35. Search Insert Position

Easy Array Binary Search

Problem Description

The problem asks us to find the position in a sorted array where a target value either is found or would be inserted to maintain the array's order. You are given a sorted array with distinct integers and a target value. The task is to check if the target is present in the array. If the target is found, you need to return its index. If the target is not found, you need to return the index where it would be inserted to keep the array sorted. The key requirement is that the solution must operate with a runtime complexity of `O(log n)`, suggesting that we must use an efficient algorithm like binary search that repeatedly divides the search interval in half.

Intuition

To achieve the `O(log n)` runtime complexity, we utilize the binary search algorithm. The intuition behind this approach is based on the sorted nature of the array. At each step, binary search compares the target value to the middle element of the array. If the target is equal to the middle element, we have found the target, and its index is returned. If the target is less than the middle element, it must be in the left sub-array, and we continue the search there; if it's more, we continue in the right sub-array.

Our search domain decreases by half with each comparison, which is why binary search is so efficient.

For the case where the element is not found, the position where it would be inserted is the point where our search space narrows down to an empty range, and `left` will have moved to the index where the target value can be inserted to maintain the sorted array property. This is because, throughout the search, `left` is maintained as the boundary where all elements to its left are less than the target, making it the correct insertion position for the target value.

Solution Approach

The solution implements a binary search algorithm to find the target's index or the insertion point in the array to maintain a `O(log n)` runtime complexity. The main steps of the algorithm used in the solution are as follows:

1. We initialize two pointers, `left` and `right`, representing the search space's boundaries within the array. `left` is set to `0`, and `right` is set to the length of the array.
2. We enter a loop that continues until `left < right`, which means there is still a range to be searched. Inside this loop, a middle index `mid` is calculated, which is the average of `left` and `right` indexes, for the current search interval. Here, we use a binary shift operation `>> 1` which is equivalent to dividing by 2.
3. We compare the middle element `nums[mid]` with the target. If the middle element is greater than or equal to the target (`nums[mid] >= target`), we know the target, if it exists, is to the left of `mid` or at `mid`. So, we move the `right` pointer to `mid`, narrowing the search space to the left half.
4. If the middle element is less than the target (`nums[mid] < target`), the target, if it exists, is to the right of `mid`. Therefore, we move the `left` pointer to `mid + 1`, narrowing the search space to the right half.
5. When the loop exits, the `left` pointer indicates the position where the target is found or should be inserted. The condition of the loop ensures that `left` and `right` pointers converge on the insertion point if the target is not found.

This iterative halving of the search space is what allows the binary search algorithm to run in `O(log n)` time.

By using binary search, we efficiently locate the target value or determine the correct insertion point with minimal comparisons, making the algorithm very effective for large sorted datasets.

Example Walkthrough

Let's consider the sorted array `nums = [1, 3, 5, 6]` and the target value `target = 5`. We want to find the index of the target value or the index where it should be inserted to maintain the sorted order.

1. Initialize two pointers, `left = 0` and `right = length_of_array` which is `4`.
2. Loop while `left < right`:
   - Calculate `mid` as `(left + right) >> 1` which is `(0 + 4) >> 1`, giving us `mid = 2`.
   - Check if `nums[mid]` is greater than, less than, or equal to the `target`:
     - `nums[2]` is `5`, which is equal to the `target` of `5`.
     - Since the middle element is equal to the target, we have found the target at index `2`.
3. The loop ends as we found the target.
4. Return the `mid` index, which is `2`.

The `5` is located at index `2` in the array, so the function returns `2` as the index where `5` is found.

Now let's consider a different target value `target = 2`.

1. Initialize two pointers, `left = 0` and `right = 4`.
2. Continue the loop while `left < right`:
   - Calculate `mid` as `(left + right) >> 1` which is `(0 + 4) >> 1`, giving us `mid = 2`.
   - Check if `nums[mid]` is greater than, less than, or equal to `target`:
     - `nums[2]` is `5`, which is greater than the `target` of `2`.
     - Since `nums[mid]` is greater than the target, update `right` to `mid` to continue searching in the left sub-array.
3. Now `left = 0` and `right = 2`; calculate new `mid` as `(0 + 2) >> 1`, which is `1`.
4. Check `nums[mid]` against `target`:
   - `nums[1]` is `3`, which is greater than the `target`.
   - Update `right` to `mid` to continue searching in the left sub-array.
5. Now `left = 0` and `right = 1`; loop exits as `left` is now equal to `right`.
6. Since the target `2` wasn't found, we return `left` which is the position where `2` would be inserted to maintain the sorted order.

The target `2` would be inserted at index `1` to maintain the sorted order, so the function returns `1`.

Solution Implementation

Python

```python
from typing import List  # Import the List type from typing module for type hints.

class Solution:
    def search_insert(self, nums: List[int], target: int) -> int:
        # Initialize two pointers for the search boundaries.
        left, right = 0, len(nums)

        # Use binary search to find the insert position.
        while left < right:
            # Calculate the middle index to compare with the target.
            mid = (left + right) // 2  # Using // for integer division in Python 3.
            # If the middle element is greater than or equal to the target,
            # search the left half including the current middle.
            if nums[mid] >= target:
                right = mid
            # If the middle element is less than the target,
            # search the right half excluding the current middle.
            else:
                left = mid + 1

        # The left pointer will end up at the insert position, so return it.
        return left

# Example usage:
# sol = Solution()
# print(sol.search_insert([1, 3, 5, 6], 5))  # Output: 2
# print(sol.search_insert([1, 3, 5, 6], 2))  # Output: 1
# print(sol.search_insert([1, 3, 5, 6], 7))  # Output: 4
# print(sol.search_insert([1, 3, 5, 6], 0))  # Output: 0
```

Java

```java
class Solution {
    public int searchInsert(int[] nums, int target) {
        // Initialize the left and right pointers.
        // Right is initially set to the length of the array rather than the last index.
        int left = 0, right = nums.length;

        // Binary search to find the target or the insertion point.
        while (left < right) {
            // Calculate the middle index.
            // Use unsigned right shift to avoid potential overflow for large left/right.
            int mid = (left + right) >>> 1;

            // If the current element at the middle index is greater than or equal to the target,
            // narrow the search to the left half (inclusive of mid).
            if (nums[mid] >= target) {
                right = mid;
            } else {
                // Otherwise, narrow the search to the right half (exclusive of mid).
                left = mid + 1;
            }
        }
        // Since the algorithm is looking for the insertion point, left will be the correct index
        // whether the target is found or not.
        return left;
    }
}
```

C++

```cpp
#include <vector> // Include necessary header for using vectors.

// Solution class as provided in the original problem statement.
class Solution {
public:
    // searchInsert function to determine the index at which the target should be inserted or is found in a sorted array.
    int searchInsert(std::vector<int>& nums, int target) {
        // Initialize the search bounds.
        int left = 0; // 'left' is the start of the search range.
        int right = nums.size(); // 'right' is set to one past the end of the search range.

        // Binary search loop.
        while (left < right) { // While the search range is not empty, continue searching.
            int mid = (left + right) / 2; // Calculate the midpoint to prevent potential overflow.

            // Compare the middle element to the target.
            if (nums[mid] >= target) {
                // If the middle element is greater than or equal to the target,
                // narrow the search to the left half, excluding the current mid.
                right = mid;
            } else {
                // Otherwise, narrow the search to the right half, excluding the current mid.
                left = mid + 1;
            }
        }

        // 'left' will be the position where the target should be inserted or where it is found.
        return left;
    }
};
```

TypeScript

```typescript
// Define the function searchInsert with input parameters 'numbers', an array of numbers,
// and 'target', the number to search for, which returns the index of the target number or the insert position.
function searchInsert(numbers: number[], target: number): number {

    // Define 'start' and 'end' variables representing the search range within the array.
    let start: number = 0;
    let end: number = numbers.length;

    // Continue searching while 'start' is less than 'end'.
    while (start < end) {

        // Calculate the middle index of the current search range.
        const mid: number = Math.floor((start + end) / 2);

        // If the element at 'mid' is greater than or equal to the 'target',
        // narrow the search range to the left half by setting 'end' to 'mid'.
        if (numbers[mid] >= target) {
            end = mid;
        }

        // Otherwise, narrow the search range to the right half by setting 'start' to 'mid + 1'.
        else {
            start = mid + 1;
        }
    }

    // Return the 'start' index, which is the insert position of the 'target'.
    return start;
}
```

```python
from typing import List  # Import the List type from typing module for type hints.

class Solution:
    def search_insert(self, nums: List[int], target: int) -> int:
        # Initialize two pointers for the search boundaries.
        left, right = 0, len(nums)

        # Use binary search to find the insert position.
        while left < right:
            # Calculate the middle index to compare with the target.
            mid = (left + right) // 2  # Using // for integer division in Python 3.
            # If the middle element is greater than or equal to the target,
            # search the left half including the current middle.
            if nums[mid] >= target:
                right = mid
            # If the middle element is less than the target,
            # search the right half excluding the current middle.
            else:
                left = mid + 1

        # The left pointer will end up at the insert position, so return it.
        return left

# Example usage:
# sol = Solution()
# print(sol.search_insert([1, 3, 5, 6], 5))  # Output: 2
# print(sol.search_insert([1, 3, 5, 6], 2))  # Output: 1
# print(sol.search_insert([1, 3, 5, 6], 7))  # Output: 4
# print(sol.search_insert([1, 3, 5, 6], 0))  # Output: 0
```

Time and Space Complexity

Time Complexity

The provided code implements a binary search algorithm. In every iteration, it halves the segment of the `nums` list it is considering by adjusting either the `left` or `right` pointers. The time complexity of binary search is `O(log n)`, where `n` is the number of elements in the `nums` list, due to the list's reduction by half in each iteration of the while loop.

Space Complexity

The space complexity of the code is `O(1)`. This is because it uses a constant amount of space for the variables `left`, `right`, and `mid` regardless of the input size. No additional memory is allocated that is dependent on the size of the input list.