[Count Subarrays Where Max Element Appears at Least K Times](https://leetcode.com/problems/count-subarrays-where-max-element-appears-at-least-k-times/)

Question :

You are given an integer array nums and a **positive** integer k.

Return *the number of subarrays where the****maximum****element of*nums*appears****at least***k*times in that subarray.*

A **subarray** is a contiguous sequence of elements within an array.

**Example 1:**

**Input:** nums = [1,3,2,3,3], k = 2

**Output:** 6

**Explanation:** The subarrays that contain the element 3 at least 2 times are: [1,3,2,3], [1,3,2,3,3], [3,2,3], [3,2,3,3], [2,3,3] and [3,3].

**Example 2:**

**Input:** nums = [1,4,2,1], k = 3

**Output:** 0

**Explanation:** No subarray contains the element 4 at least 3 times.

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 106
* 1 <= k <= 105

ANS IN CPP:

class Solution {

public:

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

        int maxElement = \*max\_element(nums.begin(), nums.end());

        long long ans = 0, start = 0;

        for (int end = 0; end < nums.size(); end++) {

            if (nums[end] == maxElement) {

                k--;

            }

            while (!k) {

                if (nums[start] == maxElement) {

                    k++;

                }

                start++;

            }

            ans += start;

        }

        return ans;

    }

};

**Overview**

The problem involves analyzing an integer array nums to count the number of subarrays in which the maximum element of nums appears at least k times, where k is given as an input.

A **subarray** is a contiguous sequence of elements within an array.

Algorithmically, solving this problem involves traversing the array and tracking the frequency of the maximum element in a dynamic range. The algorithm should efficiently update the frequency as it progresses through the array.

This problem is similar to scenarios where we need to find the frequency of occurrence of a particular event or condition within a given time frame or sequence.

* For instance in financial data analysis, one might be interested in identifying periods where a stock's price reaches its maximum value at least a certain number of times within a given timeframe. This can provide insights into potential trends or patterns.
* Similarly, in network traffic analysis, identifying subintervals where the network experiences maximum data transfer rates beyond a certain threshold can be crucial for optimizing network performance or identifying potential issues.

**Approach 1: Sliding Window**

**Intuition**

Since we are concerned with contiguous sequences and the frequency of a specific element, the sliding window algorithm emerges as a potentially effective approach. The sliding window algorithm is useful when handling contiguous segments within an array.

A sliding window is maintained by two indices, one of which indicates the start of the window, and the other the end of the window.

As we traverse the array, we should maintain the frequency of the maximum element within the window. Whenever we encounter the maximum element, we increment the frequency. The objective is to count the windows where this frequency is greater than or equal to the given threshold, k.

To achieve this objective, whenever the frequency of the maximum element in the window is greater than k, we initiate a process to shrink the window. This involves adjusting the starting point of the window (let's track this by index variable start) until the frequency of the maximum element in the window is exactly k to identify subarrays that have the maximum element appear at least k times.

The index variable start accounts for multiple starting positions for valid subarrays (where the frequency of the maximum is at least k) at the current ending position (let's track this by index variable end). By adding start to the answer, we ensure that we account for all valid subarrays ending at the current index end. This is because of the fact that for a given ending position end, there exist start + 1 possible starting positions, each contributing to a valid subarray.

As we traverse the array and execute these steps, we accumulate the count of valid subarrays. The final result is the total count of such subarrays.

1 / 17

**Algorithm**

1. **Initialization:**
   * Initialize variables max\_element, ans, start, and max\_elements\_in\_window.
   * max\_element stores the maximum element in the given array nums.
   * ans will be the final count of subarrays meeting the condition.
   * start is a pointer for the start of the window.
   * max\_elements\_in\_window stores the frequency of the max\_element within the current window.
2. **Iterating through the array:**
   * Iterate through each element in the array using a for loop with index end ranging from 0 to the length of nums.
3. **Counting frequency of max\_element in the current window:**
   * Check if the current element nums[i] is equal to max\_element.
   * If true, increment max\_elements\_in\_window as it represents the frequency of max\_element in the current window.
4. **Sliding window to meet the condition:**
   * Use a while loop to shrink the window (start pointer) until max\_elements\_in\_window is equal to k.
   * Inside the while loop, decrement max\_elements\_in\_window if the element at the window's start (nums[start]) is equal to max\_element.
   * Increment start to move the window to the right.
5. **Counting subarrays:**
   * Add start to the ans variable. This is done inside the for loop, so it accumulates the count of subarrays meeting the condition.
6. **Returning the result:**
   * After the loop completes, return the final count stored in the ans variable.

**Implementation**

**Complexity Analysis**

Let NN*N* be the length of nums.

* Time complexity: O(N)O(N)*O*(*N*).
  + Finding the maximum element in nums requires linear traversal of the array, taking O(N)O(N)*O*(*N*) computational time.
  + The outer for loop iterates through each element in the array exactly once, as indicated by the range from 000 to N−1N - 1*N*−1.
  + Inside this loop, the while loop with the start pointer performs a sliding window operation. However, note that the start pointer is increased, and max\_elements\_in\_window is decreased within this loop. The start pointer is never decreased after it is increased in the while loop. Hence, once an element is processed in the while loop, it will not be revisited. Therefore, each element is processed at most twice: once during the outer loop and at most once during the while loop.
  + In the worst case, the while loop could iterate through the entire length of the array during its lifetime. However, since each element is processed at most twice, the total number of iterations across all elements is linear, making the time complexity of the algorithm O(N)O(N)*O*(*N*).
* Space complexity: O(1)O(1)*O*(1). The space complexity is O(1)O(1)*O*(1) as the algorithm uses a constant amount of extra space regardless of the size of the input array.

**Approach 2: Track Indexes of Max Element**

**Intuition**

In the previous approach, the variable start was used to monitor potential starting positions corresponding to a given ending position within the array nums. We can also observe that for each valid subarray that began at an index r that contains a max element and ended at some index p, all subarrays starting at any index before r and ending at p are also valid subarrays. Upon examining the code, we can see that after the while loop completes, start consistently points to the index following the index containing a max\_element. With this understanding, we can store all indexes where a max\_element is found in an array. If there are more than k maximum elements within the array at any given point, we can identify the index of the max\_element that appeared k maximum elements ago.

For example:

nums = [1,3,2,3,3], k = 2

max\_element = 3

indexes\_of\_max\_elements = [1, 3, 4]

-------------------

For the index 3,

↓

[1,3,2,3,3]

index of the max element that appeared k maximum elements ago is 1

⌄ ↓

[1,3,2,3,3]

Add one to the index to find the number of possible starting positions:

1 + 1 = 2.

This indicates that the possible starting positions for the ending

position 3 are [0, 1].

-------------------

For the index 4,

↓

[1,3,2,3,3]

the index of the max element that appeared k maximum elements ago is 3

⌄ ↓

[1,3,2,3,3]

Add one to the index to find the number of possible starting positions:

1 + 3 = 4.

This indicates that the possible starting positions for the ending

position 3 are [0, 1, 2, 3].

Therefore, for any index where we've observed more than k maximum elements, the number of potential starting positions equals 1 plus the index where we encountered the max\_element k maximum elements ago.

**Algorithm**

1. **Initialization:**
   * Initialize variables max\_element, indexes\_of\_max\_elements, and ans.
   * max\_element stores the maximum element in the given array nums.
   * indexes\_of\_max\_elements is a list that stores the indexes of occurrences of the maximum element.
   * ans will be the final count of subarrays meeting the condition.
2. **Iterating through the array:**
   * Iterate through each element in the array along with its index.
3. **Finding indexes of maximum element:**
   * Check if the current element is equal to max\_element.
   * If true, append the index of the current element to the indexes\_of\_max\_elements list.
4. **Counting frequency of maximum element:**
   * Calculate the frequency of occurrences of the maximum element by finding the length of the indexes\_of\_max\_elements list.
5. **Checking condition for subarrays:**
   * Check if the frequency of the maximum element is greater than or equal to k.
   * If true, increment ans by the index of the (len(indexes\_of\_max\_elements) - k)-th occurrence of the maximum element plus 1.
   * This step counts the number of subarrays ending at the current index where the maximum element appears at least k times.
6. **Returning the result:**
   * After iterating through all elements, return the final count stored in the ans variable, which represents the total count of subarrays meeting the given condition.

**Implementation**

**Complexity Analysis**

Let NN*N* be the length of nums.

* Time complexity: O(N)O(N)*O*(*N*). Initializing max\_element incurs a time complexity of O(N)O(N)*O*(*N*) since each element of nums is checked. The for loop used to count subarrays also incurs a time complexity of O(N)O(N)*O*(*N*).
* Space complexity: O(N)O(N)*O*(*N*). In the worst case all the elements in nums are equal to max\_element. In this case, the final length of indexes\_of\_max\_elements will be N. Hence, the worst-case space complexity is O(N)O(N)*O*(*N*).