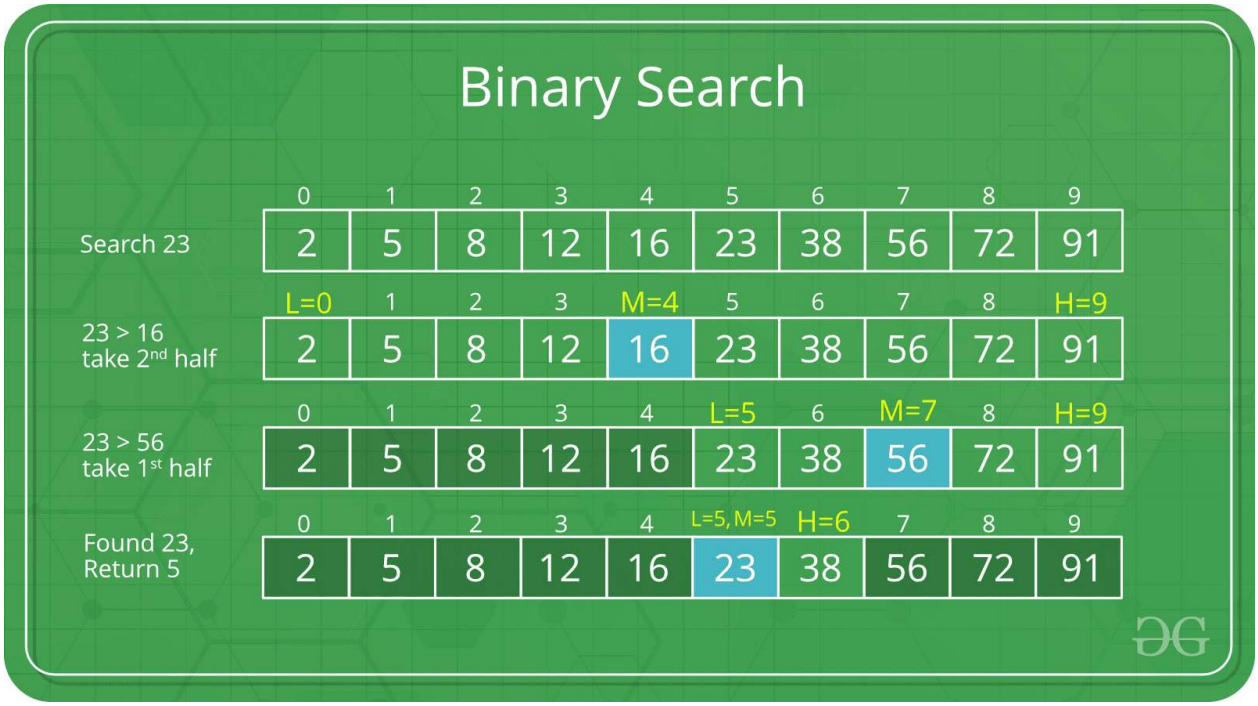
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to **O(Log n)**.



Implementation ->

1)Compare x with the middle element.

2)If x matches with the middle element, we return the mid index.

3)Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.

4)Else (x is smaller) recur for the left half

Recursive implementation of Binary Search

```
int binarySearch(int arr[], int l, int r, int x)
{
  if (r >= l) {
    int mid = l + (r - l) / 2;

    if (arr[mid] == x)
      return mid;
    if (arr[mid] > x)
      return binarySearch(arr, l, mid - 1, x);
    return binarySearch(arr, mid + 1, r, x);
  }
  return -1;
}
```

Iterative implementation of Binary Search

```
int binarySearch(int arr[], int l, int r, int x)
{
  while (l <= r) {
    int m = l + (r - l) / 2;

    if (arr[m] == x)
      return m;

    if (arr[m] < x)
      l = m + 1;
    else
      r = m - 1;
  }return -1;
}
```

Time Complexity:

The time complexity of Binary Search can be written as

T(n) = T(n/2) + c

The above recurrence can be solved either using the Recurrence Tree method or Master method. It falls in case II of the Master Method and the solution of the recurrence is **Theta(log(n))**.

Auxiliary Space: O(1) in case of iterative implementation. In the case of recursive implementation, **O(Logn)** recursion call stack space.