**Syed Nadeem G**

**Information Technology**

# SDE – Practice Day 5 (14/11/2024)

## 1. First and Last Occurances:

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:**

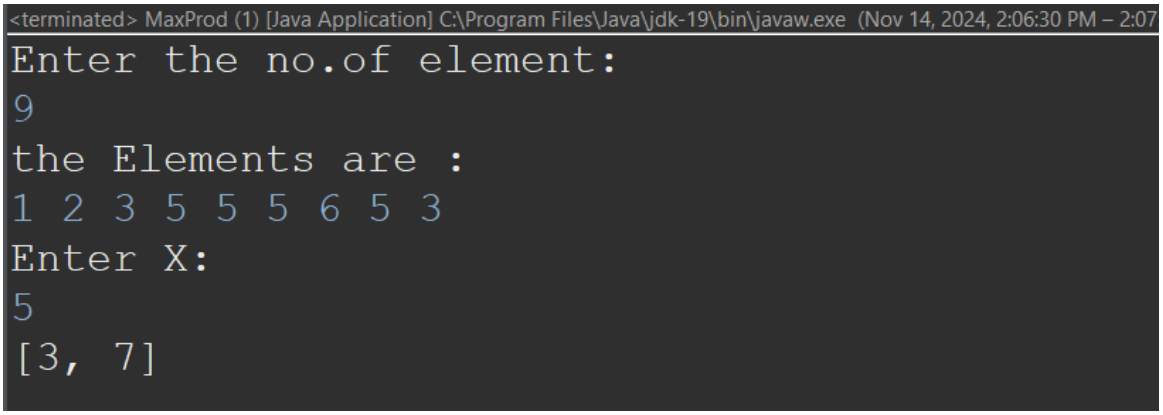
**Input:** arr[] = [1, 3, 5, 5, 5, 5, 67, 123, 125], x = 5
**Output:** [2, 5]
**Explanation:** First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5
**Input:** arr[] = [1, 3, 5, 5, 5, 5, 7, 123, 125], x = 7
**Output:** [6, 6]
**Explanation:** First and last occurrence of 7 is at index 6

CODE:

```
import java.util.*;
import java.math.*;
public class MaxProd {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no.of element:");
		int n = sc.nextInt();
		System.out.println("the Elements are :");
		int[] arr = new int[n];
		for(int i=0; i<n; i++) {
			arr[i] = sc.nextInt();
		}
		System.out.println("Enter X:");
```

```
                int x = sc.nextInt();
                ArrayList<Integer> ans = helper(arr, x);
        System.out.println(ans);
            }


            private static ArrayList<Integer> helper(int arr[], int x) {
        // code here
        ArrayList<Integer> ls = new ArrayList<>();
        int first = -1;
        int last = -1;
        for(int i=0; i<arr.length; i++){
          if(arr[i]!=x){
            continue;
          }
          else if(first == -1){
            first = i;
          }
          last = i;
        }
        ls.add(first);
        ls.add(last);
        return ls;
      }
}
```

Output:




Time Complexity:O(n)

## 2. First 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.

Code:

```
import java.util.*;
import java.math.*;
public class MaxProd {

        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the no.of element:");
                int n = sc.nextInt();
                System.out.println("the Elements are :");
                int[] arr = new int[n];
                for(int i=0; i<n; i++) {
                        arr[i] = sc.nextInt();
                }
                int ans = transitionPoint(arr);
    System.out.println(ans);
        }
  private static int transitionPoint(int arr[]) {
    int ind = -1;
    for(int i=0; i<arr.length; i++){
      if(arr[i]==1){
        ind = i;
        break;
      }
    }
    return ind;

  }
}
```

Output:

Time Complexity: 0(n)

## 3. First Repeating Character

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.*;
import java.math.*;
public class MaxProd {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no.of element:");
		int n = sc.nextInt();
		System.out.println("the Elements are :");
		int[] arr = new int[n];
		for(int i=0; i<n; i++) {
			arr[i] = sc.nextInt();
		}

		int ans = firstRepeated(arr);
	System.out.println(ans);
```

```
        }


        public static int firstRepeated(int[] arr) {
    // Your code here
    HashSet<Integer> set = new HashSet<>();
    int min = -1;
    for(int i=arr.length-1; i>=0; i--){
      if(set.contains(arr[i])){
        min = i;
      }

      set.add(arr[i]);

    }
    return min!=-1 ? min+1: -1;
  }

}
```

Output:

Time Complexity: O(n)

## 4. Remove Duplicates from 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:

```
import java.util.*;
import java.math.*;
public class MaxProd {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no.of element:");
		int n = sc.nextInt();
		System.out.println("the Elements are :");
		int[] arr = new int[n];
		for(int i=0; i<n; i++) {
			arr[i] = sc.nextInt();
		}
		ArrayList<Integer> ls = new ArrayList<>();
		for(int num:arr) {
			ls.add(num);
		}
		int ans = remove_duplicate(ls);
    System.out.println(ans);
	}


  public static int remove_duplicate(List<Integer> arr) {

    int i = 0, n = arr.size();

    for (int j = 1; j < n; j++) {

      if (!arr.get(j).equals(arr.get(i))) {
        i++;
        arr.set(i, arr.get(j));
      }
    }
    return i + 1;
```

```
    }

}
```

Output:

```
<terminated> MaxProd (1) [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe  (Nov 14, 2024,
Enter the no.of element:
6
the Elements are :
1 2 2 2 3 3
3
```

Time Complexity: O(n)

## 5. 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:**

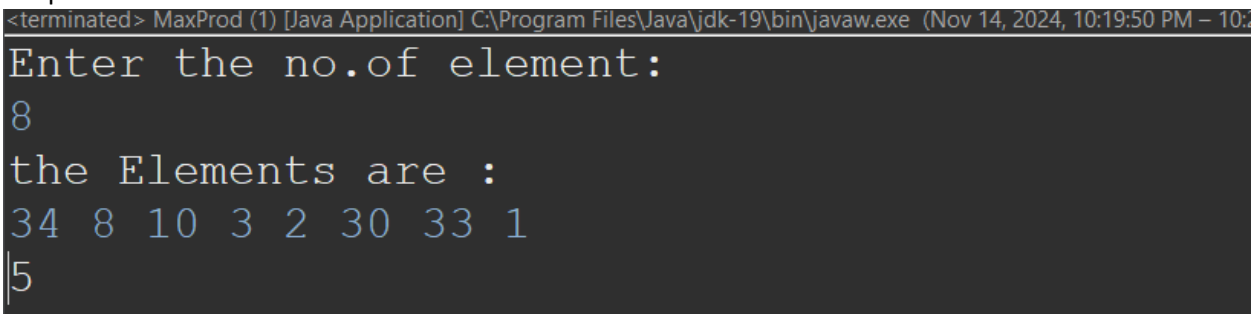
**Input:** arr[] = [1, 10]
**Output:** 1
**Explanation:** arr[0] ≤ arr[1] so (j-i) is 1-0 = 1.
**Input:** arr[] = [34, 8, 10, 3, 2, 80, 30, 33, 1]
**Output:** 6
**Explanation:** In the given array arr[1] < arr[7] satisfying the required condition(arr[i] ≤ arr[j]) thus giving the maximum difference of j - i which is 6(7-1).

Code:

```
import java.util.*;
import java.math.*;
public class MaxProd {
```

```
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the no.of element:");
                int n = sc.nextInt();
                System.out.println("the Elements are :");
                int[] arr = new int[n];
                for(int i=0; i<n; i++) {
                        arr[i] = sc.nextInt();
                }
                int ans = maximumInd(arr);
    System.out.println(ans);
        }


    public static int maximumInd(int[] arr) {
        int n = arr.length;
        if(n<=1) return 0;
        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 j=n-2; j>=0; j--) {
            rightMax[j] = Math.max(rightMax[j+1], arr[j]);
        }

        int i=0, j=0, dif = 0;
        while(i<n && j<n) {
            if(leftMin[i]<rightMax[j]) {
                    dif = Math.max(dif, j-1);
                    j++;
            }
            else {
                    i++;
            }
        }
        return dif;
    }

}
```

Output:




Time Complexity: O(3n) ➔ O(n)

## 6. 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.

Code:

```
import java.util.*;
import java.math.*;
public class MaxProd {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no.of element:");
		int n = sc.nextInt();
		System.out.println("the Elements are :");
		int[] arr = new int[n];
		for(int i=0; i<n; i++) {
			arr[i] = sc.nextInt();
		}
		waveArray(arr);
```

```
                for(int ans:arr)
                        System.out.print(ans+" ");
        }


    public static void waveArray(int[] arr) {
        int n = arr.length;
        for(int i=1; i<n; i+=2) {
            int t = arr[i];
            arr[i] = arr[i-1];
            arr[i-1] = t;
        }
        return;
    }

}
```

Output:

```
<terminated> MaxProd (1) [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe  (Nov 14, 2024, 10:27:32 PM – 10:27
Enter the no.of element:
5
the Elements are :
1 2 3 4 5
2 1 4 3 5
```

Time Complexity: 0(n)

## 7. Coin Change:

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

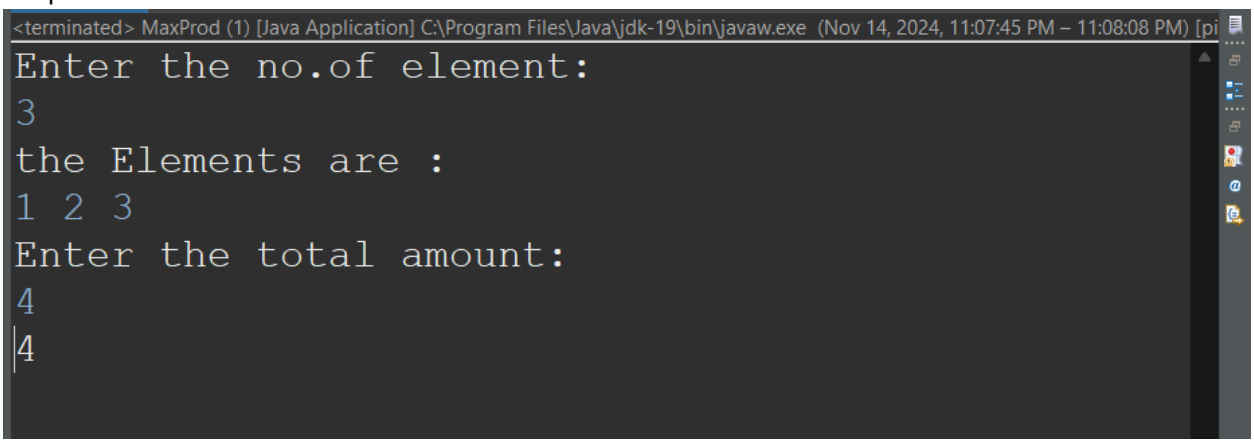
**Output:** 5

**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.*;
import java.math.*;
public class MaxProd {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no.of element:");
		int n = sc.nextInt();
		System.out.println("the Elements are :");
		int[] arr = new int[n];
		for(int i=0; i<n; i++) {
			arr[i] = sc.nextInt();
		}
		System.out.println("Enter the total amount: ");
		int amount = sc.nextInt();
		int ways = count(arr, amount);
		System.out.print(ways);
	}


	public static int count(int coins[], int sum) {
    int n = coins.length;
    int[][] dp = new int[n+1][sum+1];
    for(int i=0; i<=n; i++){
      dp[i][0] = 1;
    }
    for(int i=0; i<=sum; i++){
      dp[0][i] = 0;
    }

    for(int i=1; i<=n; i++){
      for(int j=1; j<=sum; j++){
        if(coins[i-1]<=j){
          dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]];
        }
        else{
          dp[i][j] = dp[i-1][j];
        }
      }
    }
    return dp[n][sum];
  }
}
```

Output:




Time complexity: O(n*sum)