947. Most Stones Removed with Same Row or Column Medium Depth-First Search Union Find Graph **Hash Table** 

# **Problem Description**

stones can occupy the same coordinate point. A stone can be removed if there is at least one other stone sharing the same row or column on the grid, and that other stone is still on the plane (i.e., it hasn't been removed). Our task is to find out the maximum number of stones that can be removed while following the stated rule. The stones are

In this problem, we're given n stones positioned at integer coordinate points in a 2-dimensional plane. The key rule is that no two

**Leetcode Link** 

represented in an array named stones, with each element stones[i] being a list of 2 integers: [xi, yi], which corresponds to the location of the i-th stone on the 2D plane.

Intuition

one.

the same connected component. In the context of the problem, connected components are groups of stones that are all connected by rows or columns in such a way that if you pick any stone in a connected component, you'd be able to trace to any other stone in the same connected component through shared rows or columns. The last stone in such a connected component can never be removed as there would be no other stone in the same row or column. Now, the key observation is that the maximum number of stones that can be removed is equal to the total number of stones minus

The intuition behind the solution begins by understanding that if two stones share the same row or the same column, they are part of

The provided Python solution employs the union-find (or disjoint-set) data structure to efficiently find connected components. This structure helps to keep track of elements that are connected in some way, and in this case, is used to identify stones that are in the same row or column.

the number of these connected components. To illustrate, imagine you have three stones forming a straight line either horizontally or

vertically. You can remove the two endpoint stones but have to leave the middle one, making the connected components count as

Here's the breakdown of how the union-find algorithm is applied: 1. Each stone is represented by a node, whose initial "parent" is itself, indicating that it is initially in its own connected component. 2. Iterate over all stones. For each stone (x, y), unify the component containing x with the component containing y + n (we offset

3. After all stones have been iterated over, we count unique representatives of the connected components which are the parent

4. The answer is the total number of stones minus the number of unique representatives (connected components).

By applying union-find, the solution achieves a merging process of the stones that lie in the same connected component quickly,

each node, where a node is either a row or a column index.

Initially, every element is set to be its own parent.

nodes for the rows and columns.

y by n to avoid collisions between row and column indices as they could be the same).

- leading to an efficient way to calculate the maximum number of stones that can be removed. Solution Approach
- The solution implements a Union-Find data structure to keep track of connected components. This data structure is particularly useful in dealing with problems that involve grouping elements into disjoint sets where the connectivity between the elements is an essential attribute.

• An array p is created with size 2\*n. This is because we have n possible x-coordinates (0 to n-1) and n possible y-coordinates (0

to n-1), but we need to separate the representation to avoid collision, hence n is doubled. The array p represents the parent of

• The find function is a standard function in Union-Find, which returns the representative (root parent) of the disjoint set that x

If x does not point to itself, we recursively find the parent of x until we reach the root while applying path compression. Path

compression means we set p[x] to its root directly to flatten the structure, which speeds up future find operations on elements

Here's a step-by-step explanation of the solution: Initialization

# The find function

belongs to.

in the same set.

**Unification Process** 

coordinate.

The main loop iterates over every stone.

**Counting Connected Components** 

• For a given stone at coordinates (x, y), we unify the set containing x with the set containing y + n to ensure we don't mix up rows and columns. • The unification is done by setting the parent of the set containing x to be the parent of the set containing y + n using the find function.

• A set s is used to store unique parents of the stones, which comes from applying the find function on every stone's x-

By implementing Union-Find, the solution approach efficiently identifies and unifies stones into connected components and

• len(stones) - len(s) gives us the count of removable stones, satisfying the problem's requirement.

calculates the maximum number of stones that can be removed, leading to an effective strategy for this problem.

• The number of unique parents indicates the number of connected components.

Calculating the Answer Finally, the solution is the total number of stones minus the number of connected components. To see why this is the case, for each connected component, one stone will inevitably remain, making all others removable.

### **Example Walkthrough** Let's consider a small example with 5 stones to illustrate the solution approach.

considering both x and y coordinates.

applying path compression.

3. Unification Process:

like this: stones = [[0,0], [0,2], [1,1], [2,0], [2,2]].

Next, stones[3] = [2,0], we unify 2 with 0 + 5.

Finally, stones [4] = [2,2], we unify 2 with 2 + 5.

def removeStones(self, stones: List[List[int]]) -> int:

if parent[root\_id] != root\_id:

parent = list(range(num\_nodes \* 2))

public int removeStones(int[][] stones) {

for (int[] stone : stones) {

private int find(int x) {

return parent[x];

vector<int> parents;

// Type definition for a 2D point.

function initParents(size: number): void {

for (let i = 0; i < size; i++) {

type Point = [number, number];

const parents: number[] = [];

parents[i] = i;

function find(x: number): number {

**if** (parents[x] !== x) {

const maxCoord = 10010;

initParents(maxCoord << 1);</pre>

stones.forEach(([row, col]) => {

const rowParent = find(row);

parents[rowParent] = colParent;

const uniqueRoots = new Set<number>();

stones.forEach(([row, \_]) => {

Time and Space Complexity

complexity analysis of this code is as follows:

return parents[x];

12 }

13

16

18

19

21

25

26

28

29

30

31

33

34

35

36

37

38

39

40

46

48

47 }

});

20 }

if (parent[x] != x) {

for (int i = 0; i < parent.length; ++i) {</pre>

// of the stone's adjusted column index

parent[find(stone[0])] = find(stone[1] + n);

// since each root represents a connected component

return stones.length - uniqueRoots.size();

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

return parent[root\_id]

Recursive function that finds the root of a disjoint set.

parent[root\_id] = find\_root(parent[root\_id])

parent[find\_root(x)] = find\_root(y + num\_nodes)

unique\_roots = {find\_root(x) for x, \_ in stones}

Path compression is applied to flatten the structure for efficiency.

# Given the constraints of the problem, there can be at most 20000 nodes

# Iterate through each stone and unify their row and column into the same set.

# Use a set comprehension to store the unique roots of all stones' rows and columns.

# Calculate how many stones can be removed by subtracting the number of unique sets

private int[] parent; // Array to represent the parent of each element in the Disjoint Set Union (DSU)

int n = 10010; // Representative value for scaling row and column indices

// the structure of the tree, effectively speeding up future `find` operations.

// Class member to hold the parent pointers for the Union-Find data structure

// Global variable to hold parent pointers for the disjoint-set (Union-Find) data structure.

// Function to initialize 'parents' whereby each element is its own parent.

// Function to find the root of element 'x' with path compression.

22 // Function to remove as many stones as possible while ensuring

// Define a large enough value for the maximum coordinate.

// Using a set to store unique roots after path compression,

// Initialize the 'parents' collection to twice the value of 'maxCoord'.

// at least one stone is left in the same row or column.

function removeStones(stones: Point[]): number {

// Perform union operations for the stones.

const colParent = find(col + maxCoord);

// which indicates the connected components.

parents[x] = find(parents[x]); // Path compression step

parent = new int[n << 1]; // Initialize the parent array for the DSU

parent[i] = i; // Initially, each element is its own parent

# Create an array representing the disjoint set forest with an initial parent of itself.

# because stones could be placed in rows and columns 0 through 9999.

We keep a set s and insert the root parent for each x-coordinate after unification.

the roots for our example, meaning we have 3 connected components.

Steps According to the Solution Approach: 1. Initialization: We create an array p of size 2\*n = 10. This array will help us keep track of the parent of each coordinate

2. The find function: Let's define our find function which takes an index x and recursively finds the representative of x's set, while

Looking at stones[0] = [0,0], we unify the x-coordinate (0) with the y-coordinate (0 + 5) to avoid collision with the x-

• After the unification, we have root parents for the x-coordinates: [0, 1, 2]. Let's assume union-find identifies 0, 1, and 2 as

remove is len(stones) - len(s) which is 5 - 3 = 2. Thus, we can remove a maximum of 2 stones while ensuring that no

Assume we have n = 5 stones at the following coordinates on a 2D plane: [[0,0], [0,2], [1,1], [2,0], [2,2]]. So our stones array looks

- coordinates. Then, stones[1] = [0,2], we unify 0 with 2 + 5. For stones [2] = [1,1], we unify 1 with 1 + 5.
- We have a total of 5 stones, and we've identified 3 connected components. The maximum number of stones that we can

def find\_root(root\_id):

 $num_nodes = 10000$ 

for x, y in stones:

5. Calculating the Answer:

stone is isolated.

removable stones.

Python Solution

class Solution:

8

9

10

11

12

13

14

15

19

20

21

22

23

24

25

11

12

13

21

23

24

25

26

27

28

29

30

31

32

34

33 }

4. Counting Connected Components:

This walkthrough demonstrates the solution approach using the union-find algorithm to efficiently compute the maximum number of stones that can be removed according to the given rules. By identifying the connected components where stones share the same row or column, union-find allows us to easily count these connected components and therefore calculate the maximum number of

26 # (which have to remain) from the total number of stones. 27 return len(stones) - len(unique\_roots) 28 29 # Note: This code assumes that the List class has been imported from the typing module: # from typing import List 31

// Perform union of stone's row and column by setting the parent of the stone's row to the representative

// The number of stones that can be removed is the total number of stones minus the number of unique roots,

// Recursively find the root representative of the element `x`. Path compression is applied here to flatten

```
14
           Set<Integer> uniqueRoots = new HashSet<>(); // Set to store unique roots after unions have been performed
15
           for (int[] stone : stones) {
16
               // Add the representative of each stone's row to the set of unique roots
17
18
               uniqueRoots.add(find(stone[0]));
```

C++ Solution

1 class Solution {

2 public:

Java Solution

class Solution {

```
// Function to remove as many stones as possible while ensuring at least one
 6
       // stone is left in the same row or column
       int removeStones(vector<vector<int>>& stones) {
           // Define a large enough value for the maximum coordinate
           int maxCoord = 10010;
10
           // Resize the parents vector to double the value since we are mapping
11
12
           // 2-D coordinates to a 1-D array
           parents.resize(maxCoord << 1);</pre>
13
           // Initialize the parent of each element to be itself
14
           for (int i = 0; i < parents.size(); ++i) {</pre>
15
                parents[i] = i;
16
17
18
           // Perform union operations for the stones
           for (auto& stone : stones) {
21
               parents[find(stone[0])] = find(stone[1] + maxCoord);
23
24
           // Using a set to store unique roots after path compression
25
           unordered_set<int> uniqueRoots;
26
            for (auto& stone : stones) {
                uniqueRoots.insert(find(stone[0]));
27
28
29
30
           // The number of stones that can be removed is the total number of stones
           // minus the number of unique roots in the disjoint-set forest
31
32
           return stones.size() - uniqueRoots.size();
33
34
35
       // Function to find the root of the element with path compression
       int find(int x) {
36
37
           if (parents[x] != x) {
               parents[x] = find(parents[x]); // Path compression step
38
39
           return parents[x];
40
41
42 };
43
Typescript Solution
```

### uniqueRoots.add(find(row)); 41 42 }); 43 // The number of stones that can be removed is the total number of stones 44 // minus the number of unique roots in the disjoint-set forest. 45 return stones.length - uniqueRoots.size;

**Time Complexity:** 1. The find function is at most traversing the depth of the union-find tree, which, with path compression, results in nearly constant time per operation. However, worst-case without any optimizations can be O(n), but realistically, with path compression, it is closer to  $O(\alpha(n))$ , where  $\alpha(n)$  is the Inverse Ackermann function, which grows very slowly and is practically considered constant

2. There is a loop that runs once for each stone, and inside this loop, two find operations are performed, one for the x-coordinate

Considering there are m stones, each union-find operation is  $O(\alpha(n))$ , the overall time complexity across all stones will be  $O(m * \alpha(n))$ .

and one for the y-coordinate shifted by n to keep the x and y coordinates distinct in the union-find structure.

The given code snippet uses the Union-Find algorithm to remove as many stones as possible on a 2D grid while ensuring at least one

stone is left in a connected group (a group of stones connected by either the same row or the same column). The time and space

# Since the Inverse Ackermann function $\alpha(n)$ is practically constant for all reasonable values of n, the time complexity simplifies to

**Space Complexity:** 

time.

O(m).

- 1. An array p of size n << 1 (or 2 \* n) is created to represent the union-find structure, where n is a constant representing the maximum number of different row/column values to expect, set to 10010 in the code. Hence, the space used by p is O(n).
- 2. A set s is created to store unique roots after path compression. In the worst case, this set will contain all the stones if none share the same row or column, hence an O(m) space complexity at worst.

## Combining these, the total space complexity is O(n + m) which, given the constant size of n, simplifies to O(m), where m is the number of stones.

In summary:

- Time Complexity: O(m), where m is the number of stones. Space Complexity: O(m), where m is the number of stones.