

# 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:

- 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.
- 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`.
- 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.
- 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.

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

```
1 Position: 0 1 2 3 4 5 6
2 nums:    [1, 2, 4, 4, 4, 5, 6]
3           ^  ^  ^
4 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.

```
1 Position: 0 1 2 3 4 5 6
2 nums:    [1, 2, 4, 4, 4, 5, 6]
3           ^  ^  ^
4 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

```
1 from bisect import bisect_left
2
3 class Solution:
4     def searchRange(self, nums: List[int], target: int) -> List[int]:
5         # Find the leftmost (first) index where 'target' should be inserted.
6         left_index = bisect_left(nums, target)
7
8         # Find the rightmost index by searching for the position where 'target' + 1 should be inserted.
9         # This will give us one position past the last occurrence of 'target'.
10        right_index = bisect_left(nums, target + 1)
11
12        # If 'left_index' and 'right_index' are the same, the target is not present in the list.
13        if left_index == right_index:
14            return [-1, -1] # Target not found, return [-1, -1].
15        else:
16            # Return the starting and ending index of 'target'.
17            # Since 'right_index' gives us one position past the last occurrence,
18            # we subtract one to get the actual right boundary.
19            return [left_index, right_index - 1]
20
21 # Note: List[int] is a type hint specifying a list of integers.
22 # Remember to include 'from typing import List' if you're running this code as is.
23
```

## Java Solution

```
1 class Solution {
2     // Main method to find the starting and ending position of a given target value.
3     public int[] searchRange(int[] nums, int target) {
4         // Search for the first occurrence of the target
5         int leftIndex = findFirst(nums, target);
6         // Search for the first occurrence of the next number after target
7         int rightIndex = findFirst(nums, target + 1);
8
9         // If leftIndex equals rightIndex, the target is not in the array
10        if (leftIndex == rightIndex) {
11            return new int[] {-1, -1}; // target not found
12        } else {
13            // Subtract 1 from rightIndex to get the ending position of the target
14            return new int[] {leftIndex, rightIndex - 1}; // target range found
15        }
16    }
17
18    // Helper method to search for the first occurrence of a number
19    private int findFirst(int[] nums, int x) {
20        int left = 0;
21        int right = nums.length; // Set right to the length of the array
22
23        // Binary search
24        while (left < right) {
25            int mid = (left + right) >> 1; // Find mid while avoiding overflow
26
27            // When the mid element is >= x, we might have found the first occurrence
28            // or the target might still be to the left, so we narrow down to the left half
29            if (nums[mid] >= x) {
30                right = mid;
31            } else {
32                // Otherwise, the target can only be in the right half
33                left = mid + 1;
34            }
35        }
36        return left; // When left and right converge, left (or right) is the first occurrence
37    }
38 }
39
```

## C++ Solution

```
1 #include <vector>
2 #include <algorithm> // include this to use std::lower_bound
3
4 class Solution {
5 public:
6     // This function finds the start and end indices of a given target value within a sorted array.
7     std::vector<int> searchRange(std::vector<int>& nums, int target) {
8         // Find the first position where target can be inserted without violating the ordering.
9         int leftIndex = std::lower_bound(nums.begin(), nums.end(), target) - nums.begin();
10
11        // Find the first position where the next greater number than target can be inserted.
12        // This will give us one position past the target's last occurrence.
13        int rightIndex = std::lower_bound(nums.begin(), nums.end(), target + 1) - nums.begin();
14
15        // If leftIndex equals rightIndex, target is not found.
16        if (leftIndex == rightIndex) {
17            return {-1, -1}; // Target is not present in the vector.
18        }
19
20        // Since rightIndex points to one past the last occurrence, we need to subtract 1.
21        return {leftIndex, rightIndex - 1}; // Return the starting and ending indices of target.
22    }
23 };
24
```

## Typescript Solution

```
1 function searchRange(nums: number[], target: number): number[] {
2     // Helper function that performs a binary search on the array.
3     // It finds the leftmost index at which 'value' should be inserted in order.
4     function binarySearch(value: number): number {
5         let left = 0;
6         let right = nums.length; // Note that 'right' is initialized to 'nums.length', not 'nums.length - 1'.
7
8         // Continues as long as 'left' is less than 'right'.
9         while (left < right) {
10            // Find the middle index between 'left' and 'right'.
11            const mid = Math.floor((left + right) / 2); // Using Math.floor for clarity.
12
13            // If the value at 'mid' is greater than or equal to the search 'value',
14            // tighten the right bound of the search. Otherwise, tighten the left bound.
15            if (nums[mid] >= value) {
16                right = mid;
17            } else {
18                left = mid + 1;
19            }
20        }
21        // Return the left boundary which is the insertion point for 'value'.
22        return left;
23    }
24
25    // Use the binary search helper to find the starting index for 'target'.
26    const startIdx = binarySearch(target);
27    // Use the binary search helper to find the starting index for the next number,
28    // which will be the end index for 'target' in a sorted array.
29    const endIdx = binarySearch(target + 1) - 1; // Subtract 1 to find the last index of 'target'.
30
31    // If the start index is the same as end index + 1, 'target' is not in the array.
32    // Return [-1, -1] in that case. Otherwise, return the start and end indices.
33    return startIdx <= endIdx ? [startIdx, endIdx] : [-1, -1];
34 }
35
```

## 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:  $O(1)$