2248. Intersection of Multiple Arrays

Easy Array Hash Table Counting Leetcode Link

Problem Description

In this problem, we are given a 2D array `nums`, where each subarray `nums[i]` consists of distinct positive integers. The goal is to find all the integers that are common in each of these subarrays and return a sorted list of these common elements. To clarify, an integer is part of the result list if and only if it is present in every subarray of `nums`.

For example, if `nums = [[4,9,5],[9,4,9,8,4],[4,6,8,9]]`, the integers `4` and `9` are present in all the subarrays, so the sorted result would be `[4,9]`.

Intuition

The intuition behind the solution is based on the concept of counting frequency. Since we are looking for integers that are present in all arrays, we can keep a counter for each number to track how many times it appears across all arrays. If a number appears in each subarray, its count should be equal to the number of subarrays in `nums`.

Here is the step-by-step approach:

1. Initialize a counter array `cnt` which has a fixed size of `1001` to cover the potential range of values in `nums` subarrays as specified by the problem (assuming the maximum possible integer is `1000`).
2. Iterate over each subarray in `nums` and then iterate over each integer within that subarray and increment its corresponding count in `cnt`.
3. After counting, iterate through the counter array `cnt` and pick those integers whose counter value is equal to the length of `nums`, which means these integers have appeared in each subarray.
4. Collect these common integers and return them in a sorted order. It is implicit that the integers are already in sorted order because we increment the counts while iterating over the fixed range of indices from `1` to `1000`. Thus, the resulting list of common integers is inherently sorted.
5. The final result is a list of integers that appear in every subarray of `nums`, sorted in ascending order.

The solution effectively reduces the problem to a simple frequency counting mechanism and leverages the fixed range of possible integer values to ensure sorted output.

Solution Approach

The given solution uses a simple counting approach using an array to keep track of the frequency of integers across all subarrays in `nums`. Let's break down the implementation into detailed steps:

1. A counter array `cnt` is initialized with `1001` elements, all set to `0`. This array acts as a hash table to keep track of the frequency of each integer since we know the integers are positive and the constraint hint at a maximum integer value of `1000`.
2. The outer `for` loop iterates through each subarray `arr` within the main array `nums`. For each subarray, we iterate through its elements using the inner `for` loop.
3. Inside the inner `for` loop, we use the integer `x` from the subarray as an index to increment the counter at that index in `cnt` array by `1`. This step counts the occurrence of each integer across all subarrays.

   ```
   for arr in nums:
       for x in arr:
           cnt[x] += 1
   ```

4. After filling the `cnt` array with counts, a list comprehension is used to iterate over all possible integers (using `enumerate(cnt)` which gives us both the number `x` and its count `v`), checking if the count `v` is equal to the length of `nums`. This ensures that we only select integers that are present in all subarrays.

   ```
   return [x for x, v in enumerate(cnt) if v == len(nums)]
   ```

5. The list comprehension also takes care of assembling the output list and the nature of enumeration guarantees the order will be ascending, hence satisfying the problem requirements of returning the common elements in sorted order.

The data structures utilized here are:

- An array (`cnt`) used as a frequency counter.
- A list comprehension to generate the output list.

The algorithm is a well-known counting sort technique which is very efficient when the range of potential values is known and reasonably small, as it is in this case. As a result, the complexity of this solution is very good, with time complexity being `O(N * M)` where `N` is the number of subarrays and `M` is the average length of these subarrays, and space complexity is `O(1)` since the size of the `cnt` array is fixed and does not grow with the input size.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following 2D array:

```
nums = [[1, 2, 3], [2, 3, 4], [3, 4, 5, 6]]
```

The aim is to find all numbers that appear in every subarray.

1. Initialize a counter array `cnt` with size `1001`, all elements set to `0`.
2. We go through each subarray `arr` in `nums`. For the first subarray `[1, 2, 3]`, we increment `cnt[1]`, `cnt[2]`, and `cnt[3]` by `1`. After processing the first subarray, `cnt` looks like this:

   ```
   cnt = [..., 0, 1, 1, 1, 0, 0, ...]  (positions 1, 2, and 3 are incremented)
   ```

3. Repeat step 2 for the second subarray `[2, 3, 4]`. Now, `cnt[2]`, `cnt[3]`, and `cnt[4]` are incremented by `1`, resulting in:

   ```
   cnt = [..., 0, 1, 2, 2, 1, 0, ...]  (positions 2, 3, and 4 are incremented)
   ```

4. Do the same for the third subarray `[3, 4, 5, 6]`, incrementing `cnt[3]`, `cnt[4]`, `cnt[5]`, and `cnt[6]`. After this step, `cnt` is:

   ```
   cnt = [..., 0, 1, 2, 3, 2, 1, 1, 0, ...]  (positions 3, 4, 5, and 6 are incremented)
   ```

5. After processing all subarrays, we iterate through the counter array `cnt` and look for values that are equal to the length of `nums` (which is `3` in this case). We find that `cnt[3]` is `3`.
6. The final output list will only include the number `3` as it is the only number whose count is equal to the number of subarrays in `nums`. We return `[3]`.

The solution code that accomplishes this is:

```
cnt = [0] * 1001
for arr in nums:
    for x in arr:
        cnt[x] += 1
return [x for x, v in enumerate(cnt) if v == len(nums)]
```

And our example will yield the output:

```
[3]
```

This simple example demonstrates how the counting approach finds common elements present in all subarrays reliably and efficiently.

Python Solution

```python
from typing import List

class Solution:
    def intersection(self, nums: List[List[int]]) -> List[int]:
        # Initialize a list to count the occurrences of each number, assuming numbers range from 0 to 1000.
        # This means each index represents a number, and the value at that index represents its count.
        count = [0] * 1001

        # Loop through each list in the list of lists `nums`.
        for num_list in nums:
            # Using a set to avoid counting duplicates in the same list.
            unique_nums = set(num_list)
            # Increment the count for each number found in the list.
            for num in unique_nums:
                count[num] += 1

        # Return a list of numbers (index) where the count is equal to the length of `nums`
        # i.e., the number appeared in all lists.
        return [num for num, frequency in enumerate(count) if frequency == len(nums)]

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<Integer> intersection(int[][] arrays) {
        // Array to store the count of each element (assuming the range of elements is 0-1000)
        int[] count = new int[1001];

        // Iterate through each sub-array
        for (int[] array : arrays) {
            // Count each element in the sub-array
            for (int element : array) {
                ++count[element];
            }
        }

        // List to store the result (elements present in all sub-arrays)
        List<Integer> result = new ArrayList<>();

        // Iterate through the count array to find elements with a count equal to the number of arrays,
        // which means they appear in every sub-array.
        for (int i = 0; i < 1001; ++i) {
            if (count[i] == arrays.length) {
                result.add(i); // Add to the result list if element is present in all arrays
            }
        }

        // Return the result list
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the required header for the vector container.

class Solution {
public:
    // The function 'intersection' takes a vector of vector of ints as input.
    // It returns a vector of integers that are common in all inner vectors.
    vector<int> intersection(vector<vector<int>>& nums) {
        // We use an array to keep count of integer occurrences across all inner vectors.
        int countArray[1001] = {}; // There are 1001 elements initialized to 0.

        // Iterate over the outer vector 'nums', which contains inner vectors.
        for (auto& innerArray : nums) {
            // For each integer in the inner vectors, increment its count in 'countArray'.
            for (int num : innerArray) {
                countArray[num]++;
            }
        }

        // Define a vector to hold the intersection results.
        vector<int> intersectionResult;

        // Iterate over the 'countArray' to check which numbers have a count
        // equal to the size of the outer vector (i.e., they appear in all inner vectors).
        for (int i = 0; i < 1001; ++i) {
            if (countArray[i] == nums.size()) {
                intersectionResult.push_back(i); // Add the number to the result vector.
            }
        }

        // Return the resulting intersection vector.
        return intersectionResult;
    }
};

```

Typescript Solution

```typescript
function intersection(nums: number[][]): number[] {
    // Create an array to keep track of the count of each number across all subarrays
    const count = new Array(1001).fill(0);

    // Iterate over each subarray in nums
    for (const subArray of nums) {
        // Iterate over each number in the subarray
        for (const num of subArray) {
            // Increment the count for this number
            count[num]++;
        }
    }

    // Prepare an array to hold the numbers present in all subarrays
    const result: number[] = [];

    // Iterate over the possible numbers
    for (let i = 0; i < 1001; i++) {
        // If the count of a number is equal to the length of nums,
        // it means the number is present in all subarrays
        if (count[i] === nums.length) {
            result.push(i);
        }
    }

    // Return the result array containing the intersection
    return result;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n * k + m)`, where `n` represents the number of arrays in `nums`, `k` is the average length of the arrays in `nums`, and `m` is the fixed size of counting array `cnt` (1001 in this case). The first term `n * k` comes from the nested loops where we iterate over all elements across all the arrays. The second term `m` is due to the list comprehension at the end, which iterates over the whole `cnt` array once.

The space complexity of the code is `O(m)`. This is because we use a fixed-size array `cnt` to count occurrences of each number (between 0 and 1000, inclusively). Space complexity does not depend on the input size, only on the size of the counting array `cnt`.