2537. Count the Number of Good Subarrays

Sliding Window

Problem Description

<u>Array</u>

Hash Table

Given an integer array called nums and an integer k, the task is to determine the number of 'good' subarrays in nums. A subarray is a continuous part of the original array, and it is considered 'good' if there are at least k pairs (i, j) within it where i < j and the values at positions i and j are equal (i.e., arr[i] == arr[j]). The goal of the problem is to count and return how many such 'good' subarrays exist.

Intuition

The intuition behind the solution comes from recognizing that a subarray is 'good' if it contains a certain number of duplicated

Medium

that this element can form with previous elements (if there are duplicates). The solution uses a moving window approach to check subarrays. We use the window defined by the indices i and the current index of the element being considered, growing from the start of the array to the end.

elements. For each new element we add to a subarray while iterating through the array, we increase the potential number of pairs

A Counter is used to track the number of times each element appears within the current window. For each new element x considered, we increase the count for that element in the Counter, and we add to cur, which keeps track of the current number of

pairs within the window that satisfy the condition of being a 'good' subarray.

As we move the window forward by increasing i, we need to check if the window still has at least k good pairs after potentially removing an element (since the window might shrink). If the number of good pairs is still at least k after this potential removal, then every subarray that ends with the current element is 'good'. We then calculate how many such subarrays are there, which is i + 1, and add it to the overall count ans.

of good subarrays that end at the current index, resulting in a running total that gives the final answer: the number of good subarrays in the initial array. **Solution Approach**

The implementation of the solution employs a sliding window pattern coupled with a hash map (Counter in Python) to keep track

Initialize a Counter object cnt, which will map each number to the number of times it has occurred in the current subarray.

By using this approach, every time we find a window where the count cur is at least k, we ensure that we accumulate the number

Set up two accumulator variables: ans to store the total count of 'good' subarrays, and cur to keep track of the current count

0

of pairs within the window that can potentially form a 'good' subarray. Set up an index i to represent the start of the current window, initially set to 0.

- Loop through the nums array with a variable x representing the current number.
- For each x, add the current count of x in cnt to cur, increasing the number of pairs that match the condition (since x could form pairs with the earlier occurrences of itself).

While the current number of pairs (cur), minus the count of the element at the start of the window (nums[i]), plus one (as

we are considering the case where we might exclude nums [i] from the array), is still greater than or equal to k, it means

- Increment the count of x in cnt (since we are considering x as part of the current window).
- Update cur by subtracting the updated count of nums[i] (since the potential pairs involving nums[i] are now reduced). Increment i to shrink the window from the left.

'good' subarrays where a 'good' subarray is one that contains at least k pairs with the same value.

Start with our data structures: cnt is empty, ans is 0, and cur is 0. The index i is set to 0.

We move to the next index, but since we're at the end of our array, we stop the loop.

■ Decrement the count of nums[i] in cnt (since we are removing it from the window).

we can shrink the window from the left by doing the following:

pairs is maintained while accumulating the total number of such subarrays.

of the counts of each element within the current window. The approach works as follows:

current element, and we add this to our answer ans. After the loop completes, ans contains the total number of 'good' subarrays, and we return it. In conclusion, the sliding window, along with the counter map, efficiently keeps track of the number of pairs that are duplicated

within each window. By adjusting the start of the window (i), we ensure that the property of each subarray having at least k good

If the current number of pairs cur is greater than or equal to k, there are i + 1 new 'good' subarrays ending with the

Example Walkthrough

Let's say we are provided with an integer array nums = [1, 2, 2, 1] and an integer k = 2. We want to count the number of

We would approach this example with our sliding window pattern and a Counter for tracking the occurrences of each element.

As we loop through the nums array: Element x = 1. We add cnt [1] (which is 0) to cur (now cur is also 0). We then increment cnt [1] by 1. cur remains less

Element x = 2. We add cnt [2] (which is 0) to cur (still 0). Increment cnt [2]. Since cur is still less than k, ans stays 0.

Element x = 2. Now, cnt [2] is 1, so we add this to cur, making cur = 1. We increment cnt [2], making cnt [2] = 2. With

cur now 1, we still don't have enough pairs, so we don't change ans or i.

Solution Implementation

num_counter = Counter()

for num in nums:

def countGood(self, nums: List[int], k: int) -> int:

current_sum += num_counter[num]

num counter[num] += 1

if current_sum >= k:

result += start_index + 1

Return the total count of "good" subarrays

Initialize a counter to track the count of each number

Increment the count of the current number in the counter

while current_sum - num_counter[nums[start_index]] + 1 >= k:

If the current sum meets the requirement, count the subarrays

Decrease the count of the starting number

num_counter[nums[start_index]] -= 1

Python

class Solution:

than k so ans remains 0.

- Element x = 1. We add cnt[1] (which is 1) to cur (now cur is 2). We increment cnt[1] by 1. Now cur equals k, and so we can begin checking if we can shrink the window. Since removing nums [i] (which is 1) would bring cur below k, we don't shrink the window. Therefore, we add i + 1 (which is 1) subarrays to ans, making ans = 1.
- We conclude that there are ans = 1 subarrays that meet the 'good' criteria. The subarray [2, 2] within nums is the one that meets the condition of having at least k = 2 pairs (i, j) with i < j such that nums[i] == nums[j].

We continue to try and shrink the window, but since subtracting nums[i] (which is 1) would drop cur below k, we cannot.

- from collections import Counter
- # Initialize the result and current sum and the starting index result = current_sum = start_index = 0 # Iterate over the list of numbers

Update the current sum with the number of times we have seen the current number

If the current sum exceeds the limit, adjust the window from the left

```
# Deduct the excess from the current sum
current_sum -= num_counter[nums[start_index]]
# Move the starting index to the right
start index += 1
```

return result

C++

public:

#include <vector>

class Solution {

using namespace std;

#include <unordered_map>

// Function to count the number of good subarrays.

// Loop through the elements in the array.

currentCount += elementCount[num]++;

for (int& num : nums) {

long long totalCount = 0; // The total count of good subarrays.

int startIndex = 0; // The start index for the current subarray.

long long currentCount = 0; // Current number of pairs with the same value.

while (currentCount - elementCount[nums[startIndex]] + 1 >= k) {

currentCount -= --elementCount[nums[startIndex++]];

// The '+1' accounts for the single element as a subarray.

// The '+1' accounts for the individual element as a subarray.

long long countGood(vector<int>& nums, int k) {

```
Java
class Solution {
    public long countGood(int[] nums, int k) {
       // HashMap to store the frequency of each number in the current subarray
       Map<Integer, Integer> frequencyCounter = new HashMap<>();
        long totalCount = 0; // Total count of good subarrays
        long currentSize = 0; // Number of times a number has been repeated in the current subarray
        int startIndex = 0; // Start index for the sliding window
       // Iterate over the array using 'num' as the current element
        for (int num : nums) {
           // Update currentSize for the current value
            currentSize += frequencyCounter.getOrDefault(num, 0);
            // Increase the frequency counter for num
            frequencyCounter.merge(num, 1, Integer::sum);
           // Shrink the window from the left until the number of repeated elements is less than k
           while (currentSize - frequencyCounter.get(nums[startIndex]) + 1 >= k) {
               // Decrease the currentSize by the number of times the number at startIndex is in the window
                currentSize -= frequencyCounter.merge(nums[startIndex], -1, Integer::sum);
                // Move the start index of the subarray window to the right
               startIndex++;
           // If the number of repeated elements is at least k, we count this as a 'good' subarray
           if (currentSize >= k) {
               // Add to the total number (startIndex + 1 indicates that we have a 'good' subarray up to the current index i)
                totalCount += startIndex + 1;
       // Return the total count of good subarrays
       return totalCount;
```

unordered_map<int, int> elementCount; // Hashmap to count the occurrences of each integer.

// Reduce the number of pairs by the number of occurrences of the start element.

// Increment the current count by the number of occurrences before incrementing the count for the current number.

// If the current count is greater than or equal to k after adjusting the start index, we increment the total count.

// If the current count is greater than or equal to k, we need to adjust the start index of the subarray.

```
if (currentCount >= k) {
               totalCount += startIndex + 1;
       // Return the total number of good subarrays.
       return totalCount;
};
TypeScript
// Importing the necessary module for dictionary-like data structure.
import { Map } from "es6-shim";
// Function to count the number of good subarrays.
function countGood(nums: number[], k: number): number {
    // Map to store the frequency of each element in the current window
    let elementCount: Map<number, number> = new Map();
    // Initialize the total count of good subarrays as a number.
    let totalCount: number = 0;
    // Variable to store the current count of pairs with the same value within the window.
    let currentCount: number = 0;
    // The start index for the current subarray window.
    let startIndex: number = 0;
   // Iterating over each number in the array.
    for (let num of nums) {
       // If the number is already in the map, increase its count, otherwise add it with count 1.
       elementCount.set(num, (elementCount.get(num) || 0) + 1);
       // Increment the current count of pairs by the previous count of 'num'.
       currentCount += elementCount.get(num) - 1;
       // Adjust the start index of the window if there are k or more pairs with the same value.
       while (currentCount - (elementCount.get(nums[startIndex]) - 1) >= k) {
           // Reduce the number of pairs by the occurrences of the nums[startIndex].
            currentCount -= elementCount.get(nums[startIndex]) - 1;
           // Decrease the count of the start number as we're moving the window forward.
            elementCount.set(nums[startIndex], elementCount.get(nums[startIndex]) - 1);
            // Move the window's start index forward.
            startIndex++;
       // If the current count window has k or more pairs, add to the total count.
```

Move the starting index to the right start_index += 1 # If the current sum meets the requirement, count the subarrays if current_sum >= k:

Time and Space Complexity

return result

if (currentCount >= k) {

return totalCount;

from collections import Counter

for num in nums:

num_counter = Counter()

class Solution:

totalCount += startIndex + 1;

// Return the total count of good subarrays.

def countGood(self, nums: List[int], k: int) -> int:

result = current_sum = start_index = 0

current_sum += num_counter[num]

result += start_index + 1

Return the total count of "good" subarrays

Iterate over the list of numbers

num_counter[num] += 1

Initialize a counter to track the count of each number

Initialize the result and current sum and the starting index

Increment the count of the current number in the counter

while current_sum - num_counter[nums[start_index]] + 1 >= k:

Decrease the count of the starting number

current_sum -= num_counter[nums[start_index]]

Deduct the excess from the current sum

num counter[nums[start_index]] -= 1

Update the current sum with the number of times we have seen the current number

If the current sum exceeds the limit, adjust the window from the left

Time Complexity

The given code has two nested loops. However, the inner loop (while-loop) only decreases cur down to a point where it is less than k, and since elements can only be added to cur when the outer loop (for-loop) runs, the inner loop can run at most as many times as the outer loop throughout the whole execution. Because the inner loop pointer i is only incremented and never reset, each element is processed once by both the outer and

inner loops together. This leads to a linear relationship with the number of elements in nums. Therefore, the overall time complexity is O(n), where n is the length of nums. **Space Complexity**

The space complexity is driven by the use of the Counter object that stores counts for elements in nums. In the worst case, if all elements are unique, the counter would require space proportional to n. Hence, the space complexity is also 0(n).