# **Sahana Lakshmipathy\_AI&DS\_DSA\_Day-5**

**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,

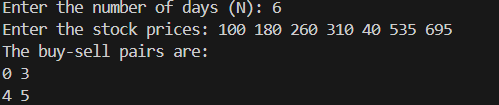
and sell it on 3rd day, which will

give us maximum profit. Now, we buy

stock on day 4 and sell it on day 6.

**Program:**

class Solution{  
 //Function to find the days of buying and selling stock for max profit.  
 ArrayList<ArrayList<Integer> > stockBuySell(int A[], int n) {  
 // code here  
   
 ArrayList<ArrayList<Integer>> result = new ArrayList<>();  
  
 // Traverse through the array  
 int i = 0;  
 while (i < n - 1) {  
 // Find local minima (buy day)  
 while (i < n - 1 && A[i + 1] <= A[i]) {  
 i++;  
 }  
  
 // If we reach the end, break  
 if (i == n - 1) {  
 break;  
 }  
   
 int buy = i++;  
   
 // Find local maxima (sell day)  
 while (i < n && A[i] >= A[i - 1]) {  
 i++;  
 }  
   
 int sell = i - 1;  
  
 // Add the pair of buy and sell days  
 ArrayList<Integer> transaction = new ArrayList<>();  
 transaction.add(buy);  
 transaction.add(sell);  
 result.add(transaction);  
 }  
   
 return result;  
 }  
}



**Time Complexity: O(n)**

**2. 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:

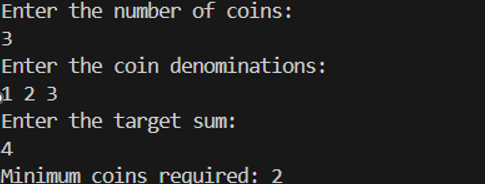
Input: coins[] = [1, 2, 3], sum = 4

Output: 4

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

**Program:**

class Solution {  
 public int count(int coins[], int sum) {  
 // code here.  
 // Create a dp array to store the number of ways to make each amount up to `sum`  
 int[] dp = new int[sum + 1];  
   
 // Base case: there's one way to make a sum of 0, which is to use no coins  
 dp[0] = 1;  
   
 // Iterate over each coin  
 for (int coin : coins) {  
 // For each coin, update the dp array for all amounts from coin to sum  
 for (int i = coin; i <= sum; i++) {  
 dp[i] += dp[i - coin];  
 }  
 }  
   
 // The answer will be in dp[sum], the number of ways to make `sum`  
 return dp[sum];  
 }  
}



**Time Complexity: O(m x sum)**

**3.First and Last Occurrence**

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

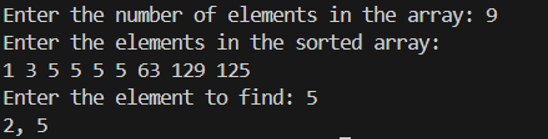
Output: [2, 5]

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

**Program:**

class GFG {  
 ArrayList<Integer> find(int arr[], int x) {  
 // code here  
 ArrayList<Integer> result = new ArrayList<>();  
   
 int first = findOccurrence(arr, x, true); // Find first occurrence  
 int last = findOccurrence(arr, x, false); // Find last occurrence  
   
 // If element is not found, return [-1, -1]  
 if (first == -1) {  
 result.add(-1);  
 result.add(-1);  
 } else {  
 result.add(first);  
 result.add(last);  
 }  
   
 return result;  
 }  
   
 private 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; // Update result to current index  
 // Move left for first occurrence or right for last occurrence  
 if (findFirst) {  
 high = mid - 1;  
 } else {  
 low = mid + 1;  
 }  
 } else if (arr[mid] < x) {  
 low = mid + 1;  
 } else {  
 high = mid - 1;  
 }  
 }  
   
 return result;  
 }  
}

class Solution {  
 int transitionPoint(int arr[]) {  
 // code here  
   
 int low = 0;  
 int high = arr.length - 1;  
 int result = -1;  
  
 // Binary search for the first occurrence of 1  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
  
 if (arr[mid] == 1) {  
 result = mid; // Update result to current index  
 high = mid - 1; // Move left to find the first occurrence  
 } else {  
 low = mid + 1; // Move right since we're still in the 0's part  
 }  
 }  
   
 return result;  
 }  
}



**Time complexity: O(logn)**

**4.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.

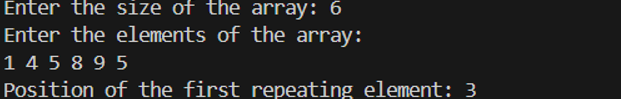
Input: arr[] = [1, 2, 3, 4]

Output: -1

Explanation: All elements appear only once so answer is -1.

**Program:**

class Solution {  
 // Function to return the position of the first repeating element.  
 public static int firstRepeated(int[] arr) {  
 HashMap<Integer, Integer> map = new HashMap<>();  
  
 // Step 1: Store the frequency of each element  
 for (int num : arr) {  
 map.put(num, map.getOrDefault(num, 0) + 1);  
 }  
  
 // Step 2: Find the first element with frequency > 1  
 for (int i = 0; i < arr.length; i++) {  
 if (map.get(arr[i]) > 1) {  
 return i + 1; // Return 1-based index  
 }  
 }  
  
 return -1; // No repeating element found  
 }  
}



**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:

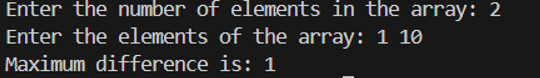
Input: arr[] = [1, 10]

Output: 1

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

**Program:**

// User function Template for Java  
class Solution {  
 // Function to find the maximum index difference.  
 int maxIndexDiff(int[] arr) {  
 int n = arr.length;  
 if (n <= 1) {  
 return 0;  
 }  
  
 // Step 1: Create leftMin and rightMax arrays  
 int[] leftMin = new int[n];  
 int[] rightMax = new int[n];  
  
 // Populate leftMin[]  
 leftMin[0] = arr[0];  
 for (int i = 1; i < n; i++) {  
 leftMin[i] = Math.min(arr[i], leftMin[i - 1]);  
 }  
  
 // Populate rightMax[]  
 rightMax[n - 1] = arr[n - 1];  
 for (int j = n - 2; j >= 0; j--) {  
 rightMax[j] = Math.max(arr[j], rightMax[j + 1]);  
 }  
  
 // Step 2: Use two-pointer approach to find the max index difference  
 int i = 0, j = 0;  
 int maxDiff = -1;  
  
 while (i < n && j < n) {  
 if (leftMin[i] <= rightMax[j]) {  
 // Update maxDiff if a larger index difference is found  
 maxDiff = Math.max(maxDiff, j - i);  
 j++;  
 } else {  
 i++;  
 }  
 }  
  
 return maxDiff;  
 }  
}



**Time complexity: 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.

Input: arr[] = [1]

Output: [1]

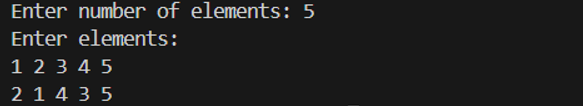
Constraints:

1 ≤ arr.size ≤ 106

0 ≤ arr[i] ≤107

**Program:**

class Solution {  
 public static void convertToWave(int[] arr) {  
 // code here  
 int n = arr.length;  
  
 // Traverse all even indexed elements  
 for (int i = 0; i < n - 1; i += 2) {  
 // Swap arr[i] and arr[i+1]  
 int temp = arr[i];  
 arr[i] = arr[i + 1];  
 arr[i + 1] = temp;  
 }  
 }  
}

  
**Time Complexity: O(n)**

**7.** **Remove duplicates in a 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 ≤ arr.size() ≤ 105

1 ≤ ai ≤ 106

**Program:**

import java.util.List;

import java.util.ArrayList;

import java.util.Scanner;

class removeDuplicatesInSortedArray {

// Function to remove duplicates from the given list

public int remove\_duplicate(List<Integer> arr) {

if (arr.size() == 0) {

return 0;

}

int j = 0;

for (int i = 1; i < arr.size(); i++) {

if (!arr.get(i).equals(arr.get(i - 1))) {

j++;

arr.set(j, arr.get(i));

}

}

return j + 1;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the size of the array: ");

int n = sc.nextInt();

List<Integer> arr = new ArrayList<>();

System.out.println("Enter the elements of the array: ");

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

arr.add(sc.nextInt());

}

removeDuplicatesInSortedArray solution = new removeDuplicatesInSortedArray();

int uniqueCount = solution.remove\_duplicate(arr);

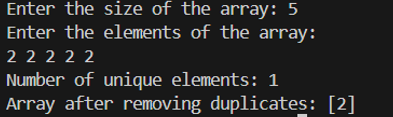
System.out.println("Number of unique elements: " + uniqueCount);

System.out.println("Array after removing duplicates: " + arr.subList(0, uniqueCount));

sc.close();

}

}

****

**Time complexity: O(n)**

**8.** **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.

**Program:**

import java.util.Scanner;

class transitionPoint {

int transitionPoint(int arr[]) {

int point = -1;

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

if (arr[i] == 1) {

point = i;

break;

}

}

return point;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the size of 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();

}

transitionPoint solution = new transitionPoint();

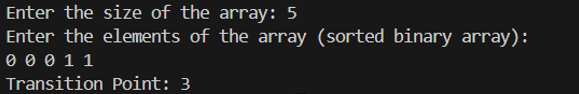
int result = solution.transitionPoint(arr);

System.out.println("Transition Point: " + result);

scanner.close();

}

}



**Time Complexity: O(n)**