Floor and Ceil in Sorted Array

Floor and Ceil in Sorted Array

Problem Statement: You're given an sorted array arr of n integers and an integer x. Find the floor and ceiling of x in arr[0..n-1].
The floor of x is the largest element in the array which is smaller than or equal to x.
The ceiling of x is the smallest element in the array greater than or equal to x.

Pre-requisite: Lower Bound & Binary Search

```
Example 1:
Input Format: n = 6, arr[] ={3, 4, 4, 7, 8, 10}, x= 5
Result: 4 7
Explanation: The floor of 5 in the array is 4, and the ceiling of 5 in the array
is 7.

Example 2:
Input Format: n = 6, arr[] ={3, 4, 4, 7, 8, 10}, x= 8
Result: 8 8
Explanation: The floor of 8 in the array is 8, and the ceiling of 8 in the array
is also 8.
```

Solution

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

Problem Link.

Solution:

We are going to solve this problem using the concepts of Lower Bound and Binary Search.

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.

What is the floor of x?

The floor of x is the largest element in the array which is smaller than or equal to x(i.e. ***largest element in the array <= x***).

What is the ceiling of x?

The ceiling of x is the smallest element in the array greater than or equal to x(i.e. ***smallest element in the array >= x***).

From the definitions, we can easily understand that the ceiling of x is basically the lower bound of x. The lower bound algorithm returns the index of x if x is present in the array and otherwise, it returns the index of the smallest element in the array greater than x.

The implementation of Ceil will be the same as the lower bound algorithm.

But we have no such algorithm prepared for the floor. So, we will implement the floor algorithm based on the Binary Search algorithm. The only difference in the algorithm compared to the lower bound algorithm will be the conditions. In this case,

- **If arr[mid] <= x:** arr[mid] is a possible answer. So, we will store it and will try to find a larger number that satisfies the same condition. That is why we will remove the left half and try to find the number in the right half.
- **If arr[mid] > x:** The arr[mid] is definitely not the answer and we need a smaller number. So, we will reduce the search space to the left half by removing the right half.

The rest of the part of the algorithm will be exactly the same.

Note:*If the algorithm returns invalid indices which means there is no floor or ceiling of x, we will return -1.*

Note: *Remember, here we are told to find the numbers not the indices. So, we will store the numbers instead of indices in the ans variable.*

Algorithm / Intuition:

Ceil:

We will declare the 2 pointers and an 'ans' variable initialized to -1(*If we don't find any index, we will return -1*).

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 arr[mid] may be an answer. So, we will update the 'ans' variable with arr[mid] and search in the left half if there is any smaller number that satisfies the same condition. Here, we are eliminating the right half.
    2. **Case 2 - If arr[mid] < x:** In this case, arr[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.

Floor:

We will declare the 2 pointers and an ‘ans’ variable initialized to -1 (*If we don’t find any index, we will return -1*).

4. **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.
5. **Calculate the ‘mid’:** Now, we will calculate the value of mid using the following formula:
   mid = (low+high) // 2 (‘//’ refers to integer division)
6. **Compare arr[mid] with x:** With comparing arr[mid] to x, we can observe 2 different cases:
   1. **Case 1 - If arr[mid] <= x:** The index arr[mid] is a possible answer. So, we will store it and will try to find a larger number that satisfies the same condition. That is why we will remove the left half and try to find the number in the right half.
   2. **Case 2 - If arr[mid] > x:** arr[mid] is definitely not the answer and we need a smaller number. So, we will reduce the search space to the left half by removing the right half.

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

Dry run: *Please refer video for it.*

Code:

```
def findFloor(arr, n, x):
    low = 0
    high = n - 1
    ans = -1

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

    return ans


def findCeil(arr, n, x):
    low = 0
    high = n - 1
    ans = -1

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

    return ans


def getFloorAndCeil(arr, n, x):
    f = findFloor(arr, n, x)
    c = findCeil(arr, n, x)
    return (f, c)


arr = [3, 4, 4, 7, 8, 10]
n = 6
x = 5
ans = getFloorAndCeil(arr, n, x)
print("The floor and ceil are:", ans[0], ans[1])
```

Output: The floor and ceil are: 4 7

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.

Follow-up Question:

Why are we not comparing the numbers to get the largest or smallest while calculating the floor or ceiling?

- In the floor algorithm, if we get a possible answer, we are reducing the search space to the right half. And the right half certainly contains larger numbers than the current answer. So, every time we are getting larger numbers as answers.
- Similarly, in the ceil algorithm, if we get a possible answer, we are reducing the search space to the left half. And the left half certainly contains smaller numbers than the current answer. So, every time we are getting smaller numbers as answers.
- Basically, in each case, we are moving in a direction such that we get the largest or the smalleset as per our need. That is why we are not checking for the largest or the smallest explicitly.

Note: *Here, we have utilized a variable called 'ans' to store the answers. However, in various instances, you may encounter a different approach where no separate variable is used. Instead, either the 'low' or 'high' pointer is employed as the answer itself. As we progress further in this topic, we will delve into this technique and explore its implementation in detail.*