Date: 14.11.2024

Sri Vaigunth.M-22AD134

**DSA Coding Problems**

**Set – 5**

1. **Remove Duplicates Elements in the List:**

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 = [1, 2, 4]

**Output:** [1, 2, 4]  
**Explation:** As the array does not contain any duplicates so you should return 3.

**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));

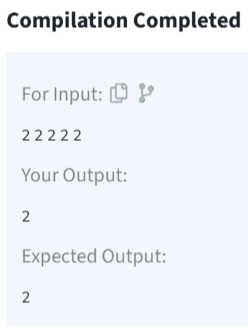
}

}

return uniqueIndex + 1;

}

}



**Time Complexity: O(n)**

1. **First Repeating Element:**

Given an array **arr[],** find the first repeating element. The element should occur ore 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 {

// Function to return the position of the first repeating element.

public static int firstRepeated(int[] arr) {

HashMap<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < arr.length; i++) {

map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);

}

for (int i = 0; i < arr.length; i++) {

if (map.get(arr[i]) > 1) {

return i + 1;

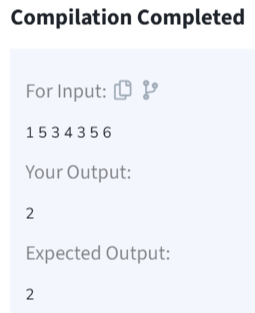
}

}

return -1;

}

}



**Time Complexity: O(n)**

1. **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**.

**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[]) {

// code here

if ((arr.length==1) && (arr[0]==1)){

return 0;

}

else if ((arr.length==1)&&(arr[0]==0)){

return -1;

}

for(int i=0;i<arr.length-1;i++){

if (arr[i]==1){

return i;

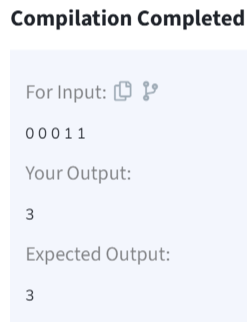
}

}

return -1;

}

}



**Time Complexity: O(n)**

1. **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

**Code:**

class Solution {

public static void convertToWave(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i += 2) {

if (i + 1 < n && arr[i] < arr[i + 1]) {

int temp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = temp;

}

if (i - 1 >= 0 && arr[i] < arr[i - 1]) {

int temp = arr[i];

arr[i] = arr[i - 1];

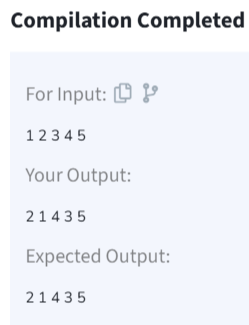
arr[i - 1] = temp;

}

}

}

}



**Time Complexity:O(n)**

1. **Stock buy and Sell:**

Given an array **prices[]**of size **n**denoting the cost of stock on each day, the task is to find the maximum total profit if we can buy and sell the stocks any number of times.

**Note:**We can only sell a stock which we have bought earlier and we cannot hold multiple stocks on any 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:**

class stock {

static int maximumProfit(int[] prices) {

int n = prices.length;

int lMin = prices[0]; // Local Minima

int lMax = prices[0]; // Local Maxima

int res = 0;

int i = 0;

while (i < n - 1) {

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

lMin = prices[i];

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

lMax = prices[i];

res += (lMax - lMin);

}

return res;

}

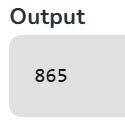
public static void main(String[] args) {

int[] prices = {100, 180, 260, 310, 40, 535, 695};

System.out.println(maximumProfit(prices));

}

}



**Time Complexity:O(n)**

1. **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.

**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:**

class Solution {

public int count(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];

}

}

****

**Time Complexity:O(n\*s)**

1. **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.

**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:**

class GFG {

ArrayList<Integer> find(int arr[], int x) {

int[] a=new int[2];

a[0]=-1;a[1]=-1;boolean falg=true;

for(int i=0;i<arr.length;i++){

if(arr[i]==x){

if(falg){

a[0]=i;a[1]=i;

falg=false;

}

else a[1]=i;

}

}

ArrayList<Integer> list = new ArrayList<>();

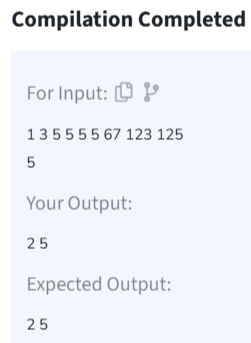
list.add(a[0]);

list.add(a[1]);

return list;

}

}

****

**Time Complexity:O(n)**

1. **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**.

**Input:** arr[] = [1, 10]

**Output:** 1

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

**Code:**

class Solution {

// Function to find the maximum index difference.

int maxIndexDiff(int[] arr) {

int n = arr.length;

int[] left = new int[n];

int[] right = new int[n];

int ans = -1;

left[0] = arr[0];

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

left[i] = Math.min(arr[i], left[i-1]);

}

right[n-1] = arr[n-1];

for(int i = n-2; i>=0; i--){

right[i] = Math.max(arr[i], right[i+1]);

}

int i =0,j=0;

while(i<n && j<n){

if(left[i]<=right[j]){

ans = Math.max(j-i,ans);

j++;

}

else{

i++;

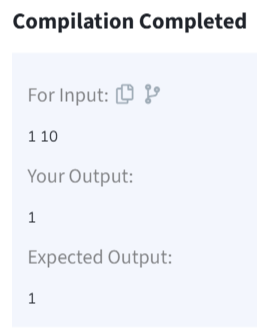
}

}

return ans;

}

}



**Time Complexity:O(n)**