1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

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

Problem Description

The problem provides an array of integers named `nums` and an integer called `limit`. The task is to find the length of the longest non-empty contiguous subarray (a sequence of adjacent elements from the array) where the absolute difference between any two elements in the subarray does not exceed the `limit` value.

For example, if the input array is `[10, 1, 2, 4, 7, 2]` and the limit is `5`, the longest subarray where the absolute difference between any two elements is less than or equal to `5` is `[1, 2, 4, 7, 2]`, which has a length of `5`.

The two key aspects of the problem are:

- Working with contiguous elements (subarray), not just any subsets of the array.
- Ensuring that **every** pair of elements in the subarray has an absolute difference of at most `limit`.

Intuition

The solution approach involves using a data structure that maintains a sorted order of elements. This allows the efficient retrieval of the smallest and largest elements in the current window (subarray) to check if their absolute difference is within the `limit`.

A suitable data structure for this problem is a SortedList, provided by the `sortedcontainers` library in Python. Here's how we can arrive at the solution:

1. Maintain a sliding window that expands and contracts as we iterate through the `nums` array.
2. In each iteration, add the current element to the SortedList, which is our window.
3. Check if the absolute difference between the smallest and largest elements in the SortedList exceeds the `limit`.
4. If it does, we remove the leftmost element from our window (which was first added when the window was last valid) to try and bring the difference back within `limit`.
5. We keep track of the maximum size of the window that satisfied the condition of staying within the `limit`.

This approach ensures that at any given point, we have the longest valid subarray ending at the current position, and we keep updating the answer with the maximum size found so far.

The reason we use a SortedList instead of sorting the window array in each iteration is the time complexity—SortedList maintains the order of elements with a far lesser time complexity for insertion and removal compared to sorting an array at each step.

Solution Approach

The implementation is based on the sliding window pattern, which is an optimization technique to reduce repeated work and maintain a range of elements that fulfill certain criteria. Here is a detailed explanation:

1. Initialize a `SortedList` named `sl` and two pointers for the window indices `i` and `j` with `i` being the right pointer and `j` being the left one. Also, initialize a variable `ans` to store the longest length of the subarray found so far.

2. Loop through the elements of `nums` with index `i` and value `v`.

   - Add the new element `v` to the `SortedList`. Because the list is always sorted, doing this helps us quickly reference the smallest and largest elements up to the current position.
3. Inside the loop, check if the current window (from `j` to `i` inclusively) is valid, meaning that the absolute difference between the largest (`sl[-1]`) and smallest (`sl[0]`) elements in the `SortedList` does not exceed the `limit`.

   - If the limit is exceeded, we need to contract the window by removing the leftmost element. This is done by removing `nums[j]` from `SortedList` since `j` corresponds to the leftmost index of the window.
   - Increment `j` to move the start of the window to the right.
4. Update the answer `ans` with the maximum length found so far. We compute the current window size with `i - j + 1`, as `i` is the end of the window and `j` is the beginning.

5. After the loop ends, return `ans`, which holds the size of the longest subarray found that satisfies the condition.

Here is a code snippet to illustrate the solution's core logic:

```python
sl = SortedList()  # Instantiate a SortedList data structure.
ans = j = 0  # Initialize the answer and the left pointer of the window to 0.
for i, v in enumerate(nums):
    sl.add(v)
    while sl[-1] - sl[0] > limit:  # If current window is invalid, contract it.
        sl.remove(nums[j])
        j += 1
    ans = max(ans, i - j + 1)  # Store the largest size of the valid window.
return ans
```

The SortedList is efficient because it keeps elements sorted at all times. Hence, we can always get the smallest and largest element in O(1) time and remove elements in O(log N) time, where `N` is the number of elements in the list. This is much more optimal than sorting a list which would take O(N log N) time for each change in the window.

Overall, the use of a sliding window algorithm with SortedList allows us to efficiently solve this problem with a time complexity that depends on inserting and removing each element into the sorted list (generally O(log N) for each operation), rather than re-evaluating the entire subarray every time.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Consider the input array `nums = [4, 2, 2, 5, 4]` with `limit = 2`.

Following the proposed solution:

1. We initialize an empty `SortedList` named `sl`, set two pointers for the window indices `j = 0` and `i = 0`, and `ans = 0`, the variable that will hold the answer.

2. We start iterating through the elements of `nums`.

   - For `i = 0` (`v = 4`): Add `4` to `sl`, so `sl = [4]`. The window `[4]` is valid because there is only one element. Update `ans` to `1`.
3. Move to `i = 1` (`v = 2`):

   - Add `2` to `sl`, leading to `sl = [2, 4]`. The window `[4, 2]` is valid as `4 - 2 = 2`, which does not exceed the limit. Update `ans` to `2`.
4. Proceed to `i = 2` (`v = 2`):

   - Add another `2` to `sl`, so `sl = [2, 2, 4]`. The window `[4, 2, 2]` is still valid for the same reasons. Update `ans` to `3`.
5. Move on to `i = 3` (`v = 5`):

   - Add `5` to `sl`, which yields `sl = [2, 2, 4, 5]`. The window `[4, 2, 2, 5]` is invalid because `5 - 2 = 3`, which is larger than the limit. We remove the leftmost element (`nums[j]` which is `4`) from `sl`, resulting in `sl = [2, 2, 5]`, and increment `j` to `1`. The updated window `[2, 2, 5]` is now valid. Update `ans` to `3`.
6. Finally, for `i = 4` (`v = 4`):

   - We add `4` to `sl` to get `sl = [2, 2, 4, 5]`. The window `[2, 2, 5, 4]` is again invalid because `5 - 2 = 3` exceeds the limit. We remove `nums[j]` (which is now the leftmost `2`) from `sl`, making it `sl = [2, 4, 5]`, and increment `j` to `2`. The new window `[2, 5, 4]` is valid and its size (`3`) is used to update `ans` if it's larger than the current `ans`.
7. Having iterated through all elements, we find that the longest subarray where the absolute difference between any two elements does not exceed the `limit` is `3`. Thus, we return `ans = 3`.

This example shows how the sliding window moves through the array and adjusts by adding new elements and potentially removing the leftmost element to maintain a valid subarray within the `limit`. The `SortedList` makes it efficient to find the minimum and maximum within the current window to decide if the subarray satisfies the condition.

Python Solution

```python
# We import SortedList from the sortedcontainers module
from sortedcontainers import SortedList
from typing import List  # Import List from typing module for type annotation

class Solution:
    def longest_subarray(self, nums: List[int], limit: int) -> int:
        # Initialize a SortedList which allows us to maintain a sorted collection of numbers
        sorted_list = SortedList()

        # Initialize variables for the answer and the start index of the window
        max_length = 0
        window_start = 0

        # Iterate through the array with index and value
        for window_end, value in enumerate(nums):
            # Add the current value to the sorted list
            sorted_list.add(value)

            # Shrink the window from the left if the condition is violated
            # The condition being if the absolute difference between the max and min values in the window exceeds the limit
            while sorted_list[-1] - sorted_list[0] > limit:
                # Remove the leftmost value from the sorted list as we're shrinking the window
                sorted_list.remove(nums[window_start])
                # Move the start of the window to the right
                window_start += 1

            # Calculate the length of the current window and compare with the max
            # Update the max_length as needed
            max_length = max(max_length, window_end - window_start + 1)

        # Return the length of the longest subarray after examining all windows
        return max_length

# Example usage
# sol = Solution()
# result = sol.longest_subarray([10,1,2,4,7,2], 5)
# print(result)  # Output: 4

```

Java Solution

```java
class Solution {
    public int longestSubarray(int[] nums, int limit) {
        // Create a TreeMap to keep track of the frequency of each number
        TreeMap<Integer, Integer> frequencyMap = new TreeMap<>();
        int maxLength = 0; // Stores the maximum length of the subarray
        int left = 0; // The left pointer for our sliding window

        // Iterate over the array using the right pointer 'right'
        for (int right = 0; right < nums.length; ++right) {
            // Update the frequency of the current number
            frequencyMap.put(nums[right], frequencyMap.getOrDefault(nums[right], 0) + 1);

            // Shrink the sliding window until the absolute difference between the max
            // and min within the window is less than or equal to 'limit'
            while (frequencyMap.lastKey() - frequencyMap.firstKey() > limit) {
                // Decrease the frequency of the number at the left pointer
                frequencyMap.put(nums[left], frequencyMap.get(nums[left]) - 1);
                // If the frequency drops to zero, remove it from the frequency map
                if (frequencyMap.get(nums[left]) == 0) {
                    frequencyMap.remove(nums[left]);
                }
                // Move the left pointer to the right, shrinking the window
                ++left;
            }

            // Update the maximum length found so far
            maxLength = Math.max(maxLength, right - left + 1);
        }
        // Return the maximum length of the subarray that satisfies the condition
        return maxLength;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to calculate the length of the longest subarray with the absolute difference
    // between any two elements not exceeding `limit`.
    int longestSubarray(vector<int>& nums, int limit) {
        // Initialize a multiset to maintain the elements in the current sliding window.
        multiset<int> window_elements;
        int longest_subarray_length = 0; // Variable to keep track of the max subarray length.
        int window_start = 0; // Starting index of the sliding window.

        // Iterate over the array using `i` as the end of the sliding window.
        for (int window_end = 0; window_end < nums.size(); ++window_end) {
            // Insert the current element into the multiset.
            window_elements.insert(nums[window_end]);

            // If the difference between the largest and smallest elements in the multiset
            // exceeds the `limit`, shrink the window from the left until the condition is satisfied.
            while (*window_elements.rbegin() - *window_elements.begin() > limit) {
                // Erase the leftmost element from the multiset and shrink the window.
                window_elements.erase(window_elements.find(nums[window_start++]));
            }

            // Calculate the length of the current subarray and update the maximum length.
            int current_subarray_length = window_end - window_start + 1;
            longest_subarray_length = max(longest_subarray_length, current_subarray_length);
        }

        // Return the length of the longest subarray found.
        return longest_subarray_length;
    }
};

```

Typescript Solution

```typescript
type CompareFunction<T> = (a: T, b: T) => number;

interface ITreapNode<T> {
    value: T;
    count: number;
    size: number;
    priority: number;
    left: ITreapNode<T> | null;
    right: ITreapNode<T> | null;
}

let compareFn: CompareFunction<any>;
let leftBound: any;
let rightBound: any;
let root: ITreapNode<any>;

function getSize(node: ITreapNode<any> | null): number {
    return node?.size ?? 0;
}

function getFac(node: ITreapNode<any> | null): number {
    return node?.priority ?? 0;
}

function createTreapNode<T>(value: T): ITreapNode<T> {
    const node: ITreapNode<T> = {
        value: value,
        count: 1,
        size: 1,
        priority: Math.random(),
        left: null,
        right: null,
    };

    return node;
}

function pushUp(node: ITreapNode<any>): void {
    let tmp = node.count;
    tmp += getSize(node.left);
    tmp += getSize(node.right);
    node.size = tmp;
}

function rotateRight<T>(node: ITreapNode<T>): ITreapNode<T> {
    const left = node.left;
    node.left = left?.right ?? null;
    if (left) {
        left.right = node;
    }
    if (node.right) pushUp(node.right);
    pushUp(node);
    return left ?? node;
}

function rotateLeft<T>(node: ITreapNode<T>): ITreapNode<T> {
    const right = node.right;
    node.right = right?.left ?? null;
    if (right) {
        right.left = node;
    }
    if (node.left) {
        pushUp(node.left);
    }
    pushUp(node);
    return right ?? node;
}

// ... (Other methods would be similarly defined as global functions, but not included here for brevity)

// Initialize the global Treap
function initTreap<T>(
    compFn: CompareFunction<T>,
    leftBnd: T = -Infinity as unknown as T,
    rightBnd: T = Infinity as unknown as T,
): void {
    compareFn = compFn as unknown as CompareFunction<any>;
    leftBound = leftBnd;
    rightBound = rightBnd;
    const treapRoot: ITreapNode<any> = createTreapNode<any>(rightBound);
    treapRoot.priority = Infinity;
    treapRoot.left = createTreapNode<any>(leftBound);
    treapRoot.left.priority = -Infinity;
    pushUp(treapRoot.left);
    pushUp(treapRoot);
    root = treapRoot;
}

// Example usage of initializing and using the treap
initTreap<number>((a, b) => a - b);
addNode(root, 10); // This assumes the addNode function is implemented globally as mentioned above.

```

Time and Space Complexity

Time Complexity

The provided code utilizes a sliding window approach within a loop and a SortedList to keep track of the order of elements. For each element in `nums`, it is added to the `SortedList`, which is typically an `O(log n)` operation due to the underlying binary search tree or similar data structure used for keeping the list sorted.

The `while` loop inside the `for` loop is executed only when the current subarray does not meet the limit condition. Within the loop, `remove(nums[j])` is called, which is also an `O(log n)` operation because the list must be searched for the value to remove it, and it might need restructuring to keep it sorted.

The variable `ans` is updated using a `max()` function which is an `O(1)` operation.

Since the `for` loop iterates over each element in `nums` once, and the inner `while` loop only processes each element once due to the sliding window mechanism, the overall time complexity is `O(n log n)`, where `n` is the number of elements in the `nums` list.

Space Complexity

The additional space used by the algorithm consists of the `SortedList` and the variables used for iteration and storing the current longest subarray's length. The space complexity of the `SortedList` depends on the number of unique elements inserted. In the worst-case scenario, all elements of `nums` are different, and the `SortedList` will contain `n` elements, leading to a space complexity of `O(n)`.

The space for the other variables is comparatively negligible (`O(1)`), so the overall space complexity is `O(n)`.