14-11-2024

Java DSA Program (SET - 5)

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.

Examples:

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

Solution:

```
import java.util.*;

class StockBuySell {
    static int maximumProfit(int[] prices) {
        int res = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1])
                res += prices[i] - prices[i - 1];
        }
        return res;
    }
    public static void main(String[] args) {
        int[] prices = { 100, 180, 260, 310, 40, 535, 695 };
        System.out.println(maximumProfit(prices));
    }
}
```

Output:

```
D:\OOPS\14-11-2024 practice.java>javac StockBuySell.java

D:\OOPS\14-11-2024 practice.java>java StockBuySell
865
```

Time complexity: O(n)

Space Complexity: O(1)

2. Coin change

Given an integer array of **coins[]** of **size n** representing different types of **denominations** and an integer **sum**, the task is to count all combinations of coins to make a given value **sum**.

Note: Assume that you have an **infinite** supply of each type of coin.

Examples:

Input: *sum = 4, coins[] = [1, 2, 3]*
Output: *4*
Explanation: *There are four solutions: [1, 1, 1, 1], [1, 1, 2], [2, 2] and [1, 3]*

Input: *sum = 10, coins[] = [2, 5, 3, 6]*
Output: *5*
Explanation: *There are five solutions:*
[2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5] and [5, 5]

Input: *sum = 10, coins[] = [10]*
Output: *1*
Explanation: *The only is to pick 1 coin of value 10.*

Input: *sum = 5, coins[] = [4]*
Output: *0*
Explanation: *We cannot make sum 5 with the given coins*

Solution:

```
import java.util.*;

class Coinchange {
    static int count(int[] coins, int sum) {
        int n = coins.length;
        int[] dp = new int[sum + 1];
        dp[0] = 1;
        for (int i = 0; i < n; i++)
            for (int j = coins[i]; j <= sum; j++)
                dp[j] += dp[j - coins[i]];

        return dp[sum];
    }
    public static void main(String[] args) {
        int[] coins = {1, 2, 3};
        int sum = 5;
        System.out.println(count(coins, sum));
    }
}
```

Output:

```
D:\OOPS\14-11-2024 practice.java>javac Coinchange.java

D:\OOPS\14-11-2024 practice.java>java Coinchange
5
```

Time complexity: O(n*sum)

Space complexity: O(n*sum)

3. **Find first and last positions of an element**

Given a sorted array **arr[]** with possibly duplicate elements, the task is to find indexes of the first and last occurrences of an element **x** in the given array.

Examples:

Input : *arr[] = {1, 3, 5, 5, 5, 5, 67, 123, 125}, x = 5*
Output *: First Occurrence = 2*
Last Occurrence = 5

Input : *arr[] = {1, 3, 5, 5, 5, 5, 7, 123, 125 }, x = 7*

Output : *First Occurrence = 6*
Last Occurrence = 6

Solution:

```
import java.util.*;

class FirstLastOccuranceOfNumber {

    public static int[] searchRange(int[] nums, int target) {

        int l = 0;

        int h = nums.length - 1;

        int[] arr = new int[2];

        arr[0] = -1;

        arr[1] = -1;

        // First occurrence

        while (l <= h) {

            int mid = (l + h) / 2;

            if (nums[mid] > target) {

                h = mid - 1;

            } else if (nums[mid] == target) {

                arr[0] = mid;

                h = mid - 1;
```

```
            } else {
                l = mid + 1;
            }
        }
        // Last occurrence
        l = 0;
        h = nums.length - 1;
        while (l <= h) {
            int mid = (l + h) / 2;
            if (nums[mid] < target) {
                l = mid + 1;
            } else if (nums[mid] == target) {
                arr[1] = mid;
                l = mid + 1;
            } else {
                h = mid - 1;
            }
        }
        return arr;
    }
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5, 5, 5, 5, 67, 123, 125};
        int target1 = 5;
        int[] result1 = searchRange(arr1, target1);
        System.out.println("First Occurrence = " + result1[0]);
        System.out.println("Last Occurrence = " + result1[1]);
        int[] arr2 = {1, 3, 5, 5, 5, 5, 7, 123, 125};
        int target2 = 7;
        int[] result2 = searchRange(arr2, target2);
        System.out.println("First Occurrence = " + result2[0]);
        System.out.println("Last Occurrence = " + result2[1]);
    }
}
```

Output:

```
D:\OOPS\14-11-2024 practice.java>javac FirstLastOccuranceOfNumber.java

D:\OOPS\14-11-2024 practice.java>java FirstLastOccuranceOfNumber
First Occurrence = 2
Last Occurrence = 5
First Occurrence = 6
Last Occurrence = 6
```

Time complexity: O(log 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.

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.

Input: arr[] = [0, 1, 1]

Output: 1

Explanation: Index 1 is the transition point where 1 starts, and before it, only 0 was observed.

SOLUTION:

```
import java.util.*;

public class FindTransPoint {

    public static void main(String[] args) {

        Solution solution = new Solution();

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

        int result = solution.transitionPoint(arr);

        if (result != -1) {
```

```
            System.out.println(result);
        } else {
            System.out.println("-1");
        }
    }
}
class Solution {
    int transitionPoint(int arr[]) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == 1) {
                return i;
            }
        }
        return -1;
    }
}
```

OUTPUT:

```
D:\OOPS\14-11-2024 practice.java>javac FindTransPoint.java

D:\OOPS\14-11-2024 practice.java>java FindTransPoint
3
```

Time Complexity: O(n)

Space Complexity: O(1)

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.

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

Output: -1

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

SOLUTION:

```
import java.util.*;
public class FirstRepeatElement{
   public static int firstRepeated(int[] arr) {
      Map<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; // 1-based index
         }
      }
      return -1;
   }
   public static void main(String[] args) {
      int[] arr = {1, 5, 3, 4, 3, 5, 6};
      int result = firstRepeated(arr);
      if (result != -1) {
         System.out.println(result);
      } else {
         System.out.println(-1);
      }
   }
}
```

OUTPUT:

```
D:\OOPS\14-11-2024 practice.java>javac FirstRepeatElement.java

D:\OOPS\14-11-2024 practice.java>java FirstRepeatElement
2
```

Time Complexity: O(n)

Space 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.

Input: arr = [1, 2, 4]

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

SOLUTION:

```
import java.util.*;

public class RemoveDuplicates{
   public static int remove_duplicate(List<Integer> arr) {
      int k = 1;
      for (int i = 1; i < arr.size(); i++) {
         if (!arr.get(i).equals(arr.get(i - 1))) {
            arr.set(k++, arr.get(i));
         }
      }
      return k;
   }
   public static void main(String[] args) {
      List<Integer> arr = new ArrayList<>();
      arr.add(1);
      arr.add(1);
      arr.add(2);
      arr.add(2);
      arr.add(3);
      arr.add(4);
      arr.add(4);
```

```
        int uniqueCount = remove_duplicate(arr);
        System.out.println(uniqueCount);
        for (int i = 0; i < uniqueCount; i++) {
            System.out.print(arr.get(i) + " ");
        }
    }
}
```

OUTPUT:

```
D:\OOPS\14-11-2024 practice.java>javac RemoveDuplicates.java

D:\OOPS\14-11-2024 practice.java>java RemoveDuplicates
4
1 2 3 4
```

Time Complexity: O(n)

Space Complexity: O(1)

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.

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).

SOLUTION:

```
import java.util.*;
class MaximumIndex{
    public static void main(String[] args) {
        int A[] = {34, 8, 10, 3, 2, 80, 30, 33, 1};
        int N = A.length;
        System.out.println("Max diff be : " + maxIndexDiff(A, N));
    }
```

```
    static int maxIndexDiff(int A[], int N) {
        Stack<Integer> stkForIndex = new Stack<>();
        for (int i = 0; i < N; i++) {
            if (stkForIndex.isEmpty() || A[stkForIndex.peek()] > A[i])
                stkForIndex.push(i);
        }
        int maxDiffSoFar = 0;
        int tempdiff;
        int i = N - 1;
        while (i >= 0) {
            if (!stkForIndex.isEmpty() && A[stkForIndex.peek()] <= A[i]) {
                tempdiff = i - stkForIndex.pop();
                if (tempdiff > maxDiffSoFar) {
                    maxDiffSoFar = tempdiff;
                }
                continue;
            }
            i--;
        }
        return maxDiffSoFar;
    }
}
```

OUTPUT:

```
D:\OOPS\14-11-2024 practice.java>javac MaximumIndex.java

D:\OOPS\14-11-2024 practice.java>java MaximumIndex
Max diff be : 6
```

Time Complexity: O(n)

Space 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[] = {10, 5, 6, 3, 2, 20, 100, 80}
Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80}

Explanation:
here you can see {10, 5, 6, 2, 20, 3, 100, 80} first element is larger than the second and the same thing is repeated again and again. large element – small element-large element -small element and so on .it can be small element-larger element – small element-large element -small element too. all you need to maintain is the up-down fashion which represents a wave. there can be multiple answers.

Input: arr[] = {20, 10, 8, 6, 4, 2}
Output: arr[] = {20, 8, 10, 4, 6, 2}

Solution:

```
import java.util.*;

class WaveArray

{

   static void swap(int arr[], int a, int b)

   {

      int temp = arr[a];

      arr[a] = arr[b];

      arr[b] = temp;

   }

   static void sortInWave(int arr[], int n)

   {

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

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

           swap(arr, i, i-1);

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

           swap(arr, i, i+1);

      }

   }

   public static void main(String args[])

   {

      int arr[] = {10, 90, 49, 2, 1, 5, 23};

      int n = arr.length;

      sortInWave(arr, n);
```

```
        for (int i : arr)
            System.out.print(i+" ");
    }
}
```

Output:

```
D:\OOPS\14-11-2024 practice.java>javac WaveArray.java

D:\OOPS\14-11-2024 practice.java>java WaveArray
90 10 49 1 5 2 23
```

Time Complexity: O(n)

Space Complexity: O(1)