



Searching

Objectives

After completing this lesson, you should be able to:

- Searching
- Linear Searches
- Binary Search, Exponential Search



Course Roadmap

Data Structures

▶ Lesson 1: Introduction to Data Structures

▶ Lesson 2: Lists and Stacks

▶ **Lesson 3: Searching**

▶ Lesson 4: Trees and Queues

▶ Lesson 5: Sorting

← You are here!

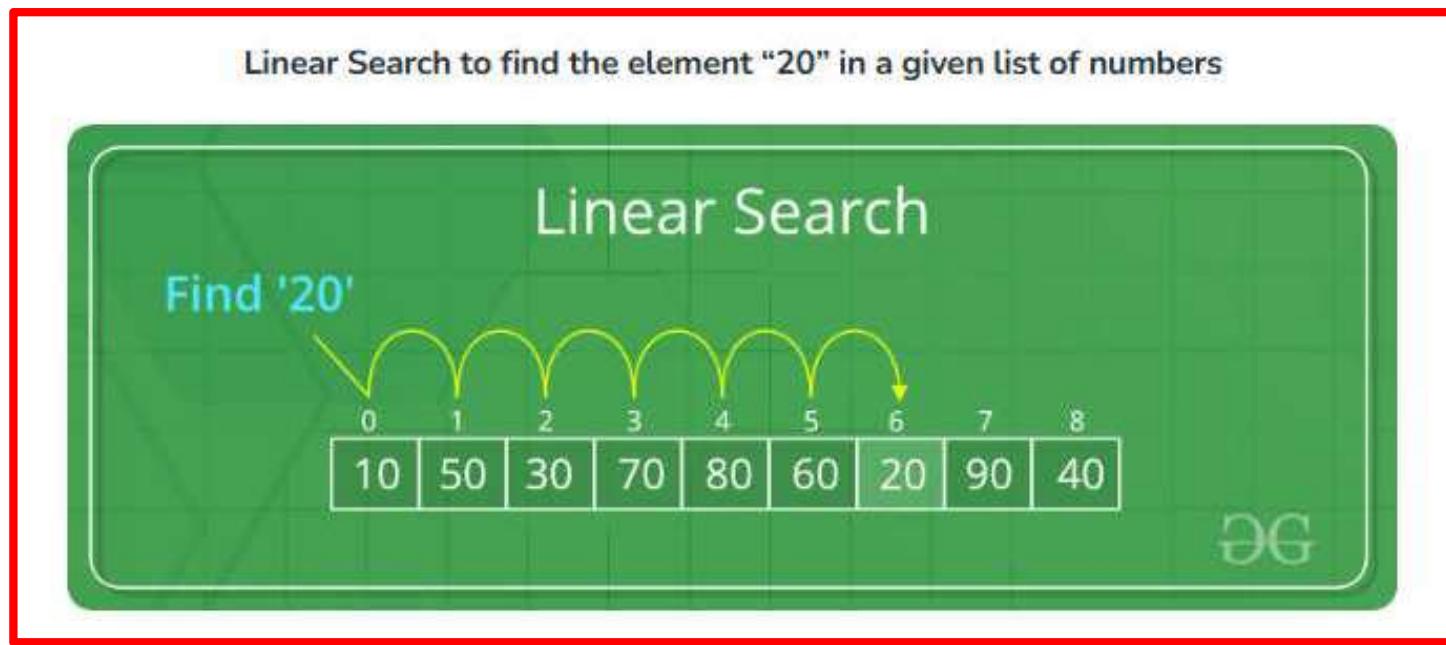


Searching

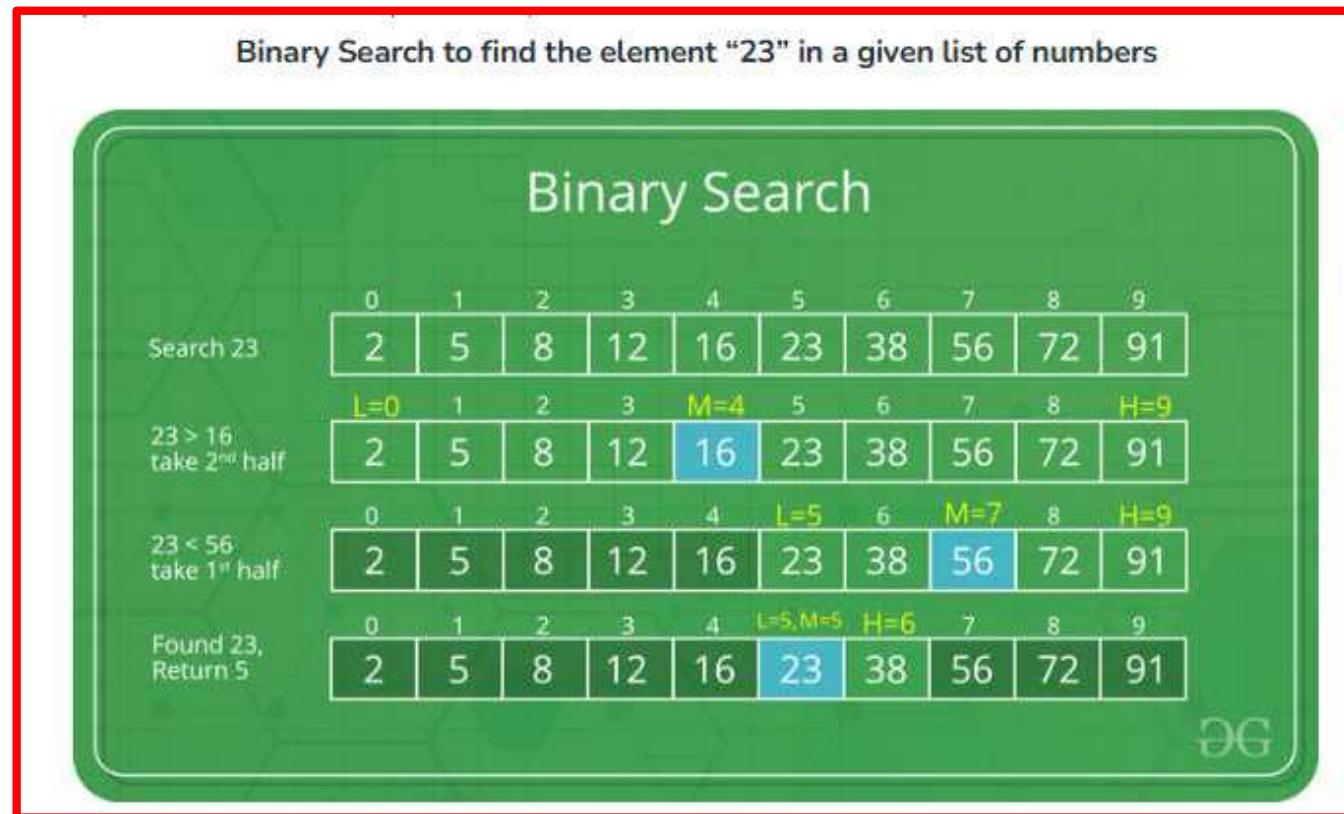
What is Searching Algorithm?

- *Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.*

- Based on the type of search operation, these algorithms are generally classified into two categories:
 1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: [Linear Search](#).



2. Interval Search: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: [Binary Search](#).

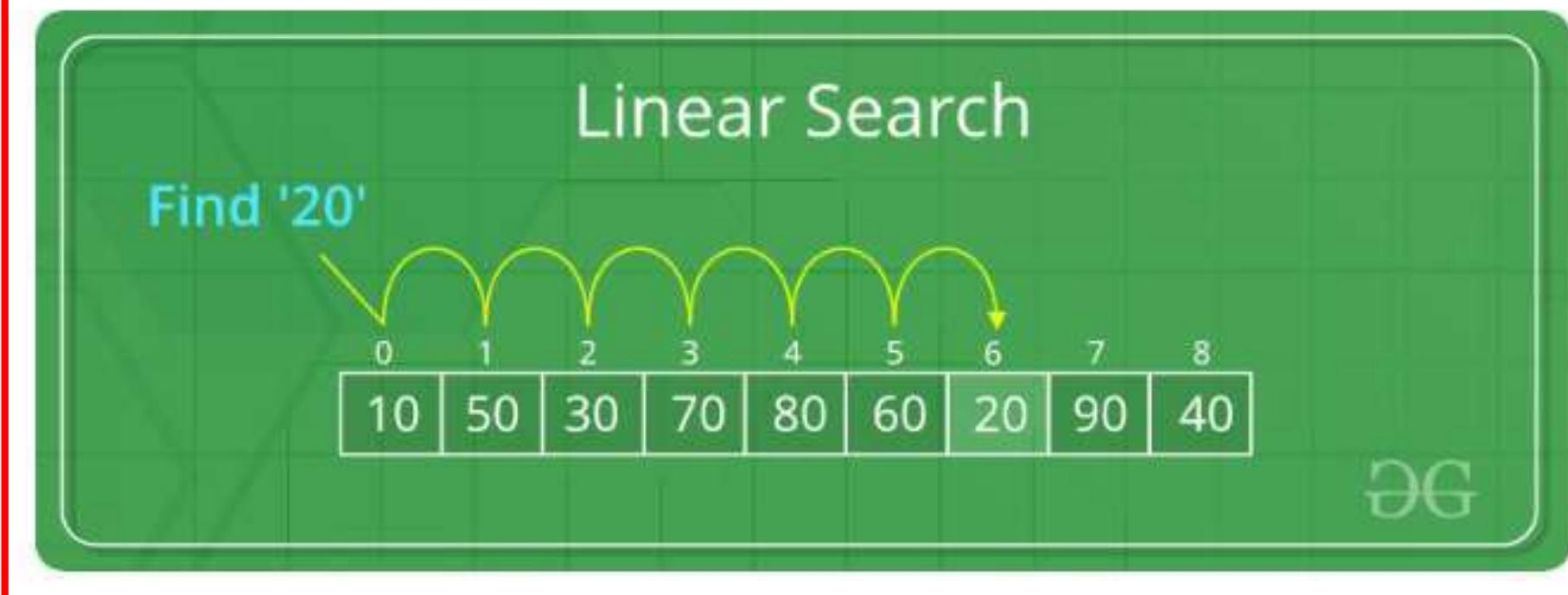




Linear Searching

Linear Search Algorithm

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



Pseudocode :

```
PROCEDURE LINEAR_SEARCH (LIST, VALUE)
    FOR EACH ITEM IN THE LIST
        IF SAME ITEM == VALUE
            RETURN THE ITEM'S LOCATION
        END IF
    END FOR
END PROCEDURE
```

How Linear Search Works?

Step 1: First, read the search element (Target element) in the array.

Step 2: Set an integer $i = 0$ and repeat steps 3 to 4 till i reaches the end of the array.

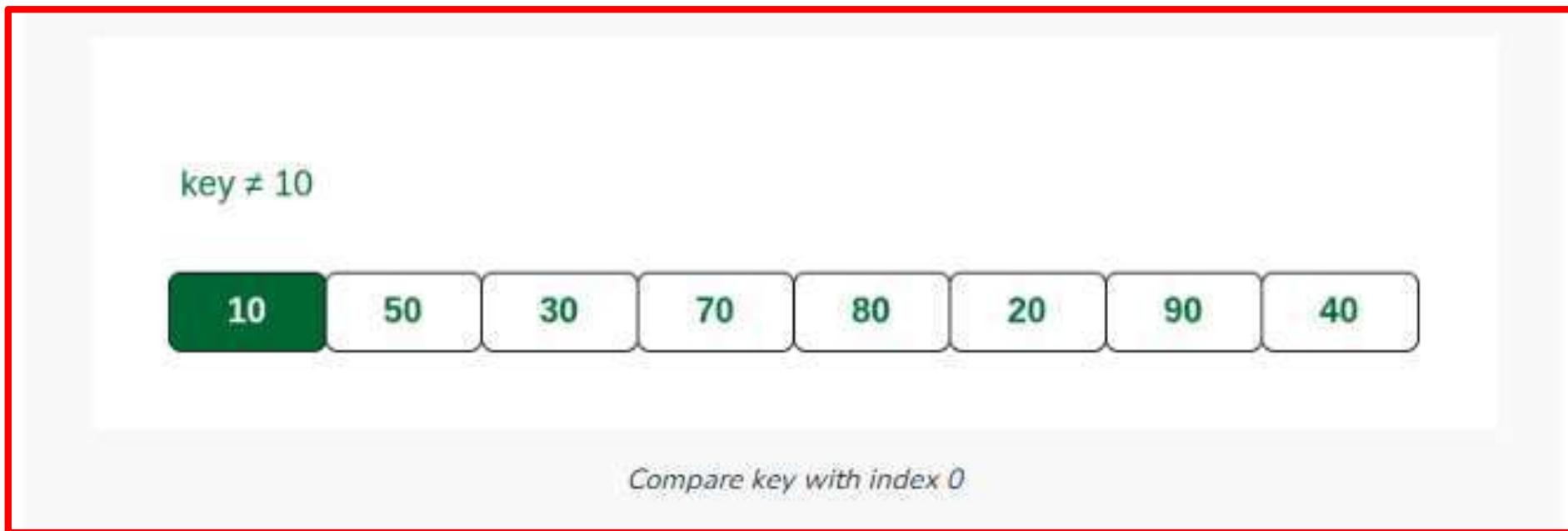
Step 3: Match the key with $\text{arr}[i]$.

Step 4: If the key matches, return the index. Otherwise, increment i by 1.

Illustration of Linear Search:

Consider the array $\text{arr}[] = \{10, 50, 30, 70, 80, 20, 90, 40\}$ and key = 20

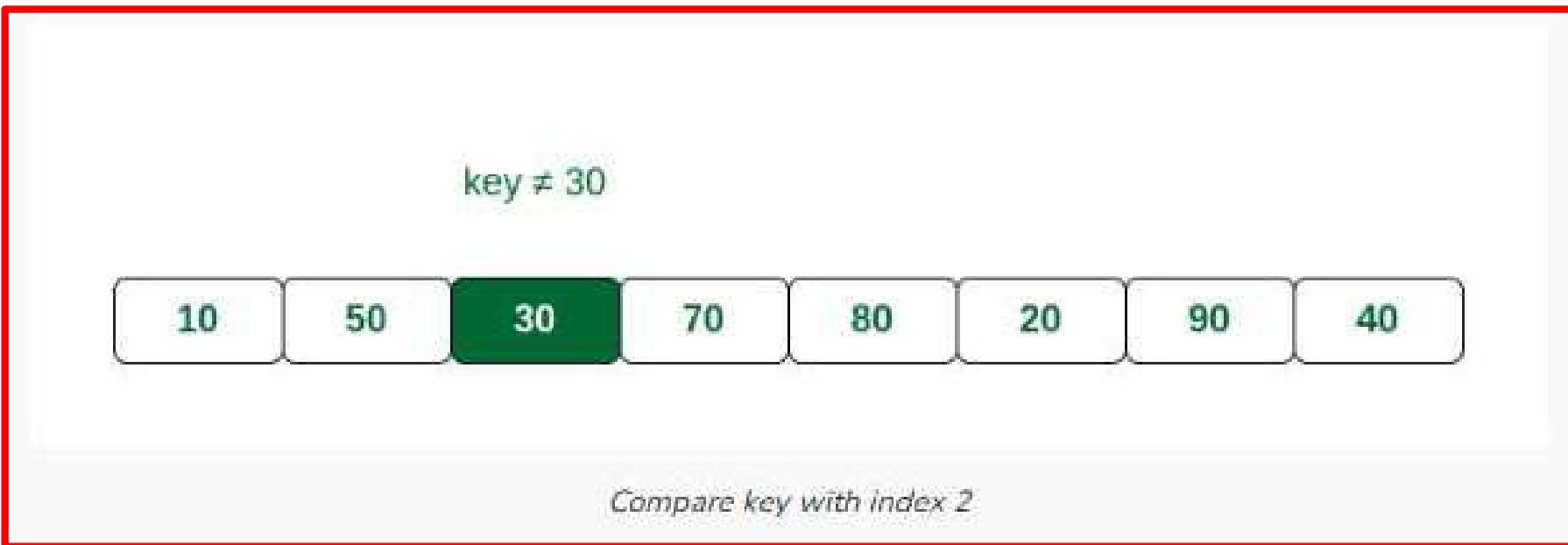
Step 1: Set $i = 0$ and check key with $\text{arr}[0]$.



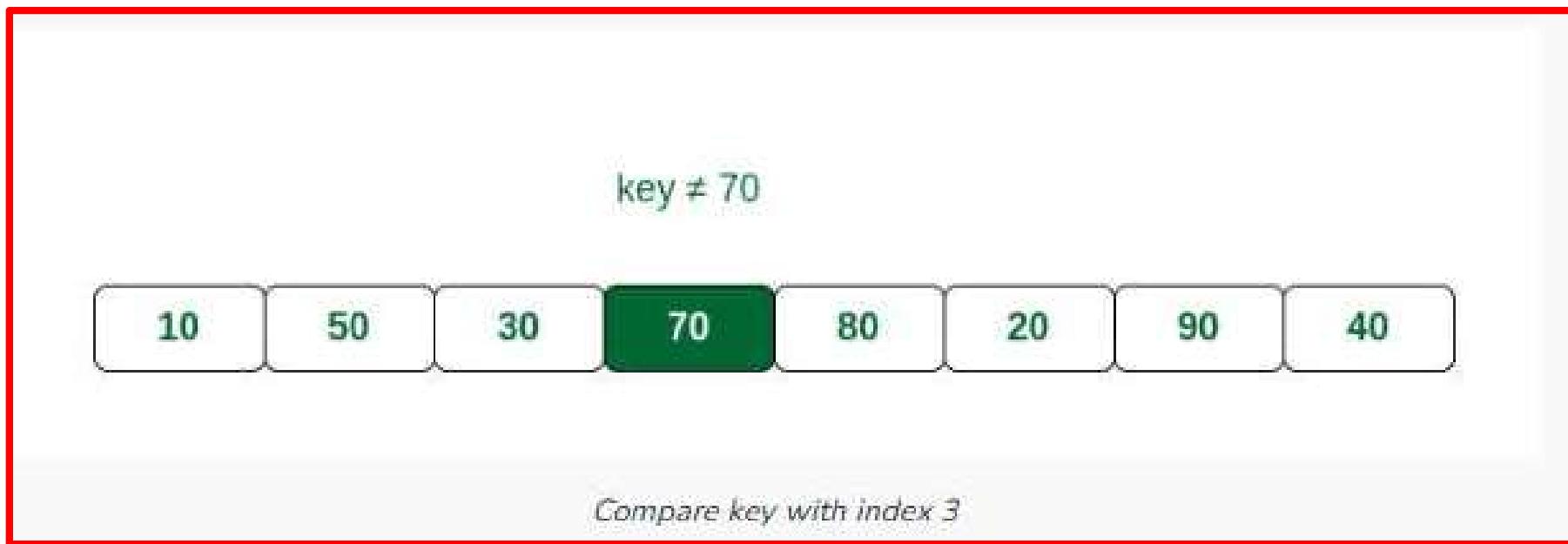
Step 2: key and arr[0] are not the same. So make i = 1 and match key with arr[1].



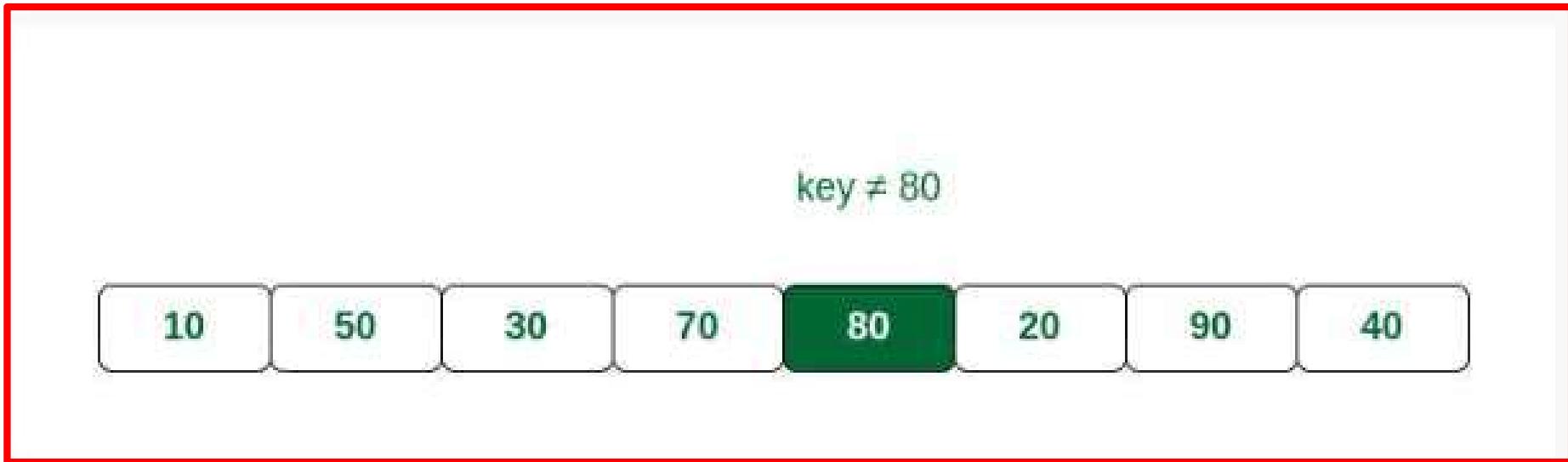
Step 3: arr[1] and key are different. Increment i and compare key with arr[2].



Step 4: arr[2] is not the same with key. Increment i and compare key with arr[3].



Step 5: key and arr[3] are different. Make i = 4 and compare key with arr[4].



Step 6: key and arr[4] are not same. Make i = 5 and match key with arr[5].



- **Result**
- *We can see here that key is present at index 5.*

Complexity Analysis of Linear Search:

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:** $O(N)$

Example

```
package training.iqgateway;
class LinearSearchExample {
    public static int search(int arr[], int x)
    {
        int N = arr.length;
        for (int i = 0; i < N; i++) {
            if (arr[i] == x)
                return i;
        }
        return -1;
    }
}
```

Example

```
// Driver code
public static void main(String args[])
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;

    // Function call
    int result = search(arr, x);
    if (result == -1)
        System.out.print("Element is not present in array");
    else
        System.out.print("Element is present at index " + result);
}
```

Recursive Approach for Linear Search:

- We can also utilize linear search using a recursive function. In this case, the iteration is done using a recursion.
- Follow the given steps to solve the problem:
 1. If the size of the array is zero then, return -1, representing that the element is not found. This can also be treated as the base condition of a recursion call.
 2. Otherwise, check if the element at the current index in the array is equal to the key or not i.e, $\text{arr}[\text{size} - 1] == \text{key}$
 3. If equal, then return the index of the found key.

Example

```
package training.iqgateway;

class RecursiveLinearSearch {
    static int arr[] = { 5, 15, 6, 9, 4 };

    // Recursive Method to search key in the array
    static int linearsearch(int arr[], int size, int key)
    {
        if (size == 0) {
            return -1;
        }
        else if (arr[size - 1] == key) {

            // Return the index of found key.
            return size - 1;
        }
        else {
            return linearsearch(arr, size - 1, key);
        }
    }
}
```

Example

```
// Driver method
public static void main(String[] args)
{
    int key = 4;

    // Function call to find key
    int index = linearsearch(arr, arr.length, key);
    if (index != -1)
        System.out.println( "The element " + key + " is found at "
            + index + " index of the given array.");

    else
        System.out.println("The element " + key + " is not found.");
}
```

Advantages of Linear Search:

1. Linear search is simple to implement and easy to understand.
2. Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
3. Does not require any additional memory.
4. It is a well suited algorithm for small datasets.

Drawbacks of Linear Search:

1. Linear search has a time complexity of $O(n)$, which in turn makes it slow for large datasets.
2. Not suitable for large arrays.
3. Linear search can be less efficient than other algorithms, such as hash tables.

When to use Linear Search?

1. When we are dealing with a small dataset.
2. When you need to find an exact value.
3. When you are searching a dataset stored in contiguous memory.
4. When you want to implement a simple algorithm.



Binary Search

- **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

Conditions for when to apply Binary Search in a Data Structure:

- To apply binary search in any data structure, the data structure must maintain the following properties:
 1. The data structure must be sorted.
 2. Access to any element of the data structure takes constant time.
- Consider an array `arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}`, and the `target = 23`.

How does Binary Search work?

First Step:

- a) Initially the search space is from 0 to 9.
- b) Let's denote the boundary by L and H where $L = 0$ and $H = 9$ initially.
- c) Now mid of this search space is $M = 4$.
- d) So compare **target** with **arr[M]**.

Second Step:

- a) As $\text{arr}[4]$ is less than **target**, switch the search space to the right of 16, i.e., [5, 9].
- b) Now **L = 5**, **H = 9** and **M** becomes **7**.
- c) Compare target with **arr[M]**.

Third Step:

- a) *arr[7] is greater than target.*
- b) *Shift the search space to the left of M, i.e., [5, 6].*
- c) *So, now L = 5, H = 6 and M = 6.*
- d) *Compare arr[M] with target.*
- e) *Here arr[M] and target are the same.*

So, we have found the target.

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 < 56 take 1 st half	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5,M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

36

When to use Binary Search?

- When searching a large dataset as it has a time complexity of $O(\log n)$, which means that it is much faster than linear search.
- When the dataset is sorted.
- When data is stored in contiguous memory.
- Data does not have a complex structure or relationships.

How to Implement Binary Search?

The basic steps to perform Binary Search are:

1. Set the low index to the first element of the array and the high index to the last element.
2. Set the middle index to the average of the low and high indices.
 - I. If the element at the middle index is the target element, return the middle index.
 - II. Otherwise, based on the value of the key to be found and the value of the middle element, decide the next search space.
 - a) If the target is less than the element at the middle index, set the high index to **middle index – 1**.
 - b) If the target is greater than the element at the middle index, set the low index to **middle index + 1**.
3. Perform step 2 repeatedly until the target element is found or the search space is exhausted.

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

1. Iterative Binary Search Algorithm
2. Recursive Binary Search Algorithm

Iterative Binary Search Algorithm:

Pseudocode of Iterative Binary Search Algorithm:

```
binarySearch(arr, x, low, high)
repeat till low = high
    mid = (low + high)/2
    if (x = arr[mid])
        return mid
    else if (x > arr[mid])
        low = mid + 1
    else
        high = mid - 1
```

Example

```
package training.iqgateway;
// Java implementation of iterative Binary Search
class BinarySearch {
    // Returns index of x if it is present in arr[],
    // else return -1
    int binarySearch(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;
            else
                r = m - 1;
        }

        // if we reach here, then element was
        // not present
        return -1;
    }
}
```

Example

```
// Driver method to test above
public static void main(String args[])
{
    BinarySearch ob = new BinarySearch();
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = arr.length;
    int x = 10;
    int result = ob.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);
}
```

Recursive Binary Search Algorithm:

Pseudocode of Recursive Binary Search Algorithm:

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x = arr[mid]
            return mid
        else if x > arr[mid]
            return binarySearch(arr, x, mid + 1, high)
        else
            return binarySearch(arr, x, low, mid - 1)
```

Example

```
package training.iqgateway;
class RecursiveBinarySearch {

    // Returns index of x if it is present in arr[l..
    // r], else return -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l) {
            int mid = l + (r - l) / 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, l, mid - 1, x);
        }
    }
}
```

```
// Else the element can only be present
// in right subarray
return binarySearch(arr, mid + 1, r, x);

}

// We reach here when element is not present
// in array
return -1;
}
```

Example

```
// Driver method to test above
public static void main(String args[])
{
    RecursiveBinarySearch ob = new RecursiveBinarySearch();
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = arr.length;
    int x = 10;
    int result = ob.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);
}
```

Advantages of Binary Search:

1. Binary search is faster than linear search, especially for large arrays. As the size of the array increases, the time it takes to perform a linear search increases linearly, while the time it takes to perform a binary search increases logarithmically.
2. Binary search is more efficient than other searching algorithms that have a similar time complexity, such as Exponential search.
3. Binary search is relatively simple to implement and easy to understand, making it a good choice for many applications.
4. Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.
5. Binary search can be used as a building block for more complex algorithms, such as those used in computer graphics and machine learning.

Drawbacks of Binary Search:

1. We require the array to be sorted. If the array is not sorted, we must first sort it before performing the search. This adds an additional $O(N * \log N)$ time complexity for the sorting step, which makes it irrelevant to use binary search.
2. Binary search requires that the data structure being searched be stored in contiguous memory locations. This can be a problem if the data structure is too large to fit in memory, or if it is stored on external memory such as a hard drive or in the cloud.
3. Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered. This can be a problem if the elements of the array are not naturally ordered, or if the ordering is not well-defined.
4. Binary search can be less efficient than other algorithms, such as hash tables, for searching very large datasets that do not fit in memory.

Applications of Binary Search:

1. Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
2. Commonly used in Competitive Programming.
3. Can be used for searching in computer graphics. Binary search can be used as a building block for more complex algorithms used in computer graphics, such as algorithms for ray tracing or texture mapping.
4. Can be used for searching a database. Binary search can be used to efficiently search a database of records, such as a customer database or a product catalog.



Exponential Search

- Exponential Search is a searching technique where range is first bound in between which the Search value exists.
- Then on that particular Range, the BINARY Search technique is applied.
- For Exponential Search
 1. The List Should be in Sorted Order
 1. First, the subarray size is taken as 1 then the last element of that Particular range is Compared with the Searched Value, then the Subarray size is taken as 2, then 4 and So On.
 2. When the last element of the Sub array is Greater than the Searched Element then the Process is Stopped.

3. Now, we can say the Searched element is present between $i/2$ and i (if the last index of Subarray greater than the Searched Element is i).
4. Then Range is $i/2$ and i becoz in the previous iteration the last element was not Greater than the Searched Element
2. Now Apply Binary Search Logic and find the Element.

Exponential search involves two steps:

1. Find range where element is present
2. Do Binary Search in above found range.

Let's say we're looking for a number at the 17th position in a very long array x . By testing the boundaries with the Exponential Search, we locate the portion of x that contains the number in 5 steps:

Step 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	31	32	...
Step 2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	31	32	...
Step 3	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	31	32	...
Step 4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	31	32	...
Step 5	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	31	32	...

How to find the range where element may be present?

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

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

Illustration

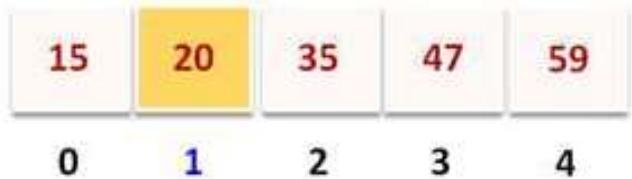
EXAMPLE:

Suppose the list is:

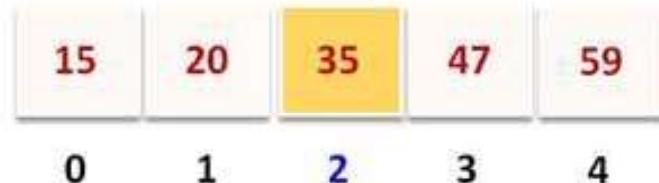
15	20	35	47	59
0	1	2	3	4

Let the **searched element** be **47**.

First the range is set to 1. Now the value of **index 1** (i.e. 20) will be compared with the **searched element (47)**.



As the searched element is not lesser than the value of **index 1** then the next range will be set to $2 * 1 = 2$. Now again the value of **index 2** (i.e. 35) will be compared with the **searched element (47)**.



As 35 is lesser than 47 then the bound will be set to $2 * 2 = 4$. The searched element 47 will now be compared with the value at index 4 i.e. 59.



Now the element is not less than 47. So, the increment of the bound will be stopped. Now, the binary search will be performed between indexes 2 and 4.

The **lower index is 2** and **the upper index is 4**.

So, the middle index will be calculated as:

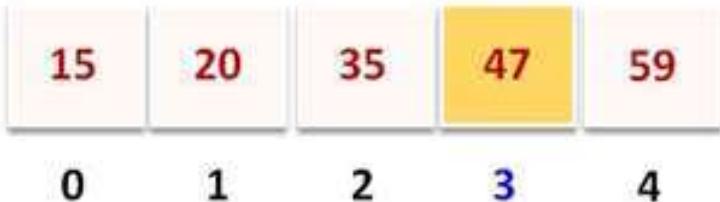
$$\text{mid} = (\text{lower index} + \text{upper index})/2$$

$$= (2 + 4)/2$$

$$= 6/2$$

$$= 3$$

Now the **searched value (47)** will be compared with the **value of index 3(47)**.



As their values are matched then the **position of the element will be printed**.

Example

```
package training.iqgateway;
import java.util.Arrays;

class ExponentialSearch
{
    // Returns position of
    // first occurrence of
    // x in array
    static int exponentialSearch(int arr[],
                                 int n, int x)
    {
        // If x is present at first location itself
        if (arr[0] == x)
            return 0;

        // Find range for binary search by
        // repeated doubling
        int i = 1;
        while (i < n && arr[i] <= x)
            i = i*2;

        // Call binary search for the found range.
        return Arrays.binarySearch(arr, i/2,
                                  Math.min(i, n-1), x);
    }
}
```

Example

```
// Driver code
public static void main(String args[])
{
    int arr[] = {2, 3, 4, 10, 40};
    int x = 10;
    int result = exponentialSearch(arr, arr.length, x);

    System.out.println((result < 0) ? "Element is not present in array" :
        "Element is present at index " + result);
}
```

ADVANTAGES:

- 1.** This searching technique is more efficient than binary search if the element is present closer to the first element.

- 2.** In some cases, the exponential search is faster than the binary search.

DISADVANTAGES:

- 1.** The list should be sorted to perform the exponential search, if the list is unsorted, it will give a wrong output.

Summary

In this lesson, you should have learned that:

- Searching
- Linear Searches
- Binary Search, Exponential Search

