IARE

INSTITUTE OF AERONAUTICAL ENGINEERING

Searching Techniques – I

- Linear search
- Binary search
- Performance Analysis, Time complexity

Dr. B. Surekha Reddy
IARE10795

Searching Techniques

- Searching is the process of locating given value position in a list of values, i.e. search is a process of finding a value in a list of values.
- Search is said to be successful if the element being searched is found or unsuccessful when the element being searched is not found.
- A search typically answers in either True or False as to whether the item is present.
- On occasion it may be modified to return where the item is found.
- Two techniques are used for searching:

    1) Linear/ Sequential Search.

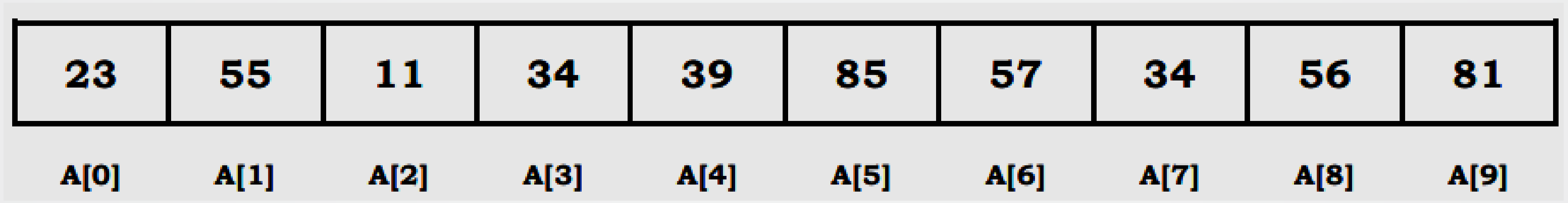
    2) Binary Search.

Linear Search

- Linear Search is used for an unsorted array. It mainly does one by one comparison of the item to be searched with array elements. It takes linear or O(n) Time.
- In Linear Search, we iterate over all the elements of the array and check if it the current element is equal to the target element.
- If we find any element to be equal to the target element, then return the index of the current element.
- Otherwise, if no element is equal to the target element, then return -1 as the element is not found.
- Linear search is also known as **sequential search**.

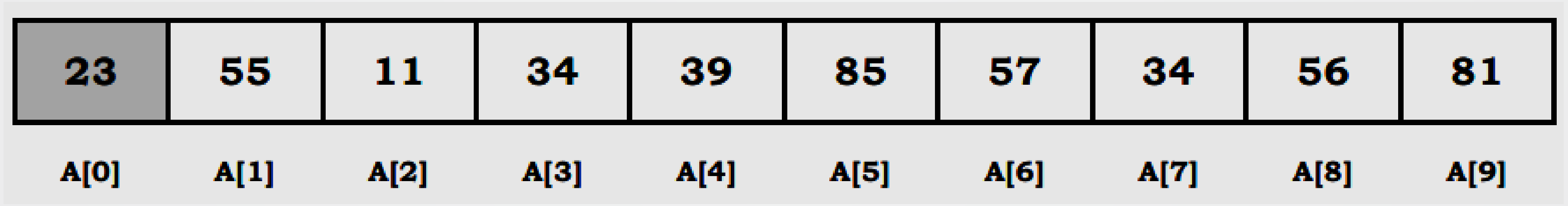
Linear Search

Example: Suppose we are given a array of 10 elements. We are to search element 72. i.e. X = 85.



23	55	11	34	39	85	57	34	56	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

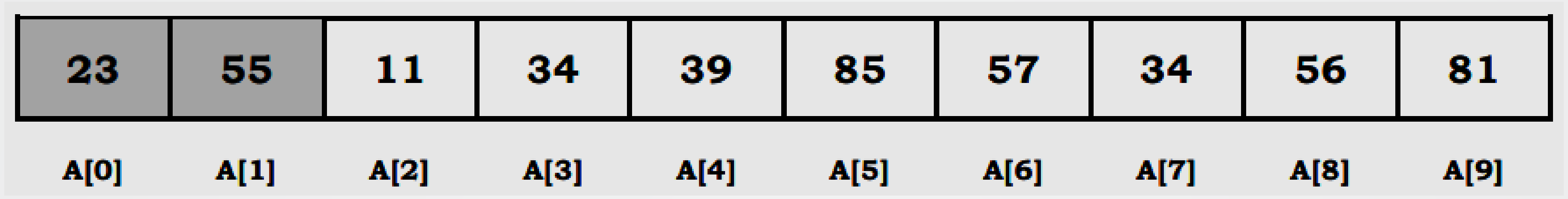
Step 1: It compares the x with first element located at index 0.Element is not matched then it moves to the next element.



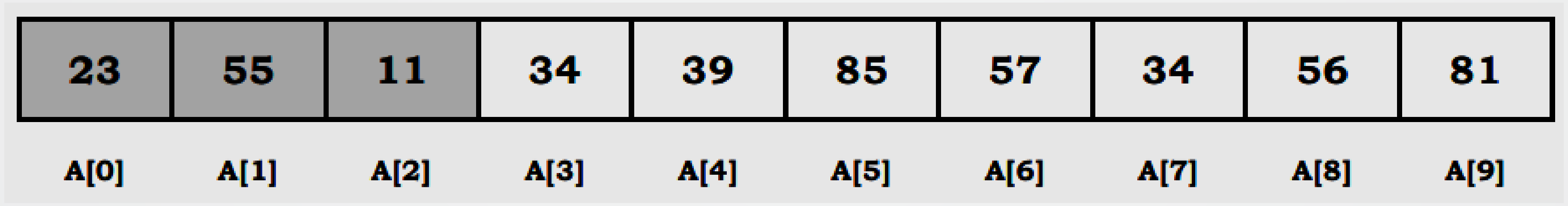
23	55	11	34	39	85	57	34	56	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Linear Search

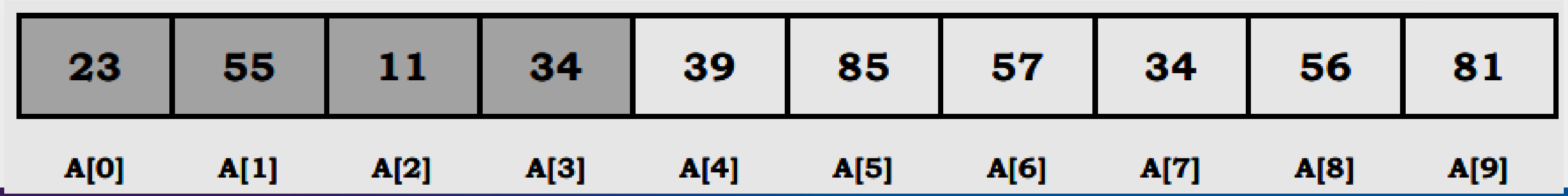
Step 2: It compares the x with next element located at index 1. Element is not matched then it moves to the next element. 23 55 11 34 39



Step 3: It compares the x with next element located at index 2. Element is not matched then it moves to the next element.

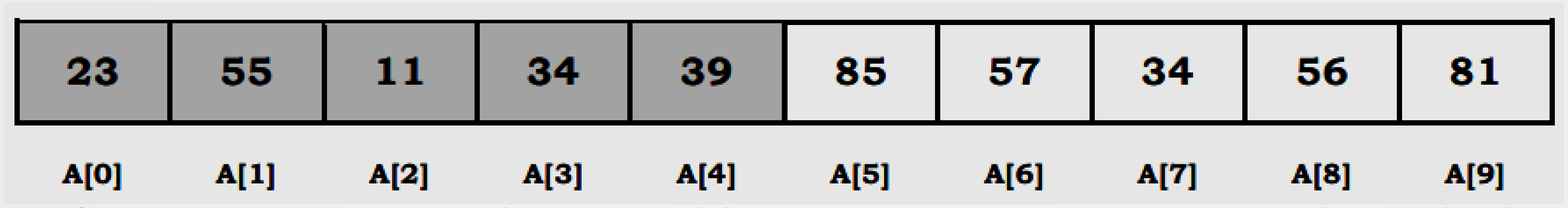


Step 4: It compares the x with next element located at index 3. Element is not matched then it moves to the next element.

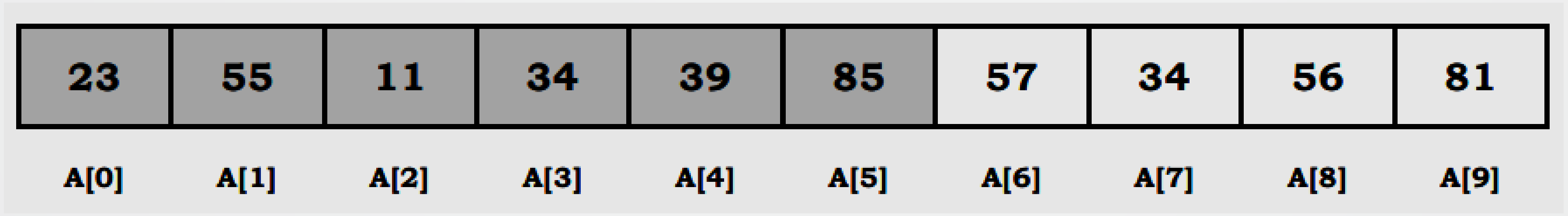


Linear Search

Step 5: It compares the x with next element located at index 4. Element is not matched then it moves to the next element.



Step 6: It compares the x with next element located at index 5. The desired element 85 is found at 5 location. Here the process is stopped and terminate the function.



Note: If the last element in the list is also doesn't match, then display "Element not found" and terminate the function.

Linear Search

Algorithm

Input

- An array `A` of `n` elements
- A value `key` to be searched

Output

- Index of `key` if found
- Otherwise, message **“Element not found”**

```
1. Start
2. Read the array size n
3. Read the array elements A[0] to A[n−1]
4. Read the search element key
5. Set i = 0
6. Repeat steps 7–8 while i < n
7. If A[i] == key, then
8. Print “Element found at position i”
9. Stop
10.Increment i by 1
11.If the loop ends and the element is not found,
12.Print “Element not found”
13.Stop
```

Linear Search

```
import java.util.Scanner;
class LinearSearch {
    static int linearSearch(int[] arr, int n, int key) {
        for (int i = 0; i < n; i++) {
            if (arr[i] == key) {
                return i;   // element found
            }
        }
        return -1;  // element not found
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of elements: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
```

```
 System.out.println("Enter array elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        System.out.print("Enter element to search: ");
        int key = sc.nextInt();
        int result = linearSearch(arr, n, key);
        if (result == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element found at
position: " + result);
    }
}
```

Linear Search

Time and Space Complexity of Linear Search Algorithm:

Linear search checks each element **one by one** until the key is found or the array ends.

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So, the best-case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N)

Auxiliary Space: O(1) as except for the variable to iterate through the list, no other variable is used.

Linear Search

Time and Space Complexity of Linear Search Algorithm:

Time Complexity Analysis

Case	Condition	Time Complexity
Best Case	Element found at first position	**Ω(1)**
Average Case	Element found in the middle	**Θ(n)**
Worst Case	Element at last position or not present	**O(n)**

Overall Time Complexity

`O(n)`

Linear Search

Applications of Linear Search Algorithm:

- **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with
- **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Linear Search

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

Linear Search

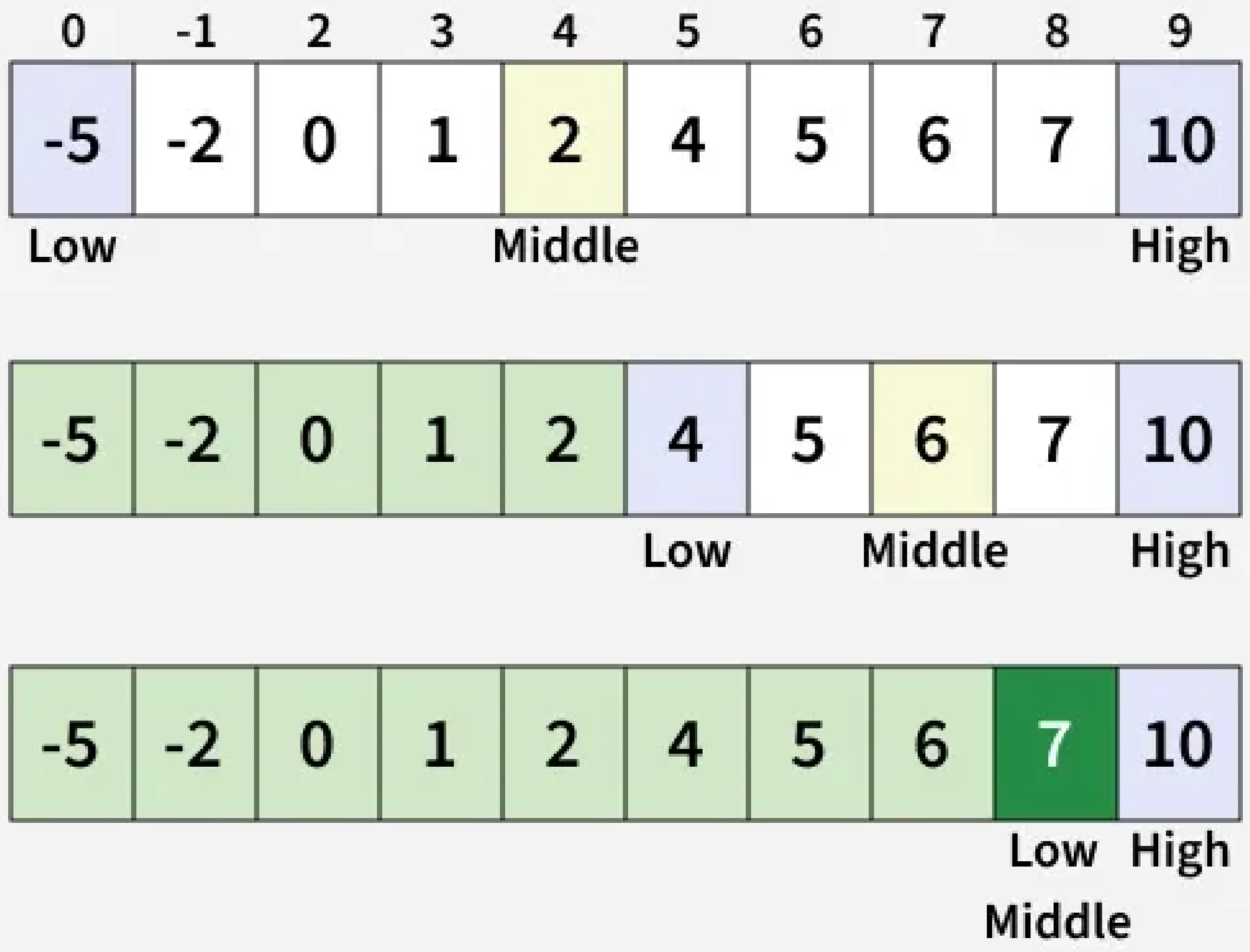
When to use Linear Search Algorithm?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

Binary Search

- Binary Search is a searching algorithm that operates on a sorted or monotonic search space, repeatedly dividing it into halves to find a target value or optimal answer in logarithmic time O(log N).
- It mainly compares the array's middle element first and if the middle element is same as input, then it returns.
- Otherwise, it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater).
- This algorithm is faster than linear search and takes O(Log n) time.

Binary Search



Searching for 7 : We repeatedly go to mid. If mid is smaller, then right half. Else left half

Binary Search

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

Binary Search

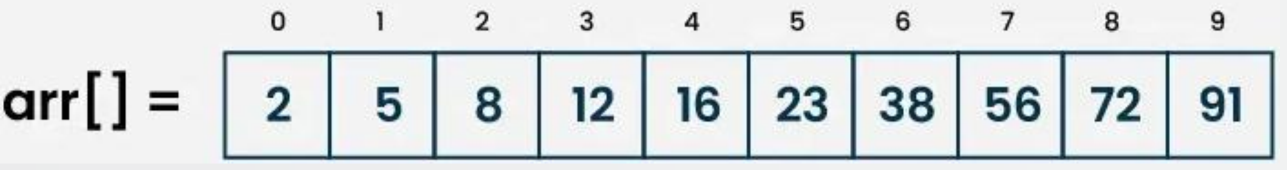
Step-by-step algorithm for Binary Search:

- Divide the search space into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
  - → If the **key** is smaller than the middle element, then the **left** side is used for next search.
  - → If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

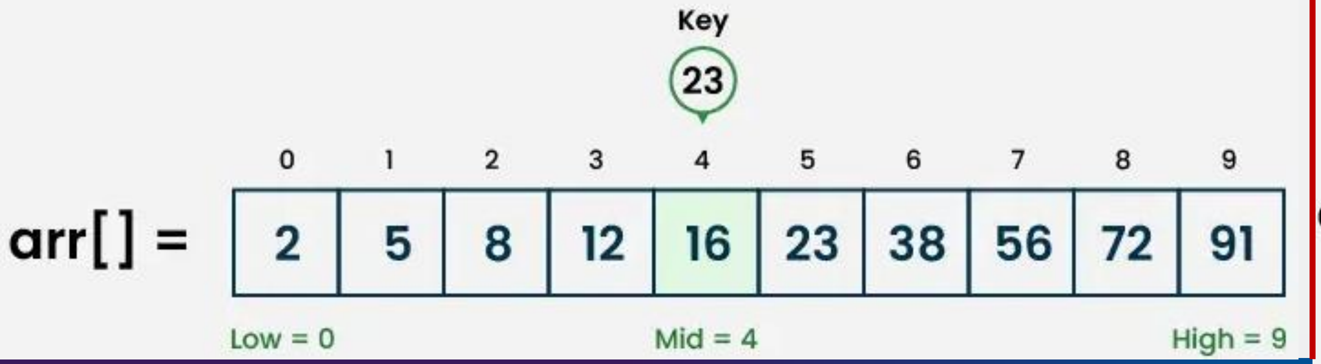
Binary Search

Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

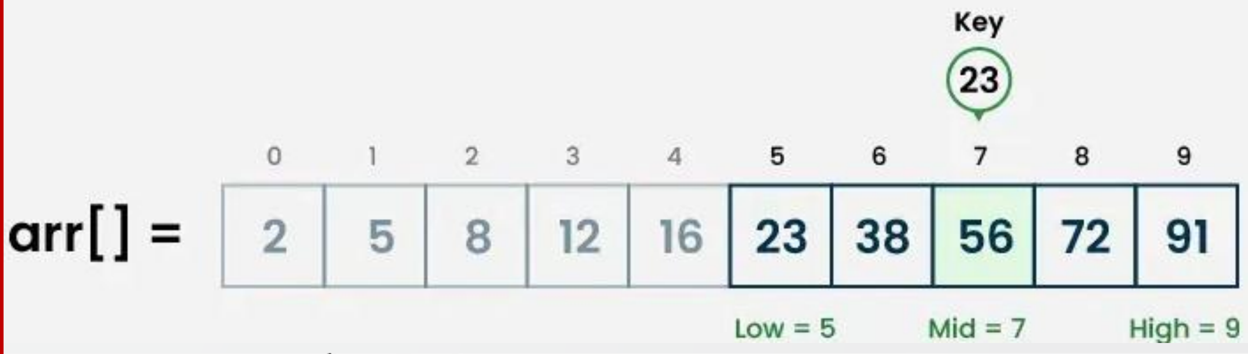
Initially | Find Key = 23 using Binary Search



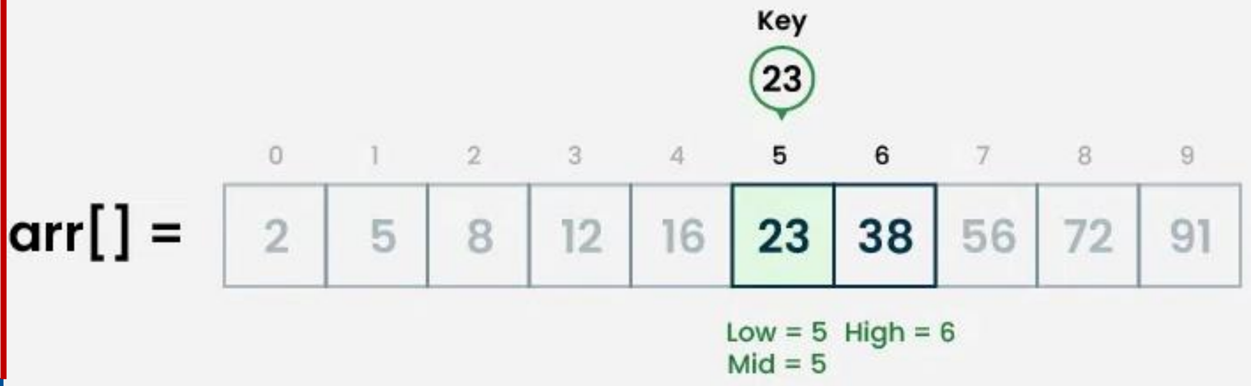
Step 1 | Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



Step 2 | Key is less than the current mid 56. The search space moves to the left.

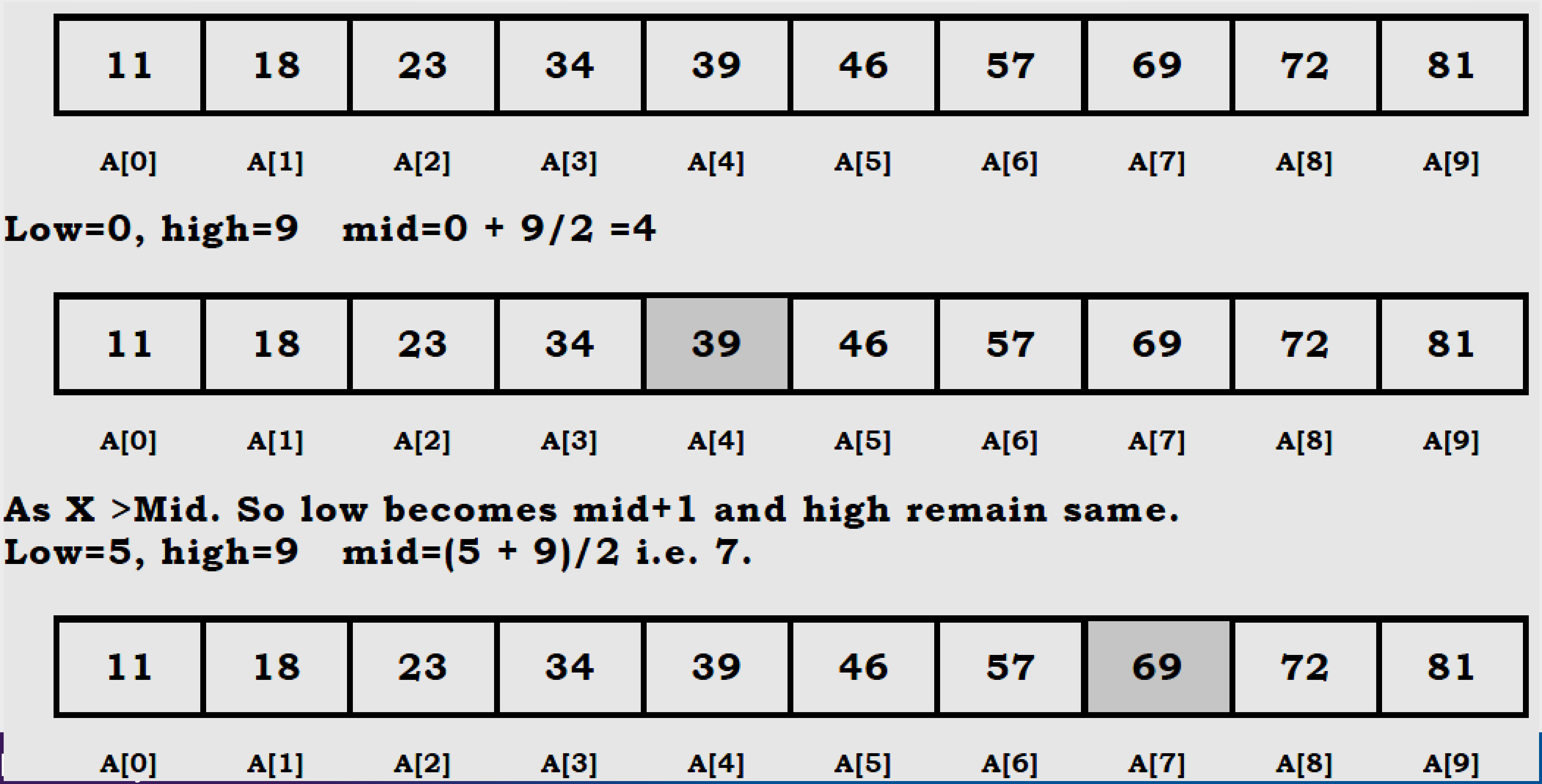


Step 3 | If the key matches the value of the mid element, the element is found and stop search.



Binary Search

Suppose we are given a sorted array of 10 elements. We are to search element 72. i.e. **X = 72.**



11	18	23	34	39	46	57	69	72	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

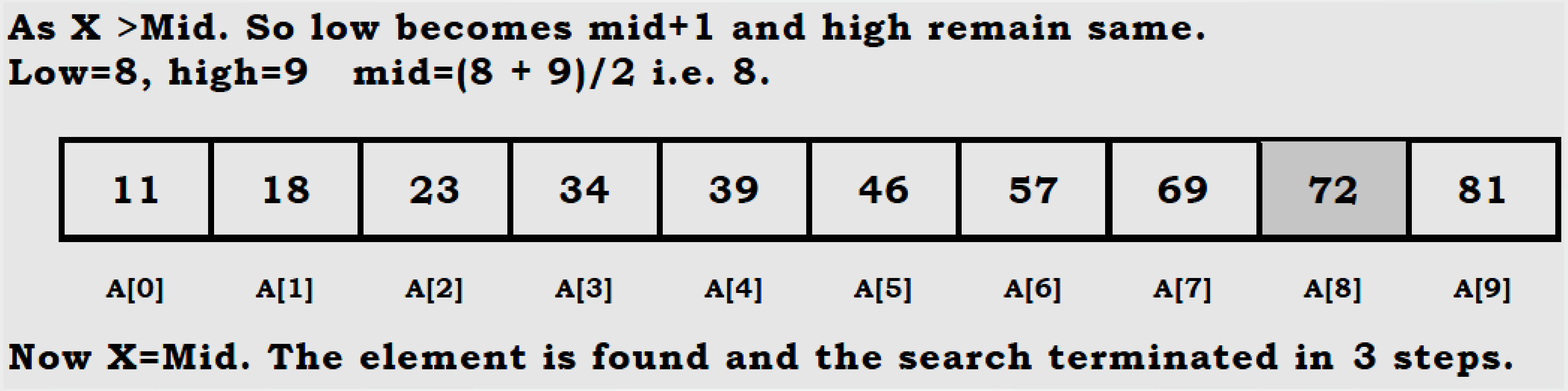
Low=0, high=9 mid=0 + 9/2 =4

11	18	23	34	39	46	57	69	72	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

As X >Mid. So low becomes mid+1 and high remain same.
Low=5, high=9 mid=(5 + 9)/2 i.e. 7.

11	18	23	34	39	46	57	69	72	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Binary Search



As X >Mid. So low becomes mid+1 and high remain same.
Low=8, high=9 mid=(8 + 9)/2 i.e. 8.

11	18	23	34	39	46	57	69	72	81
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Now X=Mid. The element is found and the search terminated in 3 steps.

Binary Search

Binary Search Procedure:

1. Find the **middle element**
2. Compare the middle element with the key
3. If equal → element found
4. If key is smaller → search **left half**
5. If key is larger → search **right half**
6. Repeat until found or search space becomes empty

Binary Search

The **Binary Search Algorithm** can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Binary Search

Iterative Algorithm: O(log n) Time and O(1) Space

Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.

Binary Search

Algorithm: Binary Search (**Iterative Algorithm**)

Input

- A **sorted array** `arr` of size `n`
- An element `x` to be searched

Output

- Index of `x` if found
- Otherwise, return `-1`

Steps

```
1.Start
2.Set low = 0
3.Set high = n - 1
4.Repeat steps 5–9 while low ≤ high
5.Calculate
mid = low + (high - low) / 2
6.If arr[mid] == x,
    • Return mid
7.Else if arr[mid] < x,
    • Set low = mid + 1
8.Else
    • Set high = mid - 1
9.End of loop
10.If the element is not found,
    • Return -1
11.Stop
```

Binary Search

```
class BinaryIterative {
    static int binarySearch(int arr[], int x) {
        int low = 0, high = arr.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            // Check if x is present at mid
            if (arr[mid] == x)
                return mid;
            // If x greater, ignore left half
            if (arr[mid] < x)
                low = mid + 1;
            // If x is smaller, ignore right half
            else
                high = mid - 1;
        }
 // If we reach here, then element was
        // not present
        return -1;
    }

    public static void main(String args[]) {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;
        int result = binarySearch(arr, x);
        if (result == -1)
            System.out.println(
                "Element is not present in array");
        else
            System.out.println("Element is present at "
                               + "index " + result);
    }
}
```

Output

```
Element is present at index 3
```

Binary Search

Recursive Algorithm: O(log n) Time and O(Log n) Space

Create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.

Binary Search

Algorithm: Binary Search **(Recursive Algorithm)**

Input

- A **sorted array** `arr`
- Lower index `low`
- Higher index `high`
- Search element `x`

Output

- Index of `x` if found
- Otherwise, return `-1`

Steps

```
1. Start
2. If high < low, then
       • Return −1 (element not found)
3. Calculate the middle index:
     mid = low + (high − low) / 2
4. If arr[mid] == x, then
     Return mid
5. Else if arr[mid] > x, then
   • Call binarySearch on the left subarray
     binarySearch(arr, low, mid − 1, x)
6. Else
   • Call binarySearch on the right subarray
     binarySearch(arr, mid + 1, high, x)
7. Repeat the above steps recursively
8. Stop
```

Binary Search

```
class BinaryRecursion {
// A recursive binary search function. It returns location
// of x in given array arr[low..high] is present, otherwise -1
    static int binarySearch(int arr[], int low, int high, int x)
{
        if (high >= low) {
            int mid = low + (high - low) / 2;
// If the element is present at the middle itself
            if (arr[mid] == x)
                return mid;
            // If element is smaller than mid, then
            // it can only be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, low, mid - 1, x);
// Else the element can only be present in right subarray
            return binarySearch(arr, mid + 1, high, x);
        }
// We reach here when element is not present in array
        return -1;    }
```

```
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int n = arr.length;
        int x = 10;
        int result = binarySearch(arr, 0, n - 1, x);
        if (result == -1)
            System.out.println(
                "Element is not present in array");
        else
            System.out.println(
                "Element is present at index "
+ result);
    }
}
```

Output

```
Element is present at index 3
```

Binary Search

Complexity Analysis

Binary search repeatedly **divides the sorted array into two halves** and searches in the appropriate half.

- **Time Complexity:**

  ➔ Best Case: O(1)

  ➔ Average Case: O(log N)

  ➔ Worst Case: O(log N)

- **Auxiliary Space:** O(1), If the recursive call stack is considered then the auxiliary space will be O(log N).

Binary Search

Complexity Analysis

- Time Complexity:

Case	Condition	Time Complexity
Best Case	Element found at middle	**Ω(1)**
Average Case	Element found after several divisions	**Θ(log n)**
Worst Case	Element found at last level or not present	**O(log n)**

Overall Time Complexity

`O(log n)`

Binary Search

Note:

Binary search requires a sorted array.

Binary Search

Linear vs Binary Search

Aspect	Linear Search	Binary Search
Array type	Sorted / Unsorted	Sorted only
Searching method	Sequential	Divide and conquer
Best case	O(1)	O(1)
Worst case	O(n)	O(log n)
Average case	Θ(n)	Θ(log n)
Efficiency	Less efficient	More efficient
Suitable for	Small or unsorted data	Large sorted data

Performance analysis evaluates algorithms based on time and space complexity; **linear search** has **O(n)** time complexity, while **binary search** has **O(log n)** time complexity.

Binary Search

Advantages

- **The binary search** is much more efficient than **the linear search**. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched.
- Binary search can have random access to the data but linear search only requires sequential access.

Disadvantage

- Binary search process only on sorted list.

Searching Techniques

MULTIPLE CHOICE QUESTIONS

1. Which of the following is not the required condition for binary search algorithm?

a) The list must be sorted

b) There should be the direct access to the middle element in any sub list

c) There must be mechanism to delete and/or insert elements in list.

d) Number values should only be present

2. The Average case occurs in linear search algorithm.

a) when item is somewhere in the middle of the array

b) when item is not the array at all

c) when item is the last element in the array

d) Item is the last element in the array or item is not there at all

Searching Techniques

<u>MULTIPLE CHOICE QUESTIONS</u>

1. Which of the following is not the required condition for binary search algorithm?

a) The list must be sorted

b) There should be the direct access to the middle element in any sub list

c) There must be mechanism to delete and/or insert elements in list.

d) Number values should only be present

2. The Average case occurs in linear search algorithm.

a) when item is somewhere in the middle of the array

b) when item is not the array at all

c) when item is the last element in the array

d) Item is the last element in the array or item is not there at all

Searching Techniques

MULTIPLE CHOICE QUESTIONS

3. Binary search algorithm cannot be applied to.

a) Sorted linked list
b) Sorted Binary tree
c) Sorted Linear Array
d) Pointer Array

4. Complexity of linear search algorithm is:

a) O(n)
b) O(log n)
c) O(n2)
d) O(n log n)

Searching Techniques

MULTIPLE CHOICE QUESTIONS

3. Binary search algorithm cannot be applied to.

a) Sorted linked list

b) Sorted Binary tree

c) Sorted Linear Array

d) Pointer Array

4. Complexity of linear search algorithm is:

a) O(n)

b) O(log n)

c) O(n2)

d) O(n log n)

Searching Techniques

MULTIPLE CHOICE QUESTIONS

5. Finding the location of a given item in a collection of items is called.

a) Discovering
b) Finding
c) Searching
d) Mining

6. What is the worst-case time complexity of linear search algorithm?

a) O(1)
b) O(n)
c) O(log n)
d) O(n^2)

Searching Techniques

MULTIPLE CHOICE QUESTIONS

5. Finding the location of a given item in a collection of items is called.

a) Discovering
b) Finding
c) Searching
d) Mining

6. What is the worst-case time complexity of linear search algorithm?

a) O(1)
b) O(n)
c) O(log n)
d) O(n^2)

Searching Techniques

FILL IN THE BLANKS

1. To compute the maximum ____________ comparisons are necessary and sufficient.
2. In Linear search, a ____________is made over all elements one by one.
3. The worst-case time complexity of linear search algorithm is____________________.
4. Binary search process only on _______________.
5. The time complexity of binary search algorithm in best case is ____________.

Searching Techniques

<u>FILL IN THE BLANKS</u>

1. To compute the maximum **n – 1** comparisons are necessary and sufficient.
2. In Linear search, a **Sequential search** is made over all elements one by one.
3. The worst-case time complexity of linear search algorithm is **O(log n)**.
4. Binary search process only on **Sorted list**.
5. The time complexity of binary search algorithm in best case is **O(n)**.

Searching Techniques

TRUE/FALSE

1. Finding the location of a given item in a collection of items is called searching.
2. The binary search is much more efficient than the linear search.
3. Linear search algorithm is used for large array list elements .
4. Complexity of binary search algorithm in worst and average case is O(n).
5. Comparisons are necessary and sufficient for computing both the minimum and the maximum is 3n-3/2.

Searching Techniques

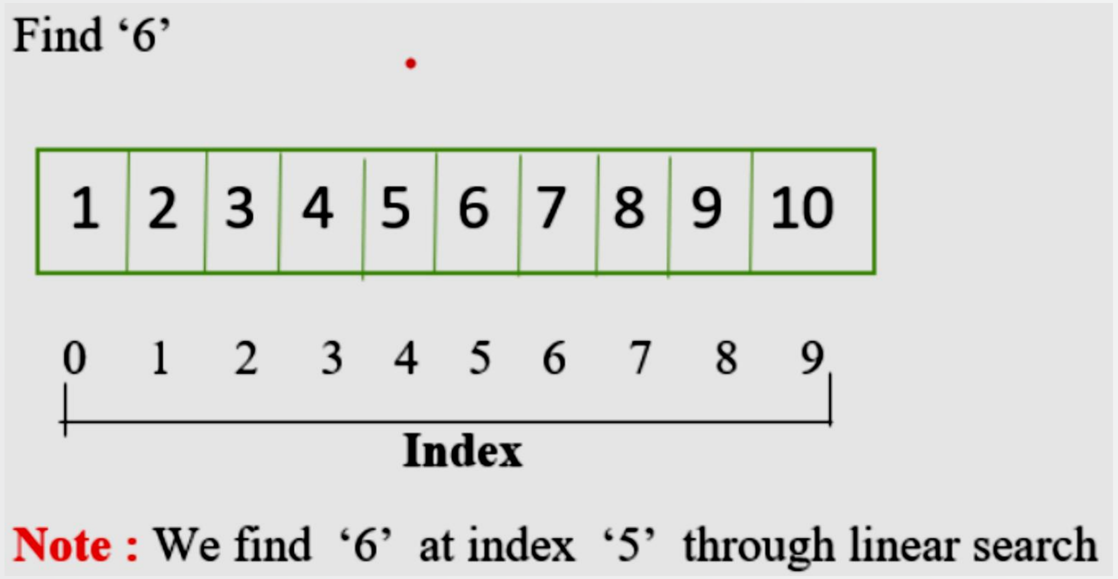
TRUE/FALSE

1. Finding the location of a given item in a collection of items is called searching. **(TRUE)**
2. The binary search is much more efficient than the linear search. **(TRUE)**
3. Linear search algorithm is used for large array list elements. **(FALSE)**
4. Complexity of binary search algorithm in worst and average case is O(n). **(FALSE)**
5. Comparisons are necessary and sufficient for computing both the minimum and the maximum is 3n-3/2. **(TRUE)**

Programs

2.1 Linear / Sequential Search

Linear search is defined as the searching algorithm where the list or data set is traversed from one end to find the desired value. Given an array arr[] of n elements, write a recursive function to search a given element x in arr[].



Linear search procedure:

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
2. If x matches with an element, return the index.
3. If x doesn’t match with any of the elements, return -1.

Programs

2.1 Linear / Sequential Search

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output: 6
Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 175;
Output: -1
Element x is not present in arr[].

Programs

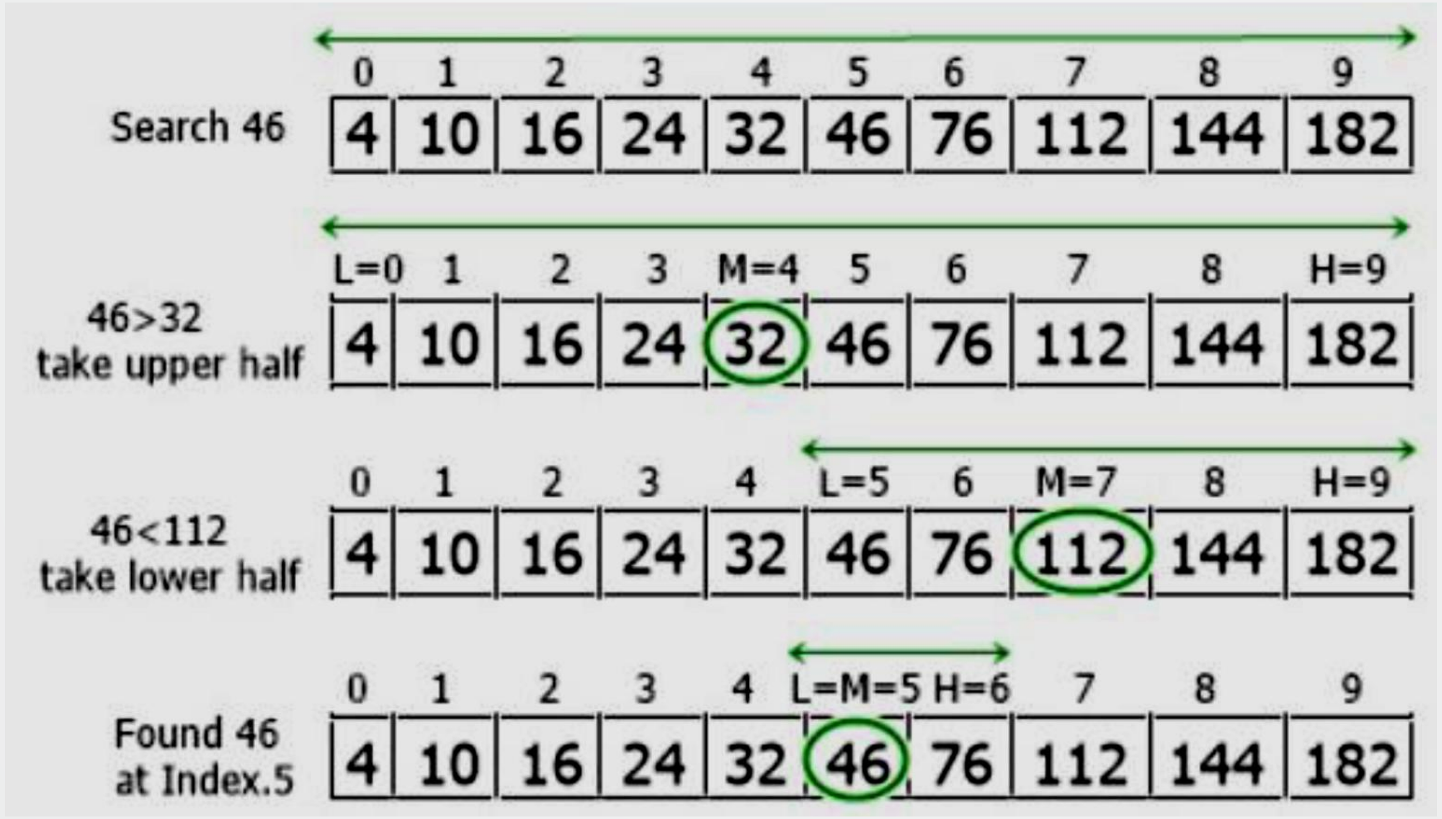
2.1 Linear / Sequential Search

```
class Main {
    // Recursive Linear Search Function
    static int linearSearch(int arr[], int x, int index)
{
        // Base case: element not found
        if (index == arr.length)
            return -1;
        // If element found
        if (arr[index] == x)
            return index;
        // Recursive call for next index
        return linearSearch(arr, x, index + 1);
    }
```

```
    public static void main(String[] args) {

        int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130,
                             170};
        int x = 110;

        int result = linearSearch(arr, x, 0);

        if (result == -1)
            System.out.println("Element x is not present
in arr[]");
        else
            System.out.println("Element x is present at
index " + result);
    }
}
```

Programs

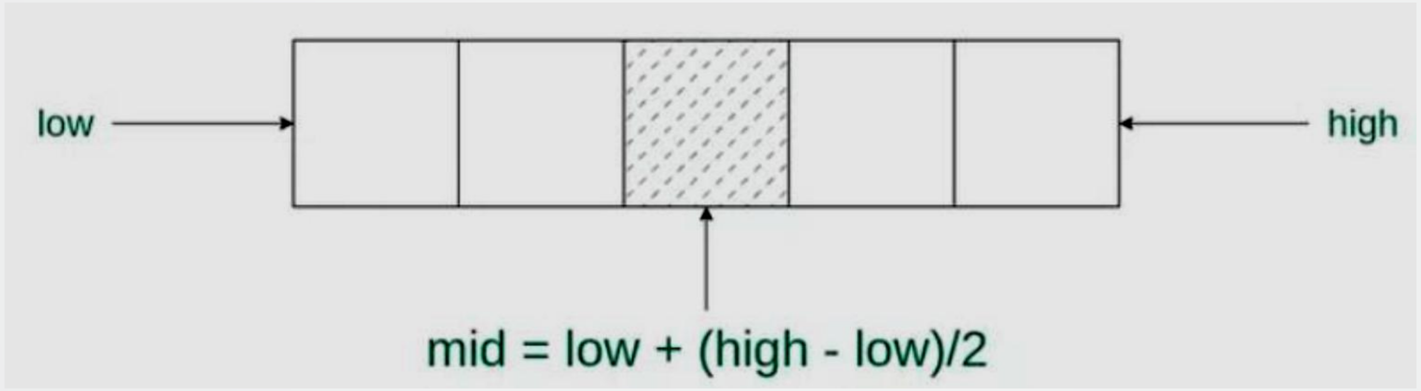
Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



Programs

Conditions for Binary Search algorithm:

1. The data structure must be sorted.
2. Access to any element of the data structure takes constant time.



Programs

Binary Search Procedure:

1. Divide the search space into two halves by finding the middle index “mid”.
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
   a) If the key is smaller than the middle element, then the left side is used for next search.
   b) If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

Programs

2.2 Binary Search

Input: arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Output: target = 23
Element 23 is present at index 5

Programs

2.2 Binary Search

Algorithm (Iterative Binary Search)

```
1. Set low = 0, high = n - 1
2. While low ≤ high:
   i. mid = (low + high) / 2
   ii. If arr[mid] == key, return mid
   iii. If arr[mid] < key, set low = mid + 1
   iv. Else set high = mid - 1
3. If element not found, return -1
```

```java
class BinarySearch {
    static int binarySearch(int[] arr, int key) {
        int low = 0;
        int high = arr.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            // Element found
            if (arr[mid] == key)
                return mid;
            // Search right half
            if (arr[mid] < key)
                low = mid + 1;
            // Search left half
            else
                high = mid - 1;
        }
        // Element not found
        return -1;
    }

    public static void main(String[] args) {

        int[] arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int key = 23;

        int result = binarySearch(arr, key);

        if (result == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element found at
index " + result);
    }
}
```