Search Insert Position - Tutorial

Search Insert Position

Problem Statement: You are given a sorted array **arr** of distinct values and a target value **x**. You need to search for the index of the target value in the array.

If the value is present in the array, then return its index. Otherwise, determine the index where it would be inserted in the array while maintaining the sorted order.

Pre-requisite: Lower Bound & Binary Search

```
Example 1:
Input Format: arr[] = {1,2,4,7}, x = 6
Result: 3
Explanation: 6 is not present in the array. So, if we will insert 6 in the 3rd
index(0-based indexing), the array will still be sorted. {1,2,4,6,7}.

Example 2:
Input Format: arr[] = {1,2,4,7}, x = 2
Result: 1
Explanation: 2 is present in the array and so we will return its index i.e. 1.
```

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem Link.

Solution:

We will solve this problem using the lower-bound algorithm which is basically a modified version of the classic Binary Search algorithm.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

On deep introspection of the given problem, we can easily understand that we have to find the correct position or the existing position of the target number in the given array.

Now, if the element is not present, we have to find the nearest greater number of the target number. So, basically, we are trying to find an element arr[ind] >= x and hence the lower bound of the target number i.e. x.

The lower bound algorithm returns the first occurrence of the target number if the number is present and otherwise, it returns the nearest greater element of the target number.

Approach:

The steps are as follows:

We will declare the 2 pointers and an ‘ans’ variable initialized to n i.e. the size of the array(*as If we don’t find any index, we will return n*).

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers like this: low will point to the first index and high will point to the last index.
2. **Calculate the ‘mid’:** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 (‘//’ refers to integer division)
3. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
    1. **Case 1 - If arr[mid] >= x:** This condition means that the index mid may be an answer. So, we will update the ‘ans’ variable with mid and search in the left half if there is any smaller index that satisfies the same condition. Here, we are eliminating the right half.
    2. **Case 2 - If arr[mid] < x:** In this case, mid cannot be our answer and we need to find some bigger element. So, we will eliminate the left half and search in the right half for the answer.

The above process will continue until the pointer low crosses high.

Dry run: *Please refer video for it.*

Code:

```
def searchInsert(arr: [int], x: int) -> int:
    n = len(arr)  # size of the array
    low = 0
    high = n - 1
    ans = n

    while low <= high:
        mid = (low + high) // 2
        # maybe an answer
        if arr[mid] >= x:
            ans = mid
            # look for smaller index on the left
            high = mid - 1
        else:
            low = mid + 1  # look on the right

    return ans

if __name__ == "__main__":
    arr = [1, 2, 4, 7]
    x = 6
    ind = searchInsert(arr, x)
    print("The index is:", ind)
```

Output: The index is: 3

Time Complexity: O(logN), where N = size of the given array.
Reason: We are basically using the Binary Search algorithm.

Space Complexity: O(1) as we are using no extra space.