2537. Count the Number of Good Subarrays



Leetcode Link

Problem Description

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

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

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

of the element being considered, growing from the start of the array to the end. 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. 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 of

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 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 counts of each element within the current window. The approach works as follows: 1. Initialize a Counter object cnt, which will map each number to the number of times it has occurred in the current subarray.

2. Set up two accumulator variables: ans to store the total count of 'good' subarrays, and cur to keep track of the current count of pairs within the window that can potentially form a 'good' subarray.

- 3. Set up an index i to represent the start of the current window, initially set to 0.
- 4. 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).
 - we are considering the case where we might exclude nums [i] from the array), is still greater than or equal to k, it means we can shrink the window from the left by doing the following: Decrement the count of nums [i] in cnt (since we are removing it from the window).

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

• 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.

 \circ Increment the count of x in cnt (since we are considering x as part of the current window).

element, and we add this to our answer ans. 5. After the loop completes, ans contains the total number of 'good' subarrays, and we return it.

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

Example Walkthrough

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

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

the window. Therefore, we add i + 1 (which is 1) subarrays to ans, making ans = 1.

subarrays where a 'good' subarray is one that contains at least k pairs with the same value. We would approach this example with our sliding window pattern and a Counter for tracking the occurrences of each element.

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 'good'

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

1. Start with our data structures: cnt is empty, ans is 0, and cur is 0. The index i is set to 0. 2. 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 than k so ans remains 0.

• 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

- 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
- 3. We continue to try and shrink the window, but since subtracting nums [i] (which is 1) would drop cur below k, we cannot. 4. We move to the next index, but since we're at the end of our array, we stop the loop.
- 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]. **Python Solution**

12 # Update the current sum with the number of times we have seen the current number 13 current_sum += num_counter[num] 14 # Increment the count of the current number in the counter 15 num_counter[num] += 1

9

10

11

16

17

18

20

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

class Solution:

1 from collections import Counter

num_counter = Counter()

for num in nums:

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

result = current_sum = start_index = 0

Iterate over the list of numbers

startIndex++;

if (currentSize >= k) {

return totalCount;

totalCount += startIndex + 1;

// Return the total count of good subarrays

1 // Importing the necessary module for dictionary-like data structure.

// Initialize the total count of good subarrays as a number.

// Map to store the frequency of each element in the current window

// Variable to store the current count of pairs with the same value within the window.

// Function to count the number of good subarrays.

function countGood(nums: number[], k: number): number {

let elementCount: Map<number, number> = new Map();

Initialize a counter to track the count of each number

Initialize the result and current sum and the starting index

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

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

Decrease the count of the starting number

num_counter[nums[start_index]] -= 1

```
21
                   # Deduct the excess from the current sum
22
                   current_sum -= num_counter[nums[start_index]]
23
                   # Move the starting index to the right
24
                   start_index += 1
25
26
               # If the current sum meets the requirement, count the subarrays
27
               if current sum >= k:
28
                   result += start_index + 1
29
30
           # Return the total count of "good" subarrays
           return result
31
32
Java Solution
 1 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
 8
           // Iterate over the array using 'num' as the current element
 9
10
           for (int num : nums) {
               // Update currentSize for the current value
11
12
               currentSize += frequencyCounter.getOrDefault(num, 0);
13
               // Increase the frequency counter for num
14
                frequencyCounter.merge(num, 1, Integer::sum);
15
16
               // Shrink the window from the left until the number of repeated elements is less than k
               while (currentSize - frequencyCounter.get(nums[startIndex]) + 1 >= k) {
17
                   // Decrease the currentSize by the number of times the number at startIndex is in the window
```

currentSize -= frequencyCounter.merge(nums[startIndex], -1, Integer::sum);

// If the number of repeated elements is at least k, we count this as a 'good' subarray

// Add to the total number (startIndex + 1 indicates that we have a 'good' subarray up to the current index i)

// Move the start index of the subarray window to the right

```
C++ Solution
 1 #include <vector>
 2 #include <unordered_map>
  using namespace std;
   class Solution {
 6 public:
       // Function to count the number of good subarrays.
       long long countGood(vector<int>& nums, int k) {
           unordered_map<int, int> elementCount; // Hashmap to count the occurrences of each integer.
            long long totalCount = 0; // The total count of good subarrays.
            long long currentCount = 0; // Current number of pairs with the same value.
11
12
            int startIndex = 0; // The start index for the current subarray.
13
           // Loop through the elements in the array.
14
           for (int& num : nums) {
               // Increment the current count by the number of occurrences before incrementing the count for the current number.
16
               currentCount += elementCount[num]++;
17
18
19
               // If the current count is greater than or equal to k, we need to adjust the start index of the subarray.
               while (currentCount - elementCount[nums[startIndex]] + 1 >= k) {
20
                   // Reduce the number of pairs by the number of occurrences of the start element.
22
                   currentCount -= --elementCount[nums[startIndex++]];
23
24
25
               // If the current count is greater than or equal to k after adjusting the start index, we increment the total count.
26
               // The '+1' accounts for the single element as a subarray.
               if (currentCount >= k) {
28
                   totalCount += startIndex + 1;
29
30
31
32
           // Return the total number of good subarrays.
33
           return totalCount;
34
35 };
36
Typescript Solution
```

23 24 25 26 27

9

10

11

12

13

import { Map } from "es6-shim";

let totalCount: number = 0;

let currentCount: number = 0;

```
14
       // The start index for the current subarray window.
15
       let startIndex: number = 0;
16
17
18
       // Iterating over each number in the array.
       for (let num of nums) {
19
           // 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);
21
22
           // 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) {
28
               // Reduce the number of pairs by the occurrences of the nums[startIndex].
29
               currentCount -= elementCount.get(nums[startIndex]) - 1;
30
               // Decrease the count of the start number as we're moving the window forward.
31
               elementCount.set(nums[startIndex], elementCount.get(nums[startIndex]) - 1);
33
34
               // Move the window's start index forward.
35
               startIndex++;
36
           // If the current count window has k or more pairs, add to the total count.
           // The '+1' accounts for the individual element as a subarray.
           if (currentCount >= k) {
               totalCount += startIndex + 1;
       // Return the total count of good subarrays.
       return totalCount;
```

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

43 44 45 46 47 } 48

Time and Space Complexity **Time Complexity**

Space Complexity

where n is the length of nums.

the outer loop throughout the whole execution.

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).

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),