2762. Continuous Subarrays

Medium · Queue · Array · Ordered Set · Sliding Window · Monotonic Queue · Heap (Priority Queue) · Leetcode Link

Problem Description

The problem presents a task where we are given an array of integers, `nums`, which is indexed starting from `0`. We are tasked with finding the total number of continuous subarrays within `nums`. The definition of a "continuous" subarray here is one in which the absolute difference between any two elements in the subarray is 2 or less. More formally, if we have subarray elements indexed from `i` to `j`, then for each pair of indices `i1` and `i2` such that `i <= i1, i2 <= j`, it is required that `0 <= |nums[i1] - nums[i2]| <= 2`. Our goal is to return the total count of these subarrays.

A subarray, in general terms, is a contiguous sequence of elements within an array, starting at any position and ending at any position after the start, and has at least one element.

The challenge, therefore, involves iterating over all possible subarrays, checking if they meet the continuous criteria, and keeping a running count of those that do.

Intuition

The intuition for solving this problem lies in recognizing that we can use a sliding window approach along with an efficiently updateable data structure to keep track of the elements within the current window (subarray).

We need two pointers: one (`i`) to mark the start of the window and another (`j`) to mark the end. As `j` moves forward to include new elements, we'll need to possibly shuffle `i` forward to maintain the "continuous" property of the subarray. A naive approach would have us re-checking all elements between `i` and `j` for every new element added to see if the continuous property holds, but this is inefficient and would lead to a solution that is too slow.

Instead, by using an ordered list to keep track of the elements in the current window, we can efficiently access the smallest and largest elements in this window. This lets us quickly determine if adding the new element at `j` violates the continuous property (if the difference between the max and min in the ordered list exceeds 2). If it does, we incrementally adjust `i`, removing elements from the ordered list that are now outside the new window boundary set by `i`.

This sliding window moves across `nums`, always holding a "continuous" subarray (as per the problem's definition). The count of such subarrays, when the window ends at `j`, would be `j - i + 1` because subarrays of all possible lengths that end at `j` are valid, as long as the smallest and largest elements in them are within 2 of each other.

Leveraging the ordered nature of the list, the solution manages to satisfy the constraints of the problem in a much more efficient way. It ensures that the check for the continuous property is done in constant time with respect to the size of the subarray, which greatly enhances the performance of the solution. The total number of continuous subarrays is obtained by summing up these counts throughout the iteration over `nums`.

Solution Approach

The implementation of the solution involves a combination of a sorted list data structure and a two-pointer technique, which allows us to maintain and iterate through a set of subarrays efficiently.

Here's a step-by-step breakdown of the approach:

1. **Initialize Variables**: We initialize two variables, `ans` to hold the final count of continuous subarrays and `i` to represent the starting index of the current window. We also prepare a `SortedList` named `sl` which will be used to maintain the elements within the current window.
2. **Iterate with Two Pointers**: We iterate over the array using a loop where the index `j` represents the end of the current subarray. For each index `j`, we do the following:
   - Add the element `nums[j]` to the sorted list `sl`. This helps us automatically keep track of the ordered elements in the current subarray.
   - **Check for "Continuous" Property**: While the difference between the maximum (`sl[-1]`) and minimum (`sl[0]`) elements in `sl` is greater than 2, we remove the element at index `i` from `sl` and increment `i`. By doing so, we shrink the window from the left to maintain the continuous property of the subarray.
3. **Count Subarrays**: Once the continuous property is ensured, we increment the answer `ans` by the length of the sorted list `sl`, which represents the count of continuous subarrays ending at index `j`. The formula is `ans += len(sl)`. This is because each element from `i` to `j` could be the starting point of a subarray that ends with element `nums[j]`.
4. **Final Answer**: After we complete the loop, the variable `ans` holds the total count of continuous subarrays in `nums`.

The sorted list is essential in this algorithm because it allows us to efficiently maintain the smallest and largest elements within the current window. The two pointers technique is an elegant way to navigate the array, as it dynamically adjusts the window to always fit the continuous property without the need to start over for every new element.

By combining these patterns, the reference solution efficiently solves the problem with a time complexity that is typically less than what would be required by a brute-force approach which would use nested loops to consider every subarray individually.

Example Walkthrough

Let's walk through a small example to illustrate how the solution approach works. Suppose we have the following array of integers:

```
nums = [1, 3, 5, 2, 4]
```

We want to find the total number of continuous subarrays where the absolute difference between any two elements is 2 or less.

Following the solution approach:

1. **Initialize Variables**: We start by setting `ans = 0` to hold the count and initialize `i = 0`. We also prepare an empty sorted list `sl`.
2. **Iterate with Two Pointers (j)**:
   - For `j = 0` (handling the first element `nums[0]` which is `1`), we add `nums[j]` to `sl` making it `[1]`. Since `sl` contains only one element, the continuous property is maintained. We increase `ans` by 1 (`len(sl)`).
   - For `j = 1` (`nums[1]` which is `3`), we add `nums[j]` to `sl` resulting in `[1, 3]`. The continuous property is satisfied (`3 - 1 <= 2`). We increase `ans` by 2 (`len(sl)`).
   - For `j = 2` (`nums[2]` which is `5`), adding `nums[j]` would change `sl` to `[1, 3, 5]`. However, the continuous property is violated (`5 - 1 > 2`). We remove `nums[i]` (which is `1`) and increment `i` to be `1`. Now `sl` becomes `[3, 5]` and continuous property is maintained. We increase `ans` by 2 (`len(sl)`).
   - For `j = 3` (`nums[3]` which is `2`), we add `nums[j]` to `sl` getting `[2, 3, 5]`. However, the difference `5 - 2` is greater than 2. We remove `nums[i]` (which is `3`) and increment `i` to be `2`. `sl` is now `[2, 5]`, still not continuous. We remove `nums[i]` again (which is `5`) and increment `i` to be `3`. Now `sl` has only `[2]` and it is continuous. We increase `ans` by 1 (`len(sl)`).
   - For `j = 4` (`nums[4]` which is `4`), after adding `nums[j]` to `sl` we have `[2, 4]`. The subarray is continuous (`4 - 2 <= 2`). We increase `ans` by 2 (`len(sl)`).
3. **Count Subarrays**: By iterating from `j` = 0 to 4, we increment `ans` at each step. Our final answer is the sum of these increments.
4. **Final Answer**: By adding up all the increments we made to `ans`, we get the total count of continuous subarrays.

The final number of continuous subarrays is 1 + 2 + 2 + 1 + 2 = 8 subarrays.

Python Solution

```python
from sortedcontainers import SortedList
from typing import List

class Solution:
    def continuousSubarrays(self, nums: List[int]) -> int:
        # Initialize the count of valid subarrays as 0
        count_of_valid_subarrays = 0

        # Initialize the start index for the sliding window
        start_index = 0

        # Create a SortedList to maintain sorted order of elements within the window
        sorted_window = SortedList()

        # Iterate through the elements of nums
        for num in nums:
            # Add the current number to the sorted window
            sorted_window.add(num)

            # Ensure the difference between the max and min within the window is at most 2
            while sorted_window[-1] - sorted_window[0] > 2:
                # Remove the element at the start index since it causes the diff > 2
                sorted_window.remove(nums[start_index])

                # Move the start index of the window to the right
                start_index += 1

            # Add the number of valid subarrays ending at the current index
            count_of_valid_subarrays += len(sorted_window)

        # Return the total count of valid subarrays found
        return count_of_valid_subarrays

```

Java Solution

```java
class Solution {
    public long countContinuousSubarrays(int[] nums) {
        // Initialize the result count
        long totalCount = 0;
        // Initialize two pointers for the sliding window
        int windowStart = 0;
        int n = nums.length;
        // TreeMap to keep track of the frequency of each number in the current window
        // The TreeMap's keys are sorted, so we can efficiently access the smallest and largest values
        TreeMap<Integer, Integer> frequencyMap = new TreeMap<>();

        // Iterate over all elements in the array using the 'windowEnd' pointer
        for (int windowEnd = 0; windowEnd < n; ++windowEnd) {
            // Increment the frequency of the current number, initializing it to 1 if it doesn't exist
            frequencyMap.merge(nums[windowEnd], 1, Integer::sum);
            // Shrink the window from the left if the condition is breached:
            // The difference between the maximum and minimum is greater than 2
            while (frequencyMap.lastEntry().getKey() - frequencyMap.firstEntry().getKey() > 2) {
                // Reduce the frequency of the number at 'windowStart'
                frequencyMap.merge(nums[windowStart], -1, Integer::sum);
                // If the frequency drops to 0, remove it from the map
                if (frequencyMap.get(nums[windowStart]) == 0) {
                    frequencyMap.remove(nums[windowStart]);
                }
                // Move the start of the window forward
                ++windowStart;
            }
            // Add the number of subarrays ending at 'windowEnd' which are valid
            totalCount += windowEnd - windowStart + 1;
        }
        // Return the total number of continuous subarrays found
        return totalCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long continuousSubarrays(vector<int>& nums) {
        long long count = 0; // This variable holds the total number of continuous subarrays found
        int start = 0; // This variable marks the start of the current subarray
        int n = nums.size(); // Size of the input array
        multiset<int> windowElements; // Multiset to store elements of the current window

        // Iterate over the input array
        for (int end = 0; end < n; ++end) {
            // Insert the current element to the multiset
            windowElements.insert(nums[end]);

            // Ensure the difference between the maximum and minimum elements in the multiset does not exceed 2
            while (*windowElements.rbegin() - *windowElements.begin() > 2) {
                // If the condition is broken, erase the element pointed by 'start' from the multiset
                windowElements.erase(windowElements.find(nums[start]));
                // Move start forward as the beginning of the current subarray is no longer valid
                start++;
            }

            // Calculate the number of subarrays ending at 'end' and starting from any
            // index from 'start' to 'end', and add to the total count
            count += end - start + 1;
        }
        // Return the final count of continuous subarrays
        return count;
    }
};

```

Typescript Solution

```typescript
function continuousSubarrays(nums: number[]): number {
    let count: number = 0; // This variable holds the total number of continuous subarrays found
    let start: number = 0; // This variable marks the start of the current subarray
    const n: number = nums.length; // Size of the input array
    const windowElements: Set<number> = new Set(); // Set to store unique elements of the current window
    const windowCounts: Map<number, number> = new Map(); // Map to count occurrences of elements in the current window

    // Iterate over the input array
    for (let end = 0; end < n; ++end) {
        let element = nums[end];
        // Insert the current element to the windowCounts, or update its count if already present
        windowCounts.set(element, (windowCounts.get(element) || 0) + 1);
        // Insert the current element to the windowElements set
        windowElements.add(element);

        // Ensure the difference between the maximum and minimum elements in the windowElements does not exceed 2
        while (Math.max(...windowElements) - Math.min(...windowElements) > 2) {
            // Fetch the element at the 'start' index
            let startElement: number = nums[start];
            let elementCount: number = windowCounts.get(startElement) || 0;

            // If only 1 instance of this element exists in the windowCounts, remove it from windowElements
            if (elementCount === 1) {
                windowElements.delete(startElement);
            }
            // Decrease the count of this element in the windowCounts
            windowCounts.set(startElement, elementCount - 1);

            // Move start forward as the beginning of the current subarray is no longer valid
            start++;
        }

        // Calculate the number of subarrays ending at 'end' and starting from any
        // index from 'start' to 'end', and add to the total count
        count += end - start + 1;
    }
    // Return the final count of continuous subarrays
    return count;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n * log n)`. Here's the breakdown of why this is:

- The main loop runs for each of the `n` elements in the `nums` list.
- Inside the loop, we have operations like `sl.add(x)` and `sl.remove(nums[i])`, which are operations on a `SortedList`.
- A `SortedList` maintains its order upon insertions and deletions, and these operations tend to have `O(log n)` complexity on average.
- Since these `O(log n)` operations are performed for each element in the loop, the total time complexity becomes `O(n * log n)`.

The space complexity of the code is `O(n)`. This is because:

- We create a `SortedList` which, in the worst case, might need to store all `n` elements of the `nums` list.
- No other data structures that grow with the size of the input are used, hence no additional space complexity that would outweigh `O(n)`.