GATE TECHNICAL TRAINING – DSA CODING PRACTICE PROBLEMS 2026

DATE: 14-11-24 **NAME: S YATHISSH – CSBS**

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.

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:
$2 \leq N \leq 106$
$0 \leq A[i] \leq 106$

CODE:

```
package JavaDSA;

import java.util.Scanner;

public class BuyAndSellStocks {
	public static int maximumProfit(int nums[])
		{ int lMin=nums[0];
		int lMax=nums[0];
		int i=0;
		int res=0;
		int n=nums.length;

		while(i<n-1) {
			while(i<n-1 && nums[i]>=nums[i+1])
				{ i++;
			}
			lMin = nums[i];

			while(i<n-1 && nums[i]<=nums[i+1])
				{ i++;
			}
			lMax = nums[i];

			res+=(lMax - lMin);

		}
		return res;
	}
	public static void main(String args[])
		{ Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no. of elements: ");
		int n = sc.nextInt();

		int nums[] = new int[n];
```

```
            System.out.println("Enter the elements in the array: ");
            for(int i=0; i<n; i++) {
                  nums[i]= sc.nextInt();
            }


            System.out.println("The Maximum Profit is " + maximumProfit(nums));

      }

}
```

OUTPUT:

```
Enter the no. of elements:
7
Enter the elements in the array:
100
180
260
310
40
535
695
The Maximum Profit is 865
```

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].

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:

```java
package JavaDSA;

import java.util.Scanner;

public class CountChange {
    static long count(int coins[], int n, int sum)
        { int dp[] = new int[sum + 1];
        dp[0] = 1;

        for (int i = 0; i < n; i++)
            for (int j = coins[i]; j <= sum; j++)
                dp[j] += dp[j - coins[i]];

        return dp[sum];
    }

    public static void main(String[] args)
        { Scanner scanner = new
        Scanner(System.in);

        System.out.print("Enter the number of coins: ");
        int n = scanner.nextInt();

        int[] coins = new int[n];
        System.out.println("Enter the values of the coins: ");
        for (int i = 0; i < n; i++) {
            coins[i] = scanner.nextInt();
        }

        System.out.print("Enter the target sum: ");
        int sum = scanner.nextInt();

        long ways = count(coins, n, sum);
        System.out.println("Number of ways to make the sum " + sum + " is: " + ways);
    }
```

OUTPUT:

```
Enter the number of coins: 3
Enter the values of the coins:
1
2
3
Enter the target sum: 4
Number of ways to make the sum 4 is: 4
```

TIME COMPLEXITY: O(n)

3. FIRST AND LAST OCCURENCES

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:
$1 \leq arr.size() \leq 106$
$1 \leq arr[i], x \leq 109$

CODE:

```
package JavaDSA;
import java.util.ArrayList;
import java.util.Scanner;

public class FirstLastOccurence {
       public static ArrayList<Integer> findFirstAndLast(int[] arr, int x)
        { ArrayList<Integer> result = new ArrayList<>();
        int n = arr.length;
        int first = -1, last = -1;

        for (int i = 0; i < n; i++)
            { if (x != arr[i])
                continue;

            if (first == -1)
                first = i;
            last = i;
        }

        result.add(first);
        result.add(last);

        return result;
    }
```

```
    public static void main(String[] args)
        { Scanner scanner = new
        Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        System.out.print("Enter the value of x to search for: ");
        int x = scanner.nextInt();

        ArrayList<Integer> result = findFirstAndLast(arr, x);

        System.out.println("First occurrence: " + result.get(0));
        System.out.println("Last occurrence: " + result.get(1));
    }

}
```

OUTPUT:

```
Enter the number of elements in the array: 7
Enter the elements of the array:
1
6
5
4
8
5
9
Enter the value of x to search for: 5
First occurrence: 2
Last occurrence: 5
```

TIME COMPLEXITY: O(n)

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.

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:
$1 \leq arr.size() \leq 105$
$0 \leq arr[i] \leq 1$

CODE:

```java
package JavaDSA;
import java.util.Scanner;

public class TransitionPoint {

    public static int transitionPoint(int arr[])
        { for (int i = 0; i < arr.length; i++) {
            if (arr[i] == 1)
                { return i;
            }
        }
        return -1;
    }

    public static void main(String[] args)
        { Scanner scanner = new
        Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        int result = transitionPoint(arr);
        if (result != -1) {
            System.out.println("The transition point is at index: " + result);
        } else {
            System.out.println("There is no transition point.");
        }
    }
```

OUTPUT:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
0
0
0
1
1
The transition point is at index: 3
```

```
Enter the number of elements in the array: 4
Enter the elements of the array:
0
0
0
0
There is no transition point.
```

TIME COMPLEXITY: OIn)

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.

Constraints:
1 <= arr.size <= 106
0 <= arr[i]<= 106

CODE:

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

public class FirstRepeatingElement {
```

```
    public static int firstRepeated(int[] arr)
        { int min = -1;
        HashSet<Integer> set = new HashSet<>();
        for (int i = arr.length - 1; i >= 0; i--) {
            if (set.contains(arr[i]))
                { min = i;
            } else {
                set.add(arr[i]);
            }
        }
        return (min == -1) ? -1 : min + 1;
    }

    public static void main(String[] args)
        { Scanner scanner = new
        Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        int result = firstRepeated(arr);
        if (result == -1) {
            System.out.println("No repeating element found.");
        } else {
            System.out.println("The position of the first repeating element is: " +
result);
        }

        scanner.close();
    }
```

OUTPUT:

```
Enter the number of elements in the array: 7
Enter the elements of the array:
1
5
3
4
3
5
6
The position of the first repeating element is: 2
```

TIME COMPLEXITY: O(n)

6. REMOVES 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.

Constraints:
$1 \leq arr.size() \leq 105$
$1 \leq ai \leq 106$

CODE:

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

public class RemoveDuplicates {

          public static int remove_duplicate(int[] arr)
              { HashSet<Integer> s = new HashSet<>();

              int idx = 0;

              for (int i = 0; i < arr.length; i++)
                  { if (!s.contains(arr[i])) {
                      s.add(arr[i]);
                      arr[idx++] = arr[i];
                  }
              }
              return idx;
          }

          public static void main(String[] args)
              { Scanner scanner = new Scanner(System.in);

              System.out.print("Enter the size of the array: ");
              int size = scanner.nextInt();

              int[] arr = new int[size];

              System.out.println("Enter the elements of the array: ");
              for (int i = 0; i < size; i++) {
                  arr[i] = scanner.nextInt();
              }
```

```
                int newLength = remove_duplicate(arr);

                System.out.println("Array after removing duplicates:");
                for (int i = 0; i < newLength; i++) {
                    System.out.print(arr[i] + " ");
                }
    }
}
```

OUTPUT:

```
Enter the size of the array: 5
Enter the elements of the array:
2
2
3
3
4
Array after removing duplicates:
2 3 4
```

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).

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

Constraints:
1 ≤ arr.size ≤ 106
0 ≤ arr[i] ≤ 109

CODE:

```
package JavaDSA;
import java.util.Scanner;

public class FindMaximum {

	int max(int x, int y) { return x > y ? x :
		y;
	}

	int min(int x, int y) { return x < y ? x :
		y;
	}

	int maxIndexDiff(int arr[], int n) { int maxDiff;
		int i, j;

		int RMax[] = new int[n]; int
		LMin[] = new int[n];

		LMin[0] = arr[0];
		for (i = 1; i < n; ++i)
			LMin[i] = min(arr[i], LMin[i - 1]);

		RMax[n - 1] = arr[n - 1];
		for (j = n - 2; j >= 0; --j)
			RMax[j] = max(arr[j], RMax[j + 1]);

		i = 0;
		j = 0;
		maxDiff = -1;
		while (j < n && i < n) {
			if (LMin[i] <= RMax[j]) {
				maxDiff = max(maxDiff, j - i); j = j + 1;
			} else {
				i = i + 1;
			}
		}

		return maxDiff;
	}

	public static void main(String[] args) { Scanner scanner = new
		Scanner(System.in);

		System.out.print("Enter the number of elements in the array: "); int n = scanner.nextInt();

		int[] arr = new int[n];
		System.out.println("Enter the elements of the array: "); for (int i = 0; i < n; i++)
		{
			arr[i] = scanner.nextInt();
		}

		FindMaximum max = new FindMaximum(); int
		maxDiff = max.maxIndexDiff(arr, n);
		System.out.println("The maximum index difference is: " + maxDiff);
	}
}
```

OUTPUT:

```
Enter the number of elements in the array: 2
Enter the elements of the array:
1
10
The maximum index difference is: 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.

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:
$1 \leq arr.size \leq 106$
$0 \leq arr[i] \leq 107$

CODE:

```java
package JavaDSA;
import java.util.Scanner;
import java.util.Arrays;

public class WaveArray {
      public static void convertToWave(int[] arr) {
        sorting(arr);
    }

    private static void swap(int arr[], int a, int b)
        { int temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
```

```
    }

    private static void sorting(int arr[])
        { int n = arr.length;
        for (int i = 0; i < n; i += 2) {
            if (i > 0 && arr[i - 1] > arr[i]) {
                swap(arr, i, i - 1);
            }
            if (i < n - 1 && arr[i + 1] > arr[i]) {
                swap(arr, i, i + 1);
            }
        }
    }

       public static void main(String[] args)
        { Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] arr = new int[n];
        System.out.println("Enter the elements of the array: ");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        convertToWave(arr);

        System.out.println("The Waved Array is: "+ Arrays.toString(arr));
}}
```

OUTPUT:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
1
2
3
4
5
The Waved Array is: [2, 1, 4, 3, 5]
```

TIME COMPLEXITY: O(n)