Assignment Problems – Searching and Sorting.

Q1:Given a 1-indexed array of integers numbers that are already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 < numbers.length.

Return the indices of the two numbers, index1, and index2, added by one as an integer array [index1, index2] of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Your solution must use only constant extra space.

Input: numbers = [2,7,11,15], target = 9

Output: [1,2]

Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

Solution:

```
public int[] twoSum(int[] numbers, int target) {
    int left = 0;
    int right = numbers.length - 1;
    while (left < right) {
        int sum = numbers[left] + numbers[right];
        if (sum == target) {
            return new int[]{left + 1, right + 1};
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    // We should never reach this point as the problem guarantees a solution.
    // We can throw an exception or return null to handle such cases.
    return null;
}
```

Q2:Given an array of integer nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If the target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Solution:

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1, -1};
        // Find the starting position of the target value
        result[0] = findFirstOccurrence(nums, target);
        if (result[0] == -1) {
           // If the starting position is not found, the target is not present in the array
            return result;
```

```
        }
        // Find the ending position of the target value
        result[1] = findLastOccurrence(nums, target);
        return result;
    }
private int findFirstOccurrence(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        int firstOccurrence = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] >= target) {
                right = mid - 1;
                if (nums[mid] == target) {
                    firstOccurrence = mid;
                }
            } else {
                left = mid + 1;
            }
        }
        return firstOccurrence;
    }
    private int findLastOccurrence(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        int lastOccurrence = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] <= target) {
                left = mid + 1;
                if (nums[mid] == target) {
                    lastOccurrence = mid;
                }
            } else {
                right = mid - 1;
            }
        }
        return lastOccurrence;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {5, 7, 7, 8, 8, 10};
        int target = 8;
        int[] result = solution.searchRange(nums, target);
```

```
        System.out.println("Output: [" + result[0] + ", " + result[1] + "]");
    }
}
```

Question:3A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

Solution:

```
public class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0;
       int right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[mid + 1]) {
                // If the current element is greater than the next one, move left
                right = mid;
            } else {
                // If the next element is greater or equal, move right
                left = mid + 1;
            }
        }

        // At this point, left and right will be equal, and it will represent a peak element
        return left;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 2, 3, 1};
        int peakIndex = solution.findPeakElement(nums);
        System.out.println("Output: " + peakIndex);
    }
}
```

Question:4 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 if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

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

Output: 2

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

Output: 4

Solution:

```
public class Solution {
    public int searchInsert(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return left;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 3, 5, 6};
        int target1 = 5;
        int target2 = 7;
        System.out.println("Output 1: " + solution.searchInsert(nums, target1)); // Output: 2
        System.out.println("Output 2: " + solution.searchInsert(nums, target2)); // Output: 4
    }
}
```

Question:5

Find the majority element in the array. A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Input: A[]={3, 3, 4, 2, 4, 4, 2, 4, 4}

Output: 4

Explanation: The frequency of 4 is 5 which is greater than half of the size of the array size.

Solution:

```
public class Solution {
    public int majorityElement(int[] nums) {
        int majority = nums[0];
        int count = 1;
```

```
        for (int i = 1; i < nums.length; i++) {
            if (count == 0) {
                majority = nums[i];
                count = 1;
            } else if (nums[i] == majority) {
                count++;
            } else {
                count--;
            }
        }

        return majority;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {3, 3, 4, 2, 4, 4, 2, 4, 4};
        System.out.println("Output: " + solution.majorityElement(nums)); // Output: 4
    }
}
```

Question:6

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad

You are given an API bool isBadVersion(version) which returns whether the version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Input: n = 5, bad = 4

Output: 4

Explanation:

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

Solution:

```
public class Solution {
    public int firstBadVersion(int n) {
        int left = 1;
        int right = n;

        while (left < right) {
            int mid = left + (right - left) / 2;

            if (isBadVersion(mid)) {
```

```
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }

    // This is just a placeholder method as it's provided by the problem
    private boolean isBadVersion(int version) {
        // Implement the actual isBadVersion API here
        // For example, if version 4 is bad, you could return version >= 4.
        return false;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int n = 5;
        int bad = 4;
        System.out.println("Output: " + solution.firstBadVersion(n)); // Output: 4
    }
}
```

Question:7

Given an array of integers, find the inversion of an array. Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

N=5, arr[] = {2, 4, 1, 3, 5}

Output: 3

Explanation: (2,1) (4,1) and (4,3) forms an inversion in an array

Solution:

```
public class Solution {
    public int countInversions(int[] arr) {
        return mergeSortAndCount(arr, 0, arr.length - 1);
    }

    private int mergeSortAndCount(int[] arr, int left, int right) {
        int count = 0;

        if (left < right) {
            int mid = left + (right - left) / 2;

            count += mergeSortAndCount(arr, left, mid);
            count += mergeSortAndCount(arr, mid + 1, right);
            count += mergeAndCount(arr, left, mid, right);
        }
```

```
        return count;
    }

    private int mergeAndCount(int[] arr, int left, int mid, int right) {
        int count = 0;
        int[] temp = new int[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
                count += (mid - i + 1);
            }
        }

        while (i <= mid) {
            temp[k++] = arr[i++];
        }

        while (j <= right) {
            temp[k++] = arr[j++];
        }

        System.arraycopy(temp, 0, arr, left, temp.length);

        return count;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] arr = {2, 4, 1, 3, 5};
        System.out.println("Output: " + solution.countInversions(arr)); // Output: 3
    }
}
```

Question:8

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

ar1[] = {1, 5, 10, 20, 40, 80}

ar2[] = {6, 7, 20, 80, 100}

ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}

Output: 20, 80

Solution:import java.util.*;

```
public class Solution {
    public List<Integer> findCommonElements(int[] ar1, int[] ar2, int[] ar3) {
        List<Integer> result = new ArrayList<>();
        int i = 0, j = 0, k = 0;

        while (i < ar1.length && j < ar2.length && k < ar3.length) {
            if (ar1[i] == ar2[j] && ar2[j] == ar3[k]) {
                result.add(ar1[i]);
                i++;
                j++;
                k++;
            } else if (ar1[i] < ar2[j]) {
                i++;
            } else if (ar2[j] < ar3[k]) {
                j++;
            } else {
                k++;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] ar1 = {1, 5, 10, 20, 40, 80};
        int[] ar2 = {6, 7, 20, 80, 100};
        int[] ar3 = {3, 4, 15, 20, 30, 70, 80, 120};
        List<Integer> commonElements = solution.findCommonElements(ar1, ar2, ar3);
        System.out.println("Output: " + commonElements); // Output: [20, 80]
    }
}
```