PRACTICE ASSIGNMENT 5

1. Stock buy and sell

The cost of stock on each day is given in an array A[] of size N. Find all the segments of days on which you buy and sell the stock such that the sum of difference between sell and buy prices is maximized. Each segment consists of indexes of two elements, first is index of day on which you buy stock and second is index of day on which you sell stock. Note: Since there can be multiple solutions, the driver code will print 1 if your answer is correct, otherwise, it will return 0. In case there's no profit the driver code will print the string "No Profit" for a correct solution.

Input: N = 7 A[] = {100,180,260,310,40,535,695}
Output: 1

N = 5 A[] = {4,2,2,2,4}
Output: 1

CODE:

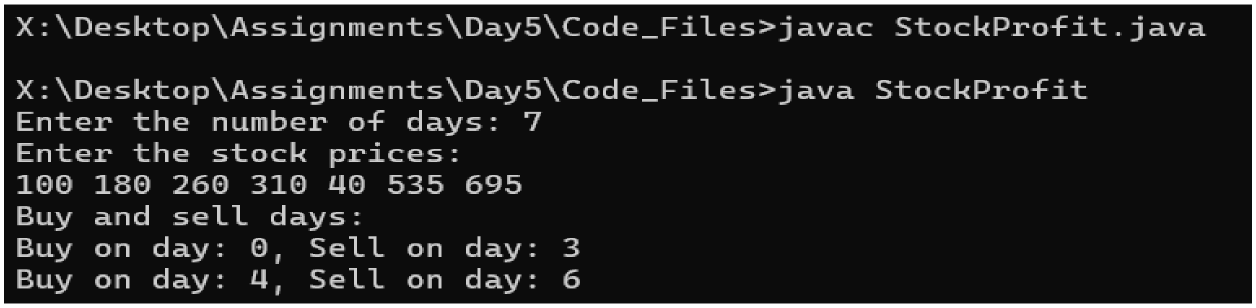
```
import java.util.*;
class StockInterval {
   int startDay;
   int endDay;
}
class StockProfit {
   ArrayList<ArrayList<Integer>> findProfitDays(int prices[], int days) {
      ArrayList<ArrayList<Integer>> output = new ArrayList<>();
      if (days < 2) {
         return output;
      }
      ArrayList<StockInterval> intervals = new ArrayList<>();
      int current = 0;
      while (current < days - 1) {
         while (current < days - 1 && prices[current + 1] <= prices[current]) {
            current++;
         }
         if (current == days - 1) break;
         StockInterval interval = new StockInterval();
         interval.startDay = current++;
```

```
            while (current < days && prices[current] >= prices[current - 1]) {
                current++;
            }
            interval.endDay = current - 1;
            intervals.add(interval);
        }

        for (StockInterval interval : intervals) {
            ArrayList<Integer> pair = new ArrayList<>();
            pair.add(interval.startDay);
            pair.add(interval.endDay);
            output.add(pair);
        }
        return output;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of days: ");
        int days = sc.nextInt();
        int[] prices = new int[days];
        System.out.println("Enter the stock prices: ");
        for (int i = 0; i < days; i++) {
            prices[i] = sc.nextInt();
        }

        StockProfit sp = new StockProfit();
        ArrayList<ArrayList<Integer>> result = sp.findProfitDays(prices, days);
        if (result.isEmpty()) {
            System.out.println("No profit can be made.");
        } else {
            System.out.println("Buy and sell days: ");
            for (ArrayList<Integer> pair : result) {
                System.out.println("Buy on day: " + pair.get(0) + ", Sell on day: " + pair.get(1));
            }
        }
        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac StockProfit.java

X:\Desktop\Assignments\Day5\Code_Files>java StockProfit
Enter the number of days: 7
Enter the stock prices:
100 180 260 310 40 535 695
Buy and sell days:
Buy on day: 0, Sell on day: 3
Buy on day: 4, Sell on day: 6
```

TIME COMPLEXITY: O(n)

---

2. Coin Change (Count Ways)

Given an integer array coins[ ] representing different denominations of currency and an integer sum, find the number of ways you can make sum by using different combinations from coins[ ].
Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want. Answers are guaranteed to fit into a 32-bit integer.

Input: coins[] = [1, 2, 3], sum = 4
Output: 4
Explanation: Four Possible ways are: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3].

Input: coins[] = [2, 5, 3, 6], sum = 10
Output: 5

Input: coins[] = [5, 10], sum = 3
Output: 0
Explanation: Since all coin denominations are greater than sum, no combination can make the target sum.

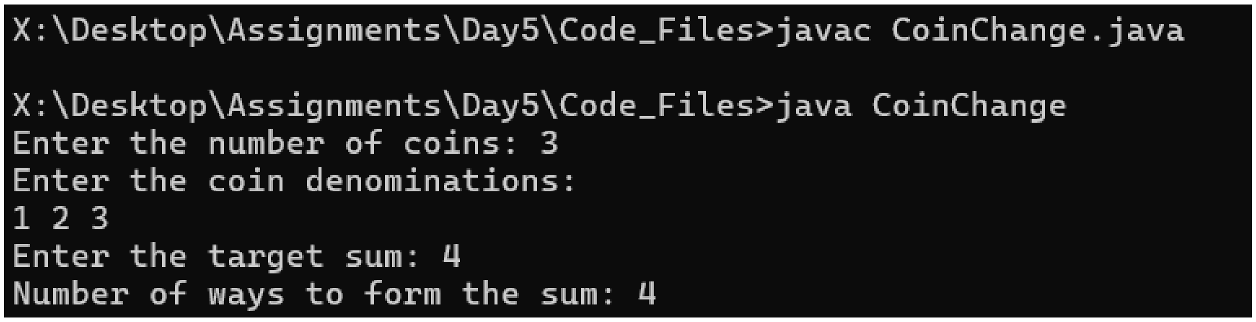
CODE:

```
import java.util.Scanner;
class CoinChange {
    public static int countWays(int[] coins, int sum) {
        int n = coins.length;
        int[] dp = new int[sum + 1];
        dp[0] = 1;

        for (int coin : coins) {
            for (int j = coin; j <= sum; j++) {
                dp[j] += dp[j - coin];
            }
        }
        return dp[sum];
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of coins: ");
        int numCoins = sc.nextInt();
        int[] coins = new int[numCoins];
        System.out.println("Enter the coin denominations: ");
        for (int i = 0; i < numCoins; i++) {
            coins[i] = sc.nextInt();
        }
        System.out.print("Enter the target sum: ");
        int sum = sc.nextInt();
        int result = countWays(coins, sum);
        System.out.println("Number of ways to form the sum: " + result);
        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac CoinChange.java

X:\Desktop\Assignments\Day5\Code_Files>java CoinChange
Enter the number of coins: 3
Enter the coin denominations:
1 2 3
Enter the target sum: 4
Number of ways to form the sum: 4
```

TIME COMPLEXITY: O(n.sum)

3. First and Last Occurrences

Given a sorted array arr with possibly some duplicates, the task is to find the first and last occurrences of an element x in the given array.
Note: If the number x is not found in the array then return both the indices as -1.

Input: arr[] = [1, 3, 5, 5, 5, 5, 67, 123, 125], x = 5
Output: [2, 5]
Explanation: First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5

Input: arr[] = [1, 3, 5, 5, 5, 5, 7, 123, 125], x = 7
Output: [6, 6]
Explanation: First and last occurrence of 7 is at index 6

Input: arr[] = [1, 2, 3], x = 4
Output: [-1, -1]
Explanation: No occurrence of 4 in the array, so, output is [-1, -1]

CODE:

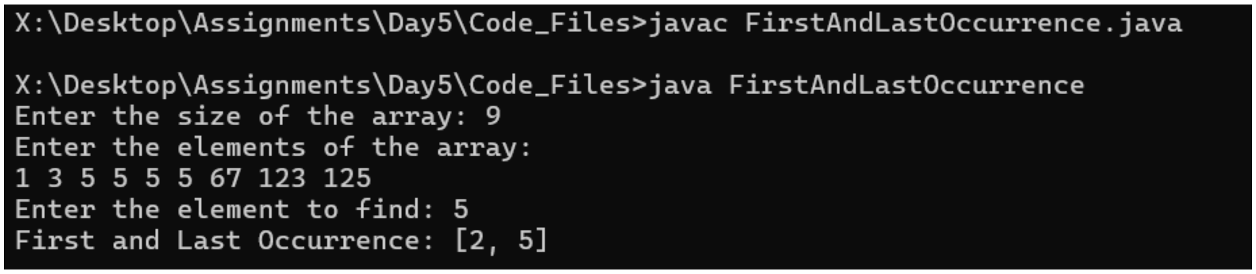
```
import java.util.Scanner;
class FirstAndLastOccurrence {
    public static int[] findFirstAndLast(int[] arr, int x) {
        int[] result = new int[2];
        result[0] = binarySearch(arr, x, true);
        result[1] = binarySearch(arr, x, false);
        return result;
    }
    public static int binarySearch(int[] arr, int x, boolean findFirst) {
        int low = 0, high = arr.length - 1, index = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == x) {
                index = mid;
                if (findFirst) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            } else if (arr[mid] < x) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return index;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        System.out.print("Enter the element to find: ");
        int x = sc.nextInt();

        int[] result = findFirstAndLast(arr, x);
        System.out.println("First and Last Occurrence: [" + result[0] + ", " + result[1] + "]");

        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac FirstAndLastOccurrence.java

X:\Desktop\Assignments\Day5\Code_Files>java FirstAndLastOccurrence
Enter the size of the array: 9
Enter the elements of the array:
1 3 5 5 5 5 67 123 125
Enter the element to find: 5
First and Last Occurrence: [2, 5]
```

TIME COMPLEXITY: O(logn)

4. Find Transition Point

Given a sorted array, arr[] containing only 0s and 1s, find the transition point, i.e., the first index where 1 was observed, and before that, only 0 was observed. If arr does not have any 1, return -1. If array does not have any 0, return 0.

Input: arr[] = [0, 0, 0, 1, 1]
Output: 3
Explanation: index 3 is the transition point where 1 begins.

Input: arr[] = [0, 0, 0, 0]
Output: -1
Explanation: Since, there is no "1", the answer is -1.

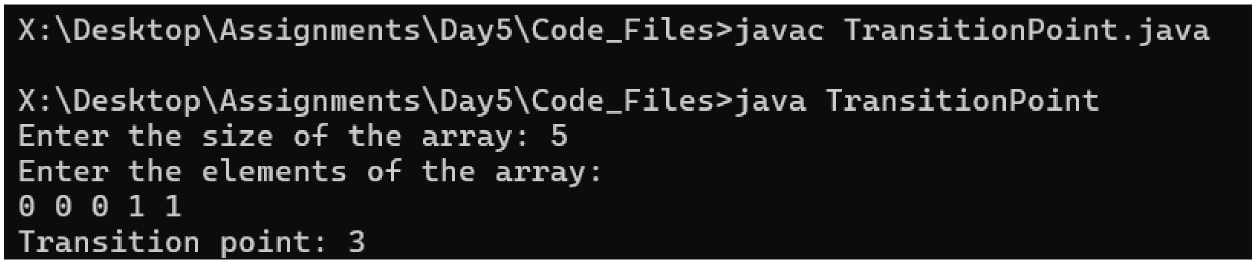
CODE:

```
import java.util.Scanner;
class TransitionPoint {
    public static int findTransitionPoint(int[] arr) {
        int low = 0, high = arr.length - 1, result = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == 1) {
                result = mid;
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return result;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        int result = findTransitionPoint(arr);
        System.out.println("Transition point: " + result);

        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac TransitionPoint.java

X:\Desktop\Assignments\Day5\Code_Files>java TransitionPoint
Enter the size of the array: 5
Enter the elements of the array:
0 0 0 1 1
Transition point: 3
```

TIME COMPLEXITY: O(logn)

---

5. First Repeating Element

Given an array arr[], find the first repeating element. The element should occur more than once and the index of its first occurrence should be the smallest.
Note:- The position you return should be according to 1-based indexing.

Input: arr[] = [1, 5, 3, 4, 3, 5, 6]
Output: 2
Explanation: 5 appears twice and its first appearance is at index 2 which is less than 3 whose first the occurring index is 3.

Input: arr[] = [1, 2, 3, 4]
Output: -1
Explanation: All elements appear only once so answer is -1.

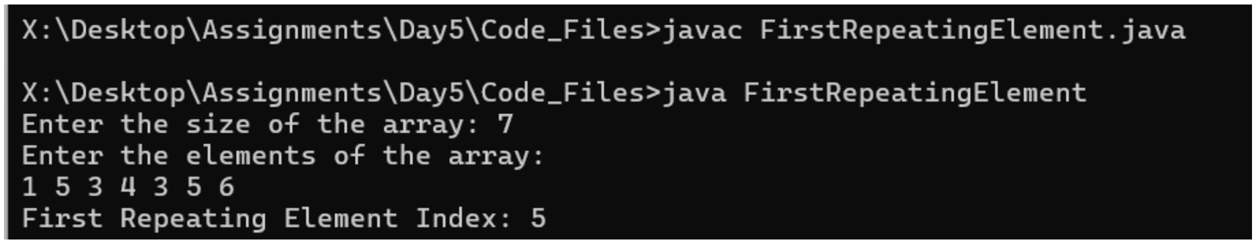
CODE:

```
import java.util.Scanner;
import java.util.HashSet;
```

```
class FirstRepeatingElement {
    public static int findFirstRepeatingElement(int[] arr) {
        HashSet<Integer> set = new HashSet<>();
        for (int i = 0; i < arr.length; i++) {
            if (set.contains(arr[i])) {
                return i + 1;
            }
            set.add(arr[i]);
        }
        return -1;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        int result = findFirstRepeatingElement(arr);
        System.out.println("First Repeating Element Index: " + result);

        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac FirstRepeatingElement.java

X:\Desktop\Assignments\Day5\Code_Files>java FirstRepeatingElement
Enter the size of the array: 7
Enter the elements of the array:
1 5 3 4 3 5 6
First Repeating Element Index: 5
```

TIME COMPLEXITY: O(n)

6. Remove Duplicates Sorted Array

Given a sorted array arr. Return the size of the modified array which contains only distinct elements.

1. Don't use set or HashMap to solve the problem.
2. You must return the modified array size only where distinct elements are present and modify the original array such that all the distinct elements come at the beginning of the original array.

Input: arr = [2, 2, 2, 2, 2]
Output: [2]
Explanation: After removing all the duplicates only one instance of 2 will remain i.e. [2] so modified array will contains 2 at first position and you should return 1 after modifying the array, the driver code will print the modified array elements.

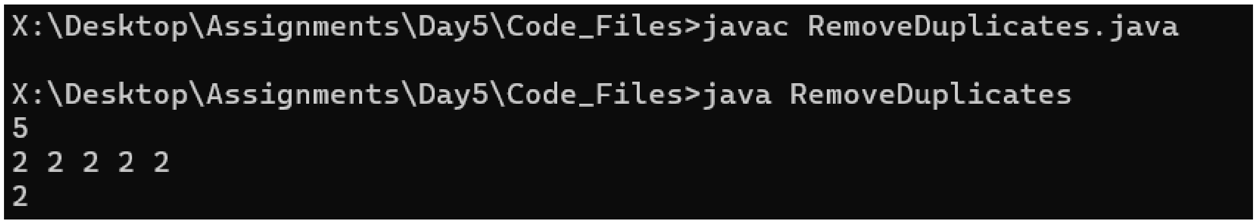
Input: arr = [1, 2, 4]
Output: [1, 2, 4]
Explation: As the array does not contain any duplicates so you should return 3.

CODE:

```
import java.util.Scanner;
class RemoveDuplicates {
    public static int removeDuplicates(int[] arr) {
        if (arr.length == 0) {
            return 0;
        }

        int i = 0;
        for (int j = 1; j < arr.length; j++) {
            if (arr[i] != arr[j]) {
                i++;
                arr[i] = arr[j];
            }
        }
        return i + 1;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        int result = removeDuplicates(arr);
        for (int i = 0; i < result; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println("\n" + result);
        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac RemoveDuplicates.java

X:\Desktop\Assignments\Day5\Code_Files>java RemoveDuplicates
5
2 2 2 2 2
2
```

TIME COMPLEXITY: O(n)

---

7. Maximum Index

Given an array arr of positive integers. The task is to return the maximum of j - i subjected to the constraint of arr[i] < arr[j] and i < j.

Input: arr[] = [1, 10]
Output: 1
Explanation: arr[0] < arr[1] so (j-i) is 1-0 = 1.

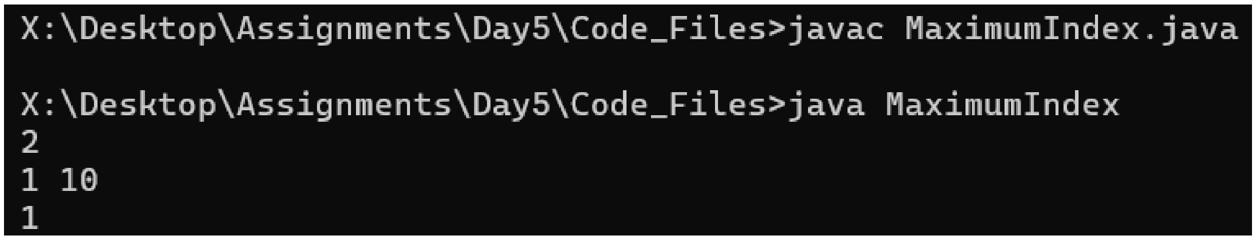
Input: arr[] = [34, 8, 10, 3, 2, 80, 30, 33, 1]
Output: 6

Explanation: In the given array arr[1] < arr[7] satisfying the required condition(arr[i] < arr[j]) thus giving the maximum difference of j - i which is 6(7-1).

CODE:

```
import java.util.Scanner;
class MaximumIndex {
    public static int maxIndexDiff(int[] arr) {
        int n = arr.length;
        int[] leftMin = new int[n];
        int[] rightMax = new int[n];
        leftMin[0] = arr[0];
        for (int i = 1; i < n; i++) {
            leftMin[i] = Math.min(arr[i], leftMin[i - 1]);
        }
        rightMax[n - 1] = arr[n - 1];
        for (int j = n - 2; j >= 0; j--) {
            rightMax[j] = Math.max(arr[j], rightMax[j + 1]);
        }
        int i = 0, j = 0;
        int maxDiff = -1;
        while (i < n && j < n) {
            if (leftMin[i] < rightMax[j]) {
                maxDiff = Math.max(maxDiff, j - i);
                j++;
            } else {
                i++;
            }
        }
        return maxDiff;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        int result = maxIndexDiff(arr);
        System.out.println(result);
        sc.close();
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac MaximumIndex.java

X:\Desktop\Assignments\Day5\Code_Files>java MaximumIndex
2
1 10
1
```

TIME COMPLEXITY: O(n)

---

8. Wave Array

Given a sorted array arr[] of distinct integers. Sort the array into a wave-like array(In Place). In other words, arrange the elements into a sequence such that arr[1] >= arr[2] <= arr[3] >= arr[4] <= arr[5].....
If there are multiple solutions, find the lexicographically smallest one.

Note: The given array is sorted in ascending order, and you don't need to return anything to change the original array.

Input: arr[] = [1, 2, 3, 4, 5]
Output: [2, 1, 4, 3, 5]
Explanation: Array elements after sorting it in the waveform are 2, 1, 4, 3, 5.

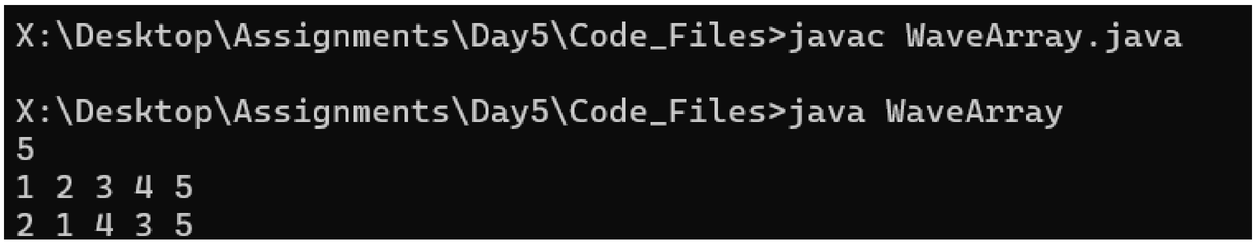
Input: arr[] = [2, 4, 7, 8, 9, 10]
Output: [4, 2, 8, 7, 10, 9]
Explanation: Array elements after sorting it in the waveform are 4, 2, 8, 7, 10, 9.

Input: arr[] = [1]
Output: [1]

CODE:

```
import java.util.Scanner;
class WaveArray {
    public static void convertToWave(int[] arr) {
        int n = arr.length;
        int temp = 0;
        for (int i = 0; i < n - 1; i = i + 2) {
            temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        convertToWave(arr);
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

OUTPUT:



```
X:\Desktop\Assignments\Day5\Code_Files>javac WaveArray.java

X:\Desktop\Assignments\Day5\Code_Files>java WaveArray
5
1 2 3 4 5
2 1 4 3 5
```

TIME COMPLEXITY: O(n)