# 2962. Count Subarrays Where Max Element Appears at Least K Times

Medium Array Sliding Window

## Problem Description

You have an array of integers, `nums`, and a positive integer, `k`. Your task is to find out how many subarrays exist within this array where the highest number in the array, which we can call `mx`, shows up at least `k` times. A subarray is defined simply as a continuous portion of the array; it does not need to include all elements, but the elements it does include must be in the same order they appear in the full `nums` array.

## Intuition

To solve this problem, we need to think about how we can track occurrences of the maximum element in each subarray. A direct approach to identifying subarrays would involve looking at every possible subarray, which would be inefficient especially for larger arrays. A smarter way to do this is by using the concept of a moving window, often known as the two-pointer technique.

We define a variable `mx` to hold the maximum value in the array and then attempt to find subarrays that have at least `k` occurrences of `mx`. Starting with the first element, we expand our window to the right until we have `k` instances of `mx`. When we reach that point, any larger subarray that includes this window will also have at least `k` instances of `mx`. Thus, we add the total possible subarrays to our answer by calculating the number of elements from the end of our window to the end of the array, inclusive. We then slide our window to the right by one and reduce our count of `mx` occurrences if the first element of our previous window was a `mx`.

The beauty of this approach lies in its linear time complexity, as each element is visited only once by each of the two pointers. We count the occurrences of `mx` within a current window and continuously adjust the window size, keeping track of how many subarrays meet our requirement. This helps us avoid reexamining portions of the array unnecessarily, turning what could be an overwhelmingly complex problem into a manageable one.

## Solution Approach

The solution implements a two-pointer algorithm to efficiently find the number of subarrays fulfilling the given condition. We use the variables `i` and `j` to represent our two pointers, which define the edges of our current window on the array. The variable `cnt` serves as a counter for how many times `mx` (the maximum element of the entire array) appears within this window. The solution also initializes an answer variable `ans` to accumulate the number of valid subarrays.

We start by finding the maximum element in the array, `mx = max(nums)`, since the problem specifically concerns occurrences of this element.

Next, we iterate through each element in the array with a for-loop, thinking of each element as the potential start of a subarray (this is the left pointer, `i`). As we iterate:

1. We use the right pointer `j` to extend the window to the right, incrementing `cnt` whenever we encounter the maximum value `mx`.
2. We continue extending the window until `cnt` is at least `k`, meaning we've found a subarray where `mx` appears at least `k` times.
3. At this point, any subarray starting from `i` and ending at or past `j-1` will contain at least `k` instances of `mx`. The total such subarrays can be counted as `n - (j - 1)`, and we add this count to our answer `ans`.
4. Before moving `i` to the next position, reducing our window from the left, we need to decrease `cnt` if the element `nums[i]` that's being removed is equal to `mx`.

The loop breaks if `j` reaches the end of the array, or if `cnt` falls below `k`, because further iteration of `i` cannot possibly find a sufficient count of `mx` to fulfill the condition.

This approach only passes through the array once with each pointer and hence, has a linear time complexity of O(n), where n is the number of elements in the array.

Here's the implementation walkthrough based on the given solution:

- `mx = max(nums)`: Find the maximum element in `nums`.
- `n = len(nums)`: Get the length of the array to know when we've reached the end.
- `ans = cnt = j = 0`: Initialize the answer (`ans`), the count for the maximum value (`cnt`), and the right pointer (`j`) to zero.
- `for x in nums`: Start the main loop over `nums`, considering each `x` to be the potential beginning of a subarray.
- `while j < n and cnt < k`: Expand the window to the right with the right pointer `j` until `k` instances of `mx` are within it or the end of the array is reached.
  - `cnt += nums[j] == mx`: Increment `cnt` when the current element is `mx`.
  - `j += 1`: Move the right pointer `j` to the next position.
- `if cnt < k`: If `k` or more instances of `mx` cannot be found, no more valid subarrays are possible, break the loop.
- `ans += n - j + 1`: Add the number of valid subarrays to `ans`.
- `cnt -= x == mx`: Before the next iteration of the loop, decrease `cnt` if the element removed from the current window is `mx`.

By using these steps, we obtain a seamless, efficient algorithm that finds the total number of valid subarrays without resorting to brute force.

### Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose the array `nums` is `[1, 2, 2, 3, 2]`, and `k` is 2. We want to count the subarrays where the maximum element appears at least twice.

First, we find the maximum element `mx` of `nums`: `mx = max(nums) = 3`.

Now, we initialize `ans = 0`, `cnt = 0`, and `j = 0`, and start iterating over the array with our left pointer, represented by the elements in the for-loop.

Here's the step-by-step process:

1. We start at `i = 0`, with `nums[i] = 1`.
2. We look to expand our window to the right; since `cnt < k`, our inner while-loop starts.
3. We increment `j` until we encounter `mx` at least `k` times. During the process, we increment `cnt` whenever `nums[j] == mx`.
   - `j = 0`, `nums[j] = 1`, `cnt` remains `0`.
   - `j = 1`, `nums[j] = 2`, `cnt` remains `0`.
   - `j = 2`, `nums[j] = 2`, `cnt` remains `0`.
   - `j = 3`, `nums[j] = 3`, `cnt` becomes `1`.
   - `j = 4`, `nums[j] = 2`, `cnt` remains `1`.
   - As `j` has reached the end of the array and we have not found `k` instances of `mx`, we break from the loop and no subarrays are counted from this position (`ans` remains `0`).
4. We move to the next starting position, `i = 1`.
   - Repeat steps 2-3, but since `j` is already at the end, we don't enter the while-loop, and no subarrays are counted from this position either.
5. We continue this process moving `i` from `1` to `n-1` (end of the array), but in this case, since `mx` appears only once and our `k` condition is at least twice, we don't find any subarray that matches the criteria.

In the end, the `ans` is `0` because there are no subarrays within `[1, 2, 2, 3, 2]` where the maximum element `3` appears at least `2` times.

Had `mx` appeared at least `k = 2` times, we would add up all the possible subarrays from `i` up to `j - 1` to our answer `ans` each time, and if `nums[i]` is an instance of `mx`, we would decrease `cnt` before moving on to the next `i`.

## Solution Implementation

### Python

```python
from typing import List  # Added for List type hint support.

class Solution:
    def countSubarrays(self, nums: List[int], k: int) -> int:
        max_value = max(nums)  # Find the maximum value in the nums list.
        n = len(nums)  # Get the length of the nums list.
        answer = 0  # Initialize the answer to 0.
        count_max = 0  # Initialize the count for the maximum value to 0.
        j = 0  # Pointer to scan through the list.

        # Iterate over all elements in nums array.
        for i, x in enumerate(nums):
            # Move the j pointer and update count_max until we have counted k max_value or reached end of list.
            while j < n and count_max < k:
                count_max += nums[j] == max_value  # Increment count_max if nums[j] is max_value.
                j += 1  # Increment j to move the window forward.

            # If we have counted k max_value, update the answer.
            if count_max < k:
                break  # If count_max is less than k, then break from the loop as further windows won't have k max_values.
            answer += n - j + 1  # Add the number of valid subarrays starting from i-th position.

            # Decrease count_max for the sliding window as we move past the i-th element.
            count_max -= x == max_value

        return answer  # Return the total number of valid subarrays.

# Example of using the modified Solution class.
sol = Solution()
result = sol.countSubarrays([1, 4, 2, 5, 3], 2)
print(result)  # Output the result.
```

### Java

```java

class Solution {
    public long countSubarrays(int[] nums, int k) {
        // Find the maximum value in the array
        int maxNum = Arrays.stream(nums).max().getAsInt();

        // Initialize the length of the array
        int n = nums.length;

        // Variable to store the answer
        long countOfSubarrays = 0;

        // Initialize the count of maximum number in the current window
        int maxCount = 0;

        // Initialize the right pointer for the window to 0
        int rightPointer = 0;

        // Iterate over each element in the nums array with index represented by leftPointer
        for (int leftPointer = 0; leftPointer < n; leftPointer++) {
            // Expand the window until we have less than k occurrences of the maximum number
            while (rightPointer < n && maxCount < k) {
                // Increase the count if the current number is the maximum number
                if (nums[rightPointer] == maxNum) {
                    maxCount++;
                }
                // Move the right pointer to the right
                rightPointer++;
            }

            // If we have found k or more occurrences, we cannot form more subarrays, so we break out
            if (maxCount < k) {
                break;
            }

            // Update the count of subarrays with the number of ways to choose the end point of subarrays
            countOfSubarrays += n - rightPointer + 1;

            // Exclude the current left pointer number from count if it's the maximum number
            if (nums[leftPointer] == maxNum) {
                maxCount--;
            }
        }

        // Return the total number of subarrays
        return countOfSubarrays;
    }
}
```

### C++

```cpp
class Solution {
public:
    long long countSubarrays(vector<int>& nums, int k) {
        // Find the maximum value in the array
        int maxVal = *max_element(nums.begin(), nums.end());
        int n = nums.size();
        long long answer = 0; // This will store the final count of subarrays
        int countMax = 0; // Counts the number of maximum elements in the current subarray
        int j = 0; // Pointer to extend the right boundary of the subarray

        // Iterate through the array, considering each element as the start of a subarray
        for (int i = 0; i < n; ++i) {
            // Extend the subarray until we either run out of elements
            // or we have enough instances of the maximum element
            while (j < n && countMax < k) {
                countMax += nums[j] == maxVal;
                ++j;
            }
            // If we have less than 'k' instances, we cannot form more subarrays
            // starting from the current element
            if (countMax < k) break;

            // Add the number of possible subarrays that start with the current element
            answer += n - j + 1;

            // Prepare for the next iteration by decrementing the count if the current
            // element is the maximum value
            countMax -= nums[i] == maxVal;
        }

        // Return the total count of qualifying subarrays
        return answer;
    }
};
```

### TypeScript

```typescript

// Counts subarrays where the occurrence of the maximum element in the array is less than k
function countSubarrays(nums: number[], k: number): number {
    // Find the maximum value in the input array nums
    const maximumValue = Math.max(...nums);
    // Get the length of the input array nums
    const totalLength = nums.length;
    // Initialize counts for maximum values and index pointer
    let countOfMax = 0;
    let indexPointer = 0;
    // Initialize answer which will hold the count of valid subarrays
    let answer = 0;

    // Iterate over each element in the nums array
    for (const current of nums) {
        // Increase indexPointer while total max-element count is less than k
        // and the indexPointer is within the array bounds
        while (indexPointer < len && countOfMax < k) {
            if (nums[indexPointer] === maximumValue) {
                countOfMax += 1;
            }
            indexPointer += 1;
        }

        // If the count is less than k at this point, we can exit the loop
        if (countOfMax < k) {
            break;
        }

        // Add the number of valid subarrays that can be formed from this position.
        answer += len - indexPointer + 1;

        // Decrease the count of maximum values if the current element was a maximum
        if (current === maximumValue) {
            countOfMax -= 1;
        }
    }

    // Return the total number of subarrays that satisfy the condition
    return answer;
}
```

```python
from typing import List  # Added for List type hint support.

class Solution:
    def count_subarrays(self, nums: List[int], k: int) -> int:
        max_value = max(nums)  # Find the maximum value in the nums list.
        n = len(nums)  # Get the length of the nums list.
        answer = 0  # Initialize the answer to 0.
        count_max = 0  # Initialize the count for the maximum value to 0.
        j = 0  # Pointer to scan through the list.

        # Iterate over all elements in nums array.
        for i, x in enumerate(nums):
            # Move the j pointer and update count_max until we have counted k max_value or reached end of list.
            while j < n and count_max < k:
                count_max += nums[j] == max_value  # Increment count_max if nums[j] is max_value.
                j += 1  # Increment j to move the window forward.

            # If we have counted k max_value, update the answer.
            if count_max < k:
                break  # If count_max is less than k, then break from the loop as further windows won't have k max_values.
            answer += n - j + 1  # Add the number of valid subarrays starting from i-th position.

            # Decrease count_max for the sliding window as we move past the i-th element.
            count_max -= x == max_value

        return answer  # Return the total number of valid subarrays.

# Example of using the modified Solution class.
sol = Solution()
result = sol.count_subarrays([1, 4, 2, 5, 3], 2)
print(result)  # Output the result.
```

## Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` array. This is because the algorithm uses a sliding window that iterates over each element of the array at most twice - once when extending the window and once when moving the starting point of the window forward. Thus, each element is visited a constant number of times, leading to a linear time complexity with respect to the size of the input array.

The space complexity is `O(1)` as the algorithm only uses a constant amount of additional space for variables like `mx`, `n`, `ans`, `cnt`, and `j`, irrespective of the size of the input array.