Ternary Search

Mohtashim Nawaz* , Harsh Aryan†, Mayank Taksande‡ and Sneha Mishra§
Indian Institute of Information Technology, Allahabad
Allahabad-211012
Email: *itm2017005@iiita.ac.in, †itm2017003@iiita.ac.in, ‡ itm2017008@iiita.ac.in, § itm2017004@iiita.ac.in

***Abstract*—This Paper introduces an algorithm to implement ternary search. The idea is to work out an efficient algorithm in terms of its Time Complexity as well as Space Complexity.**

I. Introduction

We are given an array of elements and we have to search for a specific element. The implementation of this search is implemented through Ternary Search. In this approach, we divide the array into three partitions and element is searched for depending on which range it lies and hence we search for it in that partition.

This problem is similar to the Binary Search where the array is partitioned into 2 parts and according to the magnitude the element it is recursively searched for in that part.

The problem makes use of Divide and Conquer Approach. Divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

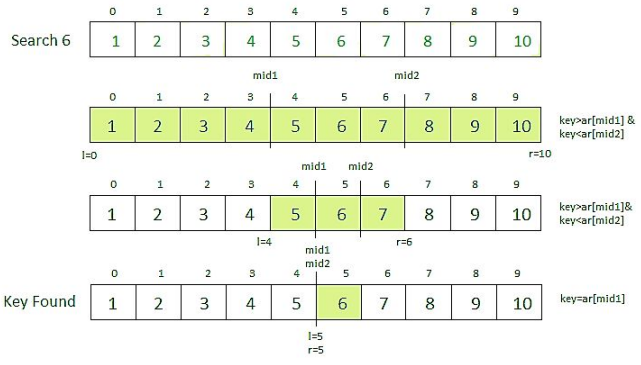

Fig 1.1:Illustration of ternary search

Here we will try to develop an efficient algorithm to implement ternary search, will use the following approach to develop the algorithm.

1) Using a recursive approach

2) Using the iterative approach

This algorithm is tested for different sample test cases for time complexity, space complexity and accuracy. In our case, we tend to focus on establishing a rule or a relationship between the time and input data.

This report further contains -

II. Algorithm Design and Analysis

III. Result

IV. Conclusion

V. References

II. Algorithm Design And Analysis

An efficient algorithm can only be created by considering every possibility. So we have to consider every possibility while designing the algorithm.

Steps for designing the algorithm include:

1). First, we compare the key with the element at mid1. If found equal, we return mid1.

2). If not, then we compare the key with the element at mid2. If found equal, we return mid2.

3). If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.

4). If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.

5). If not, then we recur to the second (middle) part.

Algorithm to Sort using Merge Sort

```
def mergeSort(arr[] : int, l : int, r : int)
        if (l < r)
                int m← l+(r-l)/2
                mergeSort(arr, l, m)
                mergeSort(arr, m+1, r)
                merge(arr, l, m, r)
```

```
def merge(arr[]: int, l: int, m: int, r: int)
        i: int , j: int, k: int, n1: int, n2: int;
        n1 ← m - l + 1
        n2 ← r - m
        L[n1]: int, R[n2]: int
        while(i < n1)
                L[i] ← arr[l + i]
        while (j < n2)
                R[j] ← arr[m + 1+ j]
        i ←0
        j ←0
        k ←l
        while (i < n1 and j < n2)
                if (L[i] <= R[j])
                        arr[k] ←L[i]
                        i←i+1
                else
                        arr[k] = R[j]
                        j←j+1
                k←k+1
        while (i < n1)
                arr[k] ←L[i]
                i←i+1
                k←k+1
        while (j < n2)
                arr[k]← R[j]
                j←j+1
                k←k+1
```

Time Analysis:
Worst Case: O(n logn)
Best Case: Ω(n logn)
Average Case: Θ(n logn)

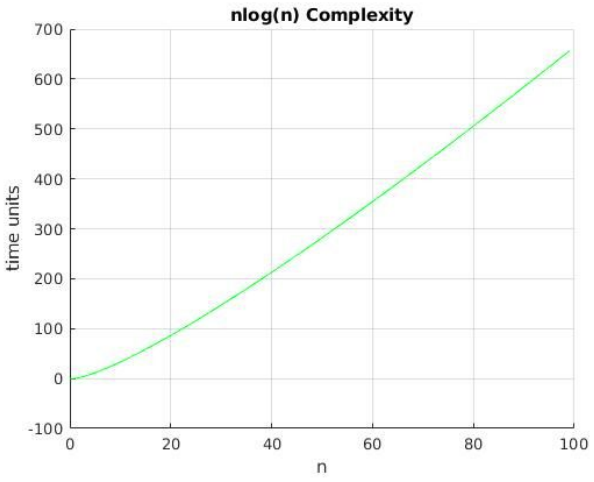

Fig 2.1:Graph depicting nlog(n) complexity

Algorithm for ternary search (using recursion)

```
def ternerySearch(ar[]:int, l:int, r:int, key:int)
        if(l<=r)
                mid1:int, mid2:int
                mid1←l+(r-l)/3
                mid2←r-(r-l)/3
                if( key = ar[mid1])
                        print "\nElement found at "
                        print "%dth position ",mid1+1
                if(key = ar[mid2])
                        print "\nElement found at "
                        print "%dth position ",mid2+1
                if(key<ar[mid1])
                        ternerySearch(ar,l,mid1-1,key)
                else if(key>ar[mid2])
                        ternerySearch(ar,mid2+1,r,key)
                else if(key>=ar[mid1] &&
key<=ar[mid2])
                ternerySearch(ar,mid1+1,mid2-1,key)
                else
                        print "\nSorry Not Found :("
```

Time Analysis:
$= 4*\log_3 n + 1$
Worst Case: $O(\log_3 n)$
Best Case: $\Omega(\log_3 n)$
Average Case: $\Theta(\log_3 n)$

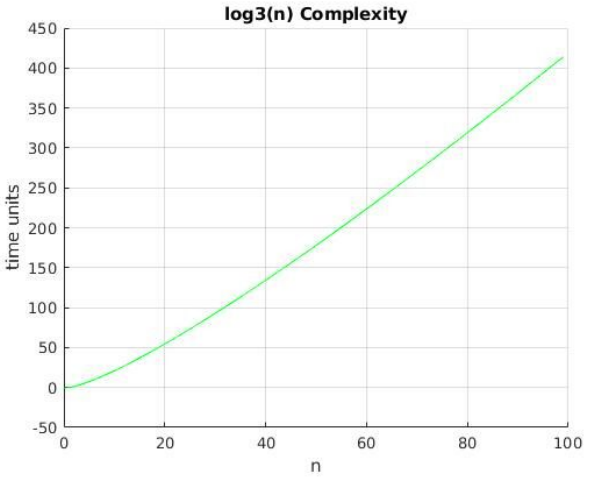

Fig 2.2:Graph depicting $\log_3 n$ *complexity*

Algorithm for Ternary Search (iterative approach)

```
def  ternarySearch(l:int, r:int, key:int, ar[]:int[])

   while (r >= l)  do
```

```
    mid1:int → l + (r - l) / 3;
    mid2:int → r - (r - l) / 3;

    if ar[mid1] == key then
       return mid1

    if ar[mid2] == key then
       return mid2

    if key < ar[mid1] then
         r → mid1 - 1

    else if key > ar[mid2] then
         l = mid2 + 1

    else
        l = mid1 + 1
        r = mid2 - 1

  return -1;
```

Time Analysis:
= $4*\log_3 n + 1$
Worst Case: $O(\log_3 n)$
Best Case: $\Omega(\log_3 n)$
Average Case: $\Theta(\log_3 n)$

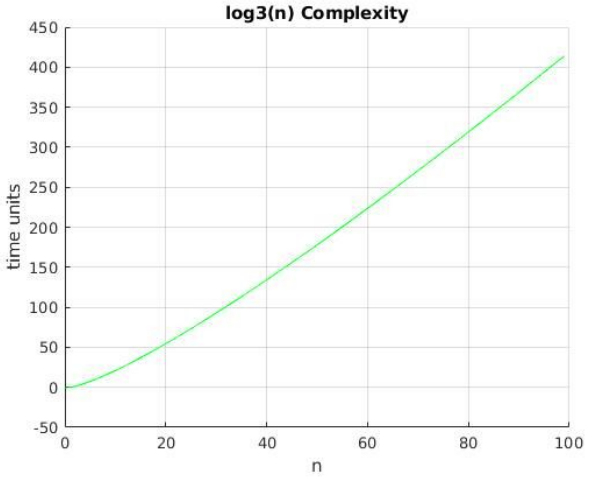

Fig 2.3:Graph depicting $\log_3 n$ *complexity*

Main function (code body)

```
def main()
    n:int, flag1:int,flag2:int
    flag1← 0
    flag2← 0
    print "Enter the size of array : "
    scan n
    int ar[n]
    i : int
    i←0
    print "\nEnter The Array : "
    While (i<n)
          scan ar[i]
    while(i<n-1)
          if(ar[i]>ar[i+1])
                flag1← flag+1
          else if(ar[i]<ar[i+1])
                flag2← flag2+1
    if(flag1=n-1 or flag2=n-1)
          print "\nSequence is Sorted! "
    else
          print "\nSequence is not Sorted! "
          print "\nSorting..."
          mergeSort(ar,0,n-1);
          printf "\n"
          sleep(1);
          while(i<n)
                print "%d ",ar[i]
    key:int
    print "\nEnter the Element to be Searched : "
    scan key
    ternerySearch(ar,0,n,key)
    return 0
```

Time Complexity:
Worst Case: $O(n \log(n)+\log_3 n)$
Best Case: $\Omega(\log_3 n)$

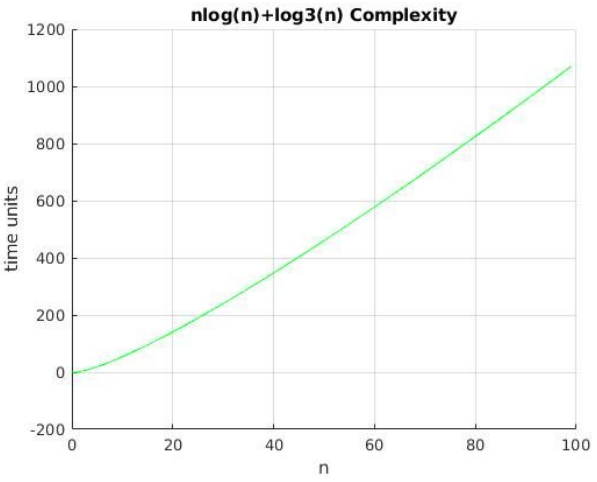

Fig 2.4:Graph depicting n log(n)+$\log_3 n$ *complexity*

Although it may seem that Ternary search with time complexity $\log_3 n$ would be better than binary search with time complexity of $\log_2 n$, but it is not quite the case.

On a closer look, Ternary search takes $4*\log_3 n$ time whereas binary search takes $2*\log_2 n$ time and in the overall scenario, binary search takes lesser time, that is lesser number of comparison for the worst case.

Hence Binary search is generally preferred over Ternary search.

III. Result

Overall Time Complexity

Time Complexity of algorithm is not equal to the actual time required to execute particular code but the number of times a statement execute.
We have come across both the iterative and the recursive algorithm for this problem.
The time complexity (for ternary search) is $O(\log_3 n)$

Space Complexity

In both Recursive Approach and Iterative Method we do not need to store the value so the space complexity will be constant(used by variables only).
The Main Function shared by both the approaches stores values in integer array which makes the overall Space complexity O(n).

IV. Conclusion

We have developed an algorithm to solve the given problem, that is to search for an element through ternary search.

We used different approaches to solve this problem and came up with both recursive and iterative solution. We have also used merge sort to sort the input array in case it is not already sorted.

This is the best possible implementation of the given problem.

V. References

[1] ”Ternary Search”, Wikipedia [Online]
Available:-https://en.wikipedia.org/wiki/Ternary_search
[Accessed: April 3 2019]

[2] ”Ternary Search”, GeeksforGeeks [Online]
Available:-https://www.google.com/amp/s/www.geeksforgeeks.org/ternary-search/amp/
[Accessed: April 4 2019]

[3] Thomas H. Corman, Introduction to Algorithms, 3rd Edition 2003

[4] Ellis Horowitz,Sartaj Sahni,Sanguthevar Rajasekaran,Fundamentals of Computer Algorithms, 2nd Edition