CHANDIGARH UNIVERSITY
UNIVERSITY INSTITUTE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Submitted By: Sandeep Kumar	**Submitted To: Santosh Sir**
Subject Name	Competitive Coding
Subject Code	20CSP-314
Branch	CSE
Semester	5th

NAME: Lipakshi
UID: 20BCS5082
SECTION:607/B

SUBJECT NAME: Competitive Coding Lab
SUBJECT CODE: 20CSP-314

Practical -2.1

Problem Satement 1

You have three stacks of cylinders where each cylinder has the same diameter, but they may vary in height. You can change the height of a stack by removing and discarding its topmost cylinder any number of times.

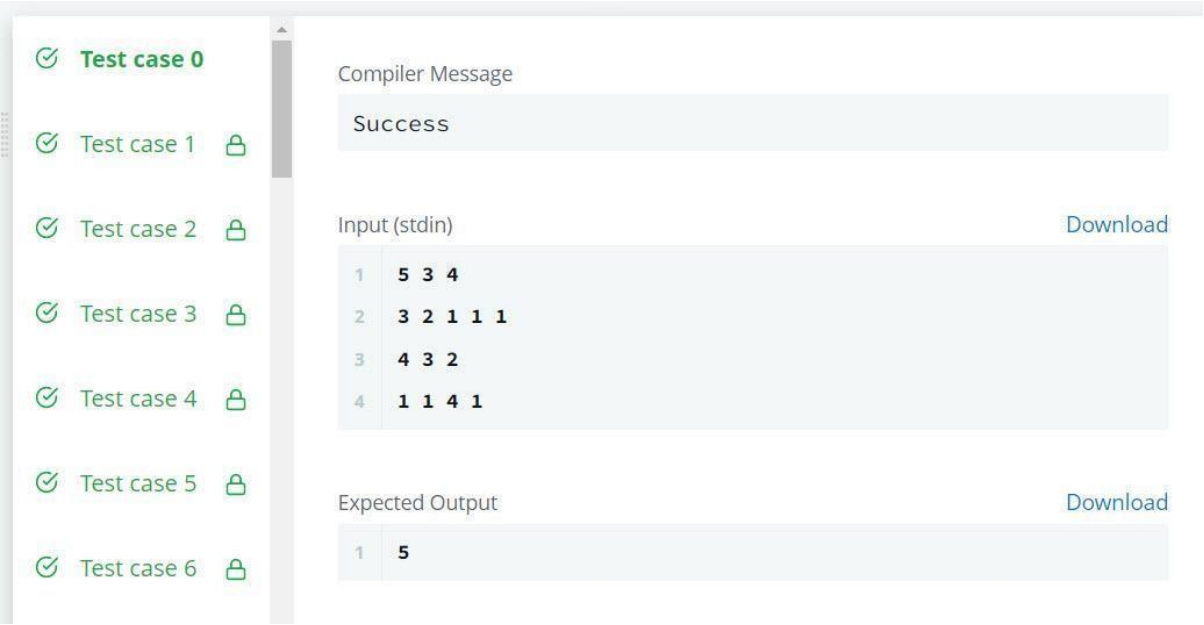
Find the maximum possible height of the stacks such that all of the stacks are exactly the same height. This means you must remove zero or more cylinders from the top of zero or more of the three stacks until they are all the same height, then return the height.

Code:

```
//Equal stacks
        int equalStacks(vector<int> h1, vector<int> h2, vector<int> h3) {
            int size=0;
            int l1=0,l2=0,l3=0;
            int top1=0,top2=0,top3=0;

            for(int i=0;i<h1.size();i++){
                l1+=h1[i];
            }
            for(int i=0;i<h2.size();i++){
                l2+=h2[i];
            }
            for(int i=0;i<h3.size();i++){
                l3+=h3[i];
            }

            do{
                if(l1==l2 && l1==l3){
                    size=l1;
                    break;
                }
                else{
                    int min_height=min(l1,min(l2, l3));
                    size=min_height;
                    if(l1>min_height && top1<h1.size()){
                        l1=l1-h1[top1];
                        h1[top1++]=0;
                    }
                    if(l2>min_height && top2<h2.size()){
                        l2=l2-h2[top2];
                        h2[top2++]=0;
                    }
                    if(l3>min_height && top3<h3.size()){
                        l3=l3-h3[top3];
                        h3[top3++]=0;
```

```
            }
        }
    }while(l1!=l2 || l1!=l3 || l2!=l3);

    return size;
}
```

Output:



Problem Statement 2

Alexa has two stacks of non-negative integers, stack and stack where index denotes the top of the stack. Alexa challenges Nick to play the following game:

- In each move, Nick can remove one integer from the top of either stack or stack .
- Nick keeps a running sum of the integers he removes from the two stacks.
- Nick is disqualified from the game if, at any point, his running sum becomes greater than some integer given at the beginning of the game.

- Nick's *final score* is the total number of integers he has removed from the two stacks.

Given , , and for games, find the maximum possible score Nick can achieve.

Example

The maximum number of values Nick can remove is . There are two sets of choices with this result.

1. Remove from with a sum of .
2. Remove from and from with a sum of .

Function Description

Complete the *twoStacks* function in the editor below.

twoStacks has the following parameters: - *int maxSum:* the maximum allowed sum

- *int a[n]:* the first stack

- *int b[m]:* the second stack

Returns

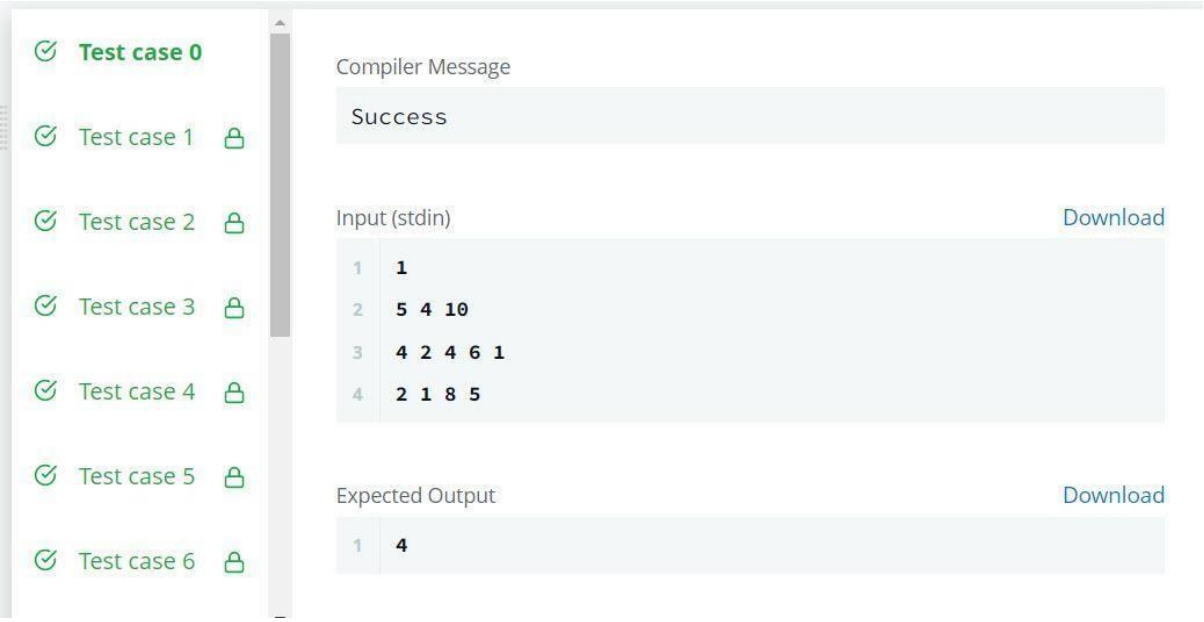
- *int:* the maximum number of selections Nick can make

Code:

```
//Game of two stacks
        int twoStacks(int maxSum, vector<int> a, vector<int> b) {
            int count=0, asize=a.size(), bsize=b.size(), acount=0, bcount=0;
            int currSum=0;

            while(acount<asize){
                if(currSum+a[acount]>maxSum){
                    break;
                }
                currSum+=a[acount++];
            }

            int res=acount;

            while(bcount<bsize){
                currSum+=b[bcount++];
                while(currSum>maxSum && acount>0){
                    currSum-=a[--acount];
                }
                res=(currSum<=maxSum)?max(res, acount+bcount):res;
            }
```

```
    return res;
}
```

Output:



Problem Statement 3

A bracket is considered to be any one of the following characters: (,), {, }, [, or].
Two brackets are considered to be a *matched pair* if the an opening bracket (i.e., (, [, or {) occurs to the left of a closing bracket (i.e.,),], or }) *of the exact same type*. There are three types of matched pairs of brackets: [], {}, and ().
A matching pair of brackets is *not balanced* if the set of brackets it encloses are not matched. For example, {[(])} is not balanced because the contents in between { and } are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, (, and the pair of parentheses encloses a single, unbalanced closing square bracket,].
By this logic, we say a sequence of brackets is *balanced* if the following conditions are met:

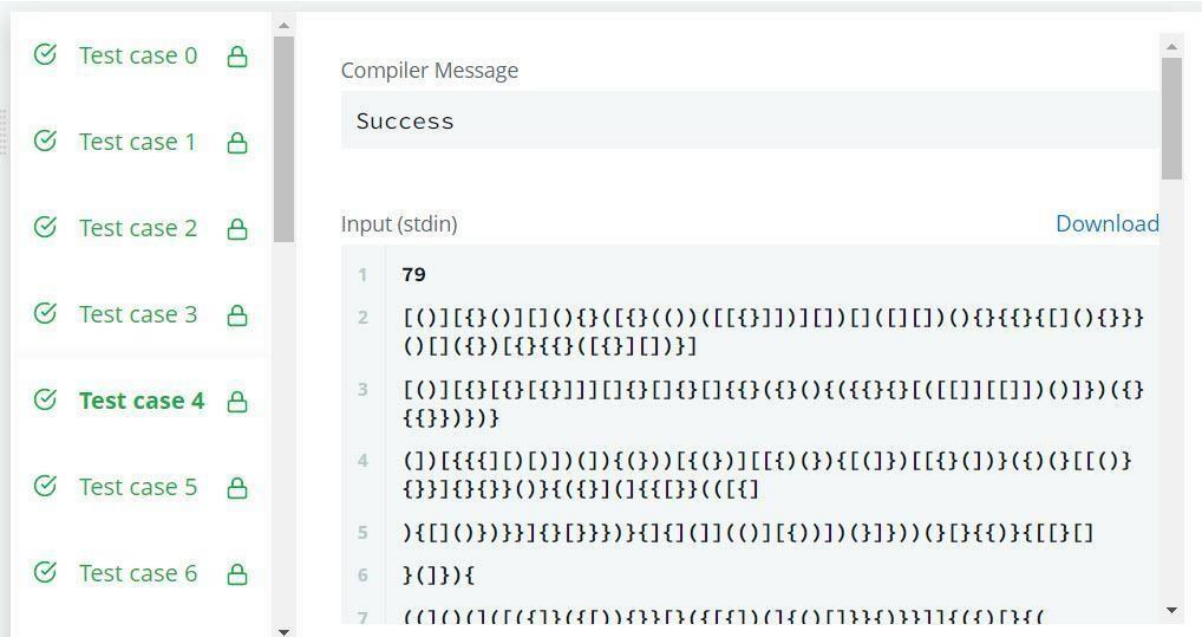
- It contains no unmatched brackets.
- The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

Given n strings of brackets, determine whether each sequence of brackets is balanced. If a string is balanced, return YES. Otherwise, return NO.

Code:

```
//Balance bracket
        string isBalanced(string s) {
            stack<char> st;
            for(int i=0;i<s.length();i++){
                char ch=s[i];
                if(ch=='{' || ch=='['||ch=='('){
                    st.push(ch);
                    continue;
                }
                else if(st.empty()){
                    return "NO";
                }
                char top=st.top();
                switch(ch){
                    case '}':
                    if(top=='{'){
                        st.pop();
                    }
                    else{
                        return "NO";
                    }
                    break;
                    case ']':
                    if(top=='['){
                        st.pop();
                    }
                    else{
                        return "NO";
                    }
                    break;
                    case ')':
                    if(top=='('){
                        st.pop();
                    }
                    else{
                        return "NO";
                    }
                    break;
                }
            }

            if(!st.empty()){
                return "NO";
            }

            return "YES";
        }
```

Output:



Problem Statement 4

You are given Q queries. Each query consists of a single number N. You can perform any of the 2 operations on in each move:

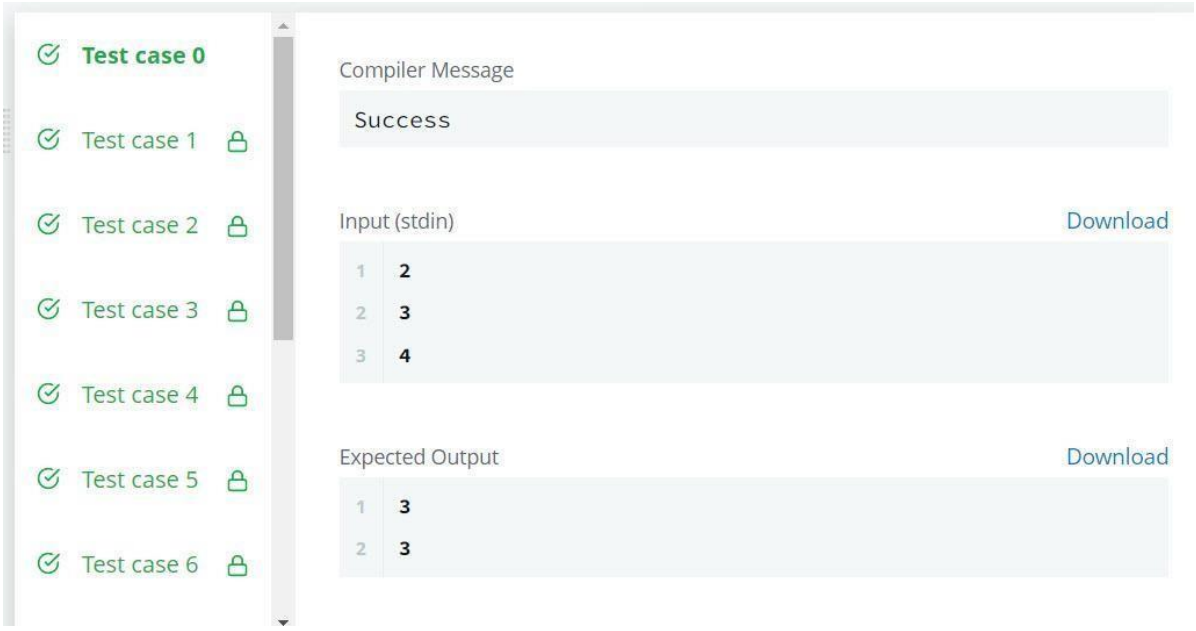
1: If we take 2 integers a and b where N=a X b (a!=1, b!=1), then we can change N=max(a,b).
2: Decrease the value of N by 1.
Determine the minimum number of moves required to reduce the value of N to 0.

Code:

```
//Down to zero
        int downToZero(int n) {
            int dp[1000000+1]={0};
            queue<int> q;

            q.push(n);
            while(q.empty()==false){
                int x=q.front();
                q.pop();

                if(x==0){
                    break;
                }
```

```
            if(dp[x-1]==0){
                dp[x-1]=dp[x]+1;
                q.push(x-1);
            }

            for(int i=sqrt(x);i>=2;i--){
                if(x%i==0 && dp[x/i]==0){
                    dp[x/i]=dp[x]+1;
                    q.push(x/i);
                }
            }
        }

        return dp[0];
    }
```

Output:



Problem Statement 5

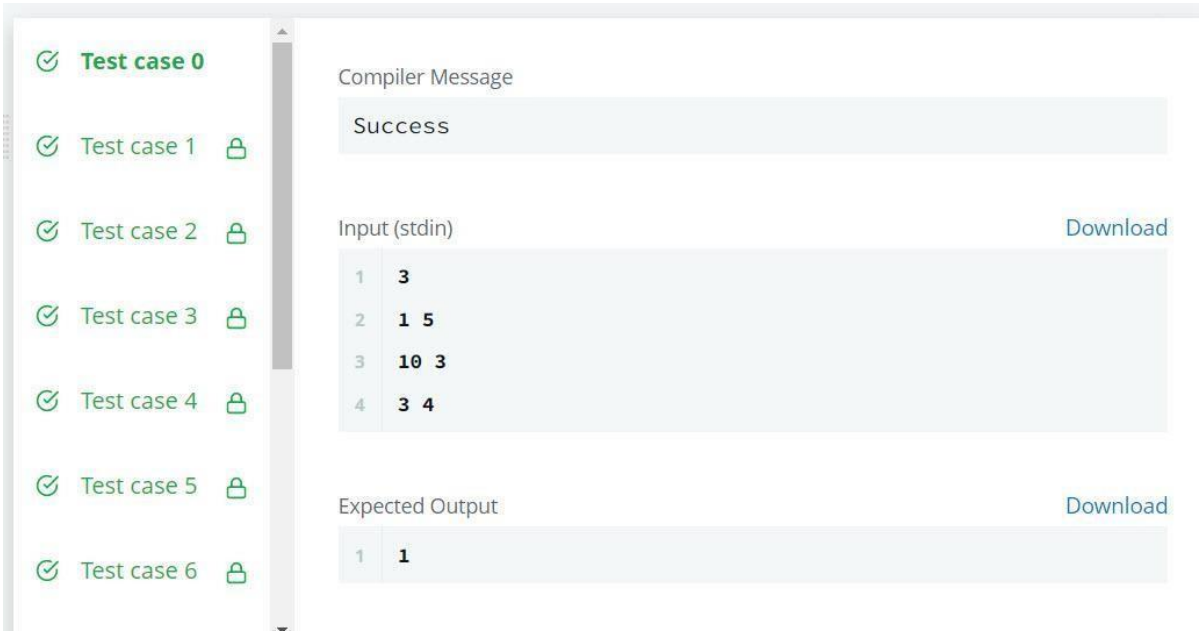
Suppose there is a circle. There are N petrol pumps on that circle. Petrol pumps are numbered 0 to (N-1) (both inclusive). You have two pieces of information corresponding to each of the petrol pump: (1) the amount of petrol that particular petrol pump will give, and (2) the distance from that petrol pump to the next petrol pump.

Initially, you have a tank of infinite capacity carrying no petrol. You can start the tour at any of the petrol pumps. Calculate the first point from where the truck will be able to complete the circle. Consider that the truck will stop at each of the petrol pumps. The truck will move one kilometer for each litre of the petrol.

Code:

```
//Truck tour
        int truckTour(vector<vector<int>> petrolpumps) {
        int ltr=0; int pp=0;
           for(int i=0+pp;i<petrolpumps.size()+pp;i++){
              ltr=ltr+petrolpumps[i%(petrolpumps.size())][0]-
        petrolpumps[i%(petrolpumps.size())][1];
              if(ltr<0){
                 ltr=0;
                 i=pp-1;
                 pp++;
                 continue;
              }
           }
           return pp-1;
        }
```

Output:



Observations/Discussions/ Complexity Analysis:

- Equal stacks: O(n) As we are traversing all the stacks once to calculate their individual sums.
- Game of two stacks: O(n) As we are traversing first stack completely then we start traversing the second stack.
- Balanced brackets: O(n) As we are traversing the string of brackets once.
- Down to zero: O(n3/2) As for each number we are looking for its factors using a for loop from n1/2 to 2.
- Truck tour: O(n2) As we start from an index and try to traverse all the rest of the elements.

Learning Outcomes

1. Arrays in data structures
2. Different conditions and loops
3. Advance learning of C++/C