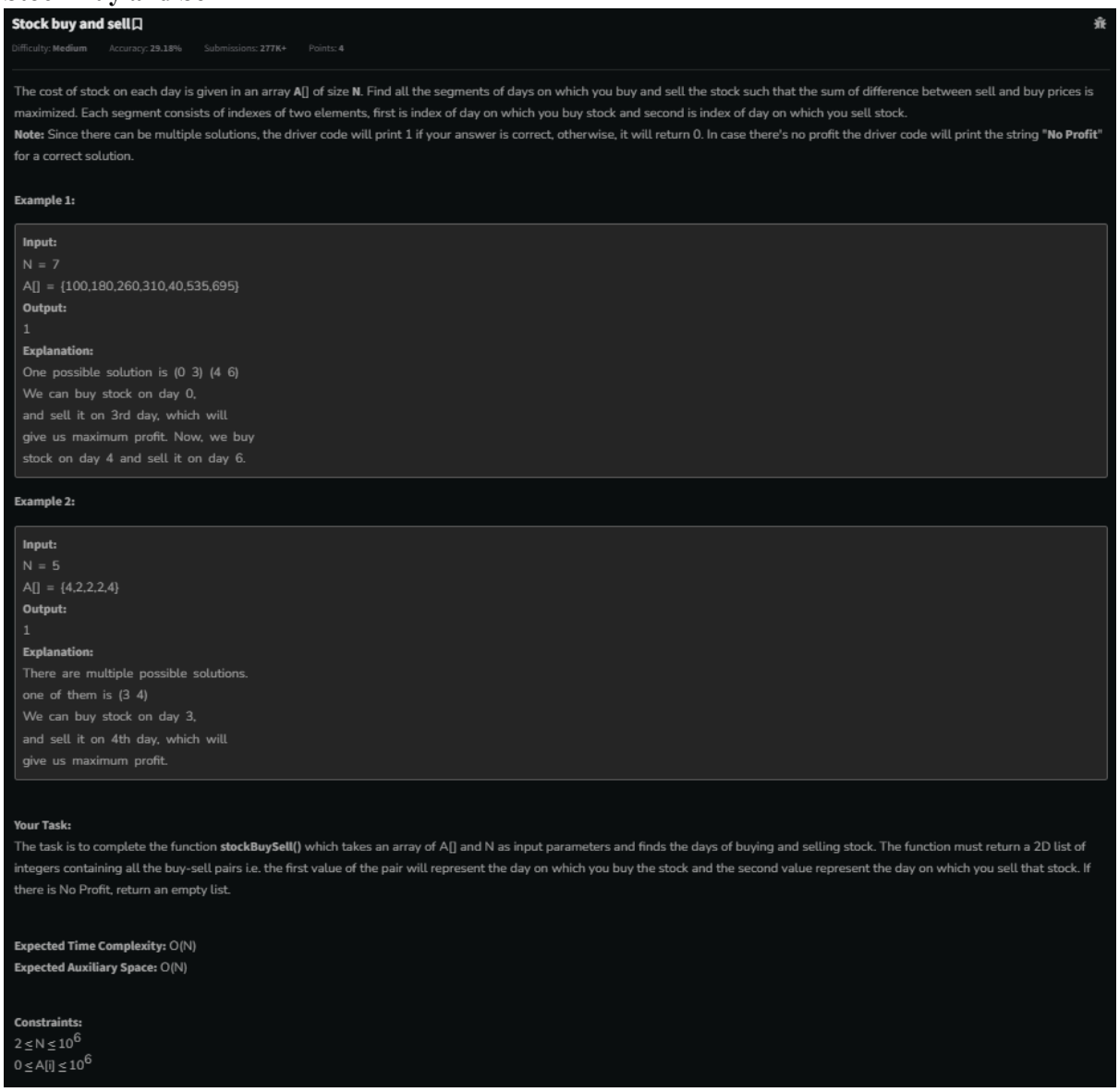
Coding Practice Set – 5

1. Stock Buy and Sell



Stock buy and sell

Difficulty: **Medium** Accuracy: **29.18%** Submissions: **277K+** Points: **4**

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.

Example 2:

Input:
N = 5
A[] = {4,2,2,2,4}
Output:
1
Explanation:
There are multiple possible solutions.
one of them is (3 4)
We can buy stock on day 3,
and sell it on 4th day, which will
give us maximum profit.

Your Task:
The task is to complete the function **stockBuySell()** which takes an array of A[] and N as input parameters and finds the days of buying and selling stock. The function must return a 2D list of integers containing all the buy-sell pairs i.e. the first value of the pair will represent the day on which you buy the stock and the second value represent the day on which you sell that stock. If there is No Profit, return an empty list.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

Constraints:
$2 \leq N \leq 10^6$
$0 \leq A[i] \leq 10^6$

Code:

```
import java.util.*;

public class StockBuyAndSell {
   public static int stockBuySell(int[] prices, int n) {
      List<int[]> result = new ArrayList<>();
      int buy = 0;
      for (int i = 1; i < n; i++) {
         if (prices[i] < prices[i - 1]) {
            if (buy < i - 1) {
               result.add(new int[]{buy, i - 1});
            }
            buy = i;
         }
      }
```

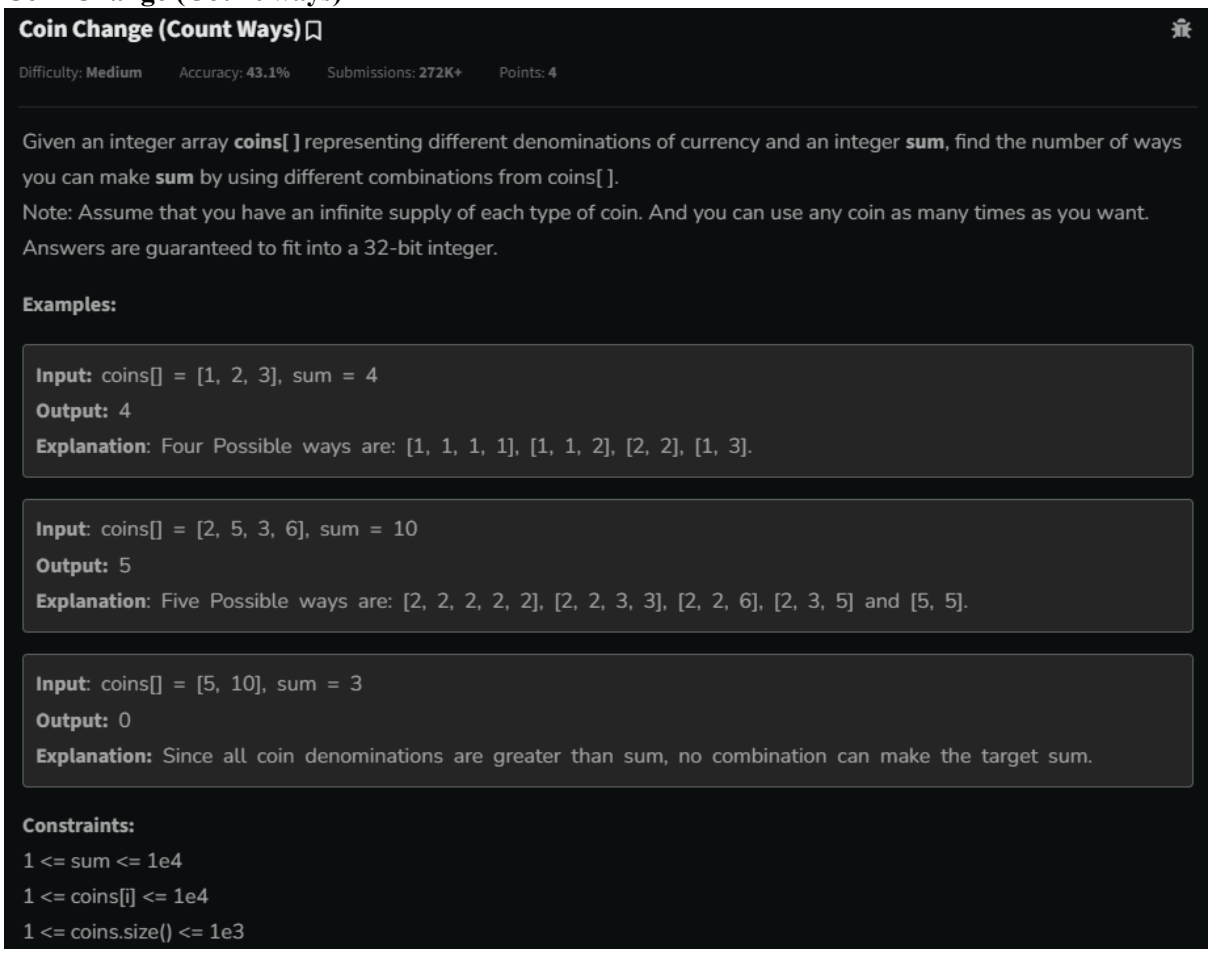
```
        if (buy < n - 1) {
            result.add(new int[]{buy, n - 1});
        }
        return result.isEmpty() ? 0 : 1;
    }

    public static void main(String[] args) {
        int[] prices = {100, 180, 260, 310, 40, 535, 695};
        int n = prices.length;
        int output = stockBuySell(prices, n);
        System.out.println(output);
    }
}
```

Output:

```
1
```

Time complexity: O(N)

2. **Coin Change (Count ways)**



Coin Change (Count Ways)

Difficulty: **Medium** Accuracy: **43.1%** Submissions: **272K+** Points: **4**

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].

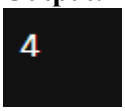
Input: coins[] = [5, 10], sum = 3
Output: 0
Explanation: Since all coin denominations are greater than sum, no combination can make the target sum.

Constraints:
1 <= sum <= 1e4
1 <= coins[i] <= 1e4
1 <= coins.size() <= 1e3

Code:

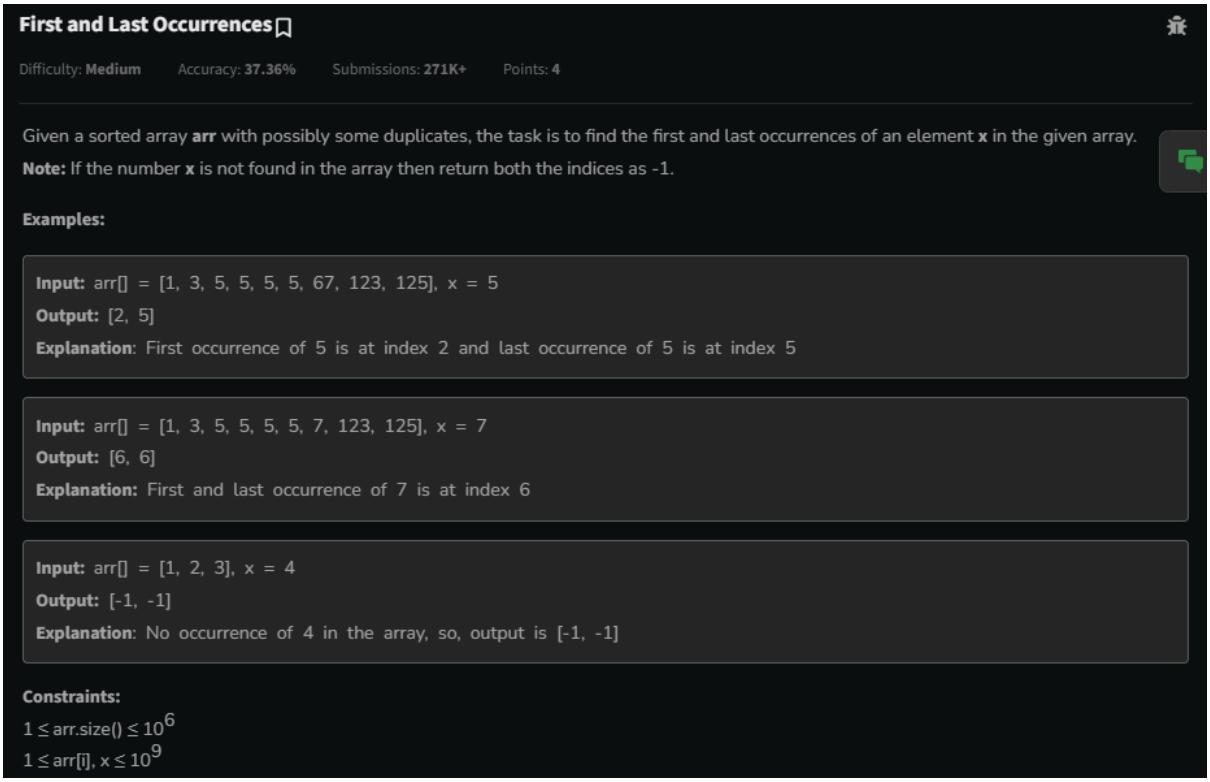
```
public class CoinChange {
    public static int countWays(int[] coins, int sum) {
        int[] dp = new int[sum + 1];
        dp[0] = 1;
        for (int coin : coins) {
            for (int j = coin; j <= sum; j++) {
                dp[j] += dp[j - coin];
            }
        }
        return dp[sum];
    }

    public static void main(String[] args) {
        int[] coins = {1, 2, 3};
        int sum = 4;
        System.out.println(countWays(coins, sum));
    }
}
```

Output:



Time Complexity: O(coins.size()*sum)

3. **First and last occurrences**



First and Last Occurrences

Difficulty: **Medium** Accuracy: **37.36%** Submissions: **271K+** Points: **4**

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]

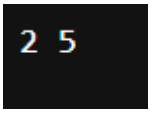
Constraints:
$1 \leq arr.size() \leq 10^6$
$1 \leq arr[i], x \leq 10^9$

Code:

```
public class FirstLastOccurrences {
    public static int[] findOccurrences(int[] arr, int x) {
        int first = findPosition(arr, x, true);
        int last = findPosition(arr, x, false);
        return new int[]{first, last};
    }

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

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 5, 5, 5, 67, 123, 125};
        int x = 5;
        int[] result = findOccurrences(arr, x);
        System.out.println(result[0] + " " + result[1]);
```

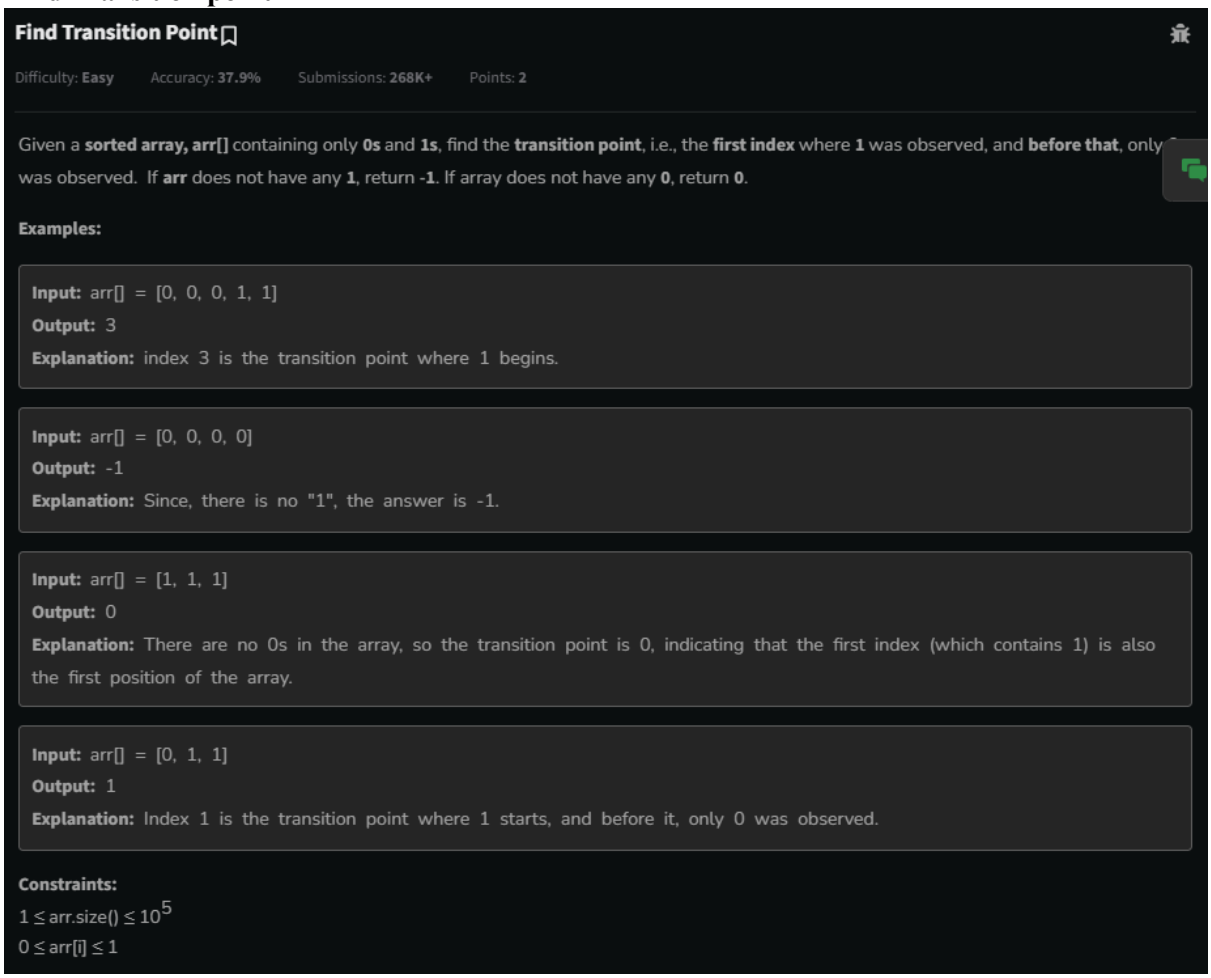
```
    }
}
```

Output:



Time Complexity: O(log n)

4. Find Transition point



Find Transition Point

Difficulty: **Easy** Accuracy: **37.9%** Submissions: **268K+** Points: **2**

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.

Input: arr[] = [0, 1, 1]
Output: 1
Explanation: Index 1 is the transition point where 1 starts, and before it, only 0 was observed.

Constraints:
$1 \leq arr.size() \leq 10^5$
$0 \leq arr[i] \leq 1$

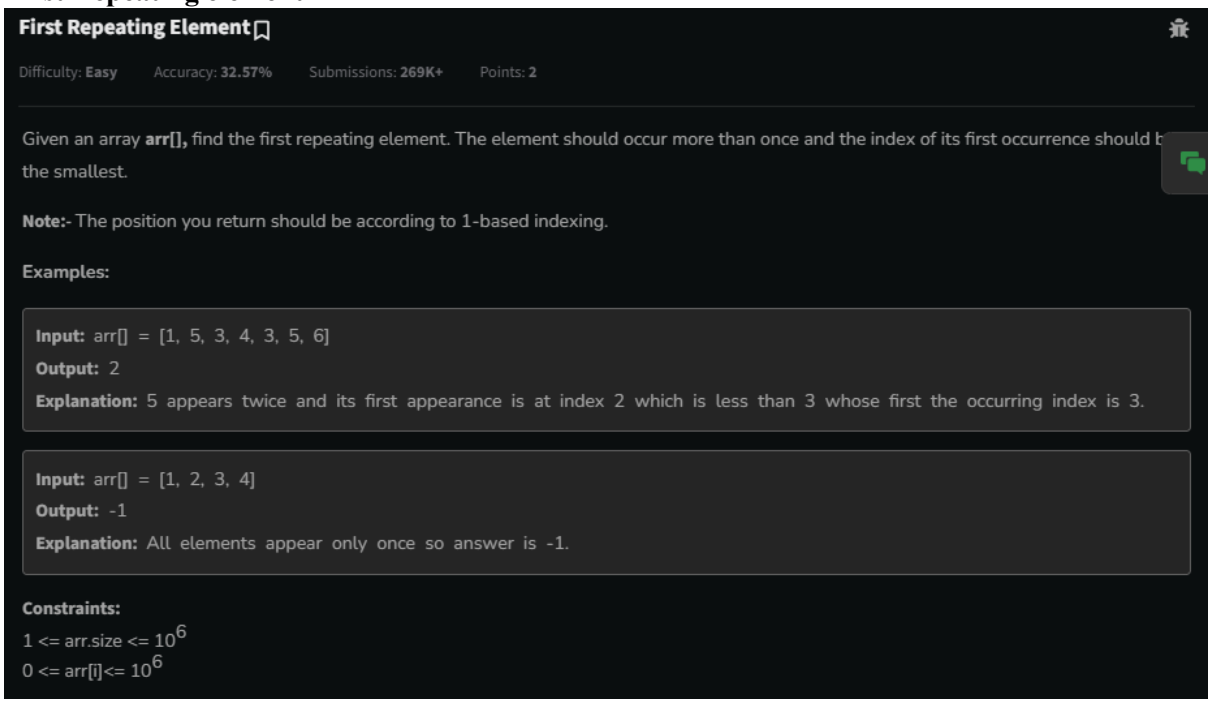
Code:

```
public class TransitionPoint {
    public static int findTransitionPoint(int[] arr) {
        int low = 0, high = arr.length - 1;
        int 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) {
        int[] arr = {0, 0, 0, 1, 1};
        System.out.println(findTransitionPoint(arr));
    }
}
```

Output:

Time Complexity: O(log n)

5. **First Repeating element**



Code:

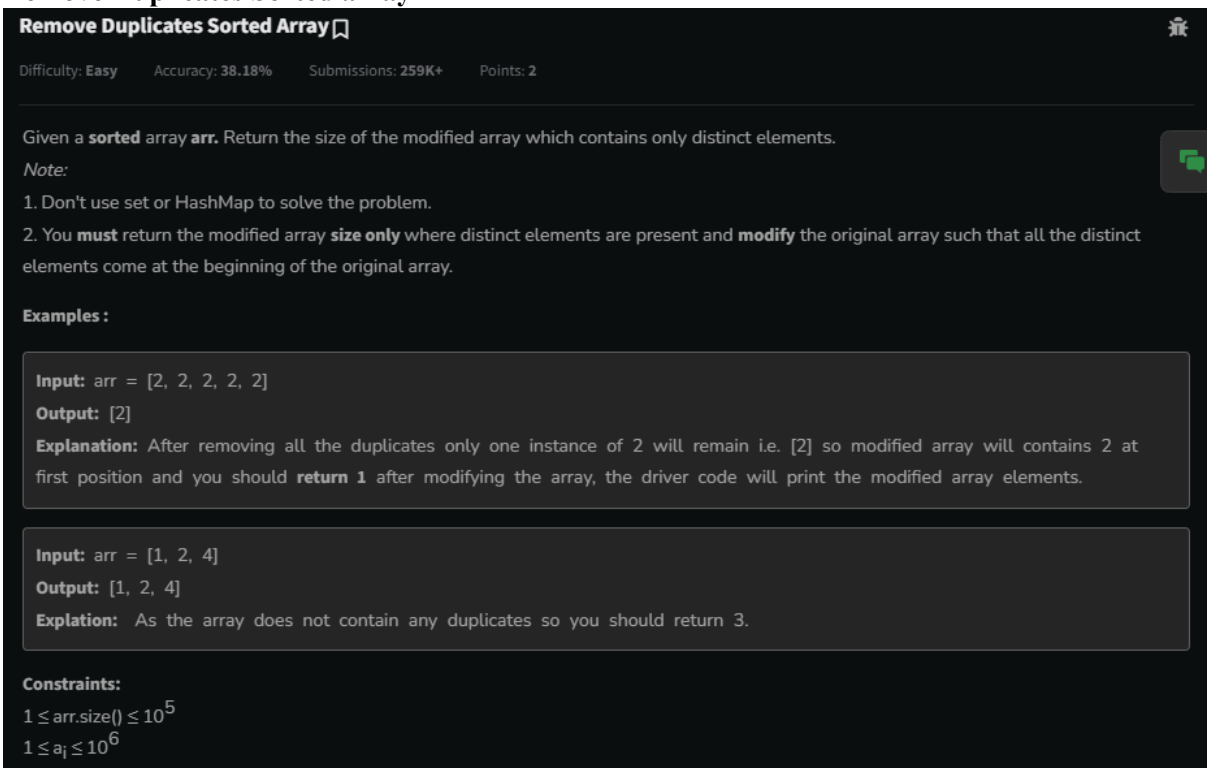
```
import java.util.*;

public class FirstRepeatingElement {
    public static int findFirstRepeatingElement(int[] arr) {
        Map<Integer, Integer> map = new HashMap<>();
        int minIndex = Integer.MAX_VALUE;
        for (int i = 0; i < arr.length; i++) {
            if (map.containsKey(arr[i])) {
                minIndex = Math.min(minIndex, map.get(arr[i]));
            } else {
                map.put(arr[i], i);
            }
        }
        return minIndex == Integer.MAX_VALUE ? -1 : minIndex + 1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 5, 3, 4, 3, 5, 6};
        System.out.println(findFirstRepeatingElement(arr));
    }
}
```

Output:

```
2
```

Time Complexity: O(n)

6. Remove Duplicates Sorted array



Remove Duplicates Sorted Array

Difficulty: **Easy** Accuracy: **38.18%** Submissions: **259K+** Points: **2**

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.

Constraints:

$1 \leq arr.size() \leq 10^5$

$1 \leq a_i \leq 10^6$

Code:

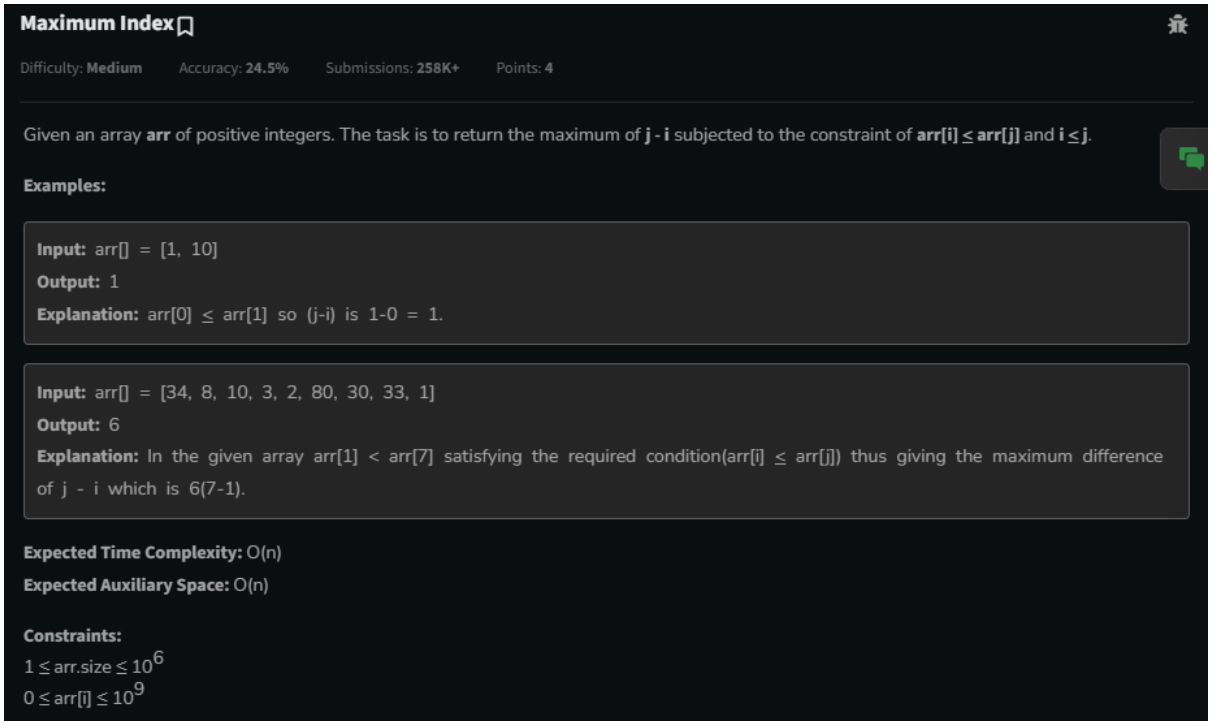
```
public class RemoveDuplicates {
    public static int removeDuplicates(int[] arr) {
        if (arr.length == 0) return 0;

        int index = 0;
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] != arr[index]) {
                index++;
                arr[index] = arr[i];
            }
        }
        return index + 1;
    }
    public static void main(String[] args) {
        int[] arr = {2, 2, 2, 2, 2};
        int size = removeDuplicates(arr);
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output:

```
2
```

Time Complexity: O(n)

7. Maximum Index



Maximum Index

Difficulty: **Medium** Accuracy: **24.5%** Submissions: **258K+** Points: **4**

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).

Expected Time Complexity: O(n)
Expected Auxiliary Space: O(n)

Constraints:
$1 \leq arr.size \leq 10^6$
$0 \leq arr[i] \leq 10^9$

Code:

```
public 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(leftMin[i - 1], arr[i]);
        }

        rightMax[n - 1] = arr[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            rightMax[i] = Math.max(rightMax[i + 1], arr[i]);
        }

        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;
    }
```

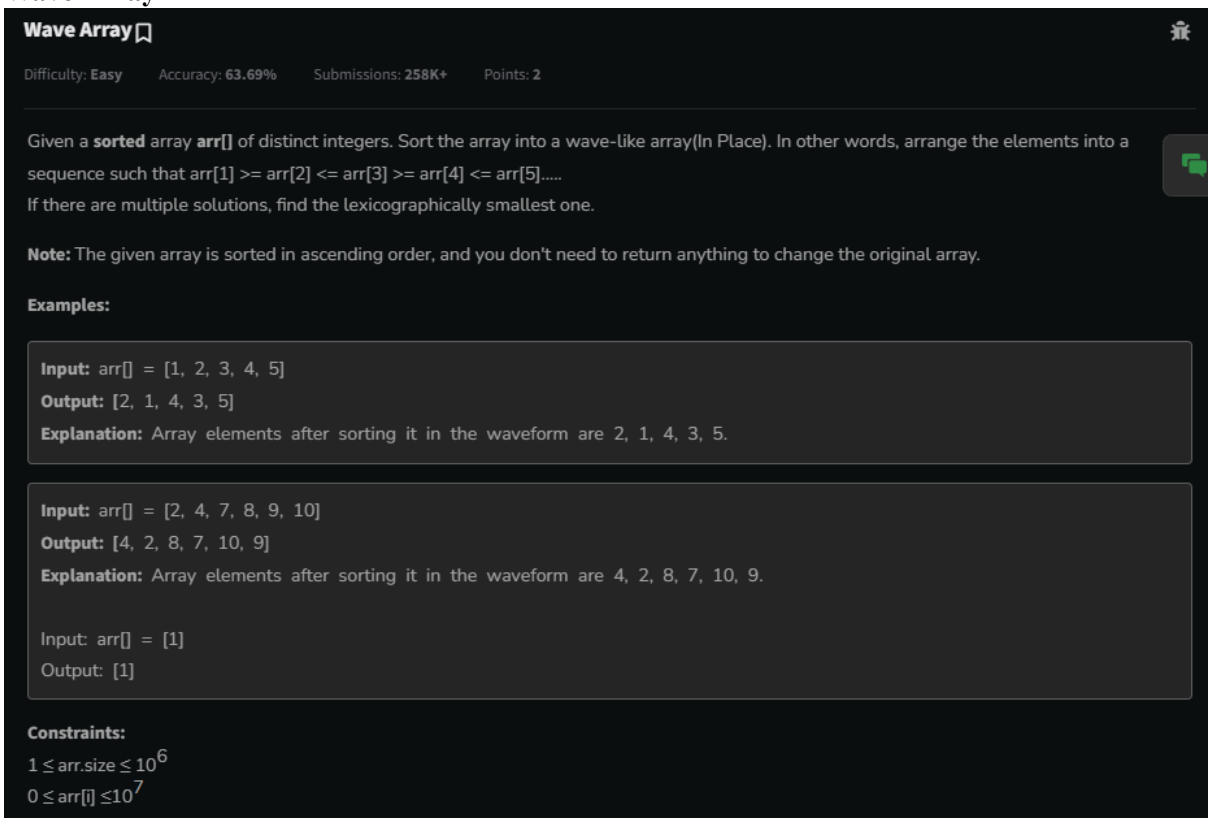
```
    public static void main(String[] args) {
        int[] arr = {34, 8, 10, 3, 2, 80, 30, 33, 1};
        System.out.println(maxIndexDiff(arr));
    }
}
```

Output:

```
6
```

Time Complexity: O(n)

8. Wave Array



Wave Array

Difficulty: **Easy** Accuracy: **63.69%** Submissions: **258K+** Points: **2**

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]

Constraints:
$1 \leq arr.size \leq 10^6$
$0 \leq arr[i] \leq 10^7$

Code:

```
public class WaveArray {
    public static void convertToWave(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) {
        int[] arr = {1, 2, 3, 4, 5};
        convertToWave(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
2 1 4 3 5
```

Time Complexity: O(n)