2965. Find Missing and Repeated Values

```
Hash Table
                        Math
                                Matrix
Easy
```

Problem Description

In this problem, we are presented with an n * n 2D integer matrix called grid. Within this matrix, every number from 1 to n^2 is supposed to appear exactly once, representing a perfect sequence with no repetitions and no missing numbers. However, there is a twist: one number, a, appears twice, while another number, b, does not appear at all—it's missing. This means that in the entire grid, there is precisely one number that's duplicated and another that's completely absent.

The challenge is to identify these two numbers: the one that occurred redundantly (a) and the one that's nowhere to be found (b).

We are asked to return these two numbers in the form of a 0-indexed integer array of size 2, where the first element is the repeating number a and the second element is the missing number b.

Intuition

each number within the matrix.

Here's the process in simple steps: Since the numbers range from 1 to n^2, we create a counting array cnt of size n^2 + 1 so that we have a dedicated spot to

To solve this problem, we leverage a very straightforward approach which is to use counting to keep track of the occurrences of

has appeared in the grid.

store the count of each number that should ideally be present in the matrix. By using a 0-indexed array where cnt[i] is meant to store the count for the number i, we reserve the count of number i at the index i. Next, we traverse through the entire grid, row by row and column by column, and increment the count in the cnt array for

each number v that we encounter along the way. The entry at cnt[v] serves as a counter for how many times the number v

After completing the count for all numbers, we check the cnt array for irregularities. There are two specific anomalies we are

interested in: a doubled count (which indicates a number has appeared twice) and a zero count (which indicates a missing number). So for each index i from 1 to n^2, we check the counters: if cnt[i] equals 2, it means that i is the repeating number a and we assign i to the first element of the solution array. Conversely, if cnt[i] equals 0, it means that i is the missing number b, and we accordingly assign i to the second element of the solution array.

My reasoning behind this simple approach is that since each number should only appear once, any deviation from this pattern

can be easily detected by its frequency counter. This method allows us to find both a and b in a single pass through the range of

possible numbers after counting all occurrences. **Solution Approach** The implementation of the solution employs a simple counting algorithm, straightforward data structures, and the pattern of

Initialization of Data Structures: First, we initialize a counting array cnt, which is sized one element larger than n^2 (because

our range is 1 to n² inclusive, and arrays are 0-indexed). We also set up an array ans of size 2 to store our final answer – the

Counting Occurrences: We iterate over each row of the grid using a for loop.

frequency analysis. Here's a detailed walk-through aligned with the code from the Reference Solution Approach:

first element for the repeating number a and the second for the missing number b.

• We increment the count of v in our cnt array. This means for every occurrence of a number, its corresponding index in the cnt array is increased by 1. After this nested loop, cnt[v] will reflect the total number of times the number v has appeared in the grid. Identifying the Repeating and Missing Numbers: ○ We run a loop ranging from 1 to n^2.

• If the count cnt[i] is 2, this means the number i is repeated. We assign this number i to the first element of ans, i.e., ans [0].

frequency of elements and easily find discrepancies. By utilizing array indices that align with the value of the numbers in the grid,

• If the count cnt[i] is 0, this implies the number i was not found in the grid and hence is missing. We assign this number i to the second element of ans, i.e., ans [1].

Inside this loop, we iterate over each value v in the row.

For each i in this range, we perform the following checks:

we eliminate the need for complex data structures or searching algorithms, providing a solution with 0(n^2) time complexity (due to the two nested for loops) and O(n^2) space complexity (due to the cnt array).

0-index which we won't use). The array cnt will look like this after initialization:

Example Walkthrough Let's assume we have a small 2x2 grid (so n=2), which contains the following numbers:

The algorithm makes use of a simple counting technique which is very effective in situations where we need to track the

[[4, 2], [2, 3]] In this grid, the number 2 appears twice (repeating number) and the number 1 is missing. Step by step, we'll apply the solution approach to this grid:

We initialize a counting array cnt of size $2^2 + 1 = 5$ (because our range of numbers is 1 to 4, and we want an extra space for the

cnt = [0, 0, 0, 0, 0]

number at index 1:

cnt = [0, 0, 1, 0, 1]

is missing.

ans = [2, 1]

Python

class Solution:

Solution Implementation

n = len(grid)

for row in grid:

answer = [0, 0]

return answer

Java

C++

public:

#include <vector>

class Solution {

for value in row:

count[value] += 1

and which is missing (count of 0).

answer[0] = i # The repeated value.

answer[1] = i # The missing value.

for i in range(1, n * n + 1):

if count[i] == 2:

elif count[i] == 0:

The size of the grid is n*n.

3. Identifying the Repeating and Missing Numbers

ans = [0, 0]

1. Initialization of Data Structures

2. Counting Occurrences We iterate over each row of the grid and for each number v we encounter, we increment cnt[v]:

Now that we have finished counting, cnt[2] is 2, indicating number 2 has appeared twice, and cnt[1] is 0, showing that number 1

We also initialize the answer array ans of size 2 which will eventually contain the repeating number at index 0 and the missing

• In the second row, we see 2 again (which will be incremented) and 3. The cnt array after processing the second row is: cnt = [0, 0, 2, 1, 1]

```
We loop through the array cnt, starting from index 1 because our numbers also start from 1. For each index i, we check the value
of cnt[i]:
```

After this step, our answer array contains the repeating and missing numbers:

• cnt[1] = 0, which means 1 is missing, so we update ans[1] = 1.

• cnt[2] = 2, which indicates that 2 is repeating, hence we set ans [0] = 2.

• In the first row, we encounter 4 and 2. After processing this row, cnt looks like:

This example illustrates the approach taken in solving the problem: by employing counting to track the frequency of elements and determining any discrepancies, we can find both the duplicating and missing numbers in the grid with relative efficiency and ease.

```
# Create a count array initialized with zeros, of size n*n + 1.
# The extra 1 is to have an index equal to the maximum possible value in the grid.
count = [0] * (n * n + 1)
# Iterate over each row and each value in the grid, incrementing the corresponding count.
```

def findMissingAndRepeatedValues(self, grid: List[List[int]]) -> List[int]:

Initialize an answer list to store the repeated and missing values.

Return the answer list containing the repeated and missing values.

// This function finds the missing and repeated values in a grid, where grid is

// It returns a vector containing two integers, where the first integer is the

std::vector<int> findMissingAndRepeatedValues(std::vector<std::vector<int>>& grid) {

std::vector<int> count(n * n + 1, 0); // Count container to keep track of occurrences of numbers

std::vector<int> answer(2); // Vector to store the answer: [repeated_value, missing_value]

// If the count for this number becomes 2, then we found the repeated number

// repeated value and the second integer is the missing value from the grid.

int n = grid.size(); // Size of the grid (NxN matrix so size is N)

// a vector of vectors of integers, representing a NxN matrix.

// Iterate over the numbers in the current row

// Increment the count for this number

// Find the missing number by looking for a count of 0

for (int number = 1; number <= n * n; ++number) {</pre>

// Iterate over the rows of the grid

for (int number : row) {

count[number]++;

if (count[number] == 0) {

break;

answer[1] = number;

if (count[number] == 2) {

answer[0] = number;

for (auto& row : grid) {

Go through the count array to find which value is repeated (count of 2)

```
class Solution {
   // Method to find the missing and repeated values in a grid.
    public int[] findMissingAndRepeatedValues(int[][] grid) {
       // Calculate the size of the grid.
       int n = grid.length;
       // Initialize a count array to keep track of occurrences of each number.
       int[] count = new int[n * n + 1]; // +1 because we are using 1-based indexing.
       // Array to store the final answer: [repeated number, missing number].
       int[] answer = new int[2];
       // Iterate over the grid to count the occurrences of each number.
        for (int[] row : grid) {
            for (int num : row) {
                // Increment the count of the current number.
                count[num]++;
               // If a number appears twice, it is the repeated number.
               if (count[num] == 2) {
                    answer[0] = num;
       // Look for the missing number in the count array.
       for (int i = 1; ; i++) {
           // If a number has never appeared, it is the missing number.
           if (count[i] == 0) {
               answer[1] = i;
               // Return the answer array with the repeated and missing numbers.
               return answer;
```

```
return answer; // Return the found repeated and missing values
  };
  TypeScript
  function findMissingAndRepeatedValues(grid: number[][]): number[] {
      // Determine the size of the grid (n by n).
      const gridSize = grid.length;
      // Initialize a counter array for numbers 1 to n^2, filled with 0s.
      const count: number[] = Array(gridSize * gridSize + 1).fill(0);
      // Initialize the answer array with two elements for repeated and missing values.
      const result: number[] = Array(2).fill(0);
      // Iterate over each row in the grid.
      for (const row of grid) {
          // Iterate over each element of the current row.
          for (const value of row) {
              // Increment the count for this value.
              count[value]++;
              // If the count reaches 2, we've found the repeated value.
              if (count[value] === 2) {
                  result[0] = value; // Store the repeated value in the result array.
      // Look for the missing value.
      for (let x = 1; x < count.length; x++) {</pre>
          // The missing value will have a count of 0.
          if (count[x] === 0) {
              result[1] = x; // Store the missing value in the result array.
              return result; // Return the result array with both repeated and missing values.
      // Note: The loop is guaranteed to return within the size range of the grid,
      // so no break condition or infinite loop risk is present here.
class Solution:
   def findMissingAndRepeatedValues(self, grid: List[List[int]]) -> List[int]:
       # The size of the grid is n*n.
        n = len(qrid)
       # Create a count array initialized with zeros, of size n*n + 1.
       # The extra 1 is to have an index equal to the maximum possible value in the grid.
        count = [0] * (n * n + 1)
       # Iterate over each row and each value in the grid, incrementing the corresponding count.
        for row in grid:
            for value in row:
                count[value] += 1
```

```
The time complexity of the provided code is indeed 0(n^2). This complexity arises due to the two nested loops, where the first
for loop runs over all rows, and the inner loop over all values in each row, leading to a total of n * n operations, which represents
```

and which is missing (count of 0).

answer[0] = i # The repeated value.

answer[1] = i # The missing value.

for i in range(1, n * n + 1):

if count[i] == 2:

Time and Space Complexity

elif count[i] == 0:

answer = [0, 0]

return answer

Initialize an answer list to store the repeated and missing values.

Return the answer list containing the repeated and missing values.

Go through the count array to find which value is repeated (count of 2)

the total number of elements in the grid. The space complexity of the code is also $0(n^2)$. This is because of the cnt list, which has n * n elements, with each spot

```
representing the count of appearances for each possible value in the range from 1 to n * n.
```