2570. Merge Two 2D Arrays by Summing Values Two Pointers

Problem Description

Hash Table

element represents an identifier (id) and the second element represents a value associated with that id. Both nums1 and nums2 are sorted in ascending order by the ids, and each id is unique within its own array. The goal is to merge nums1 and nums2 into a new array while following these rules: 1. Incorporate only the ids present in at least one of the input arrays into the result.

In this problem, we are given two 2D arrays, nums1 and nums1 array consists of subarrays where the subarray's first

2. Each id should appear only once in the merged array.

3. The value corresponding to each id in the merged array should be the sum of the values associated with the id from both arrays. If an id is

ntuition

present in only one array, its value in the other array is assumed to be 0.

The result should be an array of [id, value] pairs, sorted in ascending order by id.

To solve this problem, one intuitive approach is to treat it as a frequency counting problem combined with summation for each

unique id. Since we want to sum the values associated with an id across both arrays, we can iterate through each array, and for

where an id is present in only one array or both arrays; if it's present in both, the sum in the counter will naturally reflect the total sum across both arrays. The Python Counter class from the collections module is well-suited for this task. It allows us to keep track of the sum of values for each unique id as we iterate through nums1 + nums1 + nums1 + nums2 = nums2<a href="n Counter object which behaves like a dictionary where each key is the id and the value is the calculated sum.

each [id, value] pair encountered, increase the counter for that id by the value. This approach immediately takes care of cases

After processing all the ids, the solution involves converting the Counter object into a list of [id, value] pairs and sorting that list by id. Sorting is necessary because the Counter does not maintain the original order of the input, and the final result needs to be sorted by id.

The sorted(cnt.items()) function call transforms the Counter object into a sorted list of tuples based on the ids, directly

producing the desired output format for this problem, thereby completing the problem's requirements in an efficient and concise manner. **Solution Approach**

The implementation of the solution is fairly straightforward once the intuition is understood. The solution makes use of the Counter class from Python's collections module, which provides a convenient way to tally counts for each unique id. Here is a step-by-step breakdown of the code implementation:

Initial Counter Creation: A Counter object is initialized with no arguments, which will be used to keep a running total of the

Accumulating Counts: The solution then iterates over the concatenation of nums1 + nums2 arrays. This effectively allows the

During each iteration, the code accesses the value v associated with the id i and adds it to the count already present for i

values associated with each id.

cnt = Counter()

return sorted(cnt.items())

arrays efficient.

Example Walkthrough

nums1 = [[1, 3], [2, 2]]

nums2 = [[1, 1], [3, 3]]

iteration over all id-value pairs in both arrays as if they were in a single list. for i, v in nums1 + nums2: cnt[i] += v

```
in the Counter. If i is not present, Counter automatically starts counting from zero.
Returning the Sorted Result: Since the Counter object cnt is not ordered, and the problem expects the result to be sorted
by id, we must sort cnt.items(). The items() method returns a list of (id, sum_value) tuples.
```

The use of the Counter here is a clever application of a frequency counting pattern which, combined with the properties of the Counter class (like returning 0 for missing items instead of raising a KeyError), makes the accumulation of sums across both

The algorithm effectively operates in O(N log N) time due to the sorting operation at the end, where N is the total number of id-

value pairs in the concatenated list. The linear iteration over the concatenated list to populate the Counter has a time complexity

of O(M) where M is the size of the concatenated array (i.e., the sum of the lengths of nums1 and nums2), but since M ≤ 2N and

The sorted() function by default sorts the list of tuples by the first element of each tuple, which is the id in our case.

```
sorting dominates the complexity, O(N log N) is the practical complexity of the entire operation.
As such, this approach is an effective and elegant solution to the problem, neatly covering all edge cases thanks to the properties
of the Counter class and the sorted list of items.
```

up with a new array that includes the sum of values for each unique id. **Creating a Counter:** We create an empty **Counter** to collect the sums:

Accumulate Counts: Next, iterate over both arrays and use the **Counter** to add the values associated with each id:

Both nums1 and nums2 are sorted by ids and each id is unique within its array. The goal is to merge the arrays such that we end

For nums1 cnt[1] += 3 # The counter is now Counter({1: 3})

For nums2

the arrays.

requirements.

from typing import List

class Solution:

Java

C++

#include <vector>

class Solution {

using namespace std;

return merged;

const maxIndex = 1001:

TypeScript

class Solution {

from collections import Counter

count = Counter()

for index, value in nums1 + nums2:

for (int[] pair : nums1) {

count[pair[0]] += pair[1];

int[][] merged = new int[size][2];

if (count[i] > 0) {

 $cnt[3] += 3 \# Adds \ a \ new \ id \ 3, \ so \ Counter(\{1: 4, 2: 2, 3: 3\})$

Returning Sorted Result: Finally, we sort cnt.items() to get the sorted list by id:

sorted_result = sorted(cnt.items()) # This gives [(1, 4), (2, 2), (3, 3)]

Let's consider two sample arrays to walk through the solution approach:

cnt = Counter() # This is empty initially

 $cnt[2] += 2 \# The counter is now Counter({1: 3, 2: 2})$

 $cnt[1] += 1 \# Adds to the existing id 1, so <math>Counter(\{1: 4, 2: 2\})$

```
Now we have the merged array sorted_result which holds the unique ids with their summed values, sorted by id.
 As a result, the output will be:
[[1, 4], [2, 2], [3, 3]]
```

This output represents the sums of values associated with each id from both input arrays. Id 1 has a total value of 4 (since it's in

both nums1 and nums2), id 2 has a value of 2 (only in nums1), and id 3 has a value of 3 (only in nums2). This walkthrough

illustrates how the algorithm effectively combines the values associated with each id and sorts them to satisfy the problem

By doing this for all id-value pairs in both nums1 and nums1 and nums2, we store the sum of values for each id present in either or both of

Solution Implementation **Python**

count[index] += value # Convert the counter back to a sorted list of lists. Sorting by the first element of each pair (the index) return sorted(count.items())

Sum the values for each unique index entry

public int[][] mergeArrays(int[][] nums1, int[][] nums2) {

// Create the answer array based on the calculated size.

for (int i = 0, index = 0; i < count.length; ++i) {

// Populate the answer array with the numbers and their frequencies.

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

Initialize a Counter object to keep track of the frequencies of integers

// Iterate over the first array of pairs and update the count for each number.

Iterate through each pair in the concatenated list of nums1 and nums2

```
// Iterate over the second array of pairs and similarly update the count.
for (int[] pair : nums2) {
    count[pair[0]] += pair[1];
// Determine the size of the resulting merged array by counting non-zero frequencies.
int size = 0;
for (int freq : count) {
    if (freq > 0) {
        size++;
```

int[] count = new int[1001]; // Array to hold the frequency of numbers, assuming the range is 0-1000.

merged[index++] = new int[] {i, count[i]};// Using 'index' to fill the 'merged' array.

return merged; // Return the merged array with numbers and corresponding frequencies.

```
public:
   // Function to merge two arrays of key-value pairs where the keys are integers
   // and the values are the counts of those integers.
    vector<vector<int>> mergeArrays(vector<vector<int>>& nums1, vector<vector<int>>& nums2) {
       // Initialize a frequency array to store the counts for each number
       // as the maximum possible key is 1000 according to the problem statement.
        int frequency[1001] = {0};
       // Increment the counts for the numbers in the first array
        for (auto& pair : nums1) {
            frequency[pair[0]] += pair[1];
       // Increment the counts for the numbers in the second array
        for (auto& pair : nums2) {
            frequency[pair[0]] += pair[1];
       // The answer array to hold the merged key-value pairs.
       vector<vector<int>> merged;
       // Iterate over the frequency array.
        for (int i = 0; i < 1001; ++i) {
            // Check if there is a non-zero count for the current number.
            if (frequency[i] > 0) {
                // Push the key-value pair to the merged list,
```

```
countArray[index] += value;
// Initialize an array to hold the results
const mergedArray: number[][] = [];
```

// Increment the count for this index by the value

// Increment the count for this index by the value

// Iterate through the count array to construct the mergedArray

if (countArray[i] > 0) { // If any count exists for this index

const countArray = new Array(maxIndex).fill(0);

// Iterate over the first array of number pairs

// Iterate over the second array of number pairs

for (const [index, value] of nums1) {

for (const [index, value] of nums2) {

for (let i = 0; i < maxIndex; ++i) {</pre>

countArray[index] += value;

// where the first element is the number (key)

merged.push_back({i, frequency[i]});

function mergeArrays(nums1: number[][]. nums2: number[][]): number[][] {

// Initialize a count array with zeroes based on the maximum index

// and the second element is the count (value).

// Return the merged key-value pairs as a sorted array since we iterated in order.

// Define the maximum value for the index (1001 assumed as per the original code)

```
// Add a pair of [index, count] to the mergedArray
           mergedArray.push([i, countArray[i]]);
    // Return the mergedArray containing the sum of pairs from both input arrays
    return mergedArray;
from typing import List
from collections import Counter
class Solution:
   def mergeArrays(self, nums1: List[List[int]], nums2: List[List[int]]) -> List[List[int]]:
       # Initialize a Counter object to keep track of the frequencies of integers
        count = Counter()
       # Iterate through each pair in the concatenated list of nums1 and nums2
        for index, value in nums1 + nums2:
           # Sum the values for each unique index entry
           count[index] += value
       # Convert the counter back to a sorted list of lists. Sorting by the first element of each pair (the index)
       return sorted(count.items())
```

The time complexity of the code provided is O(N log N), where N is the total number of elements in the nums1 and nums2 arrays

Time and Space Complexity

combined. This is because, while the for-loop runs in O(N) and counter operations are on average O(1) for each, the dominant factor is the sorting operation. Sorting in Python typically uses Timsort, which has a time complexity of O(N log N) in the worst case. The space complexity of the code is O(M), where M is the number of unique keys across both nums1 and nums2. A counter object

is used to track the sums associated with each unique key. Although the lists are merged, there is no duplication of the original lists' contents, so the space required is related only to the number of unique keys. Each update to the counter is 0(1) space, thus the overall space complexity depends on the number of unique keys rather than the total number of elements.