# Searching Algorithms

**Searching** algorithms are used to find a specific element within a dataset

Two basic types: **Linear Search** (sequential) and **Binary Search** (divide-and-conquer)

Linear Search - for unordered lists
Binary Search - for ordered lists

## Key Terms in Searching Algorithms

### Successful Search

- Occurs when the **target element is found** in the data set.
- The search algorithm returns the **position/index** of the target.

Example: Searching for `23` in `[10, 15, 23, 40]` → Found at index 2.

### Unsuccessful Search

- Happens when the **target element is not present** in the data set.
  The search algorithm returns **"not found"** or a **special value** (e.g., `-1`).

Example: Searching for `99` in `[10, 15, 23, 40]` → Not found.

### Retrieval

- The process of **accessing or obtaining** data once it's located through search.
- Involves **fetching** the actual value or associated information.
- Often follows a successful search operation.

### Internal Searching

- The search is performed **in main memory (RAM)**.
  Suitable for small to medium-sized data sets.

Examples: Searching in arrays, linked lists, or in-memory databases.

### External Searching

- The search is done **on data stored in external storage** (e.g., hard disk, database).

- Used when data is **too large to fit in memory**.

  Examples: Searching in files, databases, or large indexes.

## LINEAR SEARCH

### Definition:

- A simple search technique that checks every element one by one until the target is found or end of list is reached.

### How It Works:

- Start from index `0`, compare each element with the target.
- If match is found, return index.
- If end is reached without match, return "not found".

### Use Cases:

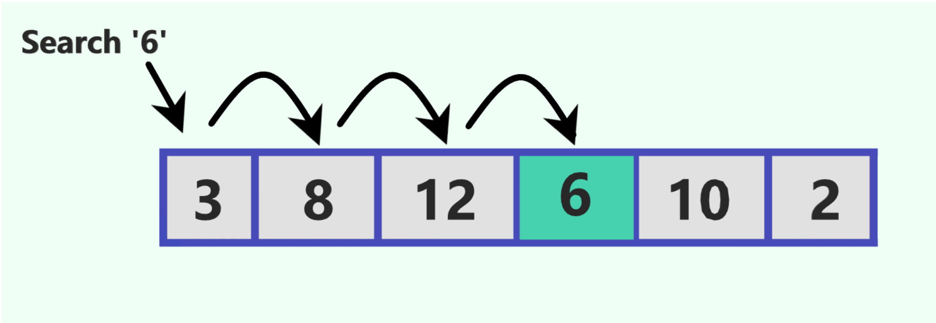
- Works on **unsorted or unordered lists**.
- Useful when the list is small or sorting is not feasible.

### Example:

Array: `[3, 8, 12, 6, 10, 2]`

Target: `6`

Checked: 3 → 8 → 12 → **6** - Match at index 3




**Time Complexity:**

- Best Case: O(1) (if element is at beginning)
- Average Case: O(n)
- Worst Case: O(n)

**Real-World Applications:**

- Searching a name in an **unsorted contact list**
- Finding a file in an **unsorted folder**

**Pseudocode:**

```
for i from 0 to n-1:
    if arr[i] == target:
        return i
return -1
```

**Linear Search – Java Code**

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;  // Successful search
            }
        }
```

```
        return -1;  // Unsuccessful search
    }

    public static void main(String[] args) {
        int[] data = {12, 45, 67, 23, 89, 34};
        int target = 23;

        int result = linearSearch(data, target);
        if (result != -1)
            System.out.println("Element found at index: " + result);
        else
            System.out.println("Element not found.");
    }
}
```

**Output:**

```
Element found at index: 3
```

### Questions:

**Q1.** Write a Java program to perform a linear search on an integer array. The array may contain duplicate elements. Your program should find and display the **first and last occurrence** of the target value.

> **Input Example**:
> Array: {3, 5, 7, 5, 9, 5, 2}
> Target: 5
>
> **Expected Output**:
> First occurrence at index 1
> Last occurrence at index 5

**Q2.** You are given a list of student names in a String array. Write a linear search function to **search for a name, ignoring case sensitivity** (e.g., "Alice" and "alice" should match).

> **Input Example**:
> Names: {"John", "Alice", "bob", "Diana"}
> Target: "ALICE"
>
> **Expected Output**:
> Name found at index 1

<u>**BINARY SEARCH**</u>

## Definition:

- Efficient search algorithm that works only on **sorted arrays** by repeatedly dividing the search interval in half.

## How It Works:

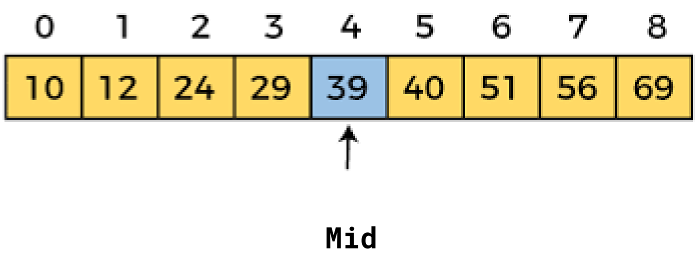
1. Find middle of the list.
2. If middle = target → return index.
3. If target < middle → search in left half.
4. If target > middle → search in right half.
5. Repeat until found or interval is empty.

## Use Cases:

- **Sorted data structures** (arrays, lists).
- Optimized for large datasets where linear search is inefficient.

## Example 1:

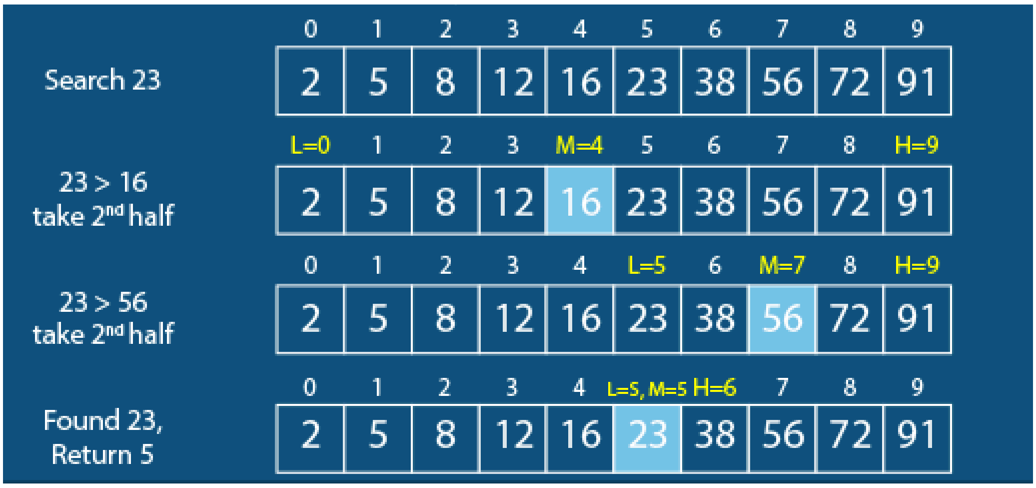
- Sorted Array: [10, 12, 24, 29, 39, 40, 51, 56, 59]
- Target: 39
- Steps: mid = 4 → arr[4] = 39 → **Match found**



### Example 2:

Sorted Array: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

M- mid L- low H- high




### Time Complexity:

- Best Case: O(1)
- Average & Worst Case: O(log n)

### Limitation:

- **Array must be sorted** before applying binary search.

### Real-World Applications:

- Searching in **phonebooks**, **dictionary words**
- Looking up **user IDs** in databases
- Finding a name in an **alphabetically ordered list**

There are 2 types

1. **Iterative Binary Search**
2. **Recursive Binary Search**

### 1. Iterative Binary Search

✅ Uses a **while** loop to repeatedly divide the search space.

✅ Maintains low, high, and mid variables within the same function.

✅ More **memory-efficient** because it doesn't use the call stack.

✅ Usually **faster** in practice due to no function call overhead.

✅ Easier to debug and widely used in real-world applications.

❌ Slightly more code to manage loop conditions manually.

**Pseudocode:**

```
function binarySearch(arr, target):
    low ← 0
    high ← length(arr) - 1

    while low ≤ high:
        mid ← (low + high) / 2
        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            low ← mid + 1
        else:
            high ← mid - 1

    return -1  // Target not found
```

## Iterative Binary Search – Java Code

```java
public class IterativeBinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == target)
                return mid; // Target found
            else if (arr[mid] < target)
                low = mid + 1; // Search right half
            else
                high = mid - 1; // Search left half
        }

        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] data = {2, 4, 6, 8, 10, 12}; // Sorted array
        int target = 8;

        int result = binarySearch(data, target);

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

**Output:**

```
Element found at index: 3
```

## 2.Recursive Binary Search

✅ Uses **function calls** to divide the problem into smaller subproblems.

✅ Each call handles a smaller part of the array (via new low and high).

✅ More elegant and **simpler to write** for learning divide-and-conquer.

❌ Uses **extra memory** due to the function call stack.

❌ Risk of **stack overflow** for very large arrays if not tail-recursive.

### Pseudocode:

```
function binarySearch(arr, low, high, target):
    if low > high:
        return -1  // Target not found

    mid ← (low + high) / 2
    if arr[mid] == target:
        return mid
    else if arr[mid] < target:
        return binarySearch(arr, mid + 1, high, target)
    else:
        return binarySearch(arr, low, mid - 1, target)
```

### Recursive Binary Search – Java Code

```
public class RecursiveBinarySearch {
    public static int binarySearch(int[] arr, int low, int high, int
target) {
        if (low > high) return -1; // Base case: not found

        int mid = (low + high) / 2;

        if (arr[mid] == target)
```

```
            return mid; // Found
        else if (arr[mid] < target)
            return binarySearch(arr, mid + 1, high, target); //
Search right
        else
            return binarySearch(arr, low, mid - 1, target); //
Search left
    }

    public static void main(String[] args) {
        int[] data = {2, 4, 6, 8, 10, 12};
        int target = 10;

        int result = binarySearch(data, 0, data.length - 1, target);

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

**Output:**

```
Element found at index: 4
```

## Questions

**Q1**:Write a Java program to implement binary search on a sorted array of integers. The program should take a target number and return the index where it is found, or print "Not found" if the number is not in the array.

**Input Example:**
Array: {2, 4, 6, 8, 10, 12}
Target: 8

**Expected Output:**
Element found at index 3

**Q2.** Write a Java program that performs **binary search recursively** on a sorted array of integers. Print the `low`, `high`, and `mid` values at each step of the recursion.

**Input Example**:
Array: `{2, 4, 6, 8, 10, 12}`
Target: `10`

**Expected Output**:
Low: 0, High: 5, Mid: 2
Low: 3, High: 5, Mid: 4
Element found at index 4

**Q3.**Modify the standard binary search algorithm to **return the first occurrence** of the target value in a sorted array that may contain duplicates.

**Input Example**:
Array: `{1, 2, 4, 4, 4, 5, 6}`
Target: `4`

**Expected Output**:
First occurrence at index 2

## Comparison: Sequential Search vs Binary Search

| Feature | Sequential Search | Binary Search |
|---|---|---|
| ***Also Called*** | Linear Search | Logarithmic Search |
| ***Array Requirement*** | Works on unsorted or sorted arrays | Works only on sorted arrays |
| ***Method*** | Checks each element one by one | Divides array in half each time |
| ***Time Complexity*** | O(n) | O(log n) |
| ***Best Case*** | O(1) - element is first | O(1) - element is middle |
| ***Worst Case*** | O(n) - element at end or not found | O(log n) - many divisions |
| ***Space Complexity*** | O(1) | O(1) iterative, O(log n) recursive |
| ***Use Case*** | Small or unsorted data | Large and sorted data |
| ***Implementation*** | Very simple | Requires careful index handling |
| ***Flexibility*** | More flexible (no sorting needed) | Less flexible (requires sorting) |