992. Subarrays with K Different Integers

Hard Array Hash Table Counting Sliding Window

Leetcode Link

Problem Description

The problem requires us to find the total number of subarrays from a given integer array `nums` such that each subarray has exactly `k` distinct integers. A subarray is defined as a contiguous sequence of elements within an array. For instance, in the array `[1,2,3,1,2]`, one subarray could be `[1,2,3]`, which contains 3 distinct integers, and if `k` equals 3, this would be considered a "good subarray."

Intuition

To solve this problem, we can use a two-pointer technique commonly used for sliding window problems. The core idea behind the solution is to create a sliding window that counts the occurrence of each number in the `nums` array and track when the number of distinct integers within the window reaches `k`. The function `f(k)` helps manage this by marking for each index `i` the furthest left position `j` such that the subarray `nums[j:i+1]` contains at most `k` distinct integers.

The counter `cnt` maintains the frequency of each number in the current window, and the array `pos` is used to record the starting position for the window that satisfies our condition for each index `i`.

The trick here is to call the helper function `f(k)` twice: once for `k` and once for `k-1` to find the number of windows with exactly `k` distinct integers and subtract the number of windows with exactly `k-1` distinct integers.

The reason behind this is that by finding the difference between these two counts, we effectively calculate the number of new subarrays that have exactly `k` distinct integers, as any window with `k-1` distinct integers cannot contribute to the count of windows with exactly `k`.

Finally, the summation `sum(a - b for a, b in zip(f(k - 1), f(k)))` calculates the total number of "good subarrays," by taking the difference of starting positions for windows with `k` and `k-1` distinct elements for each index, which corresponds to the count of "good subarrays" that end at that index.

Solution Approach

The solution uses a helper function `f(k)` to identify the last starting position of a subarray ending at each index `i` that contains at most `k` distinct integers. This function is called twice, once with `k` and another with `k-1`.

Here's the detailed algorithm:

1. Initialize `pos`, which is an array that will hold for each index `i` the furthest left position `j` such that `nums[j:i+1]` has at most `k` distinct elements.
2. Create a `Counter` object `cnt` that will help us count the frequency of each element in the window `[j, i]`.
3. Iterate through `nums` using `i` as the index for the current end of the window. For each `i`, increment the count of `nums[i]` in `cnt`.
4. Use another pointer `j` to keep track of the start of the window. If the count of distinct numbers in `cnt` exceeds `k`, remove elements from the start (increment `j`) until we're back within the limit of `k` distinct integers. Decrement the count of the element `nums[j]` in `cnt`, if the count drops to zero, remove that element from `cnt`.
5. Set the current `pos[i]` to `j`, which at this point is the left-most index for which the window `[j, i]` contains at most `k` distinct integers.
6. The function `f(k)` returns the `pos` array which we use to calculate the number of good subarrays.

For the final count, we subtract the positions found for `k-1` from the positions found for `k`. The `zip` function is used to iterate over positions of both `k` and `k-1` simultaneously, and for each pair of positions, we subtract the position for `k-1` from the position for `k` to get the number of new subarrays that have exactly `k` distinct integers.

7. The comprehension `sum(a - b for a, b in zip(f(k - 1), f(k)))` iterates through pairs of positions `(a, b)` from `pos` arrays returned by `f(k-1)` and `f(k)` respectively and calculates the difference. This difference represents the number of subarrays that end at each index having exactly `k` distinct integers. The `sum` function adds up these differences to get the total count of good subarrays.

In summary, the algorithm effectively combines a sliding window technique with a difference array approach to efficiently calculate the number of good subarrays. The sliding window ensures we consider only valid subarrays, and the difference between `f(k-1)` and `f(k)` positions allows us to count exactly those subarrays with the required number of distinct integers.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the array `nums = [1, 2, 1, 3, 4]` and `k = 3`. We're looking for the total number of subarrays with exactly `k` distinct integers.

Following our algorithm:

1. We initialize `pos = [0, 0, 0, 0, 0]` which will store the furthest left starting positions for subarrays with at most `k` distinct integers.
2. We create a frequency counter `cnt` which is initially empty.
3. Then, we iterate over `nums`. Let's walk through a few iterations:
   - For `i = 0` (element 1), `pos[0] = 0` because `[1]` has 1 distinct integer.
   - For `i = 1` (element 2), `pos[1] = 0` because `[1, 2]` has 2 distinct integers.
   - For `i = 2` (element 1, since 1 is already in `cnt`), `pos[2] = 0` because `[1, 2, 1]` still has 2 distinct integers.
4. When `i = 3` (element 3), `cnt` shows we have 3 distinct integers so far (`1, 2, 3`). We do not need to adjust `j`, so `pos[3] = 0`.
5. When `i = 4` (element 4), `cnt` would now have 4 distinct integers. Since our limit is `k`, we increment `j` to ensure we don't exceed `k` distinct integers.
   - Our count becomes `k+1` when `i = 4`. We then increment `j` to discard the first element (`1`), updating the counter. Now we have 3 distinct numbers again (`2, 3, 4`), and `pos[4] = 1`.
6. Finally, after calling function `f(k)` using our `nums` array and `k = 3`, we get a `pos` array telling us the starting positions that form valid windows, which are `[0, 0, 0, 0, 1]`.
7. For `k-1 = 2`, calling function `f(k-1)`, we would similarly get `[0, 0, 0, 1, 3]`.
8. We then zip these arrays and subtract the second from the first for each position: `[0-0, 0-0, 0-0, 0-1, 1-3] = [0, 0, 0, -1, -2]`.
9. The sum of these differences is `0 + 0 + 0 - 1 - 2 = -3`. However, we need to take its absolute value because we're interested in the count, not the signed difference: `abs(-3) = 3`. These differences represent how many new "good subarrays" we get at each index as we extend our window.

So, in the example `nums = [1, 2, 1, 3, 4]` with `k = 3`, there are 3 subarrays with exactly 3 distinct integers: `[1, 2, 1, 3]`, `[2, 1, 3, 4]`, and `[1, 2, 3, 4]`. And this matches our final count of 3. This demonstrates the use of the sliding window to isolate the regions of interest and the subtraction of counts to find a precise number of qualifying subarrays.

Python Solution

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

class Solution:
    def subarraysWithKDistinct(self, nums: List[int], k: int) -> int:
        # Helper function to calculate the number of subarrays with at most k distinct elements
        def at_most_k_distinct(k):
            # Initialize the position list to store the starting index of subarrays
            start_positions = [0] * len(nums)
            # Counter to keep track of the frequencies of elements in the current window
            count = Counter()
            left = 0  # Initialize the left pointer of the window
            # Iterate through the 'nums' list
            for right, value in enumerate(nums):
                count[value] += 1  # Increment the frequency of the current number
                # Shrink the window from the left if there are more than 'k' distinct elements
                while len(count) > k:
                    count[nums[left]] -= 1
                    # Remove the leftmost element from the counter if its frequency drops to zero
                    if count[nums[left]] == 0:
                        del count[nums[left]]
                    left += 1  # Move the left pointer to the right
                # Store the latest valid starting position where the window contains at most 'k' distinct elements
                start_positions[right] = left
            return start_positions

        # Calculate number of subarrays with exactly 'k' distinct elements by subtracting
        # the number of subarrays with at most 'k-1' distinct elements from those with at most 'k'
        return sum(end_at_most_k - end_at_most_k_minus_one for end_at_most_k, end_at_most_k_minus_one in zip(at_most_k_distinct(k), a

# Example usage:
# sol = Solution()
# result = sol.subarraysWithKDistinct([1,2,1,2,3], 2)
# print(result)  # Output: 7
```

Java Solution

```java
class Solution {
    public int subarraysWithKDistinct(int[] nums, int k) {
        // Find the positions with exactly k distinct elements and k-1 distinct elements
        int[] positionsWithKDistinct = findPositionsWithDistinctElements(nums, k);
        int[] positionsWithKMinusOneDistinct = findPositionsWithDistinctElements(nums, k - 1);

        // Initialize answer to hold the total number of subarrays with k distinct elements
        int answer = 0;

        // Calculate the difference between positions to get the count of subarrays with exactly k distinct elements.
        for (int i = 0; i < nums.length; i++) {
            answer += positionsWithKDistinct[i] - positionsWithKMinusOneDistinct[i];
        }

        return answer;
    }

    private int[] findPositionsWithDistinctElements(int[] nums, int k) {
        int n = nums.length; // Total number of elements in the input array
        int[] count = new int[n + 1]; // An array to keep track of counts of each distinct element
        int[] positions = new int[n]; // An array to store positions

        int distinctCount = 0; // A variable to keep track of current number of distinct elements

        // Two pointers - 'start' and 'end' to keep track of current window
        for (int end = 0, start = 0; end < n; end++) {
            if (++count[nums[end]] == 1) { // If it's a new distinct element
                distinctCount++;
            }
            // Shrink the window from the left if we have more than 'k' distinct elements
            while (distinctCount > k) {
                if (--count[nums[start]] == 0) { // If we've removed one distinct element
                    distinctCount--;
                }
                start++;
            }
            positions[end] = start; // Update the start position for current window size
        }
        return positions; // Return the positions array that helps in calculating the result
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>

class Solution {
public:
    // Function to calculate the number of subarrays with exactly k distinct elements
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        vector<int> subarrayStartsWithK = countSubarraysStartingPoint(nums, k);
        vector<int> subarrayStartsWithKMinusOne = countSubarraysStartingPoint(nums, k - 1);
        int totalSubarrays = 0;

        // Calculate the difference between the number of subarrays starting with k and k-1 distinct numbers
        for (int i = 0; i < nums.size(); ++i) {
            totalSubarrays += subarrayStartsWithKMinusOne[i] - subarrayStartsWithK[i];
        }

        return totalSubarrays;
    }

    // Helper function to find the earliest starting point of subarrays with at most k distinct elements for each ending point
    vector<int> countSubarraysStartingPoint(vector<int>& nums, int k) {
        int n = nums.size(); // Size of the input array
        vector<int> startPos(n); // Vector to store the starting positions
        int count[n + 1]; // Array to store the count of each number
        memset(count, 0, sizeof(count)); // Initialize the count array with zeros
        int distinctNums = 0; // Number of distinct elements

        // Two pointers technique: 'i' is the end pointer, 'j' is the start pointer
        for (int i = 0, j = 0; i < n; ++i) {
            // If we encounter a new element (count is 0), increase the number of distinct elements
            if (++count[nums[i]] == 1) {
                ++distinctNums;
            }

            // If distinct elements exceed k, move the start pointer to reduce the number of distinct elements
            for (; distinctNums > k; ++j) {
                // If after decrement count goes to zero, then one distinct element is removed
                if (--count[nums[j]] == 0) {
                    --distinctNums;
                }
            }

            // Record the starting position for the subarray ending at 'i' which has at most k distinct elements
            startPos[i] = j;
        }

        return startPos;
    }
};
```

Typescript Solution

```typescript
// Define the function to calculate the number of subarrays with exactly k distinct elements
function subarraysWithKDistinct(nums: number[], k: number): number {
    let subarrayStartsWithK = countSubarraysStartingPoint(nums, k);
    let subarrayStartsWithKMinusOne = countSubarraysStartingPoint(nums, k - 1);
    let totalSubarrays = 0;

    // Calculate the difference between the number of subarrays starting with k and k-1 distinct numbers
    for (let i = 0; i < nums.length; ++i) {
        totalSubarrays += subarrayStartsWithKMinusOne[i] - subarrayStartsWithK[i];
    }

    return totalSubarrays;
}

// Helper function to find the earliest starting point of subarrays with at most k distinct elements for each ending point
function countSubarraysStartingPoint(nums: number[], k: number): number[] {
    let n = nums.length; // Size of the input array
    let startPos = Array(n); // Array to store the starting positions for subarrays
    let count: number[] = Array(n + 1).fill(0); // Initialize an array to store the count of each number, filled with zeros
    let distinctNums = 0; // Number of distinct elements

    // Two pointers technique: 'end' is the end pointer, 'start' is the start pointer
    for (let end = 0, start = 0; end < n; ++end) {
        // If a new element is detected (count is 0), increase the number of distinct elements
        if (++count[nums[end]] == 1) {
            ++distinctNums;
        }

        // If distinct elements exceed k, move the start pointer to reduce the number of distinct elements
        for (; distinctNums > k; ++start) {
            // If after decrementing count goes to zero, then one distinct element is removed
            if (--count[nums[start]] == 0) {
                --distinctNums;
            }
        }

        // Record the starting position for the subarray ending at 'end' which has at most k distinct elements
        startPos[end] = start;
    }

    return startPos;
}
```

Time and Space Complexity

Time Complexity

The given Python function computes the number of subarrays with exactly `k` distinct elements by calculating the position of pointers for `k` and `k-1` distinct elements. The inner function `f(k)` goes through all elements of `nums` and uses a counter `cnt` to keep track of the number of distinct elements. Here's the breakdown of time complexity:

- The `f(k)` function loops over every element in `nums` exactly once with two nested loops. However, the inner while-loop does not start over for each outer iteration but continues from the last position, effectively visiting each element of the array only once. Thus, the time complexity for the `f(k)` function is `O(n)`, where `n` is the length of `nums`.
- Since `f(k)` is called twice, once with `k` and once with `k-1`, the total time for these calls is `2 * O(n) = O(n)`. The contribution of these calls to the time complexity does not change the order `O(n)`.
- The final part of the code calculates the sum of differences between the positions for `k` and `k-1` distinct elements, which takes `O(n)` time as well.

Combining all parts, the final time complexity of the `subarraysWithKDistinct` function is `O(n)` since all operations are linear with respect to the length of `nums`.

Space Complexity

The space complexity of the function depends on the size of the data structures used:

- The counter `cnt` can hold at most `min(k, n)` different integers, where `n` is the length of `nums` and `k` is the number of distinct elements we are looking for. So, it uses `O(min(k, n))` space.
- The array `pos` is of size `n`, resulting in `O(n)` space.
- Auxiliary space for indices and temporary variables is `O(1)`.

So the total space complexity is the maximum space used by any of these components, which is `O(n) + O(min(k, n))`. Since `k` is the constraint on distinct numbers and can be at most `n`, the dominant term is `O(n)`. Hence, the space complexity of the entire function is `O(n)`.