2958. Length of Longest Subarray With at Most K Frequency

Medium | Array | Hash Table | Sliding Window

Problem Description

You are given an array of integers, `nums`, and another integer, `k`. The term "frequency" refers to how often an element occurs in the array. A "good" array is defined as an array where each element's frequency is no more than `k`. The main objective is to find the length of the longest subarray within `nums` that qualifies as "good". Remember, a subarray is a consecutive sequence that forms part of the larger array.

Intuition

To find the longest "good" subarray, we can use a sliding window approach. The "window" in this context is a range of consecutive items in `nums`, which we will continuously adjust to find the longest range that meets our frequency criteria.

Here's how we think through it:

1. First, understand that a subarray is "good" when none of its elements occur more than `k` times.
2. We will have two pointers, representing the start and end of the subarray, moving through `nums`. These pointers define our current window.
3. We grow the window by moving the end pointer forward and track the frequency of each element in the window.
4. If the frequency of the new element pushes it over `k`, the subarray isn't "good" anymore. We then need to shrink the window from the start until we remove enough instances of any over-represented element to make the subarray "good" again.
5. As we shift our window, we'll remember the size of the largest "good" subarray we've seen so far. This is our answer.

The key insight is realizing that we can dynamically adjust our window to find the optimal size. By using a hash map to count element frequencies and a while loop to maintain the "good" criteria, we ensure our solution is efficient and doesn't re-scan parts of the array unnecessarily.

Solution Approach

The implementation of the solution uses the two-pointer technique combined with a hash map to achieve an efficient way of finding the longest "good" subarray. Here's a detailed walkthrough aligned with the code provided:

1. We start by initializing a `defaultdict(int)` named `cnt` to keep track of the frequency of each element in our current window.
2. We then set `ans` to 0, which will eventually hold the length of the longest "good" subarray found. Also, we initialize a pointer `j` to 0, which represents the start of the current subarray (the sliding window's left edge).
3. The `for` loop begins by iterating over each element `x` in the list `nums`, with `i` denoting the current position (forming the sliding window's right edge).
4. For each element `x`, the code `cnt[x] += 1` increments its frequency in the hash map.
5. Now we check if the subarray `[j, i]` is not "good" anymore by seeing if the frequency of `x` exceeds `k`. The `while` loop `while cnt[x] > k:` is activated if the condition is true.
6. Inside the loop, we move `j` to the right (`j += 1`) to shrink the window, decrementing the frequency of `nums[j]` in `cnt` to reflect this change. This process repeats until the frequency of `x` is `k` or less, making the current window "good" again.
7. After each iteration and making sure the window is "good", the length of the current subarray (`i - j + 1`) is compared with the maximum length found so far (`ans`), and `ans` is updated if the current length is greater. This is done using `ans = max(ans, i - j + 1)`.
8. Once we've checked all elements, the `for` loop ends, and we return `ans`, which now holds the longest length of a "good" subarray.

This algorithm efficiently compromises between expanding and shrinking the window to account for every possible "good" subarray, using the `cnt` hash map to keep track of frequencies and two pointers to keep track of the window's bounds. By updating `ans` each time we find a longer "good" subarray, we ensure the final answer holds the maximum length required.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Suppose we are given an array of integers `nums = [1, 2, 2, 3, 1, 2, 3]` and another integer `k = 2`. We are tasked with finding the length of the longest subarray where no element appears more than `k` times.

Following our solution approach:

1. We start with initializing our hash map `cnt` to keep track of the frequencies, a variable `ans` set to `0` to track the length of the longest "good" subarray, and a pointer `j` initialized to `0` for the start position of our window.
2. Next, we begin iterating over `nums` from left to right:
   - At `i = 0`, `nums[i] = 1`. We increment `cnt[1]` to `1` and since all frequencies are within `k`, we update `ans` to `1`.
   - At `i = 1`, `nums[i] = 2`. We increment `cnt[2]` to `1`. The subarray `[1, 2]` is "good" and `ans` is updated to `2`.
   - At `i = 2`, `nums[i] = 2`. We increment `cnt[2]` to `2`. The subarray `[1, 2, 2]` is still "good", so `ans` becomes `3`.
   - At `i = 3`, `nums[i] = 3`. `cnt[3]` becomes `1`, and the subarray `[1, 2, 2, 3]` is "good". Update `ans` to `4`.
   - At `i = 4`, `nums[i] = 1`. The frequency of `1` is now `2`, and the subarray `[1, 2, 2, 3, 1]` is "good", so `ans` is now `5`.
   - At `i = 5`, `nums[i] = 2`. Now `cnt[2]` becomes `3`, which is over `k`. To restore the "good" status, we shift `j` rightward. First, we decrement `cnt[1]` by `1` as we move past the first element, but `cnt[2]` is still `3`, so we shift `j` again, decrementing `cnt[2]` by `1`. Now `cnt[2]` is `2`, the subarray `[2, 2, 3, 1, 2]` is "good", and `ans` remains at `5`.
   - At `i = 6`, `nums[i] = 3`. Again, the subarray `[2, 2, 3, 1, 2, 3]` is "good" and `ans` is updated to `6`, which is the length of this "good" subarray.
3. Since we have gone through all elements, our final answer `ans` is `6`, the length of the longest "good" subarray `[2, 2, 3, 1, 2, 3]`.

This example clearly demonstrates how we use the two-pointer technique in tandem with the frequency hash map to efficiently determine the longest "good" subarray in `nums`.

Solution Implementation

Python

```python
from collections import defaultdict

class Solution:
    def max_subarray_length(self, nums: List[int], k: int) -> int:
        # Initialize a counter to keep track of the frequency of each number
        frequency_counter = defaultdict(int)

        # `max_length` stores the maximum subarray length found that meets the condition
        max_length = 0

        # `left_pointer` is used to shrink the subarray from the left
        left_pointer = 0

        # Iterate over the list of numbers using their index and value
        for right_pointer, value in enumerate(nums):
            # Increment the count of the current value
            frequency_counter[value] += 1

            # If the count of the current value exceeds k, shrink the window from the left
            while frequency_counter[value] > k:
                # Decrease the count of the number at the left_pointer position
                frequency_counter[nums[left_pointer]] -= 1

                # Move the left_pointer to the right, making the subarray smaller
                left_pointer += 1

            # Calculate the current subarray length and update `max_length` if it's larger
            current_length = right_pointer - left_pointer + 1
            max_length = max(max_length, current_length)

        # Return the maximum subarray length found
        return max_length
```

Java

```java
class Solution {
    public int maxSubarrayLength(int[] nums, int k) {
        // Map to store the frequency of each number in the current subarray
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        int maxLength = 0; // To store the length of the longest subarray

        // Use the two-pointer technique
        for (int start = 0, end = 0; end < nums.length; ++end) {
            // Increase the frequency of the current number
            frequencyMap.merge(nums[end], 1, Integer::sum);

            // If the frequency of the current number exceeds k, shrink the window from the left
            while (frequencyMap.get(nums[end]) > k) {
                frequencyMap.merge(nums[start], -1, Integer::sum);
                start++; // Move the start index to the right
            }

            // Calculate the length of the current subarray and update maxLength if it is bigger
            maxLength = Math.max(maxLength, end - start + 1);
        }

        // Return the maximum length found
        return maxLength;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    // Function to find the length of the longest sub-array with elements appearing no more than 'k' times
    int maxSubarrayLength(vector<int>& nums, int k) {
        // Dictionary to keep track of the count of each number in the current window
        unordered_map<int, int> countMap;
        // Variable to store the maximum length found
        int maxLength = 0;
        // Two pointers defining the current window's boundaries
        for (int left = 0, right = 0; right < nums.size(); ++right) {
            // Increment the count of the rightmost element in the current window
            ++countMap[nums[right]];
            // If the count of the current element exceeds k, shrink the window from the left
            while (countMap[nums[right]] > k) {
                --countMap[nums[left]];
                ++left;  // Move the left pointer to the right
            }
            // Update the maximum length if the current window is larger
            maxLength = max(maxLength, right - left + 1);
        }
        // Return the maximum length of the sub-array
        return maxLength;
    }
};
```

TypeScript

```typescript
// This function finds the maximum length of a subarray
// where the majority element (the element that appears more than "k" times) is at most "k"
function maxSubarrayLength(nums: number[], k: number): number {
    // Map to store frequency of each number in the current window
    const frequencyMap: Map<number, number> = new Map();
    // Variable to store the maximum length found
    let maxLength = 0;

    // Pointers for the sliding window
    let start = 0; // Start index of the sliding window
    let end = 0;   // End index of the sliding window

    // Iterate through the array using "end" as the end index of the sliding window
    for (end = 0; end < nums.length; ++end) {
        // Update the frequency of the end element of the window
        frequencyMap.set(nums[end], (frequencyMap.get(nums[end]) ?? 0) + 1);

        // If the current number has occurred more than "k" times, move the start index forward
        while (frequencyMap.get(nums[end])! > k) {
            // Decrease the frequency of the start element of the window
            frequencyMap.set(nums[start], frequencyMap.get(nums[start])! - 1);
            // Move the start index forward
            ++start;
        }

        // Calculate the length of the current window and update maxLength if this is larger
        maxLength = Math.max(maxLength, end - start + 1);
    }

    // Return the maximum length found
    return maxLength;
}
```


```python
from collections import defaultdict

class Solution:
    def max_subarray_length(self, nums: List[int], k: int) -> int:
        # Initialize a counter to keep track of the frequency of each number
        frequency_counter = defaultdict(int)

        # `max_length` stores the maximum subarray length found that meets the condition
        max_length = 0

        # `left_pointer` is used to shrink the subarray from the left
        left_pointer = 0

        # Iterate over the list of numbers using their index and value
        for right_pointer, value in enumerate(nums):
            # Increment the count of the current value
            frequency_counter[value] += 1

            # If the count of the current value exceeds k, shrink the window from the left
            while frequency_counter[value] > k:
                # Decrease the count of the number at the left_pointer position
                frequency_counter[nums[left_pointer]] -= 1

                # Move the left_pointer to the right, making the subarray smaller
                left_pointer += 1

            # Calculate the current subarray length and update `max_length` if it's larger
            current_length = right_pointer - left_pointer + 1
            max_length = max(max_length, current_length)

        # Return the maximum subarray length found
        return max_length
```


Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`. This is because the function has a single loop iterating through the elements of `nums` from start to end, performing a fixed set of operations for each element. Therefore, the time taken by the function scales linearly with the size of the input list `nums`.

Space Complexity

The space complexity of the code is `O(n)`. This arises due to the use of the `cnt` dictionary that stores the count of elements. In the worst case, where all elements in `nums` are unique, the dictionary would contain an entry for each distinct element, leading to a space requirement that scales linearly with the size of the input list `nums`.