Geeks Man

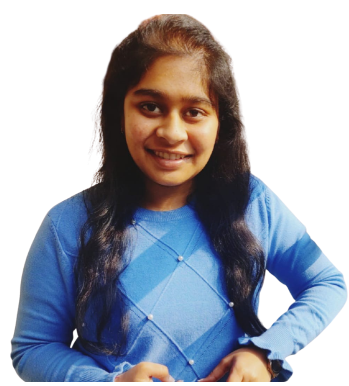
Algorithms

Lesson 4

Trilok Kaushik

Founder of Geeksman

Aditya Aggarwal



Aditya Gupta

Suhani Mittal

Team Coordinators

Topic:- Searching (Part-II)

2. Binary Search :-

Characteristics :

- It is also known as half-interval Search or logarithmic Search.
- It is more efficient than Linear Search if **Data is sorted** and Data is stored in Array.
- This Searching Technique doesn't work on unsorted sets of Data .
- In most of the cases Binary Search Algorithm is used.

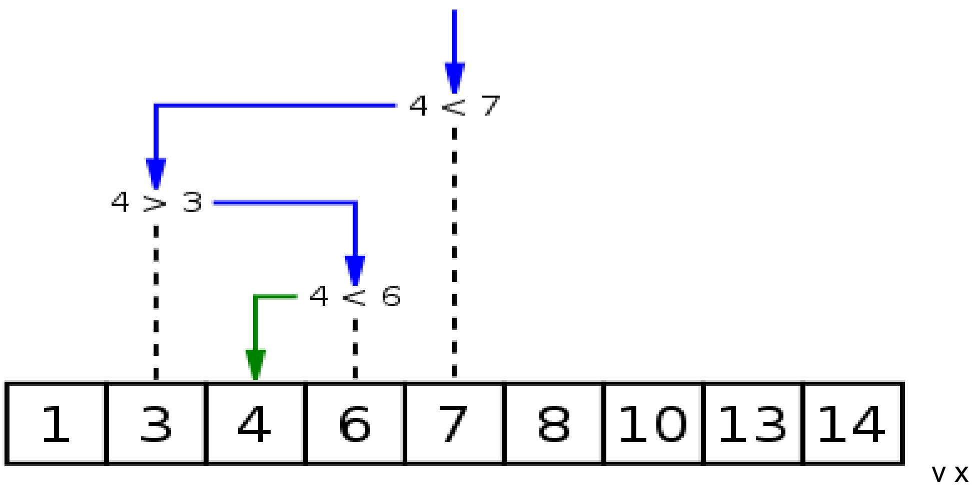
➔ *Binary Search using Array :*

- ❑ **Iterative Approach :-**

Algorithm :

1. Set beg=0 ; end= n-1;
2. Set mid=(beg+end)/2; //Begins by calculating the middle element of the array.

3. If Target value == middle element : **return mid;**
   // If matches with the middle element , its position is returned.
4. If Target value < middle element : search begins in left half by changing **end** to mid-1;
5. If Target value > middle element : Search begins in right half by changing **beg** by mid+1;
6. If beg > end : Search is unsuccessful and terminates,since we have traversed the whole array and didn't find the target value.



Indexes 0 1 2 3 4 5 6 7 8

Search Element = 4

Code :

```
// input - input array
// n - size of array
int binarySearch(int *input, int n, int val)
{
```

```
    int start=0,end=n-1;
    while(start<=end)
    {
    int mid=(start+end)/2;
    if(input[mid]==val)
        return mid;//found the target element
   else
        if(input[mid]<val)
             start=mid+1;//search in right half of array
        else
             end=mid-1;//search in left half of array
      }
      return -1;//unsuccessful search
   }
```

Here is the link for the complete code of binary search
https://sapphireengine.com/@/j8fpcs

- ### Recursive Approach :

Algorithm :

1. Find the Mid element from beg to end of the array and compare with the target element !
2. If target element gets matched with middle element return mid element
3. If target element > mid element return function call for the array from mid+1 to end
4. If target element < mid element return function call for the array from beg to mid-1
5. If beg>end : return -1   // Unsuccessful Search

Code :

```
// arr - user input sorted array
// n - size of array
// key - element to be searched

//helper function for binary search taking start, end , key ,array as input

int helper(int *arr,int start,int end,int key){
    if(start>end)return -1;//unsuccessful search
    int mid=(start+end)/2;
    if(arr[mid]==key){
        return mid;
    }
    else if(arr[mid]<key){
        return helper(arr,mid+1,end,key);
    }
    else return helper(arr,start,mid-1,key);
}

//original binary search function taking parameters array, n, key & calls helper function which return the index of key

int binarySearch(int *arr,int n,int key){
    return helper(arr,0,n-1,key);
}
```

Here is the link for the complete code of binary search using recursion
https://sapphireengine.com/@/n7dvvu

➔ Binary Search using LinkedList :

- **To perform Binary Search determination of the Middle element is important. In LinkedList , memory allocation for the singly linked list is dynamic and non-contiguous, which makes finding the middle element difficult.**

- Still , Binary Search can be implemented using the 2 pointers approach which would cause Time Complexity of O(n) in searching the middle element while in Array time Complexity would be O(logn).

- Similarly, Space Complexity also increases as the linked list is not only holding the Data but also the Address of the next node .

➔ Different Searching Techniques :

With the help of sorted Data we can perform other Searching Techniques also like :

1. **Ternary Search :**

   Code :

```
// Function to perform Ternary Search
int ternarySearch(int l, int r, int key, int ar[])

{
    while (r >= l) {

```

```
        // Find the mid1 and mid2
        int mid1 = l + (r - l) / 3;
        int mid2 = r - (r - l) / 3;

        // Check if key is present at any mid
        if (ar[mid1] == key) {
            return mid1;
        }
        if (ar[mid2] == key) {
            return mid2;
        }

        // Since key is not present at mid,
        // check in which region it is present
        // then repeat the Search operation
        // in that region

        if (key < ar[mid1]) {

            // The key lies in between l and mid1
            r = mid1 - 1;
        }
        else if (key > ar[mid2]) {

            // The key lies in between mid2 and r
            l = mid2 + 1;
        }
        else {

            // The key lies in between mid1 and mid2
            l = mid1 + 1;
            r = mid2 - 1;
        }
    }

```

```
    // Key not found
    return -1;
}
```

From the first look, it seems the ternary search does less number of comparisons as it makes $\log_3 n$ recursive calls, but binary search makes $\log_2 n$ recursive calls.

Worst case of Binary Search :

```
 T(n) = T(n/2) + 2,  T(1) = 1
```

worst case of Ternary Search :

```
 T(n) = T(n/3) + 4, T(1) = 1
```

In binary search, there are $2\log_2 n + 1$ comparisons in the worst case. In ternary search, there are $4\log_3 n + 1$ comparisons in the worst case.

Time Complexity for Binary search = $2\log_2 n + O(1)$
Time Complexity for Ternary search = $4\log_3 n + O(1)$

The value of $2\log_3 n$ can be written as $(2 / \log_2 3) * \log_2 n$. Since the value of $(2 / \log_2 3)$ is more than one, Ternary Search does more comparisons than Binary Search in the worst case.

2. <u>Jump Search :</u>

Jump Search is a searching algorithm for sorted arrays(like Binary Search). The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Algorithm :

1. Sort the Array
2. Initialize the variable jump and mention the size of jump as m
3. Start from index 0 and jump until the key element gets smaller than the jumped element.
4. If Key element > arr[jump] then go back to previous jump state and start linear Search from arr[jump-m] to arr [jump]
5. If element found return index else return -1;

What is the optimal block size to be skipped?
n= Size of array
m= block size to be jumped

In the worst case, we have to do **n/m jumps** and if the last checked value is greater than the element to be searched for, we perform **m-1 comparisons more for linear search.** Therefore the **total number of comparisons in the worst case will be ((n/m) + m-1)**. The value of the function ((n/m) + m-1) will be minimum when **m = √n**.

Therefore, **the best step size is m = √n.**

3. Exponential Search :

Just like Jump Search , Exponential search also works in this criteria by jumping the element exponentially rather than fixed size which reduces the comparisons in the previous algorithm.

Algorithm :

1. The idea is to start with subarray size 1
2. compare its last element with x,
3. then try size 2, then 4 and so on until the last element of a subarray is not greater.

4. Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i because we could not find a greater value in previous iteration

Time Complexity :- **O(Log n)**

Questions

Ques1 :- *Consider a sorted array of n numbers. What would be the time complexity of the best known algorithm to find a pair 'a' and 'b' such that |a-b| = k , k being a positive integer.*
(A) O(n)
(B) O(n log n)
(C) O(n ^ 2)
(D) O(log n)

Ques2 :- Consider the following C program that attempts to locate an element x in an array Y[] using binary search. The program is erroneous.

```
f(int Y[10], int x) {
int i, j, k;
i = 0; j = 9;
do {
          k =  (i + j) /2;
          if( Y[k] < x)  i = k; else j = k;
     } while(Y[k] != x && i < j);
if(Y[k] == x) printf ("x is in the array ") ;
else printf (" x is not in the array ") ;
}
```

On which of the following contents of Y and x does the program fail?

(A) Y is [1 2 3 4 5 6 7 8 9 10] and x < 10

(B) Y is [1 3 5 7 9 11 13 15 17 19] and x < 1

(C) Y is [2 2 2 2 2 2 2 2 2 2] and x > 2

(D) Y is [2 4 6 8 10 12 14 16 18 20] and 2 < x < 20 and x is even

Ques3 :- Given a sorted array of integers, what can be the minimum worst case time complexity to find ceiling of a number x in given array? Ceiling of an element x is the smallest element present in array which is greater than or equal to x. Ceiling is not present if x is greater than the maximum element present in array. For example, if the given array is {12, 67, 90, 100, 300, 399} and x = 95, then output should be 100.
(A) O(LogLogn)
(B) O(n)
(C) O(Logn)
(D) O(Logn * Logn)

Ques4 :- Which of the following is the correct recurrence for the worst case of Binary Search?
(A) T(n) = 2T(n/2) + O(1) and T(1) = T(0) = O(1)
(B) T(n) = T(n-1) + O(1) and T(1) = T(0) = O(1)
(C) T(n) = T(n/2) + O(1) and T(1) = T(0) = O(1)
(D) T(n) = T(n-2) + O(1) and T(1) = T(0) = O(1)

Ques5 :- Consider a sorted array of n numbers and a number x. What would be the time complexity of the best known algorithm to find a triplet with sum equal to x. For example, arr[] = {1, 5, 10, 15, 20, 30}, x = 40. Then there is a triplet {5, 15, 20} with sum 40.
(A) O(n)

(B) O(n^2)

(C) O(n Log n)

(D) O(n^3)