Day-1

1. Binary Search (Classic)

Description:

Given a **sorted array** and a **target element**, return its index if found, else return `-1`.

Example:

```
Input: arr = [1, 3, 5, 7, 9], target = 5
Output: 2
```

Java Code:

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

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

            if (arr[mid] == target) return mid;
            else if (arr[mid] < target) low = mid + 1;
            else high = mid - 1;
        }

        return -1; // not found
    }
}
```

Time Complexity:

- Time: O(log n)
- Space: O(1)

2. Count Occurrences of a Number in a Sorted Array

Description:

Given a **sorted array**, find **how many times a target number occurs**.

Example:

```
Input: arr = [2, 4, 4, 4, 6], target = 4
Output: 3
```

Java Code:

```java
public class CountOccurrences {
    public int countOccurrences(int[] arr, int target) {
        int first = findFirst(arr, target);
        if (first == -1) return 0; // not found
        int last = findLast(arr, target);
        return last - first + 1;
    }

    private int findFirst(int[] arr, int target) {
        int low = 0, high = arr.length - 1, result = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == target) {
                result = mid;
                high = mid - 1; // go left
            } else if (arr[mid] < target) low = mid + 1;
            else high = mid - 1;
        }
        return result;
    }

    private int findLast(int[] arr, int target) {
        int low = 0, high = arr.length - 1, result = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == target) {
                result = mid;
                low = mid + 1; // go right
            } else if (arr[mid] < target) low = mid + 1;
            else high = mid - 1;
        }
        return result;
    }
}
```

Time Complexity:

- Time: O(log n)
- Space: O(1)

Day-2

1. Find Peak Element in Mountain Array

Description:

A **mountain array** is defined as an array:

- Increasing up to a certain point (the peak)
- Then decreasing

Find the **index of the peak element** using binary search.

Example:

```
Input: [1, 3, 5, 7, 6, 4, 2]
Output: 3 // Peak = 7 at index 3

Input: [0, 2, 4, 6, 5, 3, 1]
Output: 3 // Peak = 6
```

Java Code:

```
public class PeakInMountainArray {
    public int peakIndexInMountainArray(int[] arr) {
        int low = 0, high = arr.length - 1;

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

            if (arr[mid] < arr[mid + 1]) {
                low = mid + 1; // Move right
            } else {
                high = mid; // Move left
            }
        }

        return low; // or high
    }
}
```

Complexity:

- Time: O(log n)
- Space: O(1)

2. Check if Array is Sorted and Rotated

Description:

Given an array, check if it is **sorted in increasing order** and then **rotated**.
Rotation means that a part of the array is moved to the end while maintaining sorted order.
e.g., `[3, 4, 5, 1, 2]` is a sorted-then-rotated array.

Example:

```
Input: [3, 4, 5, 1, 2]
Output: true

Input: [2, 1, 3, 4]
Output: false

Input: [1, 2, 3]
Output: true // it’s a sorted array (rotation = 0)
```

Java Code:

```
public class CheckSortedAndRotated {
    public boolean check(int[] nums) {
        int count = 0;
        int n = nums.length;

        for (int i = 0; i < n; i++) {
            // Compare current with next, with wrapping using %n
            if (nums[i] > nums[(i + 1) % n]) {
                count++;
            }
            if (count > 1) return false; // More than one drop → Not sorted-rotated
        }

        return true;
    }
}
```

Logic:

- Traverse the array and count how many times `nums[i] > nums[i+1]`
- If it happens more than once → it's not sorted and rotated

Complexity:

- Time: O(n)
- Space: O(1)

Day-3

1. Peak Element in an Array

Description:

A **peak element** is one that is **strictly greater than its neighbors**.
You must return **any one** peak element's index using **binary search** (O(log n)).

The array may contain multiple peaks — any one is acceptable.

Example:

```
Input: [1, 2, 3, 1]
Output: 2 // 3 is a peak

Input: [1, 2, 1, 3, 5, 6, 4]
Output: 5 // 6 is a peak (can also return 1 for 2)
```

Java Code:

```java
public class PeakElementFinder {
    public int findPeakElement(int[] nums) {
        int low = 0, high = nums.length - 1;

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

            if (nums[mid] > nums[mid + 1]) {
                // Peak is on the left including mid
                high = mid;
            } else {
                // Peak is on the right
                low = mid + 1;
            }
        }

        return low;  // or high; both point to a peak
    }
}
```

Logic:

- Use a binary search variant:
    - If `nums[mid] > nums[mid+1]`, then a peak lies to the **left**
    - Else, a peak lies to the **right**

Complexity:

- Time: O(log n)
- Space: O(1)

2. Search in Rotated Sorted Array

Description:

Given a rotated sorted array and a target value, return its index if found, otherwise return `-1`.

Example:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

Java Code:

```
public class SearchInRotatedArray {
    public int search(int[] nums, int target) {
        int low = 0, high = nums.length - 1;

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

            if (nums[mid] == target)
                return mid;

            // Left half is sorted
            if (nums[low] <= nums[mid]) {
                if (nums[low] <= target && target < nums[mid])
                    high = mid - 1;
                else
                    low = mid + 1;
            }
            // Right half is sorted
            else {
                if (nums[mid] < target && target <= nums[high])
                    low = mid + 1;
                else
                    high = mid - 1;
            }
        }

        return -1;
    }}
```

Time & Space Complexity:

- Time: O(log n)
- Space: O(1)

Day-4

1. Find First and Last Position of Element in Sorted Array

Description:

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given target value.

- If the target is not found, return `[-1, -1]`.
- Your algorithm must run in **O(log n)** time.

Example:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

Java Code:

```
public class FirstLastPosition {
    public int[] searchRange(int[] nums, int target) {
        int first = findIndex(nums, target, true);
        int last = findIndex(nums, target, false);
        return new int[]{first, last};
    }

    private int findIndex(int[] nums, int target, boolean findFirst) {
        int low = 0, high = nums.length - 1, result = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) {
                result = mid;
                if (findFirst) {
                    high = mid - 1;  // Move left
                } else {
                    low = mid + 1;   // Move right
                }
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return result;
    }
}
```

2. Search Insert Position

Description:

Given a **sorted array** of distinct integers and a target value, return the index if the target is found. If not, return the index where it **would be inserted** in order.

- Must run in **O(log n)** time.

Example:

```
Input: nums = [1,3,5,6], target = 5
Output: 2

Input: nums = [1,3,5,6], target = 2
Output: 1

Input: nums = [1,3,5,6], target = 7
Output: 4
```

Java Code:

```
public class SearchInsertPosition {
    public int searchInsert(int[] nums, int target) {
        int low = 0, high = nums.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return low;  // Insertion position
    }
}
```