2799. Count Complete Subarrays in an Array

Medium | Array | Hash Table | Sliding Window | Leetcode Link

Problem Description

In this problem, you are provided with an array called `nums`, which contains positive integers. Your task is to count what's known as "complete" subarrays within this array. A subarray is considered "complete" if it contains exactly the same distinct elements as are present in the entire array. To clarify, a subarray is a contiguous sequence of elements within the array. The main goal is to find the number of such unique complete subarrays.

Intuition

The intuition behind solving this problem lies in recognizing that a complete subarray must end at some point where all distinct elements in `nums` are included up to that point. The idea is to move through the array with a sliding window, tracking the frequency of elements in the current window using a counter.

Once a window contains all distinct elements (i.e., the length of the counter is equal to the total number of distinct elements in `nums`), we can say each subarray starting from the current beginning of the window (`i`) to the current end of the window (`j`) is a complete subarray. This is because the end, `j`, includes all unique elements, so any subarray from `i` to `j` will also include these elements.

Additionally, we know that for a fixed endpoint `j`, if a window is complete, extending that window to the right will also result in complete subarrays (as they'll also contain all distinct elements). Therefore, the number of complete subarrays ending at `j` can be determined by counting all possible starting points from the current start of the window `i` to `j`, which will be `n - j` (where `n` is the length of `nums`).

As we slide the window to the right incrementally, once a window no longer remains complete (an element's count drops to 0), we exit the inner loop, and proceed with expanding the end of the window (`j`) again to find a new complete subarray. By moving the start of the window (`i`) appropriately and using this approach, we ensure that we consider all complete subarrays exactly once.

Solution Approach

The solution follows a two-pointer technique, commonly used for problems involving contiguous subarrays or sequences. Here's a step-by-step explanation of how the algorithm is implemented:

1. First, we need to determine the total number of distinct elements in the entire array `nums`. We use a set to find distinct elements and store the count in the variable `cnt`.

```
cnt = len(set(nums))
```

2. A counter `d` from the `collections` module is used to keep track of the frequency of elements within the current window as we scan through the array.

```
d = Counter()
```

3. Two pointers are used: `i` for the start of the window and `j` for the end of the window. The variable `ans` is used to store the total number of complete subarrays found so far, and `n` represents the length of the array `nums`.

```
ans, n = 0, len(nums)
i = 0
```

4. The algorithm iterates over the elements of `nums` using a for loop, with `j` acting as the window's end.

```
for j, x in enumerate(nums):
```

5. Inside the loop, the counter `d` is updated with the current element `x`.

```
d[x] += 1
```

6. A while loop is used to check whether the current window is complete (i.e., the length of `d` equals `cnt`). If it is complete, we add `n - j` to `ans`, because all subarrays starting from `i` to `j` are complete subarrays.

```
while len(d) == cnt:
    ans += n - j
```

7. To move the window forward, we decrease the count of the element at the start of the window and, if its count reaches zero, remove it from the counter.

```
d[nums[i]] -= 1
if d[nums[i]] == 0:
    d.pop(nums[i])
```

8. After updating the counter, we increment `i` to shrink the window from the left.

```
i += 1
```

By using the two-pointer technique with the counter, we can efficiently check for complete subarrays and count them. The algorithm guarantees that no complete subarrays are missed and none are counted more than once, ensuring the correct result.

Finally, the value of `ans` reflects the total count of complete subarrays within `nums`, which is what we are asked to return.

```
return ans
```

Example Walkthrough

Let's illustrate the solution approach with a small example.

Consider the array `nums = [1, 2, 1, 3, 2]`. We want to find all "complete" subarrays where a "complete" subarray contains exactly the distinct elements present in the entire array `nums`. In this example, the distinct elements are `1`, `2`, `3`, so every complete subarray must contain each of these numbers at least once.

Step 1: Determine the number of distinct elements. For `nums`, this is `cnt = 3` (`{1, 2, 3}`).

Step 2: Initialize the counter `d` and other variables `ans = 0`, `n = 5`, and `i = 0`.

Step 3 & 4: Start iterating over the elements with two pointers `i` and `j`.

- For `j = 0`, `nums[j] = 1`. Update `d` with this element `d[1] = 1`.
- For `j = 1`, `nums[j] = 2`. Update `d` with this element `d[2] = 1`.
- For `j = 2`, `nums[j] = 1`. Update `d` with this element `d[1] = 2`.
- Up until now, `len(d) != cnt`, so no complete subarray here.

Step 5 & 6: Continue iterating.

- For `j = 3`, `nums[j] = 3`. Update `d` with this element `d[3] = 1`.
- Now `len(d) == cnt`, we have a complete window from `i = 0` to `j = 3`. Each subarray starting from `i` to `j` is complete. Since `n - j = 5 - 3 = 2`, there are two subarrays `[1, 2, 1, 3]` and `[2, 1, 3]`.

Step 7 & 8: To move the window:

- while `len(d) == cnt`, we decrease the count of `nums[i]` which is `nums[0] = 1`. Since `d[1]` becomes `1` (not zero), we don't remove it.
- Increment `i` to `1`. The window is now from `i = 1` to `j = 3` and `len(d) == cnt` still holds, so we add `n - j = 2` more subarrays `[2, 1, 3]`, `[1, 3]`.

Continue this process:

- Move `i` to `2`, `d[nums[1]]` which is `d[2]` is decremented, the count is now zero, and we remove `2` from `d`.
- The window is no longer complete (`len(d) != cnt`), so break the `while` loop.

Step 4: Increment `j` to `4`, and continue the process by filling up the counter again until we find new complete subarrays.

By continuously moving `i` and `j` and checking the completeness of the window, we can find all complete subarrays. In this example, the complete subarrays would be `[1, 2, 1, 3]`, `[2, 1, 3]`, `[1, 3]`, `[2, 1, 3, 2]`, `[1, 3, 2]`.

The count `ans` is incremented each time we find a new complete subarray. After iterating through `nums`, we return the value of `ans` as the total count of complete subarrays.

Python Solution

```
from collections import Counter

class Solution:
    def countCompleteSubarrays(self, nums):
        # Count the number of unique elements in nums
        unique_count = len(set(nums))
        # Initialize a counter to keep track of the frequency of elements
        elem_freq = Counter()
        # Initialize the answer and set the length of the nums array
        total_subarrays, length = 0, len(nums)
        # Start pointer for the sliding window
        start_index = 0

        # Iterate over nums with an end pointer for the sliding window
        for end_index, num in enumerate(nums):
            # Update the frequency count of the current element
            elem_freq[num] += 1
            # Shrink the window from the left if all unique elements are included
            while len(elem_freq) == unique_count:
                # Add the number of complete subarrays ending at the current end_index
                total_subarrays += length - end_index
                # Decrease the freq count of the element at the start of the window
                elem_freq[nums[start_index]] -= 1
                # Remove the element from the counter if its count drops to zero
                if elem_freq[nums[start_index]] == 0:
                    elem_freq.pop(nums[start_index])
                # Move the start pointer to the right
                start_index += 1

        # Return the total number of complete subarrays
        return total_subarrays
```

Java Solution

```
class Solution {
    public int countCompleteSubarrays(int[] nums) {
        // Create a map to store the frequency of each array
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        // Initialize the map with the unique numbers in the array
        for (int num : nums) {
            frequencyMap.put(num, 1);
        }

        // Store the size of the unique elements in the array
        int uniqueCount = frequencyMap.size();
        // Variable to hold the final result
        int answer = 0;
        // Length of the nums array
        int arrayLength = nums.length;

        // Clear the map for reuse
        frequencyMap.clear();

        // Sliding window approach
        for (int left = 0, right = 0; right < arrayLength; ++right) {
            // Add or update the count of the current element
            frequencyMap.merge(nums[right], 1, Integer::sum);
            // If the window contains all unique elements
            while (frequencyMap.size() == uniqueCount) {
                // Update the answer with the number of subarrays ending with nums[right]
                answer += arrayLength - right;
                // Move the left pointer, decrementing the frequency of the left-most element
                if (frequencyMap.merge(nums[left], -1, Integer::sum) == 0) {
                    // If the count goes to zero, remove the element from the map
                    frequencyMap.remove(nums[left]);
                }
                ++left;
            }
        }

        // Return the total count of complete subarrays
        return answer;
    }
}
```

C++ Solution

```
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Method to count the number of complete subarrays
    int countCompleteSubarrays(vector<int>& nums) {
        // Create a map to store the unique elements and their counts
        unordered_map<int, int> countMap;

        // Initial loop to count the unique elements in the array
        for (int num : nums) {
            countMap[num] = 1;
        }

        // Store the size of the map, which is the count of unique elements
        int uniqueCount = countMap.size();

        // Clear the map to reuse it for counting in subarrays
        countMap.clear();

        // Variable to store the answer which is the number of complete subarrays
        int completeSubarrays = 0;

        // Size of the input array
        int arraySize = nums.size();

        // Two-pointer approach to find all complete subarrays
        for (int start = 0, end = 0; end < arraySize; ++end) {
            // Increase the count of the current end element in the map
            countMap[nums[end]]++;

            // When the map size equals the count of unique elements, we found a complete subarray
            while (countMap.size() == uniqueCount) {
                // Add the number of complete subarrays that can be made with this start point
                completeSubarrays += arraySize - end;

                // Reduce the count of the start element and erase it from the map if its count becomes zero
                if (--countMap[nums[start]] == 0) {
                    countMap.erase(nums[start]);
                }

                // Move the start pointer forward
                ++start;
            }
        }

        // Return the total count of complete subarrays
        return completeSubarrays;
    }
};
```

Typescript Solution

```
function countCompleteSubarrays(nums: number[]): number {
    // Create a map to store the frequency of each unique number in the array
    const frequencyMap: Map<number, number> = new Map();

    // Populate the frequency map with the initial count of each number
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) ?? 0) + 1);
    }

    // Count the number of unique elements in the input array
    const uniqueElementCount = frequencyMap.size;

    // Clear the frequency map for reuse
    frequencyMap.clear();

    const totalNums = nums.length; // Total number of elements in nums
    let totalCompleteSubarrays = 0; // Initialize complete subarrays counter
    let start = 0; // Initialize start pointer for subarrays

    // Iterate over the array using 'end' as the end pointer for subarrays
    for (let end = 0; end < totalNums; ++end) {
        // Increment the count for the current element in the frequency map
        frequencyMap.set(nums[end], (frequencyMap.get(nums[end]) ?? 0) + 1);

        // While the current subarray contains all unique elements,
        // Keep updating the total count of complete subarrays and adjust the start pointer.
        while (frequencyMap.size === uniqueElementCount) {
            // Add the number of complete subarrays from the current subarray to the result
            totalCompleteSubarrays += totalNums - end;
            // Decrement the count of the number at the start pointer
            frequencyMap.set(nums[start], frequencyMap.get(nums[start])! - 1);
            // If the start number count hits zero, remove it from the map
            if (frequencyMap.get(nums[start]) === 0) {
                frequencyMap.delete(nums[start]);
            }
            // Move the start pointer to the right
            ++start;
        }
    }

    // Return the total count of complete subarrays
    return totalCompleteSubarrays;
}
```

Time and Space Complexity

The provided Python code calculates the number of complete subarrays in the input list `nums`. A complete subarray is defined such that all elements of the `nums` list are included in the subarray at least once. Here is the complexity analysis of the code:

Time Complexity:

The time complexity of the code is $O(n)$, where n is the number of elements in the input list `nums`. Although the code contains a nested loop, the inner loop with variable `i` does not start from the beginning for each iteration of the outer loop with variable `j`. Instead, `i` picks up where it left off in the previous iteration. Each element is processed once by both `i` and `j`, hence the complexity is linear.

Space Complexity:

The space complexity is $O(n)$ as well. The primary data structure contributing to space complexity is the `Counter` object `d`, which in the worst case contains as many unique keys as there are unique elements in `nums`. Additionally, the set of `nums`, which is created at the beginning of the method, also contributes to the space complexity if all elements are unique. However, since both structures depend on the number of unique elements in `nums`, and there can't be more unique elements than `n`, the space complexity remains $O(n)$.