**Day-5 Practice**

**1)Stock Buy And Sell**

**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:

import java.util.ArrayList;

public class StockBuyandSell {

public static void main(String[] args) {

StockBuyandSell obj=new StockBuyandSell();

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

System.out.println(obj.buysell(arr));

}

public ArrayList<ArrayList<Integer>> buysell(int[] arr){

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

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

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

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

li.add(i);

li.add(i+1);

res.add(li);

}

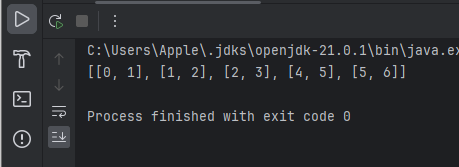
}

return res;

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

**2)Coin Change**

Program:

import java.util.\*;

public class MinimumElement {

static int minimumElementsUtil(int[] arr, int ind, int T, int[][] dp) {

if (ind == 0) {

if (T % arr[0] == 0)

return T / arr[0];

else

return (int) Math.pow(10, 9);

}

if (dp[ind][T] != -1)

return dp[ind][T];

int notTaken = 0 + minimumElementsUtil(arr, ind - 1, T, dp);

int taken = (int) Math.pow(10, 9);

if (arr[ind] <= T)

taken = 1 + minimumElementsUtil(arr, ind, T - arr[ind], dp);

return dp[ind][T] = Math.min(notTaken, taken);

}

static int minimumElements(int[] arr, int T) {

int n = arr.length;

int[][] dp = new int[n][T + 1];

for (int[] row : dp)

Arrays.fill(row, -1);

int ans = minimumElementsUtil(arr, n - 1, T, dp);

if (ans >= (int) Math.pow(10, 9))

return -1;

return ans;

}

public static void main(String[] args) {

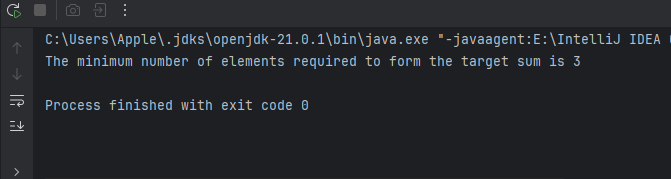
int arr[] = { 1, 2, 3 };

int T = 7;

System.out.println("The minimum number of elements required to form the target sum is " + minimumElements(arr, T));

}

Output:



 **Time Complexity**: **O(n \* T)**

 **Space Complexity**: **O(n \* T)**

**3)First and Last Occurances**

**Program:**

**import java.util.ArrayList;**

**public class FirstandLastOccurances {**

**public static void main(String[] args) {**

**FirstandLastOccurances obj= new FirstandLastOccurances();**

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

**int x=5;**

**System.out.println(obj.find(arr,x));**

**}**

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

**// code here**

**int n=arr.length;**

**ArrayList<Integer> li=new ArrayList<>();**

**li.add(first(arr,0,n-1,x));**

**li.add(last(arr,0,n-1,x));**

**return li;**

**}**

**int first(int[] arr,int start,int end,int x){**

**int res=-1;**

**while(start<=end){**

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

**if(arr[mid]>x){**

**end=mid-1;**

**}**

**else if(arr[mid]<x){**

**start=mid+1;**

**}**

**else{**

**res=mid;**

**end=mid-1;**

**}**

**}**

**return res;**

**}**

**int last(int[] arr,int start,int end,int x){**

**int res=-1;**

**while(start<=end){**

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

**if(arr[mid]>x){**

**end=mid-1;**

**}**

**else if(arr[mid]<x){**

**start=mid+1;**

**}**

**else{**

**res=mid;**

**start=mid+1;**

**}**

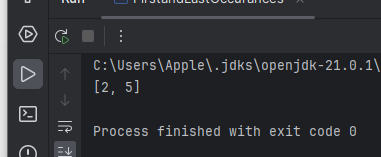
**}**

**return res;**

**}**

**}**

**Output:**

****

 **Time Complexity**: **O(n)**

 **Space Complexity**: **O(1)**

**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.

Program:

public class Transitionpoint {

public static void main(String[] args) {

Transitionpoint obj=new Transitionpoint();

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

System.out.println(obj.pointfind(arr));

}

public int pointfind(int[] arr){

int low=0;

int high=arr.length-1;

int index=0;

while(low<=high){

int mid=low+(high-low)/2;

if(arr[mid]==1){

index=mid;

high=mid-1;

}

else{

low=mid+1;

}

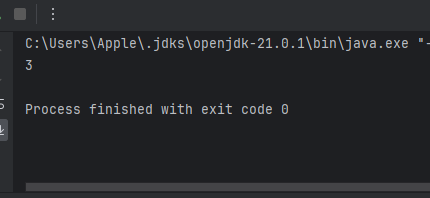
}

return index;

}

}

Output:



 **Time Complexity**: O(log n)

 **Space Complexity**: O(1)

**5)FirstRepeatingElement**

Given an array of integers **arr[]**, The task is to find the index of first repeating element in it i.e. the element that occurs more than once and whose index of the first occurrence is the smallest.

**Examples:**

***Input:*** *arr[] = {10, 5, 3, 4, 3, 5, 6}****Output:*** *5****Explanation:*** *5 is the first element that repeats*

***Input:*** *arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10}****Output:*** *6****Explanation:*** *6 is the first element that repeats*

**Program:**

import java.util.HashMap;

public class FirstRepeatingElement {

public static void main(String[] args) {

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

FirstRepeatingElement obj=new FirstRepeatingElement();

System.out.println(obj.firstRepeated(arr));

}

public static int firstRepeated(int[] arr) {

// Your code here

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

for(int i:arr){

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

}

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

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

return i+1;

}

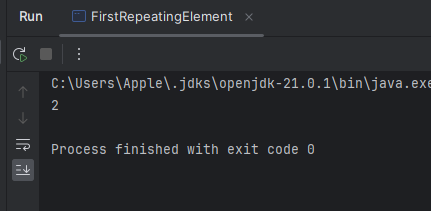
}

return -1;

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

**6)Remove the Duplicate in the sorted array**

Given a sorted array **arr[]** of size **N**, the task is to remove the duplicate elements from the array. We need keep order of the remaining distinct elements as it was in the original array.

**Examples:**

***Input:*** *arr[] = {2, 2, 2, 2, 2}****Output:*** *arr[] = {2}****Explanation:*** *All the elements are 2, So only keep one instance of 2.*

***Input:*** *arr[] = {1, 2, 2, 3, 4, 4, 4, 5, 5}****Output:*** *arr[] = {1, 2, 3, 4, 5}*

*Program:*

public class RemoveDuplicate {

public static void main(String[] args) {

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

RemoveDuplicate obj=new RemoveDuplicate();

System.out.println(obj.dublicate(arr));

}

public int dublicate(int[] arr){

int count=1;

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

if(arr[i]!=arr[i-1]){

arr[count]=arr[i];

count++;

}

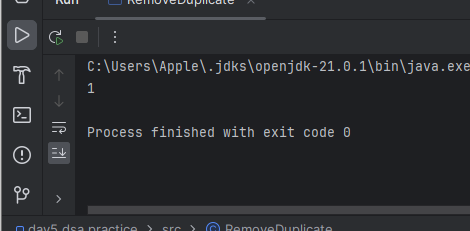
}

return count;

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

**7)MaximumIndex**

Program:

import java.util.Stack;

public class MaximumIndex {

public static void main(String[] args) {

MaximumIndex obj=new MaximumIndex();

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

System.out.println(obj.maxIndexDiff(arr));

}

int maxIndexDiff(int[] arr) {

// Your code here

Stack<Integer> stack=new Stack<>();

int ans=0;

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

if(stack.isEmpty() || arr[stack.peek()]<=arr[i]){

stack.push(i);

}

}

int i=0;

while(i<arr.length && stack.size()>0){

if(arr[i]>arr[stack.peek()]){

i++;

}

else{

ans=Math.max(ans,stack.peek()-i);

stack.pop();

}

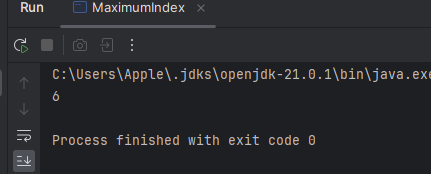
}

return ans;

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

**8)Wave Array**

Program:

import java.util.Arrays;

public class WaveArray {

public static void main(String[] args) {

WaveArray obj = new WaveArray();

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

int[] res = obj.convertToWave(arr);

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

}

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

// code here

int i = 0;

while (i < arr.length - 1) {

int temp = arr[i];

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

arr[i + 1] = temp;

i += 2;

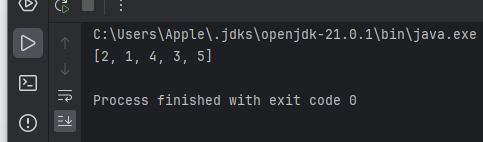
}

return arr;

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(1)