# Data Structures and Algorithms

# Types of Search Algorithms

1. Linear or Sequential Search
2. Binary Search

# Linear or Sequential Search

- It iterates through the whole array or list
- If the element found, returns its index
- Otherwise returns -1.

# Example:

- arr = [2, 12, 15, 11, 7, 19, 45]
- Suppose the target element we want to search is 7.

## •Approach for Linear or Sequential Search

- Start with index 0 and compare each element with the target
- If the target is found to be equal to the element, return its index
- If the target is not found, return -1

# Linear serarch code

```
public class LinearSearch {
    public static void main(String[] args) {
        int[] nums = {2, 12, 15, 11, 7, 19, 45};
        int target = 7;
        System.out.println(search(nums, target));
 }
static int search(int[] nums, int target) {
        for (int index = 0; index < nums.length; index++) {
            if (nums[index] == target) {
                return index;
            }
        }
        return -1;
    } }
```

# Time Complexity Analysis

**The Best Case** occurs when the target element is the first element of the array.

The number of comparisons, in this case, is 1. So, the time complexity is $O(1)$.

**The Average Case:** On average, the target element will be somewhere in the middle of the array.

The number of comparisons, in this case, will be N/2. So, the time complexity will be $O(N)$ (the constant being ignored).

**The Worst Case** occurs when the target element is the last element in the array or not in the array.
In this case, we have to traverse the entire array, and so the number of comparisons will be N. So, the time complexity will be $O(N)$.

# Binary Search

- A binary search algorithm works on the idea of neglecting half of the list on every iteration.

- It keeps on splitting the list until it finds the value it is looking for in a given list.

- A binary search algorithm is a quick upgrade to a simple linear search algorithm.

# Working of a Binary Search Algorithm

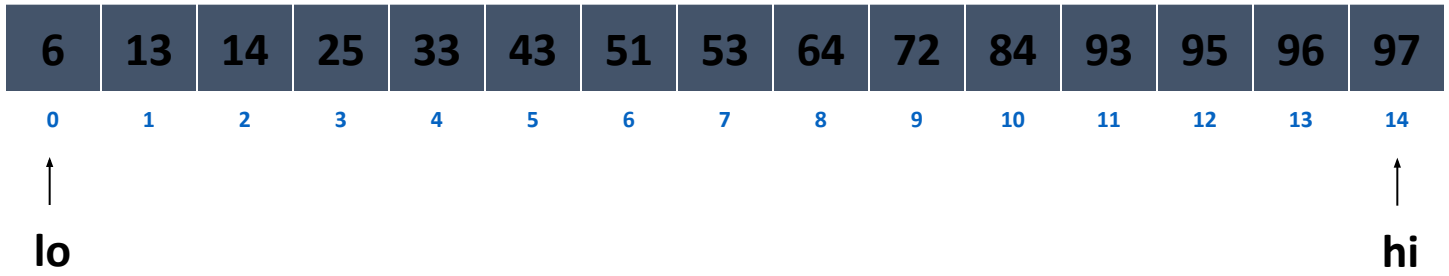
- The first thing to note is that a **binary search algorithm** always works on a sorted list.
- the first logical step is to sort the list provided

- After sorting, the median of the list is checked with the desired value.
- If the desired value is equal to the central index's worth, then the index is returned as an answer.
- If the target value is lower than the central index's deal of the list, then the list's right side is ignored.
- If the desired value is greater than the central index's value, then the left half is discarded.
- The process is then repeated on shorted lists until the target value is found.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

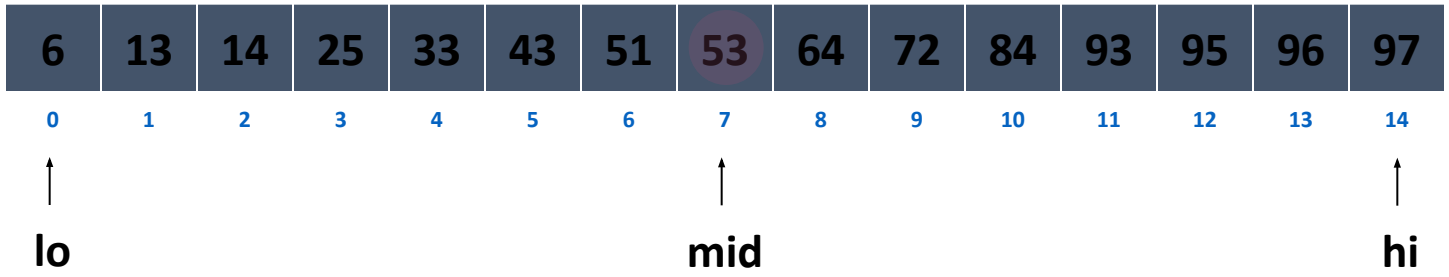



- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.




- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.


- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.


- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.


- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

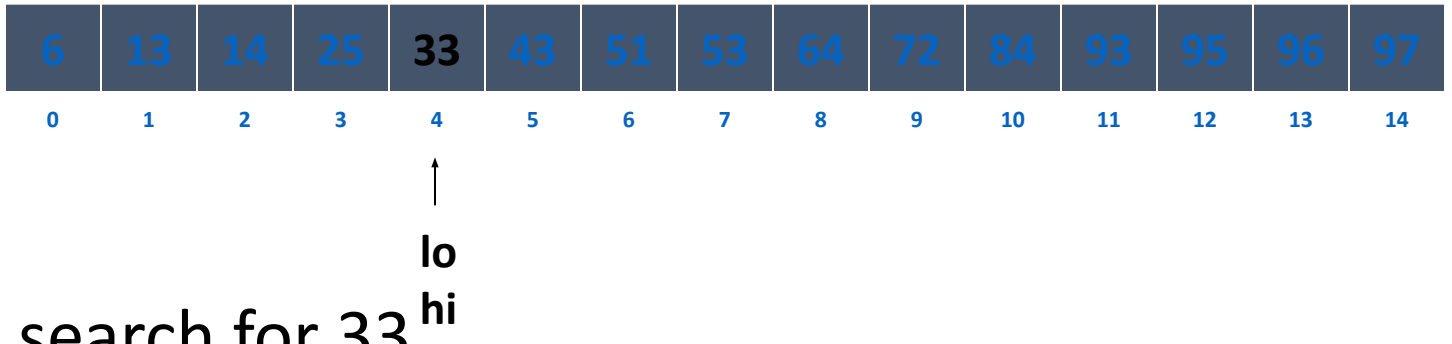
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.


- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

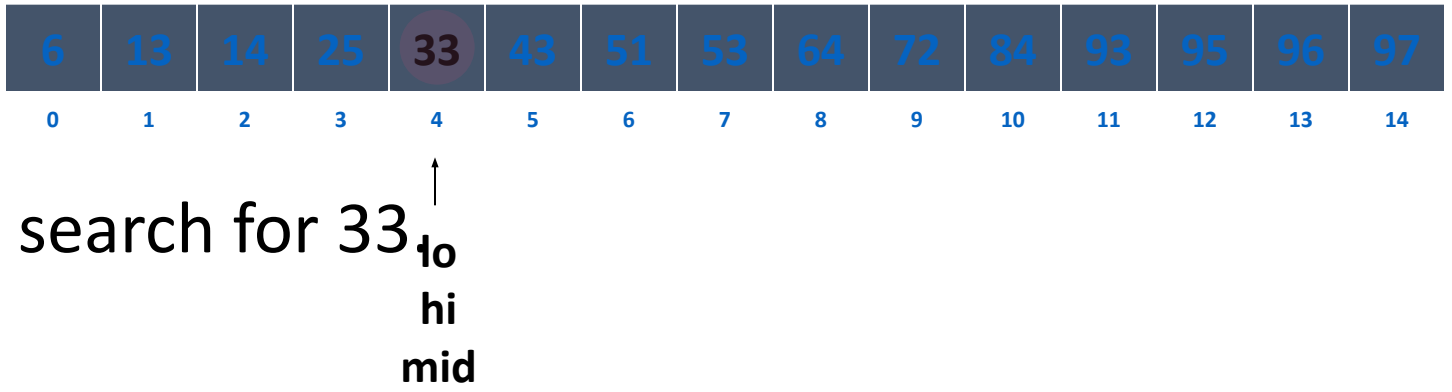



- Ex. Binary search for 33.

# Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.




- Ex. Binary search for 33

# Program

```
public static int binary(int arr[],int n,int value){
    int lo=0,hi=n-1;
   int  mid=(lo+hi)/2;
    while(lo<=hi){
        if(value==arr[mid]){
           return mid;
        }
        else if(value<arr[mid]){
            hi=mid-1;
        }
        else{
            lo=mid+1;
        }
    }
    return -1;

}
```

# Binary Search Time Complexity

# link

- https://dipeshpatil.github.io/algorithms-visualiser/#/searching