2841. Maximum Sum of Almost Unique Subarray

Medium Array Hash Table Sliding Window Leetcode Link

Problem Description

In this problem, we are provided with an integer array `nums`, and two positive integers `m` and `k`. Our task is to find the maximum sum of all *almost unique* subarrays of length `k` from `nums`. An *almost unique* subarray is defined as a subarray that contains at least `m` distinct elements. A subarray refers to a contiguous sequence of elements within the array that is non-empty. If there is no subarray that meets the criteria, then the function should return `0`.

Intuition

To tackle this problem, the intuition is to use a sliding window approach combined with a Counter (dictionary in Python) to keep track of the frequency of each element within the window. Here is the thinking process:

1. We start by creating a window of size `k` at the beginning of the array and use a Counter to track the elements' frequencies within this window.
2. We calculate the sum of the elements in the initial window as a possible candidate for the maximum sum.
3. If the number of distinct elements in this window is at least `m`, we consider the sum of this window as the initial answer.
4. We then slide the window by one position to the right, updating the Counter and sum accordingly by adding the new element to the window and removing the leftmost element that goes out of the window.
5. After updating the Counter and sum for the new window, we check if the window still contains at least `m` distinct elements. If it does, we compare the current sum with the previous maximum sum and update the maximum sum if the current sum is higher.
6. This process continues for each possible subarray of size `k` until we reach the end of the array.
7. The highest sum found during this sliding window process is returned as the result. If no such subarray is found throughout the process, the return value will be `0`.

By utilizing a sliding window and a Counter, we maintain an efficient check over the uniqueness of the elements and the sum of the elements in the subarray, thus arriving at the desired solution in a linear time complexity relative to the size of the input array.

Solution Approach

The solution uses a sliding window approach to efficiently calculate the sum of each `k`-length subarray while tracking the distinct elements within that subarray. Here's a step-by-step walkthrough of the implementation:

1. **Initialize the Counter**:
   - Create a `Counter` that will hold the frequency of each element in the current window.
   - Initialize the sum, `s`, of the first window by adding up the first `k` elements of `nums`.
2. **Initial Check**:
   - Check if the initial window (first `k` elements) is *almost unique*, meaning it has at least `m` distinct elements. If it is, store this sum as `ans`.
3. **Sliding the Window**:
   - Loop through the array starting from the `(k+1)`-th element to the end.
   - For each step in the loop:
     - Increment the count of the new element that enters the window into the Counter.
     - Decrement the count of the element that is leaving the window.
     - Update the sum, `s`, by subtracting the value of the element that leaves the window and adding the value of the element that enters the window.
4. **Update the Counter and Sum**:
   - If the count of the element that leaves the window drops to zero, remove it from the Counter since it's no longer part of the current window. This step is crucial as it ensures the Counter only contains elements present in the window.
5. **Update the Maximum Sum**:
   - After updating the Counter for the new window, check again if we have at least `m` distinct elements.
   - If the current subarray satisfies the *almost unique* condition, compare the current sum with the previously stored maximum sum, `ans`, and update `ans` if the current sum is greater.
6. **Return the Result**:
   - After processing all possible windows, return the maximum sum found, stored in `ans`. If no valid window was found, `ans` will be `0`, which is the correct return value in that case.

This algorithm efficiently computes the maximum sum of *almost unique* subarrays with a time complexity of O(n), where `n` is the length of `nums`, by performing constant work for each element in `nums`. The use of the sliding window pattern combined with a Counter for frequency tracking is what makes this approach effective.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Suppose we have the array `nums = [4, 3, 2, 3, 5, 4, 1]`, and we want to find the maximum sum of *almost unique* subarrays of length `k = 3` with at least `m = 2` distinct elements.

1. **Initialize the Counter**:
   - Create a `Counter` and initally it would be empty as no window is considered yet.
   - We start with the first window `nums[0:3]` which is `[4, 3, 2]`. The sum `s` is `4 + 3 + 2 = 9`.
2. **Initial Check**:
   - The initial window `[4, 3, 2]` has exactly `3` distinct elements, which is more than `m = 2`. Therefore, we initially store this sum as `ans = 9`.
3. **Sliding the Window**:
   - Move to the next window by sliding one position to the right. Now we consider `nums[1:4]`, which is `[3, 2, 3]`.
   - Update the Counter by decrementing the count of `4` (as it leaves the window) and incrementing the count of the new element `3` (added again but already present).
4. **Update the Counter and Sum**:
   - Remove `4` from the Counter if its count drops to `0` (in this case, it does).
   - The sum of the new window `[3, 2, 3]` is now `3 + 2 + 3 = 8`. We update `s` accordingly.
5. **Update the Maximum Sum**:
   - The new window `[3, 2, 3]` has only `2` distinct elements which meets the *almost unique* condition of at least `m = 2`.
   - We compare the sum `8` with the previous maximum `ans = 9`. Since `8` is less than `9`, we do not update `ans`.
6. **Sliding Further**:
   - Repeat the process for the next window `[2, 3, 5]`. Remove `3` from the counter, add `5`, and calculate the new sum `s = 2 + 3 + 5 = 10`.
   - Check if the new sum `10` is greater than `ans = 9`. It is, so we update `ans` to `10`.
7. **Continue**:
   - Continue with the above steps until we have slid over the entire array with the window.
   - For `nums[3:6]` → `[3, 5, 4]`, the sum is `12` and it's *almost unique*, update `ans`.
   - For `nums[4:7]` → `[5, 4, 1]`, the sum is `10` but `ans` is still greater, so no change.
8. **Return the Result**:
   - After sliding through all windows, the highest sum we found was `12` for the subarray `[3, 5, 4]`. Therefore, we return `ans = 12`.
   - If we had not found any *almost unique* subarrays, we would return `0`.

Through this example, we've demonstrated how the sliding window approach combined with a Counter can be used to efficiently solve this problem. The key is to keep updating the sum and distinct element count for each window and maintaining the maximum sum encountered that meets the *almost unique* criteria.

Python Solution

```python
from collections import Counter

class Solution:
    def maxSum(self, nums: List[int], m: int, k: int) -> int:
        # Create a counter for the first 'k' elements in 'nums'
        element_count = Counter(nums[:k])
        # Calculate the sum of the first 'k' elements
        window_sum = sum(nums[:k])
        # Initialize the maximum sum as 0
        max_sum = 0

        # If the count of unique numbers is at least 'm', update the max_sum
        if len(element_count) >= m:
            max_sum = window_sum

        # Iterate through the array, moving the window by 1 element each time
        for i in range(k, len(nums)):
            # Update the counts and the sum for the new window
            element_count[nums[i]] += 1
            element_count[nums[i - k]] -= 1
            window_sum += nums[i] - nums[i - k]

            # Remove the element from the counter if its count drops to 0
            if element_count[nums[i - k]] == 0:
                del element_count[nums[i - k]]

            # If there are at least 'm' unique numbers, update the max_sum
            if len(element_count) >= m:
                max_sum = max(max_sum, window_sum)

        return max_sum

```

Java Solution

```java
class Solution {
    // Method to find the maximum sum of a subarray of size 'k' with at least 'm' distinct numbers
    public long maxSum(List<Integer> numbers, int m, int k) {
        // Map to store the count of unique numbers in the current window of size 'k'
        Map<Integer, Integer> countMap = new HashMap<>();

        // Getting the total number of elements in the list
        int totalNumbers = numbers.size();
        // Variable to store the sum of the current window
        long currentSum = 0;

        // Initialize the window with the first 'k' elements
        for (int i = 0; i < k; ++i) {
            countMap.merge(numbers.get(i), 1, Integer::sum);
            currentSum += numbers.get(i);
        }

        // Variable to store the maximum sum of a window with at least 'm' distinct numbers
        long maxSum = 0;

        // Check if the initial window meets the condition and update the maximum sum
        if (countMap.size() >= m) {
            maxSum = currentSum;
        }

        // Slide the window of size 'k' across the list
        for (int i = k; i < totalNumbers; ++i) {
            // Add the new element to the countMap and update the sum
            countMap.merge(numbers.get(i), 1, Integer::sum);
            // Remove the leftmost element from the window and update the sum
            if (countMap.merge(numbers.get(i - k), -1, Integer::sum) == 0) {
                countMap.remove(numbers.get(i - k));
            }

            // Update the current sum by adding the new element and removing the leftmost element
            currentSum += numbers.get(i) - numbers.get(i - k);

            // Check if after sliding the window we meet the condition and possibly update the maxSum
            if (countMap.size() >= m) {
                maxSum = Math.max(maxSum, currentSum);
            }
        }

        // Return the maximum sum found that satisfies the condition
        return maxSum;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to calculate the maximum sum of any contiguous subarray of length 'k'
    // that contains at least 'm' distinct numbers.
    long long maxSum(vector<int>& nums, int m, int k) {
        // Map to store the frequency count of each number in the current window
        unordered_map<int, int> frequencyCount;
        // Variable to store the current sum of the window
        long long currentSum = 0;
        // The size of the input array
        int n = nums.size();

        // Initialize the first window and its sum
        for (int i = 0; i < k; ++i) {
            frequencyCount[nums[i]]++;
            currentSum += nums[i];
        }

        // Start with an answer of 0 (if there are no subarrays with at least 'm' distinct numbers),
        // or the current sum (if the first window has at least 'm' distinct numbers)
        long long maxSum = frequencyCount.size() >= m ? currentSum : 0;

        // Iterate through the array, sliding the window forward
        for (int i = k; i < n; ++i) {
            // Include the next element in the count and sum
            frequencyCount[nums[i]]++;
            // Exclude the oldest element in the count and sum, and remove it from the map if count falls to 0
            if (--frequencyCount[nums[i - k]] == 0) {
                frequencyCount.erase(nums[i - k]);
            }
            // Adjust the current sum by adding the new element and subtracting the old one
            currentSum += nums[i] - nums[i - k];

            // If the current window contains at least 'm' distinct numbers, update maxSum
            if (frequencyCount.size() >= m) {
                maxSum = std::max(maxSum, currentSum);
            }
        }

        // Return the maximum sum found
        return maxSum;
    }
};

```

Typescript Solution

```typescript
function maxSum(nums: number[], distinctCount: number, windowSize: number): number {
    // `n` holds the total number of elements in the input array.
    const n: number = nums.length;

    // Initialize a Map to count occurrences of each number in the current window.
    const frequencyCounter: Map<number, number> = new Map();

    // `windowSum` tracks the sum of the current window of size `windowSize`.
    let windowSum: number = 0;

    // Initialize the frequency map and window sum for the first window.
    for (let i = 0; i < windowSize; ++i) {
        frequencyCounter.set(nums[i], (frequencyCounter.get(nums[i]) || 0) + 1);
        windowSum += nums[i];
    }

    // `maxSum` stores the maximum sum of a window where the distinct element count is at least `distinctCount`.
    let maxSum: number = frequencyCounter.size >= distinctCount ? windowSum : 0;

    // Slide the window from the beginning to the end of the array.
    for (let i = windowSize; i < n; ++i) {
        // Increment the count of the new element in the window.
        frequencyCounter.set(nums[i], (frequencyCounter.get(nums[i]) || 0) + 1);

        // Decrement the count of the element that is no longer in the window.
        const countOfRemovedElement: number = frequencyCounter.get(nums[i - windowSize])! - 1;
        frequencyCounter.set(nums[i - windowSize], countOfRemovedElement);

        // If an element count goes to zero, remove it from the map to maintain the distinct elements.
        if (countOfRemovedElement === 0) {
            frequencyCounter.delete(nums[i - windowSize]);
        }

        // Update the window sum by subtracting the element that was removed and adding the new element.
        windowSum += nums[i] - nums[i - windowSize];

        // If the current window has enough distinct elements, update `maxSum` if necessary.
        if (frequencyCounter.size >= distinctCount) {
            maxSum = Math.max(maxSum, windowSum);
        }
    }

    // Return the maximum sum of a window which has at least `distinctCount` distinct elements.
    return maxSum;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed as follows:

- The initialization of `cnt` with the first `k` elements takes `O(k)` time.
- The calculation of the sum of the first `k` elements is also `O(k)`.
- The `for` loop runs for `len(nums) - k` iterations. Within each iteration, the operations of updating the counter and sum are constant time, i.e., `O(1)`.
- However, the popping of elements from the counter can occur at most once per iteration and is also an `O(1)` operation in average case for a dictionary in Python.
- Checking the length of `cnt` and updating `ans` is `O(1)`.

Considering the `for` loop is the dominant part, the time complexity is `O(n-k)`, where `n` is the length of `nums`. Since `k` is subtracted from `n`, and in the worst case `k` could be much smaller than `n`, the more generalized form to express the time complexity is `O(n)`.

Space Complexity

The space complexity of the code is mainly due to the counter `cnt`, which will store at most `k` unique integers if all elements in the first `k` elements of `nums` are unique. Therefore, the space complexity is `O(k)` in the worst case.

To summarize:

- Time Complexity: `O(n)`
- Space Complexity: `O(k)`