1213. Intersection of Three Sorted Arrays

Easy Array Hash Table Binary Search Counting

Problem Description

The goal is to identify common integers that are present in three different sorted arrays `arr1`, `arr2`, and `arr3`. These arrays are given to be sorted in strictly increasing order, which means there are no duplicate elements within each array and each subsequent element is greater than the previous one. We need to find a list of integers that are present in all three arrays and return this list in sorted order. Since the arrays are already sorted, the resulting list of common elements will also be automatically sorted.

Intuition

To find the common elements across the three sorted arrays, we could consider each element in the first array `arr1` and check if it is present in both `arr2` and `arr3`. This approach would take O(n1 * (n2 + n3)) time, where n1, n2, and n3 are the lengths of `arr1`, `arr2`, and `arr3` respectively, because we would search for each element of `arr1` in both `arr2` and `arr3`.

A more efficient way to solve this would be to use the three-pointer technique, where we can iterate through all three arrays simultaneously to find common elements. This approach takes advantage of the fact that the arrays are sorted. However, the reference solution approach suggests "Binary search".

Binding together the sorted arrays property and the proposed "Binary search" approach would involve iterating through one array, and for each element in that array, perform binary search in the other two arrays. This efficiently checks for the presence of the element in the other arrays since the arrays are sorted. Binary search greatly reduces the time complexity to O(n1 * log(n2) * log(n3)).

The solution given in the code seems to use a different approach: a frequency counter. By concatenating the arrays and counting the frequencies of the elements, we can simply look for those numbers in `arr1` (or any of the three arrays) that have a frequency of 3. This works under the assumption that the input arrays are strictly increasing and therefore there are no duplicates within the same array. Hence, an element with a count of 3 must appear once in each of the arrays. This approach has a time complexity of O(n1 + n2 + n3), the time it takes to combine the arrays and count the elements.

In summary, while the reference suggests to use binary search to enhance the searching step by capitalizing on the sorted property of arrays, the implemented solution uses a frequency counter for its simplicity and because it also leads to an efficient solution given the constraints of the problem.

Solution Approach

The solution provided in the code snippet is straightforward, and it differs from the binary search approach originally suggested. Here's the thought process behind the implementation given the reference to use a binary search approach:

1. **Binary Search**: Since the arrays are sorted, for each element in `arr1`, we can perform a binary search in `arr2` and `arr3`. If the element is found in both other arrays, then it's added to our result list. Binary search is preferred because it can drastically reduce the search time in a sorted array from linear (O(n)) to logarithmic (O(log n)).
2. **Counter for Frequency**: The actual code uses the `Counter` from Python's `collections` module. A `Counter` is a subclass of `dict` designed to count hashable objects. It's an unordered collection where elements are stored as dictionary keys and their counts as dictionary values.
3. **Concatenation and Counting**: The solution concatenates `arr1`, `arr2`, and `arr3` using the `+` operator. Once concatenated, a `Counter` object is created which tallies the count of each unique element across the three arrays.
4. **Filtering Common Elements**: We now have the frequency of each number that occurs in the three arrays. Since we're interested in numbers that appear in all three arrays, we filter out the elements which have a frequency count of 3. The code snippet does this using a list comprehension `[x for x in arr1 if cnt[x] == 3]`. Here, `cnt[x]` retrieves the count for element `x` from the `Counter` object.
5. **Sorted Output**: The given code doesn't explicitly sort the output list because it's already guaranteed to be sorted. This is because we are iterating through `arr1` (which is sorted) and we're only picking those elements that are common to all arrays, hence maintaining the original order.

The binary search approach (not implemented in the solution code), if exampled, would require iterating through `arr1` and for each element 'x', perform binary search on `arr2` and `arr3`. If 'x' is found in both these arrays, it indicates that 'x' is common across all three arrays and should be added to the result.

However, the provided solution is likely more efficient since:

1. Counting elements across all arrays is done in linear time relative to the total number of elements.
2. No additional logarithmic factor is introduced as it would be with binary searches.
3. It makes use of the problem constraint (no duplicates within the same array) to simplify the solution.

Example Walkthrough

To illustrate the solution approach, consider the following small example:

Let `arr1 = [1,2,3]`, `arr2 = [1,3,5]`, and `arr3 = [1,3,7]`.

Now, let's step through the mixed solution approach using a frequency counter as implemented in the code snippet (despite binary search being advised):

1. **Concatenation and Counting**: We start by concatenating `arr1`, `arr2`, and `arr3` to get `[1,2,3,1,3,5,1,3,7]`.
2. **Using Counter**: A `Counter` object is created, and it counts the frequency of each element in the combined list: `{1: 3, 2: 1, 3: 3, 5: 1, 7: 1}`.
3. **Filtering Common Elements**: Given the frequency count, we're interested in elements with a count of 3, indicating they are present in all three arrays. We iterate through `arr1` and use a list comprehension to find the common elements `[x for x in arr1 if cnt[x] == 3]`, resulting in `[1,3]`, since those elements have a frequency of 3.
4. **Sorted Output**: The final result is `[1,3]`, which is the list of common integers in all three arrays.

The advantage of this solution is that it operates in linear time relative to the sum of elements in all arrays without needing to sort since the arrays are already sorted. Furthermore, it avoids the potentially more complex implementation and additional search time complexity of using binary search for each element.

Solution Implementation

Python

```python
from typing import List
from collections import Counter

class Solution:
    def arrays_intersection(self, arr1: List[int], arr2: List[int], arr3: List[int]) -> List[int]:
        # Create a counter object which will store the frequency of each number across all three arrays
        elements_count = Counter(arr1 + arr2 + arr3)

        # Find the intersection by checking which numbers have a count equal to 3 - meaning they appear in all three arrays
        intersection = [number for number in arr1 if elements_count[number] == 3]

        # Return the list of numbers that are present in all three arrays
        return intersection
```

Java

```java
class Solution {

    // Method to find the intersection of three sorted arrays
    public List<Integer> arraysIntersection(int[] arr1, int[] arr2, int[] arr3) {
        // List to store the common elements across the three arrays
        List<Integer> intersection = new ArrayList<>();
        // Counter array to store the frequency of elements (assuming all elements are <= 2000)
        int[] count = new int[2001];

        // Increment the counter for every element in arr1
        for (int num : arr1) {
            count[num]++;
        }

        // Increment the counter for every element in arr2
        for (int num : arr2) {
            count[num]++;
        }

        // Check arr3 and add the number to the result list if it occurs in all three arrays
        for (int num : arr3) {
            // If the count reaches 3, it means the element is present in all three arrays
            if (++count[num] == 3) {
                intersection.add(num);
            }
        }

        // Return the list of common elements
        return intersection;
    }

}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Function to find the intersection of three sorted arrays.
    // The intersection contains elements that appear in all three arrays.
    std::vector<int> arraysIntersection(std::vector<int>& arr1, std::vector<int>& arr2, std::vector<int>& arr3) {
        // Create a result vector to store the common elements.
        std::vector<int> result;

        // Initialize a count array to store the frequency of each element.
        // The problem statement ensures no element is greater than 2000.
        int count[2001] = {}; // Initialize all counts to zero.

        // Increment the count for every element in arr1
        for (int elem : arr1) {
            ++count[elem];
        }

        // Increment the count for every element in arr2
        for (int elem : arr2) {
            ++count[elem];
        }

        // Checking the elements of arr3:
        // If the count becomes 3, it means the element is common to arr1, arr2, and arr3.
        for (int elem : arr3) {
            if (++count[elem] == 3) {
                result.push_back(elem); // Add the common element to the result vector.
            }
        }

        // Return the vector containing the intersecting elements.
        return result;
    }
};
```

TypeScript

```typescript
// Declare three arrays to hold the sorted input arrays.
let arr1: number[];
let arr2: number[];
let arr3: number[];

// Function to find the intersection of three sorted arrays.
// The intersection contains elements that appear in all three arrays.
function arraysIntersection(arr1: number[], arr2: number[], arr3: number[]): number[] {
    // Create an array to store the result of the intersection.
    let result: number[] = [];

    // Create an array to count the occurrences of each element.
    // The problem statement ensures no element is greater than 2000,
    // so we initialize an array of length 2001 to count all possible elements.
    let count: number[] = new Array(2001).fill(0);

    // Increment the count for each element in arr1.
    arr1.forEach(elem => {
        count[elem]++;
    });

    // Increment the count for each element in arr2.
    arr2.forEach(elem => {
        count[elem]++;
    });

    // Loop through the elements in arr3 and check if the count is 2 before the loop,
    // If it becomes 3 after incrementing, that means the element
    // is common to arr1, arr2, and arr3.
    arr3.forEach(elem => {
        if (++count[elem] === 3) {
            result.push(elem); // Add the common element to the result array.
        }
    });

    // Return the array containing the intersecting elements.
    return result;
}

// Example usage:
// arr1 = [1, 2, 3];
// arr2 = [1, 3, 4];
// arr3 = [1, 3, 5];
// const intersection = arraysIntersection(arr1, arr2, arr3);
// console.log(intersection); // Should log [1, 3] to the console.
```

```python
from typing import List
from collections import Counter

class Solution:
    def arrays_intersection(self, arr1: List[int], arr2: List[int], arr3: List[int]) -> List[int]:
        # Create a counter object which will store the frequency of each number across all three arrays
        elements_count = Counter(arr1 + arr2 + arr3)

        # Find the intersection by checking which numbers have a count equal to 3 - meaning they appear in all three arrays
        intersection = [number for number in arr1 if elements_count[number] == 3]

        # Return the list of numbers that are present in all three arrays
        return intersection
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n + m + l)`, where `n` is the length of `arr1`, `m` is the length of `arr2`, and `l` is the length of `arr3`. This is because the code concatenates all three arrays, which takes `O(n + m + l)` time, and then constructs a `Counter` object from the concatenated array, which also takes `O(n + m + l)` time as each element is processed once. Finally, a list comprehension is used which goes through each element in `arr1` (`O(n)` time) and checks the count, which is an `O(1)` operation due to the hash table lookup in `Counter`. Therefore, the overall time complexity is dominated by the concatenation and the Counter operation.

Space Complexity

The space complexity of the code is `O(n + m + l)` because we are creating a `Counter` object that contains all the elements from `arr1`, `arr2`, and `arr3`. This requires space proportional to the total number of elements in all three arrays. The list comprehension does not add additional space as it only contains the elements that are common in all three arrays (at most `O(n)`), but this does not change the overall space complexity dominated by the `Counter`.