MEDIUM

Maximum Stone Removal

Intuition

There are n stones at some integer coordinates points on a 2D plane. Each coordinate point may have at most 1 stone

We need to remove some stones

A stone can be removed if it shares either the same row or the same column as another stone that has not been removed

Given : array stones[i] = [xi,yi] represents location of ith stone, return the maximum possible number of stones we can remove

Eg.

X		X	
			X
	X	X	
			X

R		R	
			R
	X	R	
			X

At the end of the day we were able to remove 4 stones at the end of the day

X				
	X			
		X	X	X
		X		

X				
	X			
		R	R	X
		R		

Were able to remove 3 stones maximum

We can try to connect the components with same row and column and suppose if there are n connected stones then we can remove n-1 stones

Eg.

X				
	Y			
		M	M	M
		M		

M = 4 - 1 = 3
X = 1
Y = 1

Therefore we can say that we can find the number of elements in a component and we will be removing size(component)-1

C1	C2	C3
N1	n2	n3

N1-1 + n2-1 + n3-1 + ……. Nx - 1 = (n1+n2+...+nx)-x

This will be our answer

We will be using DSU to find the answer to the particular problem

Approach

- Calculate the maximum size of the component indexes
- Create a disjoint set of size maxRow+maxCol
- Traverse through all stones :
    - Calculate the node row as stones[i][0]
    - Calculate the node column number as stones[i][1]+maxRow+1
    - Perform the union into the disjoint set
    - Mark the row and column to be visited
- Create a variable to count the number of components
- Traverse through all stones :
    - Check if node parent is node itself ie. a new component :
        - Increment count
- Return stones-number of components

Function Code

```
class DisjointSet
{
    public:
    vector<int> parent;
    vector<int> size;
    DisjointSet(int n)
    {
        size.resize(n+1);
        parent.resize(n+1);
        for(int i=0;i<=n;i++)
        {
            parent[i]=i;
            size[i]=1;
        }
    }
    int findParent(int node)
    {
        if(parent[node]==node)return node;
        return parent[node] = findParent(parent[node]);
    }
    void unionbysize(int u,int v)
    {
        int upu = findParent(u);
        int upv = findParent(v);
        if(upu==upv)return;
        if(size[upu]<size[upv])
```

```
        {
            parent[upu]=upv;
            size[upv] += size[upu];
        }
        else
        {
            parent[upv] = upu;
            size[upu]+=size[upv];
        }
    }
};
class Solution {
  public:
    int maxRemove(vector<vector<int>>& stones, int n) {
        // calculating the dimensions of the grid
        int maxRow = 0;
        int maxCol = 0;
        for(auto it: stones)
        {
            maxRow = max(maxRow,it[0]);
            maxCol = max(maxCol,it[1]);
        }
        // creating a disjoint set
        DisjointSet ds(maxRow+maxCol+1);
        // creating an unordered map
        unordered_map<int, int> stoneNodes;
        // traversing for all elements and performing union
        for(auto it:stones)
        {
            // calculating node row number and node column number
            int noderow = it[0];
            int nodecol = it[1]+maxRow+1;
            // performing the union
            ds.unionbysize(noderow,nodecol);
            stoneNodes[noderow] = 1;
            stoneNodes[nodecol] = 1;
        }
        // creating a variable to store the count of components
        int cnt = 0;
        // traversing through stone nodes
        for(auto it: stoneNodes)
        {
            // finding the number of components using the stone parents
```

```
            if(ds.findParent(it.first)==it.first)
            {
                // incrementing the components numbers
                cnt+=1;
            }
        }
        // returning the number of stones - number of components
        return n-cnt;

    }
};
```

Time Complexity

O(N)