350. Intersection of Two Arrays II

Two Pointers

Binary Search

Problem Description

Hash Table

The problem provides us with two integer arrays, nums1 and nums2, and asks us to find the intersection of these two arrays. The intersection consists of elements that appear in both arrays. If an element occurs multiple times in both arrays, it should appear in the final result as many times as it is common to both. The result can be returned in any order.

Sorting

To elaborate, we need to identify all the unique elements that the two arrays have in common. Then, for each unique element found in both arrays, we need to include that element in our result array as many times as it occurs in both. For example, if the

number 3 appears twice in nums1 and four times in nums2, it should appear twice in the intersection array since that's the minimum count between the two arrays.

The foundation of the solution is to count the frequency of each element in the first array, which is nums1. We then iterate through

Intuition

Easy

the second array, nums2, checking if the elements appear in the counter we created from nums1. If an element from nums2 is found in the counter and has a count greater than zero, it is part of the intersection. We add this element to the result list and decrease its count in the counter by one to ensure that we don't include more occurrences of an element than it appears in both arrays. Here's how we arrive at the solution step by step:

Count Elements of nums1: By creating a counter (a specific type of dictionary) for nums1, we efficiently track how many times

Iterate Over nums2 and Collect Intersection: We go through each element in nums2. If the element is found in the counter with

each element occurs.

- a non-zero count, this indicates that the element is both in nums1 and nums2. Add to Result and Update Counter: For every such element found, we add it to our result list and then decrement the count
- for that element in the counter to ensure we only include as many instances as are present in nums1. The above steps are simple and use the property of counters to help us easily and efficiently find the intersection of the two
- **Solution Approach**

The implementation of the solution makes use of the following concepts: • Counter (from collections module): This is used to construct a hash map (dictionary) that counts the frequency of each element in nums1. It's a subclass of a dictionary which is specifically designed to count hashable objects.

The code implementation goes as follows:

res = []

arrays.

We start by importing Counter from the collections module.

from collections import Counter

We initialize an empty list res which will hold the elements of the intersection.

counter = Counter(nums1)

intersection.

res.append(num)

counter[num] -= 1

if counter[num] > 0:

Check if num exists in the counter and its count is greater than 0. This confirms whether num should be a part of the

Next, we iterate over each element num in nums2. During each iteration, we perform the following actions:

If the above condition is true, we append num to our res list. This adds num to our intersection list.

We then use Counter to create a frequency map of all the elements present in nums1.

- Then we decrement the count of num in the counter by one to ensure we do not include it more times than it appears in nums1.
- Finally, once we've completed iterating over nums2, we return the res list which contains the intersection of nums1 and nums2. return res
- conditional check and decrement operation during the iteration are constant time (O(1)) since hash map access is in constant time.

The choice of Counter and the decrementing logic ensures that each element is counted and included in the result only as many

times as it is present in both input arrays. This method is efficient because creating a counter is a linear operation (O(n)) and

iterating through the second list is also linear (O(m)), where n and m are the sizes of nums1 and nums2 respectively. The

• nums1 = [1,2,2,1]• nums2 = [2,2]We want to find the intersection of these two arrays. Following the described solution approach: Counter is used to create a frequency map for nums1. After this step, we have:

Since counter[2] is indeed greater than 0, we append 2 to our res list. Then, we decrement the count of 2 in the counter by

b. We move to the next element which is again 2. We perform the same check and find that counter[2] is still greater than 0.

4. We return the res list which is [2, 2] as the final result. This reflects the intersection of nums1 and nums2, indicating that the element 2 appears

The final intersection array is [2, 2], and it is derived through the efficient counting and iteration method described in the

counter = {1: 2, 2: 2}

res.append(2)

res = [2, 2]

Python

Java

counter[2] -= 1

res = []

counter = Counter([1,2,2,1])

Example Walkthrough

We initialize an empty list res to store the intersection.

The counter represents that the number 1 appears twice and the number 2 appears twice in nums1.

a. First, we check if 2 (the first element of nums2) exists in counter and has a count greater than 0.

Let's use a small example to illustrate the solution approach. Assume we have the following two arrays:

```
res.append(2)
counter[2] -= 1
```

So, we append this 2 to res and again decrement the counter for 2.

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

If the current element is present in the element counter

and the count is more than 0, it's part of the intersection.

Decrement the count of the current element in the counter

Iterate through each element in the second list

intersection_result.append(num)

Return the final list of intersection elements

element_counter[num] -= 1

Append the element to the result list

if element_counter[num] > 0:

return intersection_result

We iterate over each element in nums2.

one to reflect the updated count.

Now, counter = $\{1: 2, 2: 1\}$

Finally, counter = {1: 2, 2: 0} Since there are no more elements in nums2 to iterate over, our res list currently looks like this:

in both arrays and does so exactly twice, which is the minimum number of times it appears in any one array.

```
solution approach.
Solution Implementation
```

from collections import Counter

for num in nums2:

return result;

C++

public:

};

TypeScript

#include <vector>

class Solution {

using namespace std;

#include <unordered_map>

from typing import List

class Solution:

Count the occurrences of each element in the first list element_counter = Counter(nums1) # Initialize the result list which will store the intersection intersection_result = []

```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
       // Create a map to store the count of each number in nums1
       Map<Integer, Integer> numberCounts = new HashMap<>();
       // Iterate over the first array and fill the numberCounts map
        for (int number : nums1) {
           // Increment the count for the current number in the map
           numberCounts.put(number, numberCounts.getOrDefault(number, 0) + 1);
       // List to store the intersection elements
       List<Integer> intersectionList = new ArrayList<>();
       // Iterate over the second array to find common elements
        for (int number : nums2) {
           // If the current number is in the map and count is greater than 0
           if (numberCounts.getOrDefault(number, 0) > 0) {
                // Add the number to the intersection list
                intersectionList.add(number);
                // Decrement the count for the current number in the map
                numberCounts.put(number, numberCounts.get(number) - 1);
       // Convert the list of intersection elements to an array
       int[] result = new int[intersectionList.size()];
        for (int i = 0; i < result.length; ++i) {</pre>
            result[i] = intersectionList.get(i);
```

```
// Create a hash map to store the frequency of each element in nums1.
unordered_map<int, int> element_count_map;
// Iterate over the first array (nums1)
// and populate the map with the count of each element.
for (int num : nums1) {
    ++element_count_map[num]; // Increment the count for the number.
// Create a vector to store the result (intersection elements).
vector<int> intersection_result;
// Iterate over the second array (nums2).
for (int num : nums2) {
   // Check if the current number exists in the map (from nums1)
   // and it has a non-zero count, meaning it's a common element.
    if (element_count_map[num] > 0) {
        --element_count_map[num]; // Decrement the count.
        intersection_result.push_back(num); // Add to the intersection result.
// Return the result of the intersection.
return intersection_result;
```

// Function to find the intersection of two arrays.

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

function intersect(numbers1: number[], numbers2: number[]): number[] {

const numberFrequencyMap = new Map<number, number>();

// Initialize an array to store the intersection

// then add it to the intersection array

intersectionArray.push(number);

// Iterate over the second array to find common elements

// If the number is in the map and the count is not zero,

// Decrease the count of the number in the map

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

Count the occurrences of each element in the first list

for (const number of numbers1) {

const intersectionArray = [];

for (const number of numbers2) {

element_counter = Counter(nums1)

return intersection_result

Time and Space Complexity

// Create a map to keep a count of each number in the first array

// Populate the frequency map with the count of each number in numbers1

numberFrequencyMap.set(number, (numberFrequencyMap.get(number) ?? 0) + 1);

if (numberFrequencyMap.has(number) && numberFrequencyMap.get(number) !== 0) {

numberFrequencyMap.set(number, numberFrequencyMap.get(number) - 1);

// Return the final array containing the intersection of both input arrays

```
// Return the intersection array
return intersectionArray;
```

class Solution:

from collections import Counter

from typing import List

```
# Initialize the result list which will store the intersection
intersection_result = []
# Iterate through each element in the second list
for num in nums2:
    # If the current element is present in the element counter
    # and the count is more than 0, it's part of the intersection.
    if element_counter[num] > 0:
        # Append the element to the result list
        intersection_result.append(num)
        # Decrement the count of the current element in the counter
        element_counter[num] -= 1
# Return the final list of intersection elements
```

1. Building a counter from nums1 takes O(n) time, where n is the length of nums1, as each element is processed once.

Time Complexity

nums1.

2. Iterating over nums2 and updating the counter takes O(m) time, where m is the length of nums2, as each element is processed once. Hence, the overall time complexity is 0(n + m), where n is the length of nums1 and m is the length of nums2.

The time complexity of the given code can be analyzed in two parts:

Space Complexity

The space complexity of the code depends on the space used by the counter data structure: The counter keeps track of elements from nums1, therefore it uses O(n) space, where n is the unique number of elements in

The res list contains the intersected elements. In the worst case, if all elements in nums2 are present in nums1, it can take O(min(n, m)) space, where n is the length of nums1 and m is the length of nums2.

Overall, the space complexity of the code is O(n) where n represents the number of unique elements in nums1.

Taking both into account, the space complexity is $0(n + \min(n, m))$ which simplifies to 0(n) as $\min(n, m)$ is bounded by n.