2488. Count Subarrays With Median K

Hard Array Hash Table Prefix Sum Leetcode Link

Problem Description

You have been given an array named `nums` with a size `n`, which includes distinct integers ranging from `1` to `n`. You are also given a positive integer `k`. Your task is to count how many non-empty subarrays of `nums` have a median that is equal to `k`.

Remember, the median of an array is defined as the middle element once the array is sorted in ascending order. For an array with an even number of elements, there isn't a single middle element, so in that case, you take the left middle element as the median.

A subarray is any sequence of consecutive elements from the array. It can be as short as one element, or as long as the entire array.

To illustrate, consider the array `[2, 3, 1, 4]`. After sorting, this array becomes `[1, 2, 3, 4]`. The median is the left middle element, which is `2` for this array. For the array `[8, 4, 3, 5, 1]`, after sorting it becomes `[1, 3, 4, 5, 8]`. The median for this array is `4` since there is an odd number of elements.

Your objective is to find and return the number of these subarrays in which the median is exactly `k`.

Intuition

Let's try to decode the provided solution to this problem. The main idea here is to use the index of `k` in `nums` as an anchor point. Since the problem tells us `nums` consists of distinct integers from `1` to `n`, we know `k` will appear exactly once in `nums`.

We can search for the contiguous subarrays where `k` is the median. If `k` is the median of a subarray, for the subarray to remain valid when adding elements to the left or right, we must maintain a balance or near-balance in the number of elements that are less than `k` and the number of elements greater than `k`.

While iterating through the elements, we keep track of this balance with a counter `x`, which we increment if we add an element greater than `k` to our subarray, and decrement if the element is less than `k`. The counter value represents the difference between the count of elements greater than `k` and less than `k` to the right of `k`.

The solution approach calculates this balance for the elements to the right and left of `k` separately, then combines these results. For the elements on the right of `k`, we just need to keep track of instances where this balance is 0 or 1. For the elements on the left of `k`, we also have to consider the complementary counts from the right side to ensure that `k` remains the median when combining subarrays from both sides.

Essentially, the algorithm loops twice: once over the elements to the right of `k`'s index and once over the elements to the left. In each loop, it maintains a count (`cnt`) of occurrences of each balance value. This count is key to finding the number of valid subarrays when we combine elements from both sides. The final answer is the sum of all valid subarray counts.

Solution Approach

The implementation of the solution revolves around a two-part process. In the first part, we find the index `i` of the integer `k` within the array `nums` using `nums.index(k)`. This is crucial as we will compute the balance of larger and smaller elements in the subarrays with respect to `k`.

To assist in counting, a `Counter` object named `cnt` is instantiated. This data structure is a dictionary subclass from Python's `collections` module that counts hashable objects.

The core part of the algorithm operates on the premise that when we add an element greater than `k`, it could potentially disrupt the balance required for `k` to be the median. Thus, every time we encounter such an element, we increase our balance counter `x`. Conversely, an element less than `k` contributes toward maintaining the median, and thus we decrease the counter `x` by one.

Analyzing the Right of `k`

We create a subarray of `nums` starting from the element just after `k` with `nums[i + 1 :]`. With a for-loop over this subarray, for every element `v` that is greater than `k`, we increment `x` and for every element less than `k`, we decrement `x`. The conditions `0 <= x <= 1` ensure that the median of the new subarray including `v` remains `k` (as `k` should not be overtaken by a higher number of elements that are either greater or smaller). For each balance situation satisfying this condition, we increase the solution count `ans`. The `cnt` counter keeps track of how many times each `x` value occurred.

Analyzing the Left of `k`

Once the right side is processed, we reset `x` to `0` and run a second for-loop, iterating over the elements to the left of `k`. The loop decrements the index `j` each time, moving leftwards. For each element `nums[j]`, `x` is adjusted in the same way (increment for elements greater than `k`, decrement for elements less than `k`).

For each balance value `x`, two additions are made to the `ans`:

- `0 <= x <= 1` is again checked, and if true, `ans` is incremented.
- The corresponding counts for `-x` and `-x + 1` in the `cnt` Counter (the balance counts from the right side) are added to `ans`.

These additions account for the subarrays extending both left and right from element `k`, whose medians are still `k`.

Finally, the `ans` which holds the count of all qualifying subarrays is returned.

This solution leverages the balance theory, counter collections, indexing, and iteration in reverse to efficiently find all the subarrays with `k` as their median.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the array `nums = [3, 1, 4, 2]` with `n = 4` and let's find the number of subarrays where the median equals `k = 2`.

First, we need to find the index of `k` in `nums`; `k` is at index `i = 3`.

Now, let's analyze the elements to the right of `k`, which in this case, are none because `k` is at the last index. So, we move on to the next part of the process.

Next, we will analyze the elements to the left of `k`. We iterate from index `i - 1` to `0` (that is, from right to left). We initialize our balance counter `x` to `0` and a Counter to keep track of the occurrences of `x`.

- At index `i - 1` (index 2), the value is `4` (greater than `k`). We increment `x` to 1 (`x = x + 1`). There are no elements to the right, so we don't update our answer yet.
- At index `i - 2` (index 1), the value is `1` (less than `k`). We decrement `x` to 0 (`x = x - 1`). If we were to choose the subarray `[3, 1, 4, 2]` up to now, the sorted version would be `[1, 2, 3, 4]`, and `k` would be the median. We increment our answer `ans` by 1.
- At index `i - 3` (index 0), the value is `3` (greater than `k`). We increment `x` to 1 again. The subarray `[1, 4, 2]` has `k` as the median after sorting to `[1, 2, 4]`. `ans` is incremented by 1.

Finishing this process, we have found `2` subarrays where `k` is the median: `[3, 1, 4, 2]` and `[1, 4, 2]`. Therefore, by following the steps outlined in the solution approach, we would return `2` as the final answer.

Python Solution

```python
from collections import Counter

class Solution:
    def count_subarrays(self, nums: List[int], k: int) -> int:
        # Find the index of the first occurrence of k in nums.
        index_of_k = nums.index(k)

        # Initialize a counter to count occurrences of certain balances of values vs k.
        balance_counter = Counter()

        # Start with one subarray that includes k itself.
        total_subarrays = 1

        # 'balance' keeps track of the difference between the count of numbers greater than k
        # and count of numbers less than k.
        balance = 0

        # Look at the elements to the right of k in nums.
        for value in nums[index_of_k + 1:]:
            # If the number is greater than k, increment the balance; if less, decrement.
            balance += 1 if value > k else -1

            # Increment the count for subarrays where balance is 0 (equal numbers of both sides) or 1.
            total_subarrays += 0 <= balance <= 1

            # Update the counter for current balance.
            balance_counter[balance] += 1

        # Reset balance for the left side of k.
        balance = 0

        # Look at the elements to the left of k in nums.
        for j in range(index_of_k - 1, -1, -1):
            # Adjust balance.
            balance += 1 if nums[j] > k else -1

            # Count subarrays where balance is 0 or 1 just by k itself.
            total_subarrays += 0 <= balance <= 1

            # Use the counter balance to find valid subarrays that when combined
            # with the current left part forms a balanced subarray.
            total_subarrays += balance_counter[-balance] + balance_counter[-balance + 1]

        # Return the total number of subarrays.
        return total_subarrays

```

Java Solution

```java
class Solution {
    public int countSubarrays(int[] nums, int k) {
        // Getting the total number of elements in the nums array
        int numberOfElements = nums.length;
        // Iterator 'i' is used to find the position of the number 'k' in the array
        int i = 0;
        while (nums[i] != k) {
            ++i;
        }

        // Initialize an array to keep track of counts
        int[] counts = new int[numberOfElements * 2 + 1];
        // Variable 'answer' is used for storing the final answer
        int answer = 1;
        // Variable 'x' stores the accumulated count
        int x = 0;

        // Check to the right of the position where k was found
        for (int j = i + 1; j < numberOfElements; ++j) {
            // If current element is greater than k, increment x; otherwise decrement x
            x += nums[j] > k ? 1 : -1;
            // If x is in the range of 0 to 1 (inclusive), increment the answer
            if (x >= 0 && x <= 1) {
                ++answer;
            }
            // Increment the count of the position x in the counts array
            ++counts[x + numberOfElements];
        }
        // Reset 'x' for the second loop
        x = 0;
        // Check to the left of the position where k was found
        for (int j = i - 1; j >= 0; --j) {
            // If current element is greater than k, increment x; otherwise decrement x
            x += nums[j] > k ? 1 : -1;
            // If x is in the range of 0 to 1 (inclusive), increment the answer
            if (x >= 0 && x <= 1) {
                ++answer;
            }
            // Add to the answer the counts for -x and -x + 1 in the counts array
            answer += counts[-x + numberOfElements] + counts[-x + 1 + numberOfElements];
        }

        // Returning the final answer which is the total count of valid sub-arrays
        return answer;
    }
}

```

C++ Solution

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

class Solution {
public:
    int countSubarrays(std::vector<int>& nums, int k) {
        int numElements = nums.size();

        // Find the position of the first occurrence of k.
        int kPosition = std::find(nums.begin(), nums.end(), k) - nums.begin();

        // Initialize a count array with double the size of nums and an extra one, and set to all zeros.
        int countArray[(numElements << 1) | 1]; // Using bit manipulation to calculate 2*numElements+1 for size.
        std::memset(countArray, 0, sizeof(countArray));

        // Initialize answer with 1 counting the subarray that consists of only element k itself.
        int answer = 1;

        // This variable will store the relative comparison count.
        int compCount = 0;

        // Iterate through the elements to the right of k and count eligible subarrays.
        for (int rightIndex = kPosition + 1; rightIndex < numElements; ++rightIndex) {
            compCount += nums[rightIndex] > k ? 1 : -1;
            if (compCount >= 0 && compCount <= 1) {
                ++answer;
            }
            ++countArray[compCount + numElements];
        }

        // Reset comparison count for left side iteration.
        compCount = 0;

        // Iterate through the elements to the left of k and count eligible subarrays.
        for (int leftIndex = kPosition - 1; leftIndex >= 0; --leftIndex) {
            compCount += nums[leftIndex] > k ? 1 : -1;
            if (compCount >= 0 && compCount <= 1) {
                ++answer;
            }
            // Add count of subarrays matching the required condition.
            answer += countArray[-compCount + numElements] + countArray[-compCount + 1 + numElements];
        }

        // Return the total count of subarrays.
        return answer;
    }
};

```

Typescript Solution

```typescript
function countSubarrays(nums: number[], k: number): number {
    // Find the index of value 'k' in the array.
    const indexOfK = nums.indexOf(k);
    const lengthOfNums = nums.length;

    // Initialize a counting array to store frequencies of different sums.
    // The size is made to be twice the length of the input array to avoid negative indexing.
    // An extra slot is added to accommodate a '0' sum at the center.
    const count = new Array((lengthOfNums << 1) | 1).fill(0);

    // Initialize the answer with 1 to account for the 'k' itself.
    let answer = 1;
    let sum = 0;

    // Count for subarrays starting from the index after 'k'.
    for (let j = indexOfK + 1; j < lengthOfNums; ++j) {
        // Ternary operator: if the current element is greater than 'k', increment sum,
        // otherwise decrement sum.
        sum += nums[j] > k ? 1 : -1;
        // If sum is in [0,1] it is a valid subarray ending with nums[j].
        answer += sum >= 0 && sum <= 1 ? 1 : 0;
        // Increment the count of the current sum.
        ++count[sum + lengthOfNums];
    }

    // Reset sum for counting subarrays starting before 'k'.
    sum = 0;

    // Count for subarrays starting before 'k'.
    for (let j = indexOfK - 1; j >= 0; --j) {
        // Increase or decrease sum depending on whether the value is greater than 'k'.
        sum += nums[j] > k ? 1 : -1;
        // If sum is in [0,1] it is a valid subarray starting with nums[j].
        answer += sum >= 0 && sum <= 1 ? 1 : 0;
        // Add the count of subarrays that would form a valid sum when combined.
        answer += count[-sum + lengthOfNums] + count[-sum + 1 + lengthOfNums];
    }

    // Return the total count of valid subarrays.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed in the following steps:

1. Finding the index of `k` in `nums` using `nums.index(k)` – This operation is `O(n)` where `n` is the number of elements in `nums`.

2. Iterating over the elements to the right of `k`, incrementing `x`, and updating the `ans` and the `Counter` for each element – The iteration is done once for each element to the right of `k`, which is `O(n)` in the worst case (when `k` is the first element).

3. Iterating over the elements to the left of `k`, incrementing `x`, and updating `ans` based on the current `x` value and the count from the `Counter` – Similar to the step above, this is also `O(n)` in the worst case (when `k` is the last element).

Given that these steps are performed sequentially, we add the complexities resulting in a total time complexity of `O(n) + O(n) + O(n) = O(n)`.

Space Complexity

The space complexity of the code can be analyzed by considering the additional data structures used:

1. `cnt` - A `Counter` object which in the worst case stores counts for each possible distinct value of `x`. Since `x` represents a running difference between the count of values greater than `k` and less than `k`, `x` can be at most `n` in absolute value in the worst case, leading to `2n + 1` different possible values (including zero). Thus, the `Counter` can have at most `O(n)` elements.

Hence, the space complexity of the code is `O(n)`.