Implement Upper Bound - Tutorial

Problem Statement: Given a sorted array of **N integers** and an integer x, write a program to find the upper bound of x.

Pre-requisite: Binary Search algorithm

▼ **Examples**

```
Example 1:
Input Format: N = 4, arr[] = {1,2,2,3}, x = 2
Result: 3
Explanation: Index 3 is the smallest index such that arr[3] > x.

Example 2:
Input Format: N = 6, arr[] = {3,5,8,9,15,19}, x = 9
Result: 4
Explanation: Index 4 is the smallest index such that arr[4] > x.
```

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

Solution:

In the preceding article, we comprehensively explored the implementation of the Binary Search algorithm and the lower bound 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.

In this article, we will learn how to implement the upper bound algorithm using a slight modification of the Binary Search algorithm.

What is Upper Bound?

The upper bound algorithm finds the first or the smallest index in a sorted array where the value at that index is greater than the given key i.e. x.

The upper bound is the smallest index, ind, where arr[ind] > x.

But ***if any such index is not found***, the upper bound algorithm returns n i.e. size of the given array. *The main difference between the lower and upper bound is in the condition.* ***For the lower bound the condition was arr[ind] >= x*** *and here,* ***in the case of the upper bound, it is arr[ind] > x.***

▼ Brute Force Approach

▼ Algorithm / Intuition ›

Naive approach (Using linear search):

Let's understand how we can find the answer using the linear search algorithm. With the knowledge that the array is sorted, our approach involves traversing the array starting from the beginning. During this traversal, each element will be compared with the target value, x. The index, i, where the condition arr[i] > x is first satisfied, will be the answer.

▼ Code ›

```
def upperBound(arr: [int], x: int, n: int) -> int:
    for i in range(n):
        if arr[i] > x:
            # upper bound found
            return i
    return n

if __name__ == "__main__":
    arr = [3, 5, 8, 9, 15, 19]
    n = 6
    x = 9
    ind = upperBound(arr, x, n)
    print("The upper bound is the index:", ind)
```

Output: The upper bound is the index: 4

▼ Complexity Analysis ›
Time Complexity: O(N), where N = size of the given array.
Reason: In the worst case, we have to travel the whole array. This is basically the time complexity of the linear search algorithm.

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

▼ Optimal Approach
▼ Algorithm / Intuition ›

Optimal Approach (Using Binary Search):

As the array is sorted, we will apply the Binary Search algorithm to find the index. 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 upperBound(arr: [int], x: int, n: int) -> int:
    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 = [3, 5, 8, 9, 15, 19]
    n = 6
    x = 9
    ind = upperBound(arr, x, n)
    print("The upper bound is the index:", ind)
```

Output: The upper bound is the index: 4

▼ Complexity Analysis ›

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.