1852. Distinct Numbers in Each Subarray

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

Problem Description

In this problem, we are given an array of integers, `nums`, and another integer, `k`. Our task is to find the number of distinct integers in every contiguous subarray of `nums` that has a length of `k`. We construct an array `ans` such that each element `ans[i]` represents the number of distinct integers in the subarray starting from index `i` to index `i+k-1` inclusively. The result will be an array of the counts of distinct integers for each subarray of size `k`.

For example, suppose `nums` is `[1, 2, 3, 2, 2, 2, 1]` and `k` is `3`. The subarrays of size three are:

- `[1, 2, 3]`, which has 3 distinct numbers;
- `[2, 3, 2]`, which has 2 distinct numbers;
- `[3, 2, 2]`, which has 2 distinct numbers;
- `[2, 2, 1]`, which has 2 distinct numbers. The resulting `ans` would be `[3, 2, 2, 2]`.

Intuition

To find the number of distinct values in each subarray of size `k`, we need to efficiently count and update the distinct elements as we move the subarray from the beginning of `nums` to the end. A naive approach would involve recomputing the number of distinct elements for each subarray by traversing all elements within each subarray, resulting in an inefficient solution with a high time complexity.

However, as we slide the subarray window by one element from left to right, we notice that all but one element of the new subarray is the same as in the previous subarray. Specifically, each time we move the window, we remove the first element from the previous window and add a new element from the right. This suggests a more efficient approach, where we can update the counts of elements in the overlap of the two subarrays in constant time.

To achieve this, we use a Counter (in Python, a dictionary that automatically initializes values to 0 if keys are not yet present) to maintain the counts of elements in the current subarray window. For the initial subarray of size `k`, we populate the Counter with the frequency of each distinct number. We record the number of distinct elements (the size of the Counter) in our `ans` array.

As we slide the window to the right, we decrement the count of the element that is dropped from the subarray and remove the key from the Counter if its count drops to zero, ensuring that only counts of distinct elements are maintained. Then, we increment the count of the new element added to the subarray window. After each window slide, we append the current number of distinct elements (the size of the Counter) to the `ans` array.

This process continues until we have covered all subarrays of size `k`, resulting in a fully populated `ans` array with the correct counts. The use of the Counter allows us to maintain the update of distinct elements efficiently, which is the crux of this solution approach.

Solution Approach

The solution involves using a sliding window approach along with a data structure, the Counter, which is essentially a specialized dictionary for counting hashable objects in Python. These tools together provide an efficient way to dynamically track the changes in the frequency of distinct numbers as we move the window across the `nums` array.

Let's break down the code step by step:

- First, we calculate `n`, the length of the `nums` array. We also initialize the Counter with the first `k` elements of `nums`, which represents our first window.
- We add the length of the Counter `len(cnt)` to our `ans` list as our first entry, which corresponds to the number of distinct elements in the initial window.
- Next, we iterate over the remaining elements of the array starting from the `k`-th element to the end. For every iteration, we perform two operations:
  - We adjust the count of the element that's moving out of the window (`nums[i - k]`). We decrement its counter by 1, and if the count drops to 0, we remove the element from the Counter, effectively updating the distinct elements.
  - We adjust the count of the new element that's coming into the window (`nums[i]`). We increment its counter by 1: if the element was not present in the Counter, it gets added, again updating the distinct elements.
- After adjusting the Counter, we append the current length of the Counter `len(cnt)` to our `ans` list. This indicates the number of distinct elements in the current window.
- The loop continues, effectively moving the sliding window to the right by one element and updating the distinct element count on each move.
- Once we've gone through all elements from `k` to `n-1`, our `ans` array is complete and we return it.

This sliding window technique coupled with the Counter avoids the need for double loops (which would increase the time complexity significantly) and ensures that we maintain a running count of distinct elements in constant time, as only two elements change positions in the window at each step.

By intelligently managing the frequency counts and updating them in constant time as the window slides, we achieve an efficient solution with O(n) time complexity, where n is the number of elements in the `nums` array, and O(k) space complexity, where k is the window size, which corresponds to the maximum size of the Counter.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array `nums = [4, 5, 4, 3, 5]` and `k = 2`. We are to find the number of distinct numbers in each subarray of length `k`.

1. Initialize the Counter with the first `k` elements of `nums`. Here, `k = 2`, so the Counter includes `{4: 1, 5: 1}` after accounting for the first two elements `[4, 5]`.
2. Add the number of distinct elements in the Counter to the `ans` array. At this point, we have `ans = [2]` since there are two distinct numbers `[4, 5]`.
3. Start sliding the window across `nums` by one element at a time and adjust the Counter as we go.
   - Move the window one step to include `[5, 4]`, which means we should decrease the count of `4` (as we move past the first one) and then increase the count of the new `4` (the duplicate). The Counter remains unchanged `{4: 1, 5: 1}`, and we append `2` to `ans`, resulting in `ans = [2, 2]`.
   - Move to the next window `[4, 3]`. We decrease the count of `5` (dropping it off as it's no longer within the window) and include `3`. The Counter updates to `{4: 1, 3: 1}`, and `ans` becomes `ans = [2, 2, 2]`.
   - Finally, for the last window `[3, 5]`, we decrease the count of `4` (removing it from the Counter since its count goes to zero) and add `5`. Our Counter is `{3: 1, 5: 1}`. We append `2` to `ans`, ending with `ans = [2, 2, 2, 2]`.

Throughout the process, we efficiently keep track of the distinct numbers as the sliding window moved from left to right across `nums`, resulting in the final answer `ans = [2, 2, 2, 2]`, indicating that for each subarray of size `k` there are exactly two distinct numbers.

Python Solution

```python
from collections import Counter

class Solution:
    def distinctNumbers(self, nums, k):
        """
        Return a list of the count of distinct numbers in every contiguous subarray of length k.

        :param nums: List[int] - the input array of numbers.
        :param k: int - the length of each subarray to consider.
        :return: List[int] - list containing the number of distinct elements in each subarray.
        """

        # Calculate the length of the input list.
        n = len(nums)

        # Initialize a counter for the first window of size 'k'.
        num_counter = Counter(nums[:k])

        # List to store the count of distinct numbers in each window.
        distinct_count = [len(num_counter)]

        # Loop over the remaining elements starting from 'k' to the end of the list.
        for i in range(k, n):
            # 'outgoing_num' is the number exiting the window.
            outgoing_num = nums[i - k]

            # Decrease the count of the outgoing number.
            num_counter[outgoing_num] -= 1

            # If the outgoing number's count reaches 0, remove it from the counter.
            if num_counter[outgoing_num] == 0:
                del num_counter[outgoing_num]

            # 'incoming_num' is the new number entering the window.
            incoming_num = nums[i]

            # Increase the count of the incoming number.
            num_counter[incoming_num] += 1

            # Append the current number of distinct elements to the result list.
            distinct_count.append(len(num_counter))

        # Return the list of counts of distinct numbers in each window.
        return distinct_count
```

Java Solution

```java
class Solution {
    public int[] distinctNumbers(int[] nums, int k) {
        // declare and initialize an array to keep track of counts of each number
        int[] count = new int[100010];
        // distinctCount is used to keep the number of distinct elements in the current window of size k
        int distinctCount = 0;

        // Initializing count for the first window of size k
        for (int i = 0; i < k; ++i) {
            // If we haven't seen this number before in the current window, increment distinctCount
            if (count[nums[i]]++ == 0) {
                ++distinctCount;
            }
        }

        // The total number of elements in the array
        int n = nums.length;
        // Prepare an array to store the result. n - k + 1 is the number of windows we'll have
        int[] result = new int[n - k + 1];
        // Store the count of distinct numbers for the first window
        result[0] = distinctCount;

        // Iterate over the array starting from k, sliding the window one position at a time
        for (int i = k; i < n; ++i) {
            // Remove the count of the leftmost element of the previous window
            // If this number's frequency becomes 0 after decrementing, we've lost a distinct number
            if (--count[nums[i - k]] == 0) {
                --distinctCount;
            }
            // Add the count of the new element of the current window
            // If we haven't seen this number in the window before, increment distinctCount
            if (count[nums[i]]++ == 0) {
                ++distinctCount;
            }
            // Store the count of distinct numbers for the current window
            result[i - k + 1] = distinctCount;
        }

        // return the array containing counts of distinct numbers for each window
        return result;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // This function returns a vector of integers, representing the count of
    // distinct numbers in each subarray of size 'k' within the input 'nums' vector.
    vector<int> distinctNumbers(vector<int>& nums, int k) {
        // Initialize a counting array to keep track of the frequency of each number.
        // Assuming the range of numbers in the input vector does not exceed 100010.
        int frequencyCounter[100010] = {0};

        // Variable to keep track of the number of distinct elements.
        int distinctCount = 0;

        // Count distinct numbers in the first window of size 'k'.
        for (int i = 0; i < k; ++i) {
            if (frequencyCounter[nums[i]]++ == 0) { // If the number appears for the first time.
                ++distinctCount;
            }
        }

        // Get the size of the input vector.
        int n = nums.size();

        // Initialize the answer vector with appropriate size.
        vector<int> answer(n - k + 1);

        // The count for the first window is already found.
        answer[0] = distinctCount;

        // Loop through the rest of the array, sliding the window by one each time.
        for (int i = k; i < n; ++i) {
            // Remove the count of the first element in the previous window.
            if (--frequencyCounter[nums[i - k]] == 0) { // If the number's frequency drops to zero.
                --distinctCount;
            }
            // Add the count for the new element in the current window.
            if (frequencyCounter[nums[i]]++ == 0) { // If the number appears for the first time in the window.
                ++distinctCount;
            }
            // Record the number of distinct elements for this window.
            answer[i - k + 1] = distinctCount;
        }

        // Return the final vector containing counts of distinct numbers for all windows.
        return answer;
    }
};
```

Typescript Solution

```typescript
// Importing necessary components for TypeScript.
import { Vector } from 'prelude-ts';

// This function computes a vector of integers, representing the count of distinct numbers
// in each subarray of length 'k' within the input 'nums' vector.
function distinctNumbers(nums: number[], k: number): number[] {
    // Assuming the range of numbers in the input array does not exceed 100010.
    // Initialize a frequency counter array to keep track of the frequency of each number.
    const frequencyCounter: number[] = new Array(100010).fill(0);

    // Variable to keep track of the number of distinct elements.
    let distinctCount: number = 0;

    // Count distinct numbers in the first window of size 'k'.
    for (let i = 0; i < k; ++i) {
        if (frequencyCounter[nums[i]]++ === 0) {
            // If the number appears for the first time.
            ++distinctCount;
        }
    }

    // Variable to capture the size of the input array.
    const n: number = nums.length;

    // Initialize the answer array with appropriate size.
    const answer: number[] = new Array(n - k + 1);

    // The count for the first window has already been calculated.
    answer[0] = distinctCount;

    // Loop through the rest of the array, sliding the window by one element each iteration.
    for (let i = k; i < n; ++i) {
        // Decrease the count of the element exiting the window.
        if (--frequencyCounter[nums[i - k]] === 0) {
            // If the element's frequency drops to zero, decrement the distinct count.
            --distinctCount;
        }
        // Process the new element entering the window.
        if (frequencyCounter[nums[i]]++ === 0) {
            // If the element is unique within the current window, increment the distinct count.
            ++distinctCount;
        }
        // Store the number of distinct elements for this window in the answer array.
        answer[i - k + 1] = distinctCount;
    }

    // Return the computed array containing counts of distinct numbers for all windows.
    return answer;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed by looking at the operations performed within the main for-loop which runs from `k` to `n` (where `n` is the length of `nums`):

- Each iteration of the for-loop performs a constant time operation to decrement the counter for the outgoing element (`x = nums[i - k]` and `cnt[x] -= 1`).
- If the outgoing element's count drops to zero, it's removed from the counter with `cnt.pop(x)`, which also takes constant time since `Counter` is a subclass of a Python dictionary which generally has O(1) complexity for addition and removal.
- Increment the counter for the new incoming element – this also takes constant time `cnt[nums[i]] += 1`.
- Append the current number of distinct elements in the current window (`ans.append(len(cnt))`) – getting the length of `cnt` is constant time as well.

Given these operations, solving all of them have a time complexity of O(1) and we're running these for `n - k` times, the overall time complexity of the for-loop is `O(n - k)`.

However, we also have an initial setup cost where we initialize the counter with the first `k` elements of `nums`, this also takes `O(k)` time.

So, the overall time complexity of the function is `O(k) + O(n - k)` which simplifies to `O(n)`.

Space Complexity

For space complexity, we need to consider:

- The `cnt` counter, which can hold at most `min(k, k)` unique values if all elements are distinct within any window of size `k`. Hence the space complexity for the counter would be `O(min(k, k))`.
- The answer list `ans`, which will contain `n - k + 1` elements corresponding to the number of distinct elements in each window. Hence, its space complexity is `O(n - k + 1)`.

The overall space complexity of the function is the maximum of the space complexities of `cnt` and `ans`, which would be `O(n)` since `O(min(n, k))` is bounded by `O(k)` and `O(n - k + 1)` simplifies to `O(n)`.

Therefore, both the time complexity and space complexity of the code are `O(n)`.