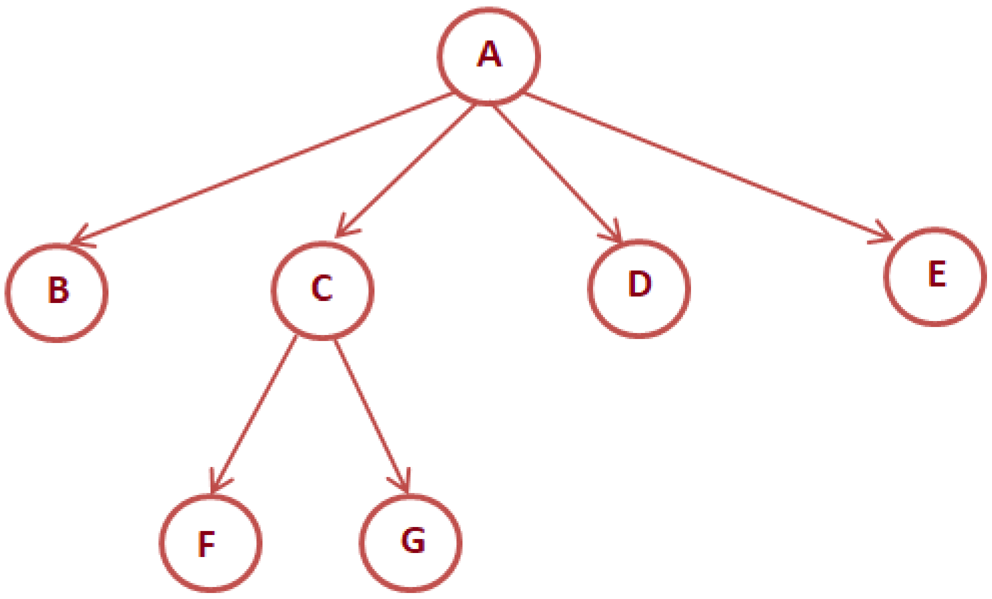
Introduction

Binary Search is an algorithmic technique where we make two partitions of the problem and then throw away one instance of it based on a certain condition. The primary condition for using the Binary Search technique is **monotonicity**. When we can prove that the value of a boolean condition will be true for some time and then become false for the rest of the search space, or vice versa, it often implies that Binary Search can be used there.



Binary Search in Sorted Array

Problem Statement: You are given an array of n elements which is sorted. You are also given an element target which you need to find out whether it is present in the array or not.

We will first discuss the naive algorithm of Linear search and then we will exploit the property that the given array is sorted using Binary Search.

Linear Search

It is a simple sequential search over all the elements of the array, and each element is checked for a match. If a match is found, return the element, otherwise, the search continues until we reach the end of the array.

Pseudocode:

```
/*
    array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function linearSearch(arr, leftidx , rightidx , target)

      // Search for the target from the beginning of arr
      for idx = 0 to arr.length-1
            if arr[idx] == target
                  return idx
      //   target is not found
      return -1
```

Time complexity: O(N), as we traverse the array only once to check for a match for the target element.

Space complexity: O(1), as no extra space is required.

Binary Search

Suppose you want to find a particular element in the sorted array, then following this technique, you have to traverse all the array elements for searching one element but guess if you only have to search at most half of the array indices for performing the same operation. This can be achieved through binary search.

Here consider the monotonic condition: Uptill certain index the value of all elements will be <= x and after that, the value of all elements will be > x. So condition **<=** is monotonic in our case. Hence we can apply Binary search!

Let us consider the array:

0 1 2 3 4

1 2 3 4 5

Given an array of size 5 with elements inside the boxes and indices above them. Our target element is 2.

Steps:

1. Find the middle index of the array.
2. Now, we will compare the middle element with the target element. In case they are equal then we will simply return the middle index.
3. In case they are not equal, then we will check if the target element is less than or greater than the middle element.
- In case, the target element is less than the middle element, it means that the target element, if present in the given array, will be on the left side of the middle element as the array is sorted.
- Otherwise, the target element will be on the right side of the middle element.
1. This helps us discard half of the length of the array each time and we reduce our search space to half of the current search space.

In the example above, the middle element is 3 at index 2, and the target element is 2 which is less than the middle element, so we will move towards the left part. Now marking start = 0, and end = n/2-1 = 1, now middle = (start + end)/2 = 0. Now comparing the 0-th index element with 2, we find that 2 > 1, hence we will be moving towards the right. Updating the start = 1 and end = 1, the middle becomes 1, comparing the 1-st index of the array with the target element, we find they are equal, meaning from here we will simply return 1 (the index of the element).

Recursive Pseudocode:

```
/*
    array of size N from 0 to N-1 is considered
*/
function binarySearch(arr, N, target, lo , hi)

    // base condition if element is not found
    if lo > hi
        return -1
    // Finding the mid of search space from lo to hi
    mid = lo + (hi-lo)/2
    //  If the target element is present at mid
    if arr[mid] == target
        return mid
    /*
        If the target element is less than arr[mid], then if the
        target is present, it must be in the left half.
    */
    if target < arr[mid]
        return binarySearch(arr, N, target, lo, mid - 1)
    // Otherwise if the target is present, it must be in the right half
    else
        return binarySearch(arr, N, target, mid + 1, hi)
```

Iterative Pseudocode:

```
/*
    array of size N from 0 to N-1 is considered
*/
function binarySearch(arr, N, target)

    // Initializing lo and hi pointers
    lo = 0
    hi = N-1
    // Searching for the target element until lo<=hi
    while lo <= hi
        // Finding the mid of search space from lo to hi
        mid = lo + (hi-lo)/2
        //  If the target element is present at mid
        if arr[mid] == target
            return mid
        /*
            If the target element is less than arr[mid], then if the
            target is present, it must be in the left half.
        */
        if target < arr[mid]
            hi = mid-1

        else
            // Otherwise if the target is present, it must
            // be in the right half
            lo = mid+1

    // If the target is not found return -1
    return -1
```

Time complexity: O(logN), where N is the number of elements in the array, given the array is sorted. Since we search for the target element in one of the halves every time, reducing our search space to half of the current search space.

Since we go on searching for the target until our search space reduces to 1, so

Iteration 1- Initial search space: N

Iteration 2 - Search space: N/2

Iteration 3 - Search space: N/4

Let after ‘k’ iterations search space reduces to 1

So, N/(2k) = 1

=> N = 2k

Taking Log2 on both sides:

=> k = log2N

Hence, the maximum number of iterations ‘k’ comes out to be log2N.

Space complexity: O(1), as no extra space is required.

Generic Binary Search

It is easier to think about binary search when solving a more general problem than finding an element in a sorted list. It is quite a handy algorithm when you are trying to solve a problem that has the following properties. The answer to the problem is a number. You don’t know the answer, but given the answer, you can code the algorithm to check if the number is an answer to the problem in a relatively straightforward way. So, essentially you have a predicate function ok(x) that returns true when x is a potential solution. Your task is to find the minimal x so that ok(x) returns true.

There is one crucial property that needs to be satisfied to apply a binary search algorithm. The predicate has to be *monotonic*. It means that if x is the lowest number making the predicate true, then it is also true for all the larger values of x. For example, the following table shows the values of a monophonic predicate over integers in the range from 1 to 6:

x : 1 2 3 4 5 6

```
ok(x) :  F  F  T  T  T  T
```

In a binary search algorithm, we introduce two variables. One variable keeps the index of the left side of the currently searched range and is commonly shortened to (l), while the second one keeps the index of the right side (r). The algorithm maintains the following *invariant* throughout its operation*:*

- ok(l) is false
- ok(r) is true

Binary search is a primitive example of a *divide and conquer* algorithm. It repeatedly halves the range from l to r while maintaining this invariant until l and r only differ by one, thus exactly pinpointing the indices at which monotonic predicate turns from false to true:

```
 x  :  1  2  3  4  5  6

ok(x) :  F  F  T  T  T  T

            ^   ^

            l    r     // at the end
```

So the whole solution is to run the following loop:

```
while (r - l > 1)
        // compute the midpoint of l..r interval
        m = (l + r) / 2
        if (ok(m))
        // Maintain invariant: ok(r) is true
                r = m
        else
        // Maintain invariant: ok(l) is false
                l = m
```

If you keep the loop invariant in mind when writing this code it is easy to see what exactly needs to be put into each of the branches of the if statement. No chance for any kind of off-by-one errors.

Advantages of Binary search:

1. This searching technique is fast and easier to implement.
2. Requires no extra space.
3. Reduces time complexity of the program to a greater extent i.e O(logN), where N is the number of the elements in the search space.

Disadvantages of Binary search:

1. Binary Search can be only applied to monotonic problem space. For example, if the array is not sorted, we need to go back to the linear search or sort the array, both of which are very expensive operations
2. Binary search can only be applied to data structures that allow direct access to elements. If we do not have direct access then we can not apply binary search on it eg. we can not apply binary search to Linked List.