Search Insert Position

Problem Statement:

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Examples:

Example 1:

Input: nums = [1,3,5,6], target = 5
Output: 2

Example 2:

Input: nums = [1,3,5,6], target = 2
Output: 1

Example 3:

Input: nums = [1,3,5,6], target = 7
Output: 4

Constraints:

1 <= nums.length <= 10^4
-10^4 <= nums[i] <= 10^4
nums contains distinct values sorted in ascending order.
-10^4 <= target <= 10^4

Explanation:

The task is to find the index of the target value in a sorted array. If the target is not present, return the index where it should be inserted to maintain the sorted order. To achieve O(log n) runtime complexity, a binary search algorithm is used.

Explained More Simply:

We have a sorted list of unique numbers and a target number. We need to find where the target number is or where it should go in the list to keep the list sorted. We will use a fast search method called binary search to do this.

Code to Solve the Problem:

```
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1  # Initialize pointers for binary
search

        while left <= right:  # Continue searching while the pointers have not
crossed
            mid = (left + right) // 2  # Find the midpoint of the current
segment
            if nums[mid] == target:  # If the midpoint is the target, return
its index
                return mid
            elif nums[mid] < target:  # If the target is greater, move the
left pointer
                left = mid + 1
            else:  # If the target is smaller, move the right pointer
                right = mid - 1

        return left  # If not found, left is the insert position
```

Explanation of Each Line of Code:


```
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1  # Initialize pointers for binary
search

        while left <= right:  # Continue searching while the pointers have not
crossed
            mid = (left + right) // 2  # Find the midpoint of the current
segment
            if nums[mid] == target:  # If the midpoint is the target, return
its index
                return mid
            elif nums[mid] < target:  # If the target is greater, move the
left pointer
                left = mid + 1
            else:  # If the target is smaller, move the right pointer
                right = mid - 1

        return left  # If not found, left is the insert position
```


class Solution:: Defines a class Solution.
def searchInsert(self, nums: List[int], target: int) -> int:: Defines a method searchInsert within the Solution class that takes a list of integers nums and an integer target and returns an integer.
left, right = 0, len(nums) - 1: Initializes pointers for binary search. left starts at the beginning of the list, and right starts at the end.
while left <= right:: Continues the search while the left pointer is less than or equal to the right pointer.
mid = (left + right) // 2: Calculates the midpoint of the current segment.
if nums[mid] == target:: Checks if the midpoint value is equal to the target value. If so, returns the midpoint index.
elif nums[mid] < target:: If the target value is greater than the midpoint value, moves the left pointer to mid + 1.

else:: If the target value is less than the midpoint value, moves the right pointer to mid - 1.
return left:: If the target value is not found, returns the left pointer as the insert position. This is the position where the target value would be inserted to maintain the sorted order.