

■ Search Insert Position

Find where a number belongs — even if it's not there yet

● Binary Search

■ EASY

■ What You Will Learn

- The 'lower bound' idea — where would this element sit?
- Why the left pointer magically lands on the insert position
- How this is binary search with a twist ending
- The pattern for 'first position $\geq$ target' problems

■ Intuition First

Imagine a sorted queue of numbered tickets: [10, 20, 30, 50, 60]. Someone with ticket 35 arrives. Where do they stand? Between 30 and 50 — at index 3. That's search insert position. You binary search for 35, and when you don't find it, the left pointer tells you exactly where it belongs.

■ Key Insight

When binary search ends without finding the target, the 'left' pointer always points to the first index where you COULD insert the target to keep the array sorted. This is the magic of lower bound.

■ Visual Explanation

```
Array: [1, 3, 5, 6]   Target = 5
L=0, R=3 → mid=1 → arr[1]=3 < 5 → L=2
L=2, R=3 → mid=2 → arr[2]=5 == 5 → return 2 ✓
```

```
Array: [1, 3, 5, 6]   Target = 2
L=0, R=3 → mid=1 → arr[1]=3 > 2 → R=0
L=0, R=0 → mid=0 → arr[0]=1 < 2 → L=1
L=1, R=0 → loop ends → return L = 1 ✓
(insert 2 between index 0 and 1)
```

```
Array: [1, 3, 5, 6]   Target = 7
(goes past all elements → L becomes 4)
return L = 4 (insert at end) ✓
```

■■ How It Works

1. Standard binary search — search for target
2. If found → return mid

3. If not found → return **left** (this is the insert position!)

Why does left work? When the loop exits ($\text{left} > \text{right}$), left has 'crossed' right and landed exactly where the target would need to go. It's been narrowing from above, so it points to the first element greater than target.

■ Java Code

```
1  class Solution {
2  public int searchInsert(int[] nums, int target) {
3      int left = 0, right = nums.length - 1;
4
5      while (left <= right) {
6          int mid = left + (right - left) / 2;
7
8          if (nums[mid] == target) {
9              return mid; // Exact match found
10         } else if (nums[mid] < target) {
11             left = mid + 1;
12         } else {
13             right = mid - 1;
14         }
15     }
16     // left is now the insert position
17     return left;
18 }
19 }
```

■■ Complexity

Complexity	Value	What it means
Time	$O(\log n)$	Same as binary search — halving each step
Space	$O(1)$	No extra memory used

■■ Common Mistakes

■■ Mistake

Returning right instead of left when not found. Always return left — it's the lower bound pointer.

■ Pro Tips

■ Pro Tip

This exact pattern (return left after loop) is the foundation for 'lower bound' in C++ and `Arrays.binarySearch()` in Java. Once you master this, dozens of problems become easy.

■ Practice Problems

#	Problem	Difficulty
1	Search Insert Position (LeetCode 35)	Easy
2	Find First Bad Version (LeetCode 278)	Easy
3	Leftmost Column with Atleast a One (LeetCode 1428)	Medium
4	Count of Smaller Numbers After Self (LeetCode 315)	Hard

■ Quick Revision

✓	Same as standard binary search — only change: return left at end
✓	Left pointer = lower bound = first valid insert position
✓	Works even if target is smaller than all elements or larger than all
✓	Time $O(\log n)$, Space $O(1)$