



## LeetCode 34 — Find First and Last Position of Element in Sorted Array

### 1. Problem Title & Link

- **Title:** LeetCode 34 — Find First and Last Position of Element in Sorted Array
- **Link:** <https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

### 2. Problem Statement (Short Summary)

Given a **sorted array** `nums` and a `target`,  
return the **start and end index** of `target` in the array.

If `target` does NOT exist → return `[-1, -1]`.

You MUST run in  **$O(\log n)$**  time → binary search.

### 3. Examples (Input → Output)

#### Example 1

Input: `nums = [5,7,7,8,8,10]`, `target = 8`

Output: `[3,4]`

#### Example 2

Input: `nums = [5,7,7,8,8,10]`, `target = 6`

Output: `[-1,-1]`

#### Example 3

Input: `nums = []`, `target = 0`

Output: `[-1,-1]`

### 4. Constraints

- $0 \leq \text{nums.length} \leq 10^5$
- Array is **sorted** in ascending order
- Must use **binary search**

### 5. Core Concept (Pattern / Topic)

#### Binary Search — Two Variants

1. Find **first occurrence**
2. Find **last occurrence**

Use **modified binary search**:

- First occurrence → move right pointer inward
- Last occurrence → move left pointer inward

### 6. Thought Process (Step-by-Step Explanation)



### Why can't we use a normal binary search?

Normal binary search finds **one** occurrence of target, but here we need **both ends** of the range.

#### Strategy:

1. Write a function firstPos():
  - Standard binary search
  - When `nums[mid] == target` → shrink right to `mid-1`
  - Store and update answer
2. Write a function lastPos():
  - Standard binary search
  - When `nums[mid] == target` → move left to `mid+1`
  - Store and update answer

Both functions run in  **$O(\log n)$** .

### 7. Visual / Intuition Diagram

Array:

[5, 7, 7, 8, 8, 10]

    ↑  ↑  
    first last

Binary search finds:

- First 8 at index 3
- Last 8 at index 4

### 8. Pseudocode

```
function firstPos():
    ans = -1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] >= target:
            right = mid - 1
        else:
            left = mid + 1
        if nums[mid] == target:
            ans = mid
    return ans

function lastPos():
    ans = -1
    while left <= right:
        mid = (left + right) // 2
```



```
if nums[mid] <= target:
    left = mid + 1
else:
    right = mid - 1
if nums[mid] == target:
    ans = mid
return ans
```

## 9. Code Implementation

### ✓ Python

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        def firstPos():
            left, right = 0, len(nums) - 1
            ans = -1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] >= target:
                    right = mid - 1
                else:
                    left = mid + 1
                if nums[mid] == target:
                    ans = mid
            return ans

        def lastPos():
            left, right = 0, len(nums) - 1
            ans = -1
            while left <= right:
                mid = (left + right) // 2
                if nums[mid] <= target:
                    left = mid + 1
                else:
                    right = mid - 1
                if nums[mid] == target:
                    ans = mid
            return ans

        return [firstPos(), lastPos()]
```

### ✓ Java



```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        return new int[]{firstPos(nums, target), lastPos(nums, target)};
    }

    private int firstPos(int[] nums, int target) {
        int left = 0, right = nums.length - 1, ans = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] >= target) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }

            if (nums[mid] == target) ans = mid;
        }
        return ans;
    }

    private int lastPos(int[] nums, int target) {
        int left = 0, right = nums.length - 1, ans = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] <= target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }

            if (nums[mid] == target) ans = mid;
        }
        return ans;
    }
}
```



## 10. Time & Space Complexity

Metric	Complexity
Time	$O(\log n) + O(\log n) = O(\log n)$
Space	$O(1)$

## 11. Common Mistakes / Edge Cases

- ✗ Using linear scan → too slow
- ✗ Not updating answer when  $\text{mid} == \text{target}$
- ✗ Using strict  $<$  instead of correct  $<=$
- ✗ Failing in arrays like  $[2,2]$ ,  $[2]$ , or empty array

Edge cases:

- $\text{nums} = []$
- target smaller than min or larger than max
- Only one occurrence in array
- All elements are the target

## 12. Detailed Dry Run (Step-by-Step Table)

Input:

$\text{nums} = [5, 7, 7, 8, 8, 10]$

target = 8

**Searching FIRST position:**

left	mid	right	nums[mid]	Action
0	2	5	7	left=3
3	4	5	8	ans=4, right=3
3	3	3	8	ans=3, right=2

First occurrence = 3

**Searching LAST position:**

left	mid	right	nums[mid]	Action
0	2	5	7	left=3
3	4	5	8	ans=4, left=5
5	5	5	10	right=4



Last occurrence = 4

Final Output:

[3,4]

### 13. Common Use Cases

- Finding first/last log entry for timestamp
- Range searching
- Duplicate block detection
- Index boundaries in sorted arrays

### 14. Common Traps

- Wrong binary search boundary conditions
- Not storing mid before moving left/right
- Forgetting multiple occurrences
- Using one binary search instead of two (possible but harder)

### 15. Builds To (Related Problems)

- **LC 35** — Search Insert Position
- **LC 852** — Peak Index in Mountain Array
- **LC 33** — Search in Rotated Sorted Array
- **LC 153/154** — Min in Rotated Sorted Array

### 16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Linear Scan	$O(n)$	$O(1)$	Too slow
2× Binary Search	$O(\log n)$	$O(1)$	Best ✓
Single Binary Search Variant	$O(\log n)$	$O(1)$	More complex

### 17. Why This Solution Works (Short Intuition)

Using two modified binary searches ensures we find the **lowest** and **highest** index where target appears, while maintaining  $O(\log n)$  time.

### 18. Variations / Follow-Up Questions

- Find FIRST number  $\geq$  target
- Find LAST number  $\leq$  target



- With duplicates allowed in rotated array (harder)
- Find count of target in sorted array:  
→  $\text{lastPos} - \text{firstPos} + 1$