CODING PRACTICE PROBLEMS

(Sowmya A) DATE:14/11/24

1.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

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.

CODE:

```
class Solution {
    public int remove_duplicate(List<Integer> arr) {
     if (arr == null || arr.size() == 0) {
            return 0;
       }
       int uniqueIndex = 0;
       for (int i = 1; i < arr.size(); i++) {
          if (!arr.get(i).equals(arr.get(uniqueIndex))) {
             uniqueIndex++;
             arr.set(uniqueIndex, arr.get(i));
          }
       }
       return uniqueIndex + 1;
    }
}
```

For Input:

2 2 2 2 2

Your Output:

2

Expected Output:

2

Time Complexity:O(n)

2. 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.

CODE:

```
class Solution {
    public static int firstRepeated(int[] arr) {
        HashMap<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;
    }
}
```

Time Complexity:O(n)

3. 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**.

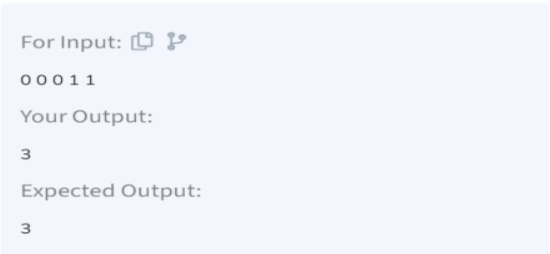
Input: arr[] = [0, 0, 0, 1, 1]

Output: 3

Explanation: index 3 is the transition point where 1 begins.

CODE:

```
class Solution {
   int transitionPoint(int arr[]) {
      int n=arr.length;
      for(int i=0;i<n;i++){
         if(arr[i]==1){
            return i;
         }
      }
      return -1;
   }
}
```



Time Complexity:O(n)

4. 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.

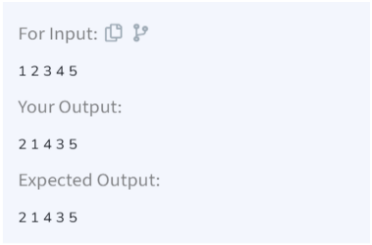
Input: arr[] = [1, 2, 3, 4, 5]

Output: [2, 1, 4, 3, 5]

Explanation: Array elements after sorting it in the waveform are 2, 1, 4, 3, 5.

CODE:

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



Time Complexity:O(n)

5.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]

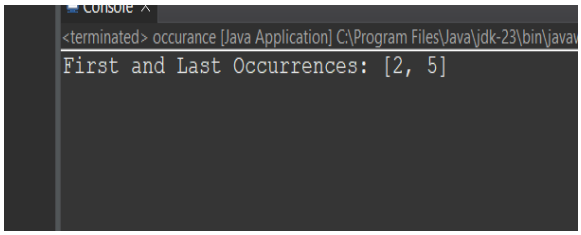
Explanation: First occurrence of 5 is at index 2 and last occurrence of 5 is at index 5

CODE:

```
package util;

public class occurance {
```

```
public static int[] findFirstAndLast(int[] arr, int x) {
    int[] result = {-1, -1};
    result[0] = findOccurrence(arr, x, true);
    result[1] = findOccurrence(arr, x, false);
    return result;
}
private static int findOccurrence(int[] arr, int x, boolean findFirst) {
    int low = 0, high = arr.length - 1;
    int 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 = findFirstAndLast(arr, x);
    System.out.println("First and Last Occurrences: [" + result[0] + ", " + result[1] + "]");
```

```
    }
}
```



Time Complexity:O(logn)

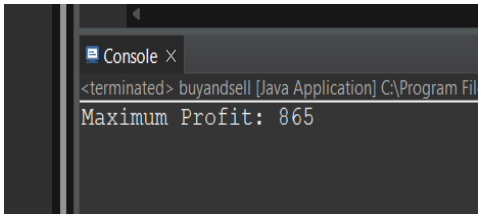
6. The cost of stock on each day is given in an array **price[]**. Each day you may decide to either buy or sell the stock i at **price[i]**, you can even buy and sell the stock on the same day. Find the **maximum profit** that you can get.

Note: A stock can only be sold if it has been bought previously and multiple stocks cannot be held on any given day.

Input: prices[] = [100, 180, 260, 310, 40, 535, 695]
Output: 865
Explanation: Buy the stock on day 0 and sell it on day 3 => 310 – 100 = 210. Buy the stock on day 4 and sell it on day 6 => 695 – 40 = 655. Maximum Profit = 210 + 655 = 865.

CODE:

```
package util;

public class buyandsell {
        public static int maxProfit(int[] prices) {
            int maxProfit = 0;
            for (int i = 1; i < prices.length; i++) {
                if (prices[i] > prices[i - 1]) {
                    maxProfit += prices[i] - prices[i - 1];
                }
            }
            return maxProfit;
        }
        public static void main(String[] args) {
            int[] prices = {100, 180, 260, 310, 40, 535, 695};
            System.out.println("Maximum Profit: " + maxProfit(prices));
        }
    }
```



Time Complexity:O(n)

7. 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

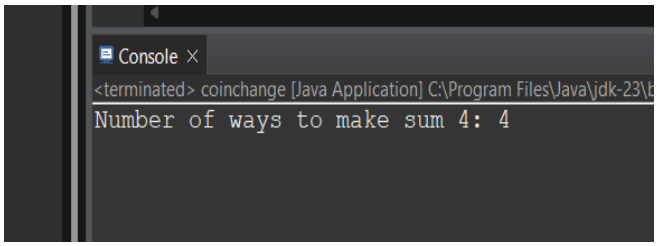
Output: 4

Explanation: Four Possible ways are: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3].

CODE:

```
package util;

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 i = coin; i <= sum; i++) {
		         dp[i] += dp[i - coin];
		      }
		   }
		   return dp[sum];
		}
		public static void main(String[] args) {
		   int[] coins = {1, 2, 3};
		   int sum = 4;
		   System.out.println("Number of ways to make sum " + sum + ": " + countWays(coins, sum));
		}
```

```
    }
```



TimeComplexity:O(n*sum)

8. 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]

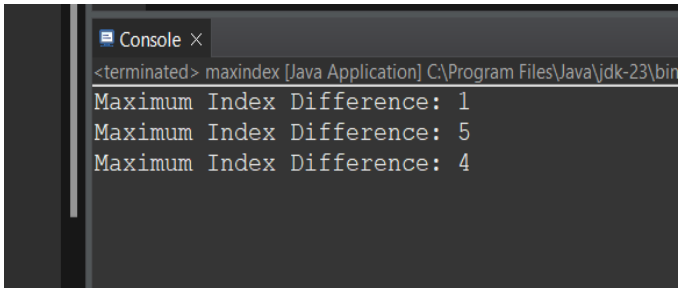
Output: 1

Explanation: arr[0] ≤ arr[1] so (j-i) is 1-0 = 1.

CODE:

```
package util;

public class maxindex{

   public static int maxIndexDiff(int[] arr) {

      int n = arr.length;

      if (n < 2) 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(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, 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[] arr1 = {1, 10};
    System.out.println("Maximum Index Difference: " + maxIndexDiff(arr1));
    int[] arr2 = {1, 2, 3, 4, 5, 6};
    System.out.println("Maximum Index Difference: " + maxIndexDiff(arr2));
    int[] arr3 = {7, 1, 3, 4, 5, 2};
    System.out.println("Maximum Index Difference: " + maxIndexDiff(arr3));
  }
}
```



Time Complexity:O(n)