

Target Position Finder in Sorted List

Problem Statement:

Given a sorted list of unique integers in ascending order, find the position of a target number within the list. If the target exists, return its position. Otherwise, return the position where it would need to be inserted to maintain the order.

Constraints:

- Time Complexity: $O(\log n)$
- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- All values are unique and sorted in ascending order
- $-10^4 \leq \text{target} \leq 10^4$

Solution Approach:

Since the list is sorted, we use Binary Search with $O(\log n)$ complexity:

1. Initialize two pointers (low and high).
2. Divide the search range in half.
3. If the target is found, return its index.
4. If not found, return the position where it should be inserted.

Python Implementation:

```
def search_insert_position(nums, target):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return low # The correct insertion position

# Example Test Cases
nums1, target1 = [1, 3, 5, 6], 5
nums2, target2 = [1, 3, 5, 6], 2
nums3, target3 = [1, 3, 5, 6], 7

print(search_insert_position(nums1, target1))
print(search_insert_position(nums2, target2))
print(search_insert_position(nums3, target3))
```

Complexity Analysis:

Time Complexity: $O(\log n)$ due to binary search.

Space Complexity: $O(1)$, as we use only a few integer variables.

Conclusion:

This approach efficiently finds the target's position or its correct insertion index in $O(\log n)$ time using binary search. It ensures optimal performance for large inputs while maintaining clarity and correctness.