2657. Find the Prefix Common Array of Two Arrays

Problem Description

Hash Table

Medium <u>Array</u>

exactly once in both arrays, but the order of numbers might differ between A and B. The task is to construct a new array C, referred to as the "prefix common array," where each C[i] represents the total count of numbers that are present both in the A array and the B array up to the index i (including i itself). In other words, at each index i, you count how many numbers from both A[0...i] and B[0...i] have been encountered thus far and represent the same set.

In this problem, you are given two integer arrays A and B, both of which are permutations of integers from 1 to n (inclusive),

representing that they include all integers within this range without repeats. This means each number from 1 through n appears

The goal is to return this "prefix common array" C by comparing elements at corresponding indices of the given permutations A and B.

Intuition

The solution builds upon the idea of incrementally computing the intersection count of numbers between two permutations A

and B up to a certain index. Since A and B are permutations of the same length containing all numbers from 1 through n, we know that every number will eventually appear. The approach uses two counters, cnt1 and cnt2, to track the frequency of numbers appeared in A and B, respectively, as we go through the arrays. It employs a for loop to go through A and B simultaneously with the help of the zip function. At every step

of this loop, we increment the count of the respective current number from A in cnt1 and from B in cnt2.

unique numbers) in cnt1. For each key x in cnt1, we determine the minimum occurrence value between cnt1[x] and cnt2[x], because commonality requires the number to appear in both permutations up to the current index, and its count in the common prefix array will be the minimum of the its occurrences in A and B so far.

After updating the counts, we calculate the intersection up to the current index by iterating over the keys (which represent

Adding these minimum values together gives the total count of common numbers up to index i, which gets appended to the array ans. The process repeats for each index, finally leading to a complete ans array that acts as the required "prefix common array." Solution Approach

iterating through the permutations A and B. Here's how the algorithm unfolds: Initialization: Create empty lists ans, cnt1, and cnt2. Here, ans will store the final response, being the "prefix common array". cnt1 and cnt2 are counters in the form of dictionaries (from the collections module in Python) that will keep track

The implementation of the solution follows a step-by-step approach to build the "prefix common array" ans progressively by

Simultaneous Traversal: Using the zip() function to simultaneously iterate over both A and B. The zip() function pairs the items of A and B with the same indices together so that they can be processed in pairs.

of the frequency of each number in A and B, respectively.

permutations. This value is appended to the ans list.

directly calculate the size of the intersection up to each index.

We start with the first elements in A and B, which are 1 and 2.

Count Incrementation: On each iteration of the for loop, for the current elements a from A and b from B, the algorithm updates cnt1[a] and cnt2[b]. This action increments the counts of the elements within our counters cnt1 and cnt2, corresponding to the number of occurrences so far.

Calculating Intersection: After incrementing the occurrence counts for the latest elements, the next task is to compute the intersection size. The algorithm uses a comprehension together with the sum() function to calculate the sum of minimum counts for each unique number that has appeared up to the current index i. The minimum is taken between cnt1[x] and

Appending to ans: The value calculated in the previous step reflects the total count of common numbers at index i in the

Return Result: After the loop terminates, the ans list, which contains the prefix intersection size at each index, is returned as

cnt2[x] for each number x, meaning the number of times x has appeared in both A and B up to i.

Iterate Until the End: Repeat steps 3 to 5 until the algorithm reaches the end of the arrays A and B.

it represents the solution to the problem. The algorithm efficiently computes the intersection sizes using dictionary-based counters, offering a dynamic approach to tracking the elements as we traverse the permutations. The use of a for loop along with zip() ensures that we are always

comparing the correct pair of elements from A and B with the corresponding index. Summing the minimum counts allows us to

- Let's illustrate the solution approach with a small example. Suppose we have two integer arrays A and B which are permutations
- We want to build the prefix common array C. Let's go step by step: Initialization: We create empty ans, cnt1, and cnt2 lists/dictionaries. **Simultaneous Traversal**: We begin iterating over both A and B using the zip() function.

Example Walkthrough

A = [1, 3, 2]

B = [2, 1, 3]

of integers from 1 to 3:

Count Incrementation for index 0:

We increment cnt1[1] and cnt2[2].

Continue Traversal for index 1:

We increment cnt1[3] and cnt2[1].

We update cnt1[2] and cnt2[3].

Calculating Intersection:

• Finally, ans = [0, 1, 3].

Solution Implementation

from collections import Counter

count 1 = Counter()

count_2 = Counter()

count 1[a] += 1

 $count_2[b] += 1$

from typing import List

result = []

class Solution:

Return Result:

10.

12.

Python

Java

C++

public:

#include <vector>

class Solution {

#include <algorithm>

using namespace std;

class Solution {

Calculating Intersection:

We append 0 to ans.

Appending to ans:

 \circ Now, ans = [0, 1].

Now, ans = [0].

 \circ Now, cnt1 = {1: 1} and cnt2 = {2: 1}.

 We determine the minimum count for each number that has appeared. So far, 1 has not appeared in B and 2 has not appeared in A, thus the common count is 0.

- **Calculating Intersection:** At this point, 1 is the only common number appearing in both A and B.
- We append the common count 1 to ans. Appending to ans:

We have finished iterating through both arrays, and the completed ans list is [0, 1, 3].

def findThePrefixCommonArray(self, array 1: List[int], array 2: List[int]) -> List[int]:

Increment the count of the current element 'a' in array_1's counter

Increment the count of the current element 'b' in array_2's counter

Calculate the total number of common elements by iterating through each

total_common = sum(min(count, count_2[element]) for element, count in count_1.items())

element in the first array's counter and summing up the minimum count

int n = A.length; // Get the length of the array, assumed to be of same length.

int[] countA = new int[n + 1]; // Arrav to count occurrences in A, 1-indexed.

int[] countB = new int[n + 1]; // Array to count occurrences in B, 1-indexed.

int[] ans = new int[n]; // Array to store the count of common elements for each prefix.

// Calculate the number of common elements for each prefix (up to the current i).

ans[i] += Math.min(countA[j], countB[j]); // Add the minimum occurrences among both arrays.

Final Traversal for index 2: For the last element, we have 2 from A and 3 from B.

Now we take the second elements 3 from A and 1 from B.

 \circ Now, cnt1 = {1: 1, 3: 1} and cnt2 = {2: 1, 1: 1}.

• The total common count is the sum of occurrences of each number, which is 3. Appending to ans:

Each number has appeared once in both A and B.

 \circ Now, cnt1 = {1: 1, 3: 1, 2: 1} and cnt2 = {2: 1, 1: 1, 3: 1}.

By following these steps, we built the prefix common array for permutations A and B using the algorithm. Each element of ans indicates the intersection size of A and B up to that index, which fulfills the goal of the problem.

This ans list is the prefix common array C that we wanted to construct.

Initialize the result list to store the common prefix counts

Iterate through both arrays simultaneously using zip

This will process the arrays as pairs of elements (a, b)

occurring in both arrays (element-wise minimum)

// Function to find the prefix common element count between two arrays.

public int[] findThePrefixCommonArray(int[] A, int[] B) {

// Iterate over the arrays A and B simultaneously.

for (int i = 1: i <= n: ++i) {

return ans; // Return the array of counts.

// Function to find the prefix common element count array

// Initialize the array to store the result

// Iterate through elements of arrays A and B

for (let i = 1: i <= length: ++i) {

for a, b in zip(array 1, array 2):

count 1[a] += 1

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

++countA[A[i]];

++countB[B[i]];

from collections import Counter

return result;

from typing import List

result = []

const result: number[] = new Array(length).fill(0);

// Increment the count for the current elements in A and B

// Add the minimum occurrence of the current element in A and B to result

def findThePrefixCommonArray(self, array 1: List[int], array 2: List[int]) -> List[int]:

Increment the count of the current element 'a' in array_1's counter

Increment the count of the current element 'b' in array_2's counter

// Return the result array containing counts of common elements for each prefix

Initialize the result list to store the common prefix counts

Iterate through both arrays simultaneously using zip

This will process the arrays as pairs of elements (a, b)

// Check for common elements upto the current index

result[i] += Math.min(countA[j], countB[j]);

vector<int> findThePrefixCommonArray(vector<int>& arrA, vector<int>& arrB) {

in array 1 and array_2, respectively

for a, b in zip(array 1, array 2):

Create two Counter objects to keep track of the counts of elements

Append the total number of common elements to the result list result.append(total_common) # Return the final result list containing the common prefix counts return result

```
for (int i = 0; i < n; ++i) {
   // Count the occurrences of each element in both arrays.
   ++countA[A[i]];
   ++countB[B[i]];
```

```
int size = arrA.size();
        vector<int> prefixCommonCount(size); // Result array to hold prefix common counts
        vector<int> countArrA(size + 1, 0); // Count array for elements in arrA
        vector<int> countArrB(size + 1, 0); // Count array for elements in arrB
        // Iterate through each element in the input arrays
        for (int i = 0; i < size; ++i) {
            // Increment the count of the current elements in arrA and arrB
            ++countArrA[arrA[i]];
            ++countArrB[arrB[i]];
            // Calculate the common elements count for the prefix ending at index i
            for (int i = 1; i <= size; ++i) {</pre>
                // Add the minimum occurrence count of each element seen so far in both arrays
                prefixCommonCount[i] += min(countArrA[j], countArrB[j]);
        // Return the resulting prefix common count array
        return prefixCommonCount;
};
TypeScript
function findThePrefixCommonArray(A: number[], B: number[]): number[] {
    // Determine the length of the arrays
    const length = A.length;
    // Initialize count arrays for both A and B with zeroes
    const countA: number[] = new Array(length + 1).fill(0);
    const countB: number[] = new Array(length + 1).fill(0);
```

Create two Counter objects to keep track of the counts of elements # in array 1 and array_2, respectively count 1 = Counter() count_2 = Counter()

class Solution:

count_2[b] += 1 # Calculate the total number of common elements by iterating through each # element in the first array's counter and summing up the minimum count # occurring in both arrays (element-wise minimum) total_common = sum(min(count, count_2[element]) for element, count in count_1.items()) # Append the total number of common elements to the result list result.append(total_common) # Return the final result list containing the common prefix counts return result Time and Space Complexity

Time Complexity The given code has a time complexity of O(n^2). This is due to the nested loop implicitly created by the sum function, which

sums over items of cnt1 inside the for a, b in zip(A, B) loop. Since the sum operation has to visit each element in the cnt1 dictionary for every element of arrays A and B, the total number of operations will be proportional to n^2, where n is the length

of arrays A and B. **Space Complexity** The space complexity is O(n) because we use two Counter objects cnt1 and cnt2 that, in the worst-case scenario, may contain

as many elements as there are in arrays A and B, which leads to a linear relationship with n. Additionally, an answer array ans of

size n is maintained.