DAA432C Group-29 Assignment-04

B. Tech IT 4^{th} Semester Sec-A

Indian Institute of Information Technology, Allahabad

IIT2019089	IIT2019088	IIT2019087
Redrouthu Pranav Sai	Ritik Kumar	Gautam Kumar
iit2019089@iiita.ac.in	iit2019088@iiita.ac.in	iit2019087@iiita.ac.in

***Abstract*— This electronic document discusses about the design and analysis of algorithm that finds the floor of x in a sorted array where floor of x is the largest element in an array smaller than or equal to x.**

***Keywords*— Binary Search, Profiling, linear search, floor of an element**

I. INTRODUCTION

This paper discusses about the algorithm designed to find the floor of x in a sorted array where floor of x is the largest element in an array smaller than or equal to x. We have also analysed about the time and space complexities of an algorithm. The time complexity for varying array size for different algorithms can be seen in the graph followed.

By the end of the paper, we will be able to understand the components of algorithm design and be exposed to different ways of analysing the algorithms. We will conclude with identifying the best algorithm for the given problem.

II. ALGORITHM DESIGN

According to the given problem i.e., to find the floor of an element in a sorted array, there can be various algorithms to solve it. But to do it without using any in-built function, the algorithm is as follows.

We will be using two approaches for the given problem, one of those two approaches uses linear search algorithm and the other uses divide an conquer approach (binary search algorithm).

A. Linear Search Approach

1) Approach: The idea is simple, traverse through the array and find the first element greater than x. The element just before the found element is the floor of x.

2) Algorithm:

1. Traverse through the array from start to end.
2. If the current element is greater than x print the previous number and break the loop.

3. If there is no number greater than x then print the last element
4. If the first number is greater than x then print -1

B. Binary Search Approach

1) Approach: There is a catch in the problem, the given array is sorted. The idea is to use Binary Search to find the floor of a number$x$ in a sorted array by comparing it to the middle element and dividing the search space into half.

2) Algorithm:

1. The algorithm can be implemented recursively or through iteration, but the basic idea remains the same.
2. There is come base cases to handle.
   (a) If there is no number greater than x then print the last element
   (b) If the first number is greater than x then print -1
3. create three variables$low = 0$, mid and $high = n\text{-}1$ and another variable to store the answer
4. Run a loop or recurse until and unless low is less than or equal to high.
5. check if the middle ($(low + high)\ /2$) element is less than x, if yes then update the low, i.e $low = mid + 1$, and update answer with the middle element. In this step we are reducing the search space to half.
6. Else update the low, i.e $high = mid$ - 1
7. Print the answer.

III. PSEUDO CODE

This program finds the floor of x in a sorted array using linear search approach

```
Function floor(Argument arr[], Argument n, Argument x)
{
  If x is greater than or equal to arr[n - 1]
    return n - 1;
  if x is less than arr[0]
    return -1;
  Iterate i over 1 to n-1
    if arr[i] is greater than x
      return (i - 1);
  end of for loop
  return -1;
end
}
In the main function(){
  Initialize integer array arr[]
  Initialize n as size of array
  Initialize x
  Call the floor function and store the result in index
  if index is equal to -1
    print "Floor of x doesn't exist in array "
  else
```

print "Floor of x is arr[index]"
return 0;
}

This following program finds the floor of x in a sorted array using binary search approach

Function floor(Argument arr[], Argument low, Argument high, Argument x)
{
if low is greater than high
return -1;
if (x >= arr[high])
return high;
initialize mid = (low + high) / 2
if arr[mid] is equal to x
return mid;
if mid is greater than 0 and arr[mid - 1] is less than or equal to x and x is less than arr[mid]
return mid - 1;
if x is less than arr[mid]
return floor(arr, low, mid - 1, x)
return floor(arr, mid + 1, high, x)
end
}

In the main function(){
Initialize integer array arr[]
Initialize n as size of array
Initialize x
Call the floor function and store the result in index
if index is equal to -1
print "Floor of x doesn't exist in array "
else
print "Floor of x is arr[index]"
return 0;
}

IV. ALGORITHM ANALYSIS

A. Analysis of naive approach

In the naive approach we are using linear search method in which a for loop will be running for n times in the worst case possibility.

As there is only a single for loop the time complexity will be as follows

Time Complexity: O(n)

Where n is the size of the sorted array

No extra space is required for any additional array or such so the space complexity will be

Space Complexity: O(1)

TABLE 1

TIME COMPLEXITY OF LINEAR SEARCH APPROACH

Class	**Time**
Worst case Performance	O(n)
Best Case Performance	O(1)

B. Analysis of Binary Search Approach

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until we've narrowed down the possible locations to just one.

Complexities like O(1) and O(n) are simple to understand. O(1) means it requires constant time to perform operations like to reach an element in constant time as in case of dictionary and O(n) means, it depends on the value of n to perform operations such as searching an element in an array of n elements.

2) Calculating time complexity: let say the iteration in Binary Search terminates after k iterations.

- At each iteration, the array is divided by half. So let's say the length of array at any iteration is n.

- at iteration 1, length of array = n

- at iteration 2, length of array = n/2

- at iteration 3, length of array = (n/2)/2 = $n/2^2$

- after iteration k, length of array = $n/2^k$

- also we know that after k divisions, the length of array becomes 1

- therefore length of array = $n/2^k$ = 1

=> n = 2^k

Appling log function on both sides

$\mathbf{k = \log_2 (n)}$

Hence, the **time complexity of Binary Search is $\log_2$ (n).**

Best Case Analysis

When we use the binary search approach the best case arises when the word we are searching is located exactly in the middle of the dictionary. Then in this case the searching takes constant time. And the time complexity will become O(1).

TABLE 2

TIME COMPLEXITY OF BINARY SEARCH APPROACH

Class	**Time**
Worst case Performance	O(log n)
Best Case Performance	O(1)

No extra space is required for any additional array or such so the space complexity will be constant.

Space Complexity: O(1)

VI. PROFILING

A. Naive Approach:

So, after the above tabular analysis of *Apriori Analysis*, we come to the *Posteriori Analysis or Profiling.* Now let us have the glimpse of time graph and then comparison between both the approaches as a follow-up.

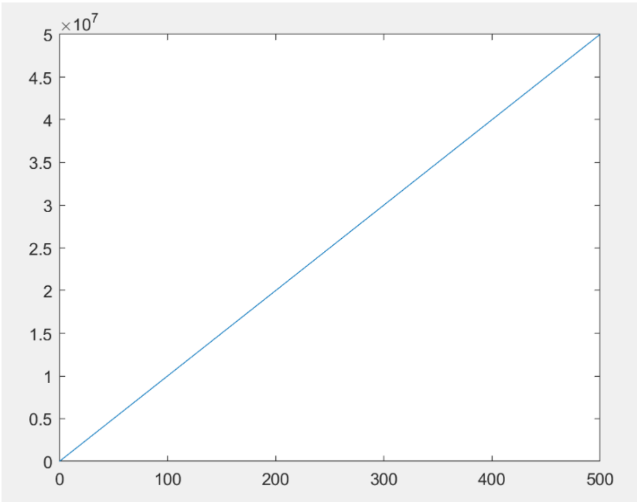

Figure 1: Naive Approach

B. Binary Search Approach

The graph which is an almost experimental result of time complexity of binary search approach used to find the floor of x, shows that there is not much increase in the time taken by the program to execute even if we increase the input value 'n'.

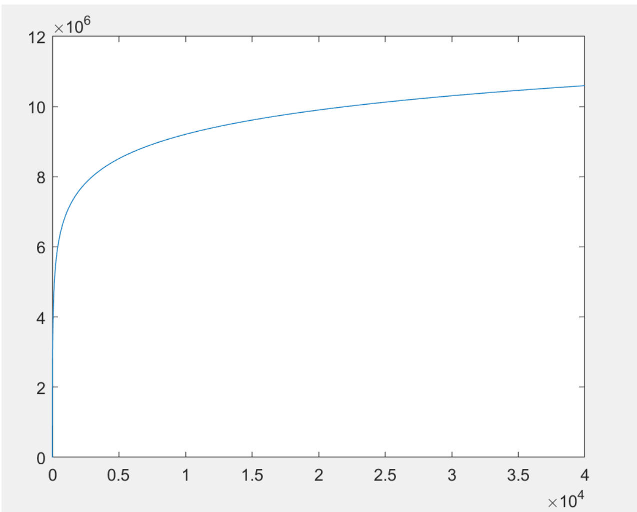

Figure 2: Binary Search

The following picture describes about the time complexity between the naive approach and the binary search approach discussed in the previous sections. Where n is the number of elements present in the sorted array.

—— Naive Approach

—— Binary Search Approach

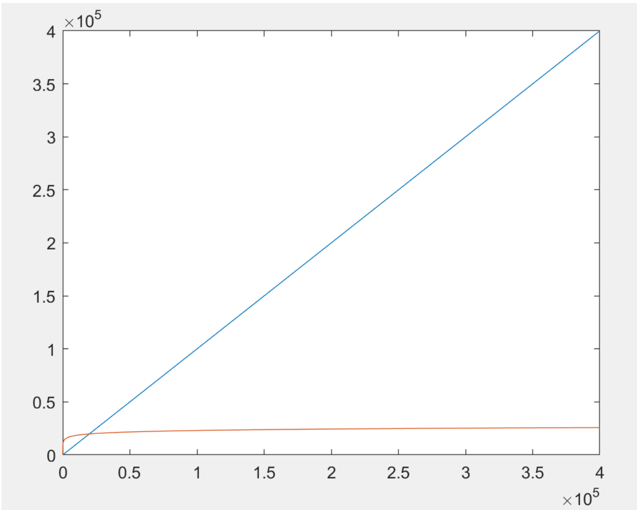

Figure 3: Comparison between naive and binary search approach

VII. CONCLUSION

We can conclude that with the above discussed algorithms the algorithm with binary search approach has the least time and space complexity to find the floor of x in a sorted array where floor of x is the largest element in an array smaller than or equal to x.

The worst case time complexity for linear search approach and the divide and conquer approach are O(n) and O(log n) respectively. And the space complexity for both approaches is constant i.e. O(1) as no approach uses any extra space such as arrays.

REFERENCES

[1] Introduction to Algorithms / Thomas H. Cormen … [et al.]. - 3^{rd} edition.

[2] The Design and Analysis of Algorithms (Pearson) by A V Aho, J E Hopcroft, and J D Ullman

[3] Algorithm Design (Pearson) by J Kleinberg, and E Tard

[4] https://en.wikipedia.org/wiki/Binary_search_algori

[5] https://www.geeksforgeeks.org/floor-in-a-sorted-array/