TECHNICAL TRAINING DSA - CODING PRACTICE PROBLEMS

Name: Rishi Kumar S
Dept: CSBS
Date: 13-11-2024

Question 1: Kth Smallest

Given an array arr[] and an integer k where k is smaller than the size of the array, the task is to find the kth smallest element in the given array.

Follow up: Don't solve it using the inbuilt sort function.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:
    int kthSmallest(vector<int> &arr, int k) {
        priority_queue <int> pq;
        int n=arr.size();
        for(int i=0;i<n;i++){
            pq.push(arr[i]);
            if(pq.size()>k){
                pq.pop();
            }
        }
        return pq.top();
    }
};

int main() {
    int test_case;
    cin >> test_case;
    cin.ignore();
    while (test_case--) {

        int k;
        vector<int> arr, brr, crr;
        string input;
        getline(cin, input);
```

```
        stringstream ss(input);
        int number;
        while (ss >> number) {
            arr.push_back(number);
        }
        getline(cin, input);
        ss.clear();
        ss.str(input);
        while (ss >> number) {
            crr.push_back(number);
        }
        k = crr[0];
        int n = arr.size();
        Solution ob;
        cout << ob.kthSmallest(arr, k) << endl << "~\n";
    }
    return 0;
}
```

Time Complexity: O(n+m)
Space Complexity: O(m)

Question 2:Minimize the Heights II

Given an array arr[] denoting heights of N towers and a positive integer K.

For each tower, you must perform exactly one of the following operations exactly once.

Increase the height of the tower by K
Decrease the height of the tower by K
Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.
Note: It is compulsory to increase or decrease the height by K for each tower. After the operation, the resultant array should not contain any negative integers.

Code:

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Solution {
  public:
    int getMinDiff(vector<int> &arr, int k) {
        int n = arr.size();
        vector<pair<int, int>> v;
        vector<int> taken(n);
        for (int i = 0; i < n; i++) {
            if (arr[i] - k >= 0) {
                v.push_back({arr[i] - k, i});
            }
            v.push_back({arr[i] + k, i});
        }
        sort(v.begin(), v.end());
        int elements_in_range = 0;
        int left = 0;
        int right = 0;
        while (elements_in_range < n && right < v.size()) {
            if (taken[v[right].second] == 0) {
                elements_in_range++;
            }
            taken[v[right].second]++;
            right++;
        }
        int ans = v[right - 1].first - v[left].first;
        while (right < v.size()) {
            if (taken[v[left].second] == 1) {
                elements_in_range--;
            }
            taken[v[left].second]--;
            left++;

            while (elements_in_range < n && right < v.size()) {
                if (taken[v[right].second] == 0) {
                    elements_in_range++;
                }
                taken[v[right].second]++;
                right++;
            }

            if (elements_in_range == n) {
                ans = min(ans, v[right - 1].first - v[left].first);
            } else {
```

```
            break;
        }
    }
    return ans;
  }
};

int main() {
  int t;
  cin >> t;
  cin.ignore();
  while (t--) {
    int n, k;
    cin >> k;
    cin.ignore();
    vector<int> a, b, c, d;
    string input;
    getline(cin, input);
    stringstream ss(input);
    int num;
    while (ss >> num)
      a.push_back(num);

    Solution ob;
    auto ans = ob.getMinDiff(a, k);
    cout << ans << "\n";
  }
  return 0;
}
```

Time Complexity: O(n logn)
Space Complexity: O(n)

Question 3: Parenthesis Checker
You are given a string s representing an expression containing various types of brackets: {}, (), and []. Your task is to determine whether the brackets in the expression are balanced. A balanced expression is one where every opening bracket has a corresponding closing bracket in the correct order.

CODE:

```
#include <bits/stdc++.h>
```

```
using namespace std;

class Solution {
  public:
    bool isParenthesisBalanced(string& s) {
        stack<int> st;
        for(char ch:s){
            if(ch==')' && st.size()!=0 && st.top()=='(') st.pop();
            else if(ch=='}' && st.size()!=0 && st.top()=='{') st.pop();
            else if(ch==']' && st.size()!=0 && st.top()=='[') st.pop();
            else st.push(ch);
        }
        return st.size()==0;
    }
};

int main() {
    int t;
    string a;
    cin >> t;
    while (t--) {
        cin >> a;
        Solution obj;
        if (obj.isParenthesisBalanced(a))
            cout << "true" << endl;
        else
            cout << "false" << endl;

        cout << "~"
             << "\n";
    }
}
```

Time Complexity: O(n+m)
Space Complexity: O(n+m)

Question 4: Equilibrium Point

Given an array arr of non-negative numbers. The task is to find the first equilibrium point in an array. The equilibrium point in an array is an index (or position) such that the sum of all elements before that index is the same as the sum of elements after it.

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

CODE:

```
#include <bits/stdc++.h>
using namespace std;


class Solution {
  public:
    int equilibriumPoint(vector<int> &arr) {
        int n=arr.size();
        int prev[n], suff[n];
        prev[0]=arr[0];
        suff[n-1]=arr[n-1];
        for(int i=1;i<n;i++){
            prev[i]=arr[i]+prev[i-1];
        }
        for(int i=n-2;i>=0;i--){
            suff[i]=arr[i]+suff[i+1];
        }
        for(int i=0;i<n;i++){
            if(prev[i]==suff[i]) return i+1;
        }
        return -1;
    }
};

int main() {
    int t;
    cin >> t;
    cin.ignore(); // To discard any leftover newline characters
    while (t--)   // while testcases exist
    {
        vector<int> arr;
        string input;
        getline(cin, input); // Read the entire line for the array elements
        stringstream ss(input);
        int number;
        while (ss >> number) {
            arr.push_back(number);
        }
```

```
        Solution ob;
        cout << ob.equilibriumPoint(arr) << endl;
        cout << "~" << endl;
    }
}
```

Time Complexity: O(n)
Space Complexity: O(n)

Question 5: Binary Search

Given a sorted array arr and an integer k, find the position(0-based indexing) at which k is present in the array using binary search.
Note: If multiple occurrences are there, please return the smallest index.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:
    int binarysearch(vector<int> &arr, int k) {
        int left=0,right=arr.size()-1;
        while(left<=right){
            int mid=(left+right)/2;
            if(arr[mid]==k) return mid;
            else if (arr[mid]<k) left=mid+1;
            else right=mid-1;
        }
        return -1;
    }
};

int main() {
    int t;
    cin >> t;
    while (t--) {
        int k;
        cin >> k;
        vector<int> arr;
        string input;
        cin.ignore();
```

```
        getline(cin, input);
        stringstream ss(input);
        int number;
        while (ss >> number) {
            arr.push_back(number);
        }
        Solution ob;
        int res = ob.binarysearch(arr, k);
        cout << res << endl;
        cout << "~" << endl;
    }
    return 0;
}
```

Time Complexity: O(log n)
Space Complexity: O(1)

QUESTION 6: Next Greater Element

Given an array arr[ ] of integers, the task is to find the next greater element for each element of the array in order of their appearance in the array. Next greater element of an element in the array is the nearest element on the right which is greater than the current element.
If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

void printNGE(int arr[], int n)
{
    stack<int> s;
    s.push(arr[0]);
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }
        while (s.empty() == false && s.top() < arr[i]) {
```

```
            cout << s.top() << " --> " << arr[i] << endl;
            s.pop();
        }
        s.push(arr[i]);
    }
    while (s.empty() == false) {
        cout << s.top() << " --> " << -1 << endl;
        s.pop();
    }
}
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Time Complexity: O(n)
Space Complexity: O(n)

QUESTION 7: Union of Two Arrays with Duplicate Elements

Given two arrays a[] and b[], the task is to find the number of elements in the union between these two arrays.
The Union of the two arrays can be defined as the set containing distinct elements from both arrays. If there are repetitions, then only one element occurrence should be there in the union.

Note: Elements are not necessarily distinct.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
  public:arrays.
    int findUnion(vector<int>& a, vector<int>& b) {
        // code here
        unordered_map<int,bool> map;
        for(int i:a) map[i]=true;
        for(int i:b){
```

```
            map[i]=true;
        }
        return map.size();
    }
};

int main() {
    int t;
    cin >> t;
    cin.ignore();

    while (t--) {
        vector<int> a;
        vector<int> b;

        string input;
        getline(cin, input); // Read the entire line for the array elements
        stringstream ss(input);
        int number;
        while (ss >> number) {
            a.push_back(number);
        }

        getline(cin, input); // Read the entire line for the array elements
        stringstream ss2(input);
        while (ss2 >> number) {
            b.push_back(number);
        }

        Solution ob;
        cout << ob.findUnion(a, b) << endl;
        cout << '~' << endl;
    }
    return 0;
}
```

Time Complexity: O(n+m)
Space Complexity: O(n+m)