DSA Practice 4 (13-11-2024)

Name: Atchayaa T

Department: B.Tech CSBS

Register No.: 22CB004

1. **K'th Smallest Element in Unsorted Array**
   Given an array arr[] of N distinct elements and a number K, where K is smaller than the size of the array. Find the K'th smallest element in the given array.

   Examples:
   ***Input**: arr[] = {7, 10, 4, 3, 20, 15}, K = 3*
   ***Output**: 7*
   ***Input**: arr[] = {7, 10, 4, 3, 20, 15}, K = 4*
   ***Output**: 10*

   Code:

```java
import java.util.Arrays;
import java.util.Collections;
public class KthSmallestElementInAnUnsortedArray {
    public static int solution(int[] arr,int num){
        int m=arr[0];
        for (int i=1;i<arr.length;i++){
            if (arr[i]>m){
                m=arr[i];
            }
        }
        int[] freq = new int[m+1];
        Arrays.fill(freq,0);
        for (int i=0;i<arr.length;i++){
            freq[arr[i]]++;
        }
        int c=0;
        for (int i=0;i<m+1;i++){
            if (freq[i]!=0){
                c+=freq[i];
            }
            if (c>=num){
                return i;
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] arr = { 12, 3, 5, 7, 19 };
        int num = 2;
        System.out.println(solution(arr, num));

    }}
```

Output: 5
Time Complexity: O(n)

2. **Minimize the maximum difference between the heights**
Given the heights of n towers and a positive integer k, increase or decrease the height of all towers by k (only once). After modifications, the task is to find the minimum difference between the heights of the tallest and the shortest tower.

Examples:
Input: *arr[] = {12, 6, 4, 15, 17, 10}, k = 6*
Output: *8*
Explanation: *Update arr[] as {12 – 6, 6 + 6, 4 + 6, 15 – 6, 17 – 6, 10 – 6} = {6, 12, 10, 9, 11, 4}. Now, the minimum difference is 12 – 4 = 8.*
Input: *arr[] = {12, 6, 4, 15, 17, 10}, k = 3*
Output: *8*
Explanation: *Update arr[] as {1 + 3, 5 + 3, 10 – 3, 15 – 3} = {4, 8, 7, 12}. Now, the minimum difference is 8.*

Code:

```java
import java.util.Arrays;

public class Minimize_Maximum_Difference_Between_Heights {
    public static int solution(int[]arr,int k){
        Arrays.sort(arr);
        int n = arr.length;
        int res = arr[n-1] - arr[0];

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

            if (arr[i]-k<0) continue;

            int min1 = arr[0];
            int min2 = arr[i-1];
            int max1 = arr[i];
            int max2 = arr[n-1];

            int minval = Math.min(min1+k,max1-k);
            int maxval = Math.max(min2+k,max2-k);

            res = Math.min(res,maxval-minval);
        }
        return res;
    }
    public static void main(String[] args) {
        int[] arr = {12, 6, 4, 15, 17, 10};
        int k = 6;
        System.out.println(solution(arr, k));
    }
}
```

Output: 8
Time Complexity: O (n logn)

3. **Valid Parentheses in an Expression**
Given an expression string s, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the given expression.

Examples:
Input*: s = “[()]{}{[()()]()}”*
Output*: true*
Explanation: *All the brackets are well-formed*
Input*: s = “[(])”*
Output*: false*
Explanation: *1 and 4 brackets are not balanced because there is a closing ‘]’ before the closing ‘(‘*

Code:

```
import java.util.*;
public class ValidParanthesis {
    public static boolean solution(String s) {
        Stack<Character> stk = new Stack<>();
        for (int i = 0; i < s.length(); i++) {

            if (s.charAt(i) == '(' || s.charAt(i) == '{' || s.charAt(i) == '[') {
                stk.push(s.charAt(i));
            }
            else {
                if (!stk.empty() &&
                    ((stk.peek() == '(' && s.charAt(i) == ')') ||
                     (stk.peek() == '{' && s.charAt(i) == '}') ||
                     (stk.peek() == '[' && s.charAt(i) == ']'))) {
                    stk.pop();
                }
                else {
                    return false;
                }
            }
        }

        return stk.empty();
    }

    public static void main(String[] args) {
        String s = "{()}[]";
        if (solution(s))
            System.out.println("true");
        else
            System.out.println("false");
    }

}
```

Output: true
Time Complexity: O(n)

4. **Equilibrium index of an array**

Given an array arr[] of size n, return an equilibrium index (if any) or -1 if no equilibrium index exists. The equilibrium index of an array is an index such that the sum of elements at lower indexes equals the sum of elements at higher indexes.

Note: Return equilibrium point in 1-based indexing. Return -1 if no such point exists.

Examples:
Input*: arr[] = {-7, 1, 5, 2, -4, 3, 0}*
Output*: 4*
Explanation: *In 1-based indexing, 4 is an equilibrium index, because: arr[1] + arr[2] + arr[3] = arr[5] + arr[6] + arr[7]*
Input: *arr[] = {1, 2, 3}*
Output*: -1*
Explanation: *There is no equilibrium index in the array.*

Code:

```
public class EquilibriumIndex {
    public static int solution(int[] arr){
        int s=0;
        for (int i =0 ; i<arr.length;i++){
            s+=arr[i];
        }

        int l = 0;
        for (int i=0; i<arr.length;i++){
            s-=arr[i];
            if (s==l) return i;
            l+=arr[i];
        }
        return -1;

    }
    public static void main(String[] args) {
        int[] arr = {1, 7, 3, 6, 5, 6};
        System.out.println(solution(arr));

    }

}
```

Output: 3
Time Complexity: O(n)

5. **Binary Search**

Code:

```java
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) return mid;
            else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9, 11};
        int target = 7;
        int result = binarySearch(arr, target);
        System.out.println(result);
    }
}
```

Output: 3
Time Complexity: O(log n)

6. **Next Greater Element (NGE) for every element in given Array**

Given an array, print the Next Greater Element (NGE) for every element.
Note: The Next greater Element for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.

Examples:
***Input**: arr[] = [4 , 5 , 2 , 25]*
***Output**: 4 –> 5*
5 –> 25
2 –> 25
25 –> -1
***Explanation:** Except **25** every element has an element greater than them present on the right side*
***Input**: arr[] = [13 , 7, 6 , 12]*
***Output**: 13 –> -1*
7 –> 12
6 –> 12
12 –> -1
***Explanation: 13** and **12** don't have any element greater than them present on the right side*

Code:

```java
class NextGreaterElement{
    static void solution(int arr[], int n)
    {
        int next, i, j;
        for (i = 0; i < n; i++) {
            next = -1;
```

```
            for (j = i + 1; j < n; j++) {
                if (arr[i] < arr[j]) {
                    next = arr[j];
                    break;
                }
            }
            System.out.println(arr[i] + " -- " + next);
        }
    }

    public static void main(String args[])
    {
        int arr[] = { 11, 13, 21, 3 };
        int n = arr.length;
        solution(arr, n);
    }
}
```

Output:
11 -- 13
13 -- 21
21 -- -1
3 -- -1
Time Complexity: O(n^2)

7. **Union of two arrays with duplicate elements**
   We are given two sorted arrays a[] and b[] and the task is to return union of both the arrays in sorted order. Union of two arrays is an array having all distinct elements that are present in either array. The input arrays may contain duplicates.

   Examples:
   ***Input**: a[] = {1, 1, 2, 2, 2, 4}, b[] = {2, 2, 4, 4}*
   ***Output**: {1, 2, 4}*
   ***Explanation**: 1, 2 and 4 are the **distinct** elements present in either array.*
   ***Input**: a[] = {3, 5, 10, 10, 10, 15, 15, 20}, b[] = {5, 10, 10, 15, 30}*
   ***Output**: {3, 5, 10, 15, 20, 30}*
   ***Explanation**: 3, 5, 10, 15, 20 and 30 are the **distinct** elements present in either array.*

Code:
```
import java.util.*;

public class UnionOfTwoSortedArrays {

    static ArrayList<Integer> findUnion(int[] a, int[] b) {
        ArrayList<Integer> res = new ArrayList<>();
        int n = a.length;
        int m = b.length;
        int i = 0, j = 0;
        while (i < n && j < m) {
            if (i > 0 && a[i - 1] == a[i]) {
                i++;
```

```
                continue;
            }
            if (j > 0 && b[j - 1] == b[j]) {
                j++;
                continue;
            }
            if (a[i] < b[j]) {
                res.add(a[i]);
                i++;
            } else if (a[i] > b[j]) {
                res.add(b[j]);
                j++;
            } else {
                res.add(a[i]);
                i++;
                j++;
            }
        }
        while (i < n) {
            if (i > 0 && a[i - 1] == a[i]) {
                i++;
                continue;
            }
            res.add(a[i]);
            i++;
        }
        while (j < m) {
            if (j > 0 && b[j - 1] == b[j]) {
                j++;
                continue;
            }
            res.add(b[j]);
            j++;
        }
        return res;
    }
    public static void main(String[] args) {
        int[] a = {1, 1, 2, 2, 2, 4};
        int[] b = {2, 2, 4, 4};
        List<Integer> res = findUnion(a, b);
        for (int x : res) {
            System.out.print(x + " ");
        }
    }
}
```

Output: 1 2 4

Time Complexity: O(n+m)