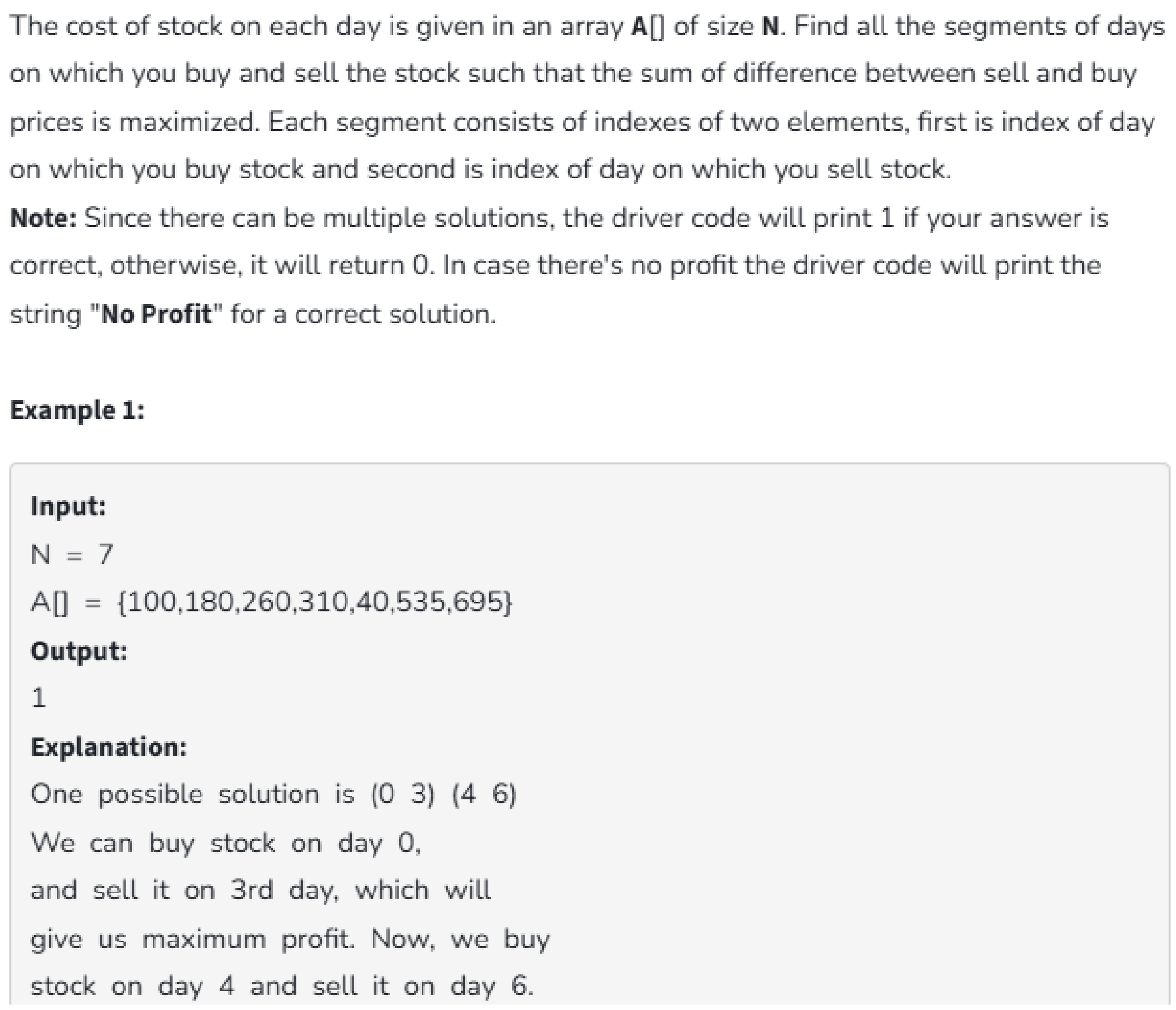
Coding Practice-5

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.

Example 1:

```
Input:
N = 7
A[] = {100,180,260,310,40,535,695}
Output:
1
Explanation:
One possible solution is (0 3) (4 6)
We can buy stock on day 0,
and sell it on 3rd day, which will
give us maximum profit. Now, we buy
stock on day 4 and sell it on day 6.
```

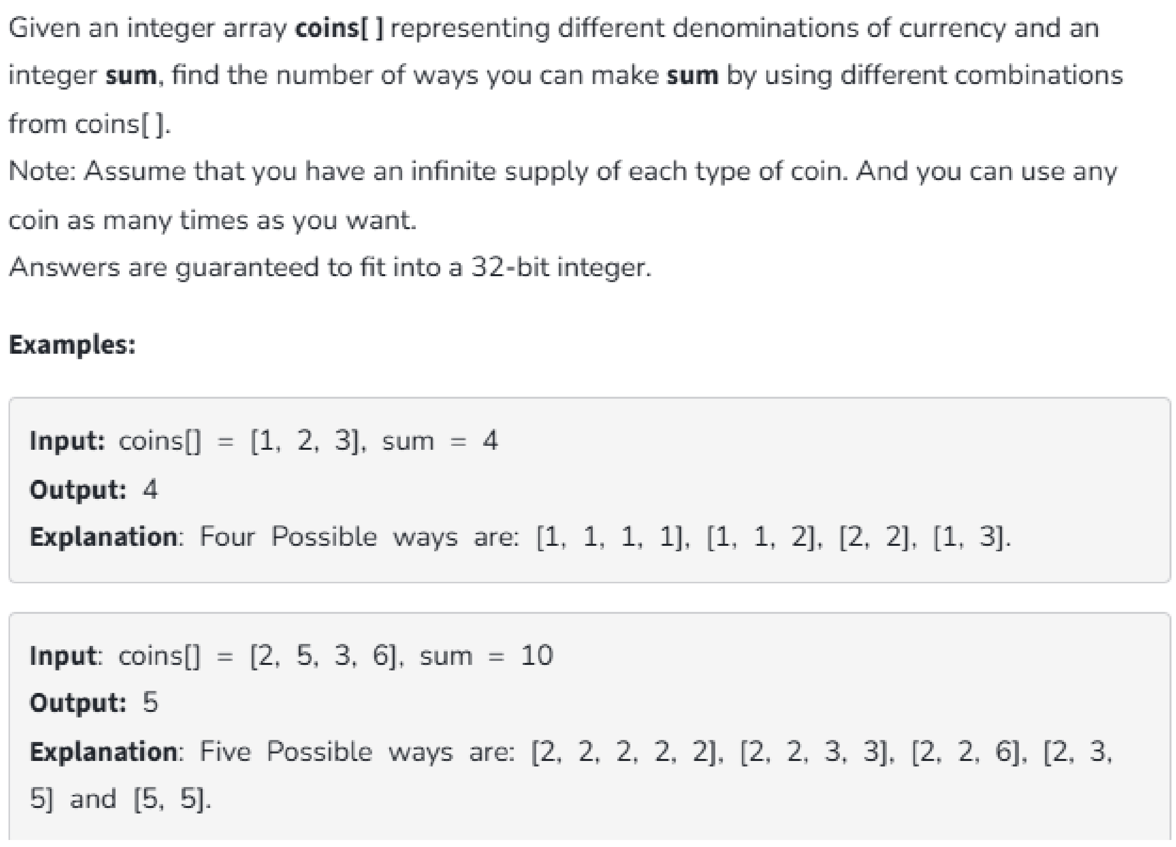
Code:

```
import java.util.ArrayList;
import java.util.List;
public class Stock {
    public static int stockBuySell(int[] prices) {
        int n = prices.length;
        List<String> segments = new ArrayList<>();
        int i = 0;
        while (i < n - 1) {
            while (i < n - 1 && prices[i + 1] <= prices[i]) {
                i++;
            }
            if (i == n - 1) {
                break;
            }
            int buy = i++;
            while (i < n && prices[i] >= prices[i - 1]) {
                i++;
            }
            int sell = i - 1;
            segments.add("(" + buy + " " + sell + ")");
        }
        return segments.isEmpty() ? 0 : segments.size();
    }
    public static void main(String[] args) {
        int[] prices = {100, 180, 260, 310, 40, 535, 695};
        System.out.println(stockBuySell(prices));
    }
}
```

Output:2

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.

Examples:

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
Explanation: Five Possible ways are: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5] and [5, 5].

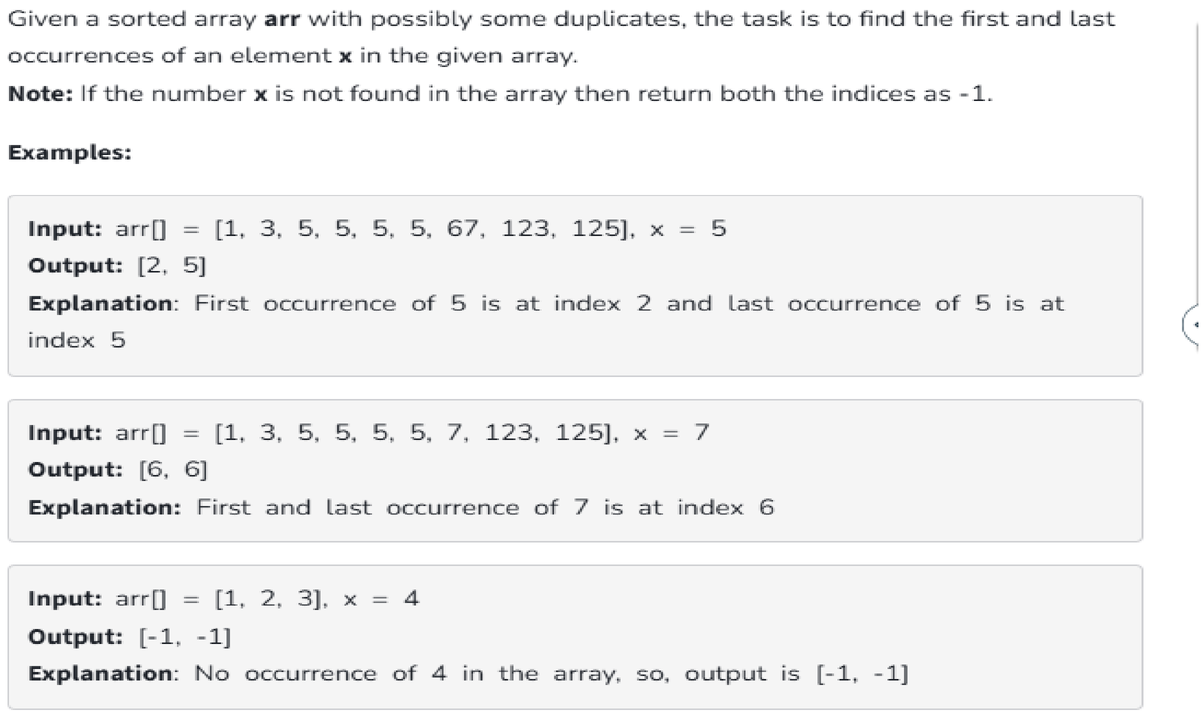
Code:

```
import java.util.*;


class CoinChange {
    static int count(int[] coins, int sum, int n, int[][] dp) {
        if (sum == 0)
            return dp[n][sum] = 1;
        if (n == 0 || sum < 0)
            return 0;
        if (dp[n][sum] != -1)
            return dp[n][sum];
        return dp[n][sum] = count(coins, sum - coins[n - 1], n, dp) + count(coins, sum, n - 1, dp);
    }


    public static void main(String[] args) {
        int tc = 1;
        while (tc != 0) {
            int n = 3, sum = 4;
            int[] coins = {1, 2, 3};
            int[][] dp = new int[n + 1][sum + 1];
            for (int[] row : dp)
                Arrays.fill(row, -1);
            int res = count(coins, sum, n, dp);
            System.out.println(res);
            tc--;
        }
    }
}
```

Output:4

Time Complexity:O(n^2)

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.

Examples:

```
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]
```

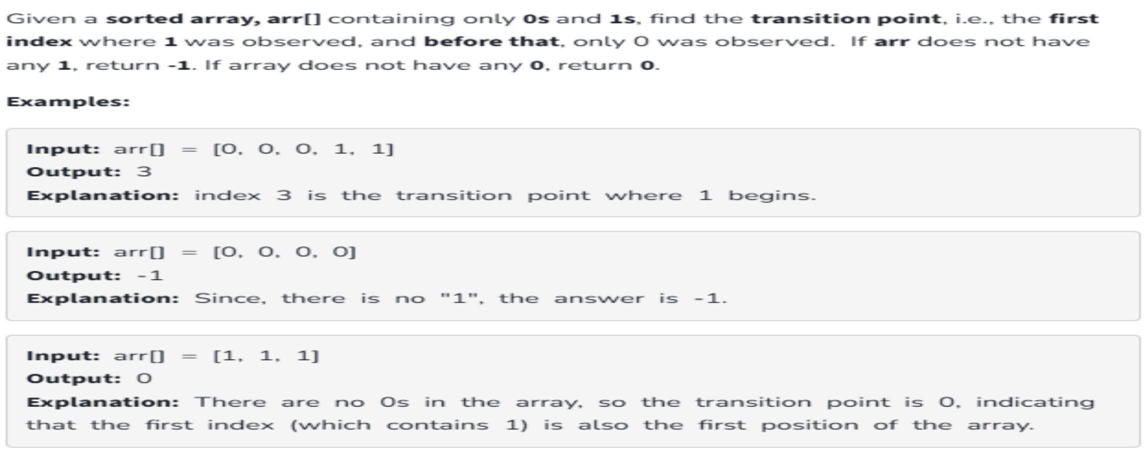
Code:

```
public class FirstLast {nb
    public static int findFirstOccurrence(int[] arr, int x) {
        int left = 0, right = arr.length - 1;
        int result = -1;

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

            if (arr[mid] == x) {
                result = mid;
               right = mid - 1;
            } else if (arr[mid] < x) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return result;
    }
    public static int findLastOccurrence(int[] arr, int x) {
        int left = 0, right = arr.length - 1;
        int result = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (arr[mid] == x) {
                result = mid;
                left = mid + 1;
            } else if (arr[mid] < x) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return result;
    }
    public static int[] findFirstAndLast(int[] arr, int x) {
        int[] result = new int[2];
        result[0] = findFirstOccurrence(arr, x);
        result[1] = findLastOccurrence(arr, x);
        return result;
    }
    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 5, 5, 5, 67, 123, 125};
        int x = 5;
        int[] result = findFirstAndLast(arr, x);
        System.out.println("First and last occurrences of " + x + ": [" + result[0] + ", " +
result[1] + "]");
    }
}
```

Output: First and last occurrences of 5: [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**.

Examples:

```
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.
```

```
Input: arr[] = [1, 1, 1]
Output: 0
Explanation: There are no 0s in the array, so the transition point is 0, indicating
that the first index (which contains 1) is also the first position of the array.
```

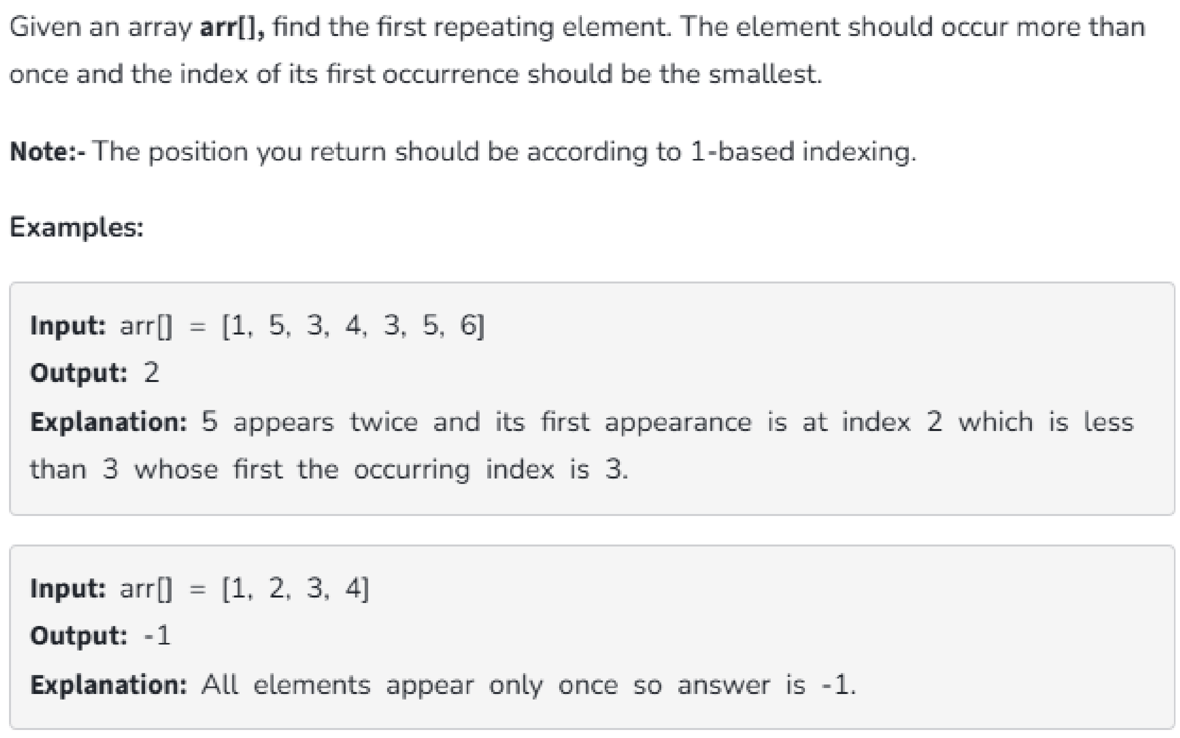
Code:

```
public class TransitionPoint {
    public static int findTransitionPoint(int[] arr) {
        int left = 0, right = arr.length - 1;
        int result = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == 1) {
                result = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        if (result == -1) {
            return arr[0] == 1 ? 0 : -1;
        }
        return result;
    }

    public static void main(String[] args) {
        int[] arr1 = {0, 0, 0, 1, 1};
        System.out.println("Transition point in arr1: " + findTransitionPoint(arr1));
    }
}
```

Output: Transition point in arr1: 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.

Examples:

```
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.
```

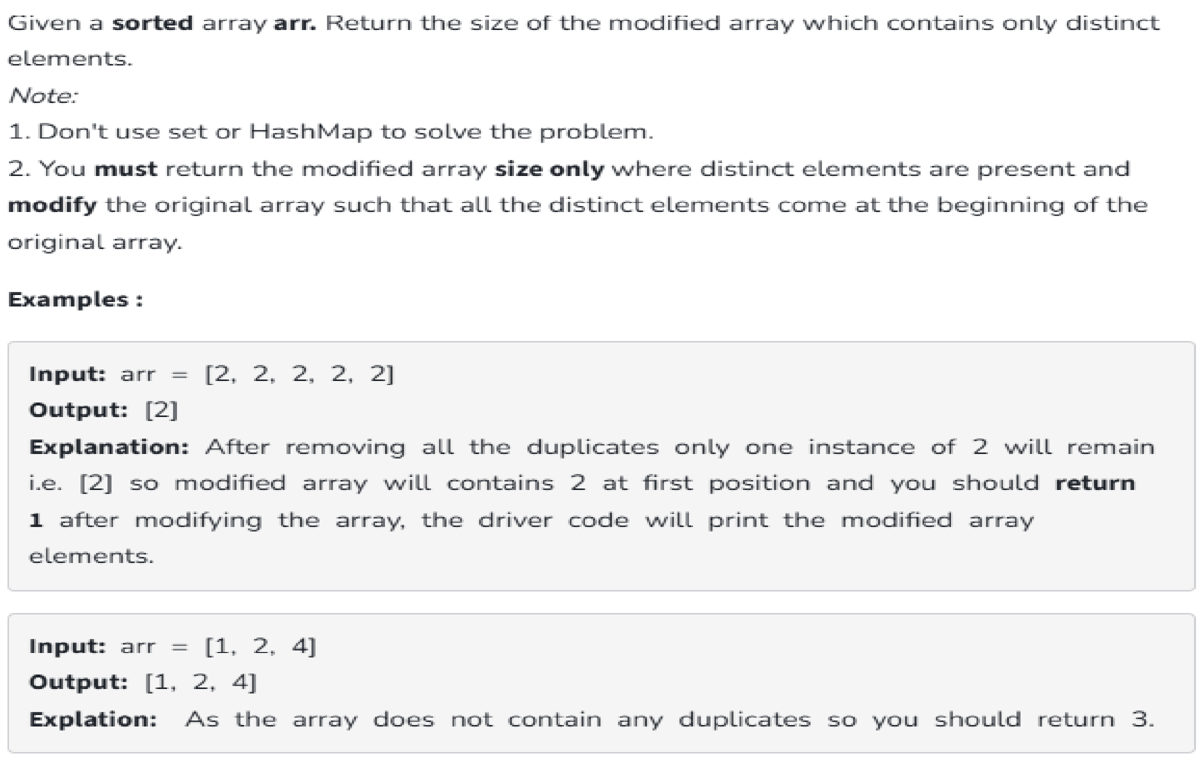
Code:

```
import java.util.HashMap;

public class Repeat {

    public static int findFirstRepeating(int[] arr) {
        HashMap<Integer, Integer> elementIndexMap = new HashMap<>();

        for (int i = 0; i < arr.length; i++) {
            if (elementIndexMap.containsKey(arr[i])) {
                return elementIndexMap.get(arr[i]) + 1;
            } else {
                elementIndexMap.put(arr[i], i);
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] arr1 = {1, 5, 3, 4, 3, 5, 6};
        System.out.println("First repeating element index in arr1: " + findFirstRepeating(arr1)); // Output: 2
    }
}
```

Output: First repeating element index in arr1: 3

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.

Note:

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.

Examples :

```
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:

```
public class Duplicate {
    public int removeDuplicates(int[] arr) {
        if (arr.length == 0) {
            return 0;
        }

        int index = 1;
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] != arr[i - 1]) {
                arr[index] = arr[i];
                index++;
            }
        }

        return index;
    }

    public static void main(String[] args) {
        Duplicate obj = new Duplicate();

        int[] arr = {2, 2, 2, 2, 2};
        int newSize = obj.removeDuplicates(arr);

        System.out.println("Modified array size: " + newSize);
        System.out.print("Modified array elements: ");
        for (int i = 0; i < newSize; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

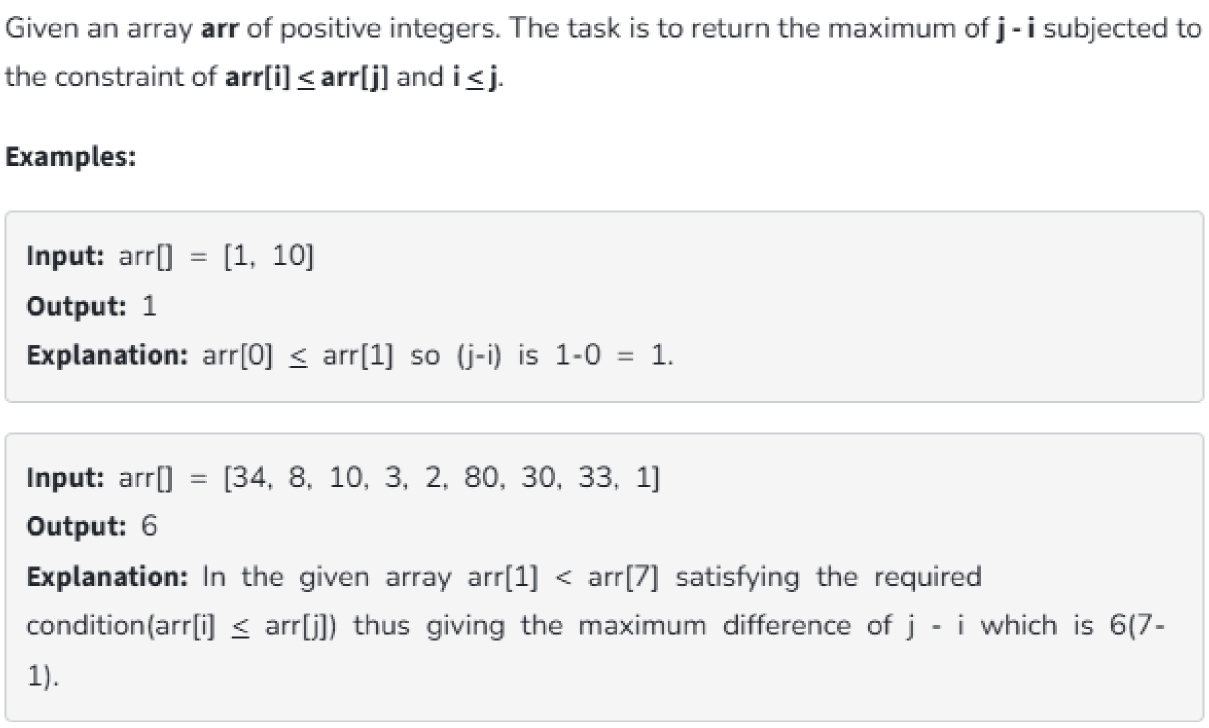
Output:

```
Modified array size: 1
```

Modified array elements: 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**.

Examples:

```
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).
```

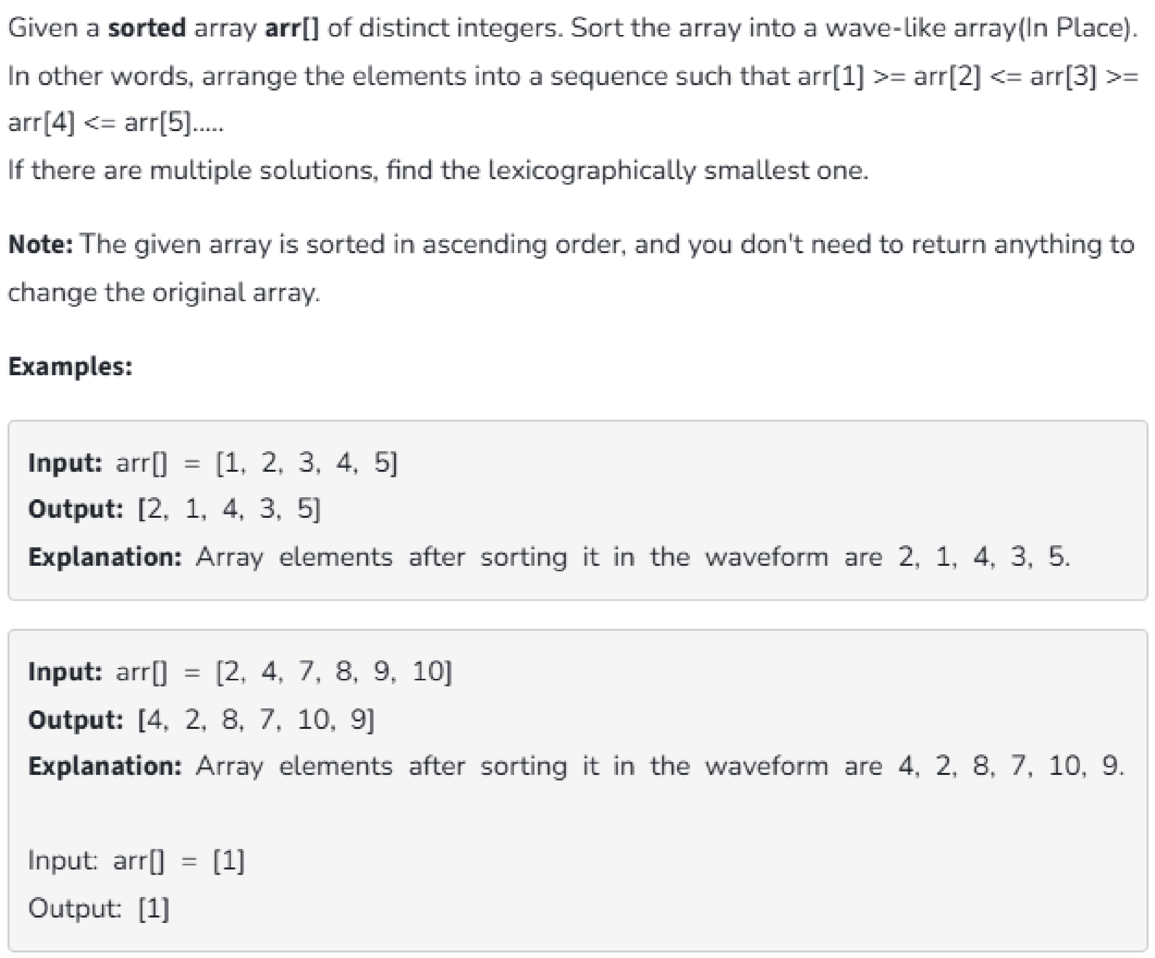
Code:

```
public class MaxIndex {
    public int maxIndexDiff(int[] arr) {
        int n = arr.length;


        int[] leftMin = new int[n];
        leftMin[0] = arr[0];
        for (int i = 1; i < n; i++) {
            leftMin[i] = Math.min(arr[i], leftMin[i - 1]);
        }


        int[] rightMax = new int[n];
        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, 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) {
        MaxIndex obj = new MaxIndex();
        int[] arr2 = {34, 8, 10, 3, 2, 80, 30, 33, 1};
        System.out.println("Maximum index difference: " + obj.maxIndexDiff(arr2));
    }
}
```

Output: Maximum index difference: 6

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.

Examples:

```
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:

```java
public class WaveSort {
    public void waveSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i += 2) {
            int temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }

    public static void main(String[] args) {
        WaveSort obj = new WaveSort();

        int[] arr1 = {1, 2, 3, 4, 5};
        obj.waveSort(arr1);
        for (int num : arr1) {
            System.out.print(num + " ");
        }
    }
}
```

Output: 2 1 4 3 5

Time Complexity:O(n)