2956. Find Common Elements Between Two Arrays

Hash Table Easy <u>Array</u>

Problem Description

In this problem, we are provided with two arrays nums1 and nums2, each containing a set of integers. These arrays are 0-indexed, meaning that their indices start from 0. The main objective is to determine two specific values: The first value is the number of elements in nums1 that are also present in nums2. This doesn't mean exact index matching; as

- long as a value from nums1 shows up anywhere in nums2, it counts. The second value is similar but in the opposite direction: we count the number of elements in nums2 that are also present in
- nums1. Our goal is to return these two counts as an array containing two integers, with the first value derived from nums1 and the second

from nums2.

ntuition

For this problem, the intuitive approach is to effectively and efficiently determine whether elements of one array occur within the other. We can simplify the task by converting the arrays into sets, which allow us to check for the presence of an element in constant time (ignoring the initial cost of creating the set). This is due to the fact that sets in Python are implemented as hash

We follow these steps: Convert nums1 and nums2 into sets s1 and s2. This not only provides us with quick lookups but also removes duplicates, which

other.

For the first count, we iterate through each element in nums1 (not s1, as we want the original array's indices) and check if it exists in s2. We sum these checks to get the total.

are irrelevant to our counts since we're only interested in whether an element from one array is present at least once in the

For the second count, we perform a similar iteration, but now we loop over nums2 and check for the presence of each element

- in s1. We again sum the checks to get the total. These two sums give us the required values, and we return them as an array of size 2.
- **Solution Approach**

The implementation of the solution uses the hash table data structure in Python, which is represented by the set. Sets are

particularly useful in this scenario because they automatically handle duplicates and allow for efficient membership tests, which is exactly what we need.

tables, providing an average-case time complexity of O(1) for querying the presence of an item.

Convert Lists to Sets: • We begin by converting the input lists nums1 and nums2 into sets, named s1 and s2. Converting to sets eliminates duplicates and enables us

Here is a walkthrough of the algorithm:

to perform quick lookups.

 This is done using the syntax s1, s2 = set(nums1), set(nums2). Count Elements in Intersection: Since we are tasked with counting the elements of nums1 that are present in nums2 and vice versa, we proceed as follows:

• For the first value, we iterate over each element in the original list nums1 using list comprehension and check if the element is in s2 (the set

version of nums2). We use the expression sum(x in s2 for x in nums1) to count the number of True outcomes (which happen when x is in

s2), effectively giving us the count of intersection elements pertaining to nums1.

- ∘ Similarly, for the second value, we iterate over nums2 and check each element's presence in s1. Using the expression sum(x in s1 for x in nums2), we get the count of intersection elements pertaining to nums2. **Return the Result:**
- After calculating both counts, we combine them into a list [sum(x in s2 for x in nums1), sum(x in s1 for x in nums2)] and return this list as our final result. In terms of complexity:

• Creating the sets from the lists has a time complexity of O(n) for nums1 and O(m) for nums2, where n and m are the lengths of the respective lists.

• Thus, the overall time complexity of this solution is O(n + m), which is quite efficient given that the lookups in the sets are O(1) operations on

• The list comprehensions that count the presences will also have a time complexity of O(n) and O(m) for nums1 and nums2 respectively.

- By using sets for membership checks and list comprehensions for the counts, we achieve a concise and efficient implementation that aligns closely with the problem's requirements.
- nums1 = [1, 2, 2, 3]• nums2 = [2, 3, 4, 2]We want to find the number of elements in nums1 that are present in nums2 and vice versa.

First, we convert nums1 and nums2 to sets to remove duplicates and allow efficient lookups. After conversion:

Next, we count the elements in nums1 that are also in s2:

• $s1 = \{1, 2, 3\}$

• $s2 = \{2, 3, 4\}$

Example Walkthrough

average.

• Element 1 in nums1 is not in s2, count = 0 • Element 2 in nums1 is in s2, count = 1

Let's illustrate the solution approach with a small example. Suppose we have two arrays:

twice as it appears twice in the original array.

• Element 4 in nums2 is not in s1, count = 2

We repeat the process for nums2 against s1:

• Element 3 in nums1 is in s2, count = 3

• Element 2 in nums2 is in s1, count = 1 • Element 3 in nums2 is in s1, count = 2

Element 2 in nums2 is in s1, count = 3 (again, we count the duplicate)

We find there are 3 elements in nums2 that are present in nums1.

• Element 2 in nums1 is in s2, count = 2 (repeated element, but we still count it)

Lastly, we return the result as an array: [3, 3], representing the counts as required.

Through this example, we see how the solution approach handles array elements and their presence in the other array by using

// Arrays to store the presence of elements with a maximum value of 100 (since the array indices go from 0 to 100)

So, there are 3 elements in nums1 that are present in nums2. We note that even though '2' is a duplicate in nums1, it is counted

Solution Implementation

Python

Java

class Solution:

Calculate the sum of elements in nums1 that are also in set_nums2 # This counts the number of elements in nums1 present in the intersection intersection_count1 = sum(x in set_nums2 for x in nums1)

intersection_count2 = sum(x in set_nums1 for x in nums2)

The function returns a list with both counts as elements

set_nums1, set_nums2 = set(nums1), set(nums2)

def findIntersectionValues(self, nums1: List[int], nums2: List[int]) -> List[int]:

Convert both lists to sets to remove duplicates and to allow for O(1) lookups

set operations and iteration to count the number of occurrences efficiently.

return [intersection_count1, intersection_count2] # Note: You need to import List from typing to use it as a type hint from typing import List

Calculate the sum of elements in nums2 that are also in set_nums1

This counts the number of elements in nums2 present in the intersection

- // This class represents a solution for finding the intersection values between two arrays. class Solution { // This method finds the intersection values between two integer arrays. public int[] findIntersectionValues(int[] nums1, int[] nums2) {
- // Array to store the intersection count for nums1 and nums2 int[] intersectionCount = new int[2]; // Count the number of items in nums1 that are also in nums2. for (int num : nums1) {

intersectionCount[0] += presenceNums2[num];

intersectionCount[1] += presenceNums1[num];

// Return the counts as an array of two elements.

// Count the number of items in nums2 that are also in nums1.

// Function to find the count of intersection values in both vectors

function findIntersectionValues(nums1: number[], nums2: number[]): number[] {

// Populate the seenInNums1 array based on the values present in nums1.

// Populate the seenInNums2 array based on the values present in nums2.

// Initialize an array to keep the count of values found in both nums1 and nums2.

// Iterate through the range of possible number values to find common values in nums1 and nums2.

// Value i is present in both nums1 and nums2, add it to the intersection array.

const seenInNums1: number[] = new Array(101).fill(0);

const seenInNums2: number[] = new Array(101).fill(0);

if (seenInNums1[i] === 1 && seenInNums2[i] === 1) {

for (const num of nums1) {

for (const num of nums2) {

seenInNums2[num] = 1;

for (let i = 0; i <= 100; i++) {

return intersectionValues;

intersectionValues.push(i);

// Return the array of intersection values.

seenInNums1[num] = 1;

// Initialize two arrays to keep track of the numbers seen in nums1 and nums2.

// Assuming the possible number values range between 0 and 100 (inclusive).

vector<int> findIntersectionValues(vector<int>& nums1, vector<int>& nums2) {

int[] presenceNums1 = new int[101];

int[] presenceNums2 = new int[101];

presenceNums1[num] = 1;

presenceNums2[num] = 1;

for (int num : nums1) {

for (int num : nums2) {

for (int num : nums2) {

return intersectionCount;

// Mark the presence of each element in nums1.

// Mark the presence of each element in nums2.

#include <vector> using std::vector;

public:

class Solution {

C++

```
// Initialize storage for elements' existence, assuming elements range from 0 to 100
       // No extra space is required as the maximum value is known (100)
       int existenceNums1[101]{}; // Initialize all elements to 0 for nums1
        int existenceNums2[101]{}; // Initialize all elements to 0 for nums2
       // Mark the existence of each element from nums1
        for (int num : nums1) {
            existenceNums1[num] = 1;
       // Mark the existence of each element from nums2
        for (int num : nums2) {
            existenceNums2[num] = 1;
       // Vector to store the result, which will hold two values
       vector<int> intersectionCount(2);
       // Calculate the intersection count for nums1 by checking existence in nums2
        for (int num : nums1) {
            intersectionCount[0] += existenceNums2[num]; // If a number exists in nums2, increment the count
       // Calculate the intersection count for nums2 by checking existence in nums1
        for (int num : nums2) {
            intersectionCount[1] += existenceNums1[num]; // If a number exists in nums1, increment the count
       // Return the resulting counts as a vector
       return intersectionCount;
};
```

```
// It appears the original intent was to have 2 elements, but the purpose is unclear from the context.
// The usage seems incorrect for finding intersection values.
// Instead, let's return a list of unique values which are present in both nums1 and nums2.
const intersectionValues: number[] = [];
```

TypeScript

```
class Solution:
   def findIntersectionValues(self, nums1: List[int], nums2: List[int]) -> List[int]:
       # Convert both lists to sets to remove duplicates and to allow for O(1) lookups
        set_nums1, set_nums2 = set(nums1), set(nums2)
       # Calculate the sum of elements in nums1 that are also in set_nums2
       # This counts the number of elements in nums1 present in the intersection
        intersection_count1 = sum(x in set_nums2 for x in nums1)
       # Calculate the sum of elements in nums2 that are also in set_nums1
       # This counts the number of elements in nums2 present in the intersection
        intersection_count2 = sum(x in set_nums1 for x in nums2)
       # The function returns a list with both counts as elements
       return [intersection count1, intersection count2]
# Note: You need to import List from typing to use it as a type hint
from typing import List
Time and Space Complexity
  The given code determines the number of intersection values between two lists, nums1 and nums2. The computational complexity
  is as follows:
```

Time Complexity The time complexity of the code is O(n + m), where n is the length of nums1 and m is the length of nums2.

Initially, we convert both lists nums1 and nums2 into sets, which takes O(n) and O(m) time respectively. Then we compute the sum of the boolean expressions checking if each element of nums1 is in s2 and if each element of nums2 is in s1. Both of these

operations are done in 0(n) and 0(m) respectively, since set lookup is 0(1) on average. Hence, in total, the time complexity is O(n) + O(m) for both conversions and the sum operations. **Space Complexity**

The space complexity of the code is 0(n + m) as well. This is because it creates two additional sets from the lists nums 1 and nums2, holding up to n and m unique elements respectively, assuming the worst case where all elements in the lists are unique. There is no other significant use of additional space.