**SANJAY\_M – CSE – DSA – PRACTICE – 5**

**Q1. 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.

CODE:

import java.util.ArrayList;

class buysell {

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

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

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

if (A[i + 1] > A[i]) {

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

al.add(i);

al.add(i + 1);

ans.add(al);

}

}

return ans;

}

public static void main(String[] args) {

buysell sol = new buysell();

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

int n = A.length;

ArrayList<ArrayList<Integer>> result = sol.stockBuySell(A, n);

if (result.isEmpty()) {

System.out.println("No Profit");

} else {

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

expected.add(new ArrayList<Integer>() {{add(0); add(1);}});

expected.add(new ArrayList<Integer>() {{add(1); add(2);}});

expected.add(new ArrayList<Integer>() {{add(2); add(3);}});

expected.add(new ArrayList<Integer>() {{add(4); add(5);}});

expected.add(new ArrayList<Integer>() {{add(5); add(6);}});

if (result.equals(expected)) {

System.out.println(1);

} else {

System.out.println(0);

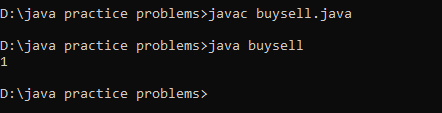
}

}

}

}

OUTPUT:



Time Complexity: O(n)

**Q2. 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[ ]. 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].

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].

CODE:

import java.util.Arrays;

class coinchange {

static long find(int[] c, int n, int sum, int j, long[][] dp) {

if (sum < 0 || j == n) return 0;

if (sum == 0) return 1;

if (dp[j][sum] != -1) return dp[j][sum];

long s = 0;

s += find(c, n, sum - c[j], j, dp);

s += find(c, n, sum, j + 1, dp);

return dp[j][sum] = s;

}

public long count(int coins[], int N, int sum) {

long[][] dp = new long[N + 1][sum + 1];

for (long[] t : dp) Arrays.fill(t, -1);

return find(coins, N, sum, 0, dp);

}

public static void main(String[] args) {

coinchange cc = new coinchange();

int[] coins = {1, 2, 3};

int N = coins.length;

int sum = 4;

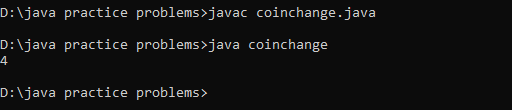
long result = cc.count(coins, N, sum);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(n\*m)

**Q3. 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.

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

**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

CODE:

import java.util.ArrayList;

class firstlast {

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

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

int left = 0, right = n - 1;

list.add(-1);

list.add(-1);

while (left <= right) {

if (arr[left] == x && arr[right] == x) {

list.set(0, left);

list.set(1, right);

break;

}

if (arr[left] != x) left++;

if (arr[right] != x) right--;

}

return list;

}

public static void main(String[] args) {

firstlast sol = new firstlast();

int arr[] = {1, 3, 5, 5, 5, 5, 67, 123, 125};

int x = 5;

int n = arr.length;

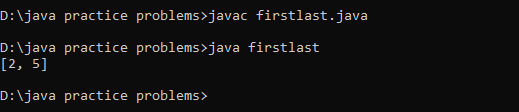
ArrayList<Integer> result = sol.find(arr, n, x);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(n)

**Q4. 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.

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

**Output:** -1

**Explanation:** Since, there is no "1", the answer is -1.

CODE:

import java.util.\*;

class transpnt {

int transitionPoint(int arr[]) {

int left, right, mid, n, index;

n = arr.length;

left = 0;

right = n - 1;

index = -1;

while(left <= right) {

mid = (left + right) / 2;

if(arr[mid] == 1) {

index = mid;

right = mid - 1;

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

left = mid + 1;

}

}

return index;

}

public static void main(String[] args) {

transpnt sol = new transpnt();

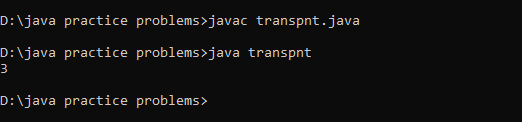
int arr1[] = {0, 0, 0, 1, 1, 1};

System.out.println(sol.transitionPoint(arr1));

}

}

OUTPUT:



Time Complexity: O(log n)

**Q5. 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. 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.

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

**Output:** -1

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

CODE:

import java.util.HashMap;

class firstrepeated {

public static int firstRepeated(int[] arr) {

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

int index = -1;

for (int x : arr) {

map.put(x, map.getOrDefault(x, 0) + 1);

}

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

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

index = i + 1;

break;

}

}

return index;

}

public static void main(String[] args) {

firstrepeated sol = new firstrepeated();

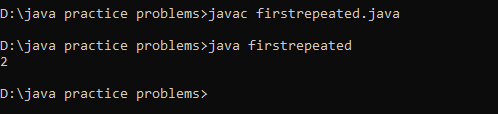
int[] arr1 = { 1, 5, 3, 4, 3, 5, 6 };

System.out.println(sol.firstRepeated(arr1));

}

}

OUTPUT:



Time Complexity: O(n)

**Q6. Remove Duplicates Sorted Array**

Given a **sorted** array**arr.** Return the size of the modified array which contains only distinct elements.  
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.

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

**Output:** [1, 2, 4]

**Explation:** As the array does not contain any duplicates so you should return 3.

CODE:

import java.util.\*;

class removedup {

public int removeDuplicate(int[] arr) {

int i = 0;

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

if (arr[j] != arr[i]) {

i++;

arr[i] = arr[j];

}

}

return i + 1;

}

}

public class removeduplicate {

public static void main(String[] args) {

int[] arr = {2, 2, 2, 2, 2};

removedup rd = new removedup();

int newLength = rd.removeDuplicate(arr);

System.out.print("[");

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

System.out.print(arr[i]);

if (i < newLength - 1) {

System.out.print(", ");

}

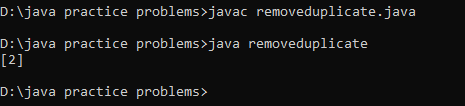
}

System.out.println("]");

}

}

OUTPUT:



Time Complexity: O(n)

**Q7. 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.

**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).

CODE:

import java.util.\*;

class maxindex {

public int maxIndexDiff(int[] arr) {

int n = arr.length;

int[] minLeft = new int[n];

int[] maxRight = new int[n];

minLeft[0] = arr[0];

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

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

}

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

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

maxRight[j] = Math.max(arr[j], maxRight[j + 1]);

}

int i = 0, j = 0;

int maxDiff = -1;

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

if (minLeft[i] <= maxRight[j]) {

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

j++;

} else {

i++;

}

}

return maxDiff;

}

public static void main(String[] args) {

maxindex mi= new maxindex();

int[] arr = { 34, 8, 10, 3, 2, 80, 30, 33, 1 };

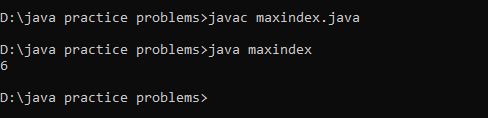
int result = mi.maxIndexDiff(arr);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(n)

**Q8. 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. 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:

import java.util.Arrays;

class wavearray {

public static void convertToWave(int[] arr) {

int i = 0;

while (i < arr.length - 1) {

int temp = arr[i];

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

arr[i + 1] = temp;

i += 2;

}

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5};

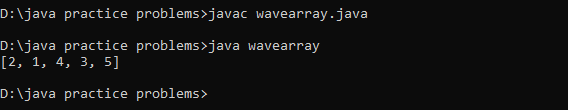
convertToWave(arr);

System.out.println(Arrays.toString(arr));

}

}

OUTPUT:



Time Complexity: O(n)