers
        for(auto it:operators)
        {
            int row = it[0];
            int col = it[1];
            // checking if we have already visited
            if(visited[row][col]==1)
            {
                ans.push_back(count);
                continue;
            }
            // mark the cell as visited
            visited[row][col] = 1;
            // increment the counter
            count++;
            // check if the adjacent are connected
            int delRow[] = {-1,0,1,0};
            int delCol[] = {0,1,0,-1};
            for(int i=0;i<4;i++)
            {
                int nrow = row+delRow[i];
                int ncol = col+delCol[i];
                // checking for validity
                if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
                {
                    // checking if an island
                    if(visited[nrow][ncol]==1)
                    {
                        // finding the node number
                        int nodenumber = row*m+col;
                        int adjnodenumber = nrow*m+ncol;
                        // checking for connectivity
```

```
if(ds.findParent(nodenumber)!=ds.findParent(adjnodenumber))
                        {
                            // decrement the count
                            count-=1;
                            // connect to the disjoint set
                            ds.unionbyrank(nodenumber,adjnodenumber);
                        }
                    }
                }
            }
            // storing the answer
            ans.push_back(count);
        }
        // returning the result vector
        return ans;
    }
};
```

Time Complexity

O((n*m)*log(N))