34. Find First and Last Position of Element in Sorted Array

Medium Array Binary Search [Leetcode Link]

Problem Description

Given an array of integers, `nums`, which is sorted in non-decreasing order, we want to find the starting and ending position of a specified `target` value within that array. The problem specifically asks us for the indices of the first and last occurrence of the `target` in `nums`. If the `target` is not present in the array, the function should return the array `[-1, -1]`.

Since the array is sorted, we can leverage binary search to find the required indices efficiently. The algorithm we need to implement should have a runtime complexity of O(log n), which is characteristic of binary search algorithms. This suggests that a simple linear scan of the array to find the `target` is not sufficient, as it would have a runtime complexity of O(n) and would not meet the efficiency requirement of the problem.

Intuition

To find the positions efficiently, one approach is to perform two binary searches. The first binary search finds the left boundary (the first occurrence) of the `target`, and the second binary search finds the right boundary (the last occurrence).

The Python solution uses the `bisect_left` function from the `bisect` module to perform binary searches. This function is handy for finding the insertion point for a given element in a sorted array, which is equivalent to finding the lower bound of the `target`.

For the left boundary, `bisect_left(nums, target)` finds the index `l` where `target` should be inserted to maintain the sorted order, which is also the first index where `target` appears in `nums`.

For the right boundary, we search for `target+1` using `bisect_left(nums, target + 1)` to get the insertion point `r` for `target+1`. The index immediately before `r` will be the last position where the `target` appears in `nums`.

Finally, if `l == r`, it means that `target` was not found in the array, as the insertion points for `target` and `target+1` are the same. In such a case, we return `[-1, -1]`. If `target` is found, we return `[l, r - 1]`, as `r - 1` is the index of the last occurrence of `target`.

The solution employs a modified binary search (through `bisect_left`) and cleverly manipulates the `target` value to find both the starting and ending positions of the `target` in a sorted array, all while maintaining the required O(log n) runtime complexity.

Solution Approach

The solution provided uses the `bisect` module from Python's standard libraries, which is specifically designed to perform binary search operations. The key functions used are `bisect_left` and a slight variant of it to find the right boundary. The `bisect_left` function finds an insertion point for a specified element in a sorted list, and we use this functionality to find the left and right boundaries of the `target` value. Let's walk through the implementation process by breakdown:

1. **Finding the Left Boundary**: When we search for `target` using `bisect_left(nums, target)`, we get the left boundary. This function returns the index at which `target` could be inserted to maintain the sorted order of the array. Since the array is sorted non-decreasingly, this index is also the first occurrence of `target` in the array if it exists. If `target` is not present, `bisect_left` will return the position where `target` would fit if it were in the list.

2. **Finding the Right Boundary**: The right boundary is a bit trickier. We could implement another binary search to find the last position of `target`, or we could use a simple trick: search for `target + 1` using `bisect_left(nums, target + 1)`. This will give us the index where `target + 1` should be inserted to maintain the sorted order of the array. The index just before this position is the last occurrence of the `target`.

3. **Determining if `target` Was Found**: After finding the left boundary `l` and the potential right boundary `r`, we need to check if `target` was found in the list. If `l == r`, this indicates that `target` was not found because the insertion points for `target` and `target + 1` are the same. In this case, we return `[-1, -1]` as per the problem statement.

4. **Returning the Result**: If `target` was found, `l` must be less than `r`, and `l` will be the first occurrence while `r - 1` will be the last occurrence. We return `[l, r - 1]`.

The reference solution approach provides two templates for binary search in Java, and while the Python solution does not directly use these templates, it embodies the same principle:

- **Template 1** is a standard binary search to find the lower bound of a value.
- **Template 2** finds the upper bound of a value but is inclusive, so you may need to adjust the return value by subtracting 1 to get the actual index of the last occurrence of `target`.

By using these templates or the `bisect` module in Python, we can write effective binary search algorithms that perform the required operations efficiently, adhering to the `O(log n)` runtime complexity constraint.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Suppose we have the sorted array `nums` as follows and we're trying to find the starting and ending positions of the `target` value which is `4`.

```
nums = [1, 2, 4, 4, 4, 5, 6]
target = 4
```

Step 1: **Finding the Left Boundary**

We use `bisect_left(nums, 4)` to find the insertion point for the target value `4`. This function returns the index at which the integer `4` could be inserted to maintain the sorted order of the array. In this example, `bisect_left` would return `2`.

Indeed, the first occurrence of `4` in `nums` is at index `2`.

```
Position: 0  1  2  3  4  5  6
nums:     [1, 2, 4, 4, 4, 5, 6]
             ^     ^
Index:       2 (left boundary)
```

Step 2: **Finding the Right Boundary**

Next, we find where the integer `5` (`target + 1`) would fit into `nums` by using `bisect_left(nums, 4 + 1)`. This returns the index `5`, signifying where we would insert `5`, had it not already been in the array.

The index right before `5` is the last occurrence of `4` in `nums`, which occurs at index `4`.

```
Position: 0  1  2  3  4  5  6
nums:     [1, 2, 4, 4, 4, 5, 6]
                  ^     ^
Index:                4 (right boundary - 1)
```

Step 3: **Determining if `target` Was Found**

Since the left boundary `l` is `2` and the right boundary `r` is `5`, and `l` is not equal to `r`, we conclude that the `target` was found.

Step 4: **Returning the Result**

Finally, since `l` is less than `r`, we return `[l, r - 1]`, which translates to `[2, 4 - 1]`, resulting in `[2, 3]`.

Therefore, our function returns `[2, 3]` as the starting and ending positions of the target value `4` in the sorted array `nums`.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        # Find the leftmost (first) index where `target` should be inserted.
        left_index = bisect_left(nums, target)

        # Find the rightmost index by searching for the position where `target + 1` should be inserted.
        # This will give us one position past the last occurrence of `target`.
        right_index = bisect_left(nums, target + 1)

        # If `left_index` and `right_index` are the same, the target is not present in the list.
        if left_index == right_index:
            return [-1, -1]  # Target not found, return [-1, -1].
        else:
            # Return the starting and ending index of `target`.
            # Since `right_index` gives us one position past the last occurrence,
            # we subtract one to get the actual right boundary.
            return [left_index, right_index - 1]

# Note: List[int] is a type hint specifying a list of integers.
# Remember to include `from typing import List` if you're running this code as is.

```

Java Solution

```java
class Solution {
    // Main method to find the starting and ending position of a given target value.
    public int[] searchRange(int[] nums, int target) {
        // Search for the first occurrence of the target
        int leftIndex = findFirst(nums, target);
        // Search for the first occurrence of the next number after target
        int rightIndex = findFirst(nums, target + 1);

        // If leftIndex equals rightIndex, the target is not in the array
        if (leftIndex == rightIndex) {
            return new int[] {-1, -1}; // target not found
        } else {
            // Subtract 1 from rightIndex to get the ending position of the target
            return new int[] {leftIndex, rightIndex - 1}; // target range found
        }
    }

    // Helper method to search for the first occurrence of a number
    private int findFirst(int[] nums, int x) {
        int left = 0;
        int right = nums.length; // Set right to the length of the array

        // Binary search
        while (left < right) {
            int mid = (left + right) >>> 1; // Find mid while avoiding overflow

            // When the mid element is >= x, we might have found the first occurrence
            // or the target might still be to the left, so we narrow down to the left half
            if (nums[mid] >= x) {
                right = mid;
            } else {
                // Otherwise, the target can only be in the right half
                left = mid + 1;
            }
        }
        return left; // When left and right converge, left (or right) is the first occurrence
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // include this to use std::lower_bound

class Solution {
public:
    // This function finds the start and end indices of a given target value within a sorted array.
    std::vector<int> searchRange(std::vector<int>& nums, int target) {
        // Find the first position where target can be inserted without violating the ordering.
        int leftIndex = std::lower_bound(nums.begin(), nums.end(), target) - nums.begin();

        // Find the first position where the next greater number than target can be inserted.
        // This will give us one position past the target's last occurrence.
        int rightIndex = std::lower_bound(nums.begin(), nums.end(), target + 1) - nums.begin();

        // If leftIndex equals rightIndex, target is not found.
        if (leftIndex == rightIndex) {
            return {-1, -1}; // Target is not present in the vector.
        }

        // Since rightIndex points to one past the last occurrence, we need to subtract 1.
        return {leftIndex, rightIndex - 1}; // Return the starting and ending indices of target.
    }
};

```

Typescript Solution

```typescript
function searchRange(nums: number[], target: number): number[] {
    // Helper function that performs a binary search on the array.
    // It finds the leftmost index at which 'value' should be inserted in order.
    function binarySearch(value: number): number {
        let left = 0;
        let right = nums.length; // Note that 'right' is initialized to 'nums.length', not 'nums.length - 1'.

        // Continues as long as 'left' is less than 'right'.
        while (left < right) {
            // Find the middle index between 'left' and 'right'.
            const mid = Math.floor((left + right) / 2); // Using Math.floor for clarity.

            // If the value at 'mid' is greater than or equal to the search 'value',
            // tighten the right bound of the search. Otherwise, tighten the left bound.
            if (nums[mid] >= value) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        // Return the left boundary which is the insertion point for 'value'.
        return left;
    }

    // Use the binary search helper to find the starting index for 'target'.
    const startIdx = binarySearch(target);
    // Use the binary search helper to find the starting index for the next number,
    // which will be the end index for 'target' in a sorted array.
    const endIdx = binarySearch(target + 1) - 1; // Subtract 1 to find the last index of 'target'.

    // If the start index is the same as end index + 1, 'target' is not in the array.
    // Return [-1, -1] in that case. Otherwise, return the start and end indices.
    return startIdx <= endIdx ? [startIdx, endIdx] : [-1, -1];
}

```

Time and Space Complexity

The provided code utilizes the binary search algorithm by employing `bisect_left()` from Python's bisect module to find the starting and ending position of a given `target` in a sorted array `nums`. The time and space complexity analysis is as follows:

Time Complexity

The `bisect_left()` function is a binary search operation that runs in `O(log n)` time complexity, where `n` is the number of elements in the array `nums`. Since the function is called twice in the code, the total time complexity remains `O(log n)` because constant factors are ignored in the Big O notation.

Therefore, the time complexity of the entire function is: `O(log n)`

Space Complexity

The code does not use any additional space that scales with the size of the input array `nums`, thus the space complexity is constant.

Hence, the space complexity of the function is: `O(1)`