**Most Stones Removed with Same Row or Column**

Question

On a 2D plane, we place n stones at some integer coordinate points. Each coordinate point may have at most one stone.

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 an array stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return *the largest possible number of stones that can be removed*.

**Example 1:**

**Input:** stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]

**Output:** 5

**Explanation:** One way to remove 5 stones is as follows:

1. Remove stone [2,2] because it shares the same row as [2,1].

2. Remove stone [2,1] because it shares the same column as [0,1].

3. Remove stone [1,2] because it shares the same row as [1,0].

4. Remove stone [1,0] because it shares the same column as [0,0].

5. Remove stone [0,1] because it shares the same row as [0,0].

Stone [0,0] cannot be removed since it does not share a row/column with another stone still on the plane.

**Example 2:**

**Input:** stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]

**Output:** 3

**Explanation:** One way to make 3 moves is as follows:

1. Remove stone [2,2] because it shares the same row as [2,0].

2. Remove stone [2,0] because it shares the same column as [0,0].

3. Remove stone [0,2] because it shares the same row as [0,0].

Stones [0,0] and [1,1] cannot be removed since they do not share a row/column with another stone still on the plane.

**Example 3:**

**Input:** stones = [[0,0]]

**Output:** 0

**Explanation:** [0,0] is the only stone on the plane, so you cannot remove it.

**Constraints:**

* 1 <= stones.length <= 1000
* 0 <= xi, yi <= 104
* No two stones are at the same coordinate point.

#### **Solution Approach 1: Depth-First Search**

**Intuition**

Let's say two stones are connected by an edge if they share a row or column, and define a connected component in the usual way for graphs: a subset of stones so that there doesn't exist an edge from a stone in the subset to a stone not in the subset. For convenience, we refer to a component as meaning a connected component.

The main insight is that we can always make moves that reduce the number of stones in each component to 1.

Firstly, every stone belongs to exactly one component, and moves in one component do not affect another component.

Now, consider a spanning tree of our component. We can make moves repeatedly from the leaves of this tree until there is one stone left.

**Algorithm**

To count connected components of the above graph, we will use depth-first search.

For every stone not yet visited, we will visit it and any stone in the same connected component. Our depth-first search traverses each node in the component.

For each component, the answer changes by -1 + component.size.

#### Coding Solution

Java

|  |
| --- |
| class Solution {  public int removeStones(int[][] stones) {  int N = stones.length;  // graph[i][0] = the length of the 'list' graph[i][1:]  int[][] graph = new int[N][N];  for (int i = 0; i < N; ++i)  for (int j = i+1; j < N; ++j)  if (stones[i][0] == stones[j][0] || stones[i][1] == stones[j][1]) {  graph[i][++graph[i][0]] = j;  graph[j][++graph[j][0]] = i;  }  int ans = 0;  boolean[] seen = new boolean[N];  for (int i = 0; i < N; ++i) if (!seen[i]) {  Stack<Integer> stack = new Stack();  stack.push(i);  seen[i] = true;  ans--;  while (!stack.isEmpty()) {  int node = stack.pop();  ans++;  for (int k = 1; k <= graph[node][0]; ++k) {  int nei = graph[node][k];  if (!seen[nei]) {  stack.push(nei);  seen[nei] = true;  }  }  }  }  return ans;  }  } |

Python

|  |
| --- |
| class Solution(object):  def removeStones(self, stones):  graph = collections.defaultdict(list)  for i, x in enumerate(stones):  for j in xrange(i):  y = stones[j]  if x[0]==y[0] or x[1]==y[1]:  graph[i].append(j)  graph[j].append(i)  N = len(stones)  ans = 0  seen = [False] \* N  for i in xrange(N):  if not seen[i]:  stack = [i]  seen[i] = True  while stack:  ans += 1  node = stack.pop()  for nei in graph[node]:  if not seen[nei]:  stack.append(nei)  seen[nei] = True  ans -= 1  return ans |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of stones.
* Space Complexity: O(N^2)*O*(*N*2).

#### **Approach 2: Union-Find**

**Intuition**

As in Approach 1, we will need to consider components of an underlying graph. A "Disjoint Set Union" (DSU) data structure is ideal for this.

We will skip the explanation of how a DSU structure is implemented. Please refer to <https://leetcode.com/problems/redundant-connection/solution/> for a tutorial on DSU.

**Algorithm**

Let's connect row i to column j, which will be represented by j+10000. The answer is the number of components after making all the connections.

Note that for brevity, our DSU implementation does not use union-by-rank. This makes the asymptotic time complexity larger.

#### Coding Solution

Java

|  |
| --- |
| class Solution {  public int removeStones(int[][] stones) {  int N = stones.length;  DSU dsu = new DSU(20000);  for (int[] stone: stones)  dsu.union(stone[0], stone[1] + 10000);  Set<Integer> seen = new HashSet();  for (int[] stone: stones)  seen.add(dsu.find(stone[0]));  return N - seen.size();  }  }  class DSU {  int[] parent;  public DSU(int N) {  parent = new int[N];  for (int i = 0; i < N; ++i)  parent[i] = i;  }  public int find(int x) {  if (parent[x] != x) parent[x] = find(parent[x]);  return parent[x];  }  public void union(int x, int y) {  parent[find(x)] = find(y);  }  } |

Python

|  |
| --- |
| class DSU:  def \_\_init\_\_(self, N):  self.p = range(N)  def find(self, x):  if self.p[x] != x:  self.p[x] = self.find(self.p[x])  return self.p[x]  def union(self, x, y):  xr = self.find(x)  yr = self.find(y)  self.p[xr] = yr  class Solution(object):  def removeStones(self, stones):  N = len(stones)  dsu = DSU(20000)  for x, y in stones:  dsu.union(x, y + 10000)  return N - len({dsu.find(x) for x, y in stones}) |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of stones. (If we used union-by-rank, this can be O(N \* \alpha(N))*O*(*N*∗*α*(*N*)), where \alpha*α* is the Inverse-Ackermann function.)
* Space Complexity: O(N)*O*(*N*).

