**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.

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.

**Code**

class Solution {

ArrayList<ArrayList<Integer>> stockBuySell(int A[], int n) {

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

int l = 0;

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

if (A[r] < A[r - 1]) {

if (l < r - 1) {

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

pair.add(l);

pair.add(r - 1);

result.add(pair);

}

l = r;

}

}

if (l < n - 1) {

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

pair.add(l);

pair.add(n - 1);

result.add(pair);

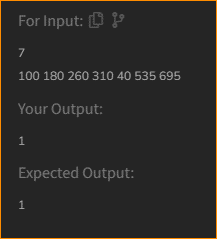
}

return result;

}

}}

**Output**



**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].

**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 i = coin; i <= sum; i++) {

dp[i] += dp[i - coin];

}

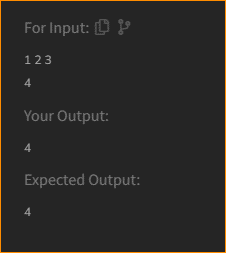
}

return dp[sum];

}

}

**Output**



**Time Complexity**

O(n \* m)

**3. First and Last Occurrences**

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

**Code**

class GFG {

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

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

int first = -1, last = -1;

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

if (arr[i] == x) {

if (first == -1) {

first = i;

}

last = i;

}

}

result.add(first);

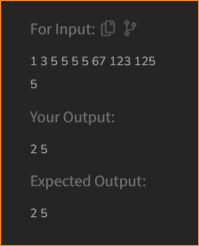
result.add(last);

return result;

}

}

**Output**



**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.

**Code**

class Solution {

int transitionPoint(int arr[]) {

int start = 0;

int end = arr.length - 1;

if (arr[0] == 1) {

return 0;

}

if (arr[end] == 0) {

return -1;

}

while (start <= end) {

int mid = start + (end - start) / 2;

if (arr[mid] == 1 && (mid == 0 || arr[mid - 1] == 0)) {

return mid;

} else if (arr[mid] == 0) {

start = mid + 1;

} else {

end = mid - 1;

}

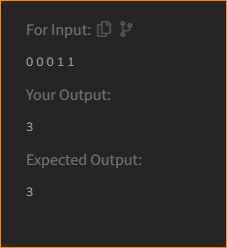
}

return -1;

}

}

**Output**



**Time Complexity**

O(log n)

**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.

**Code**

class Solution {

public static int firstRepeated(int[] arr) {

HashSet<Integer> seen = new HashSet<>();

int minIndex = -1;

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

if (seen.contains(arr[i])) {

minIndex = i + 1;

} else {

seen.add(arr[i]);

}

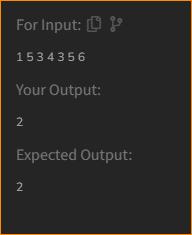
}

return minIndex;

}

}

**Output**



**Time Complexity**

O(n)

**6. Remove 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.

**Code**

class Solution {

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

int i = 0;

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

if (!arr.get(i).equals(arr.get(j))) {

i++;

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

}

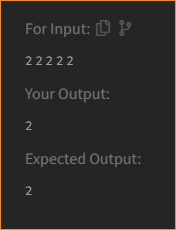
}

return i + 1;

}

}

**Output**



**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.

**Code**

class Solution {

int maxIndexDiff(int[] arr) {

int n = arr.length;

if (n == 1) {

return 0;

}

int maxDiff = -1;

int[] LMin = new int[n];

int[] RMax = new int[n];

LMin[0] = arr[0];

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

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

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

int i = 0, j = 0;

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

if (LMin[i] <= RMax[j]) {

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

j++;

} else {

i++;

}

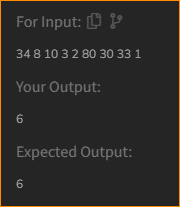
}

return maxDiff;

}

}

**Output**



**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.

**Code**

**class Solution {**

**public static void convertToWave(int[] arr) {**

**for (int i = 0; i < arr.length - 1; i += 2) {**

**int temp = arr[i];**

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

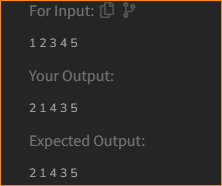
**arr[i + 1] = temp;**

**}**

**}**

**}**

**Output**



**Time Complexity**

O(n)