2100030424

P.Sneharika

LAB SESSION 04:
STACKS AND QUEUES APPLICATION

Pre-Lab:

1. You are given an array *A* of *N* integers. Now, two functions F(X) and G(X) are defined:
   F(X) : This is the smallest number *Z* such that X<Z≤N and A[X]<A[Z]
   G(X) : This is the smallest number *Z* such that X<Z≤N and A[X]>A[Z]
   Now, you need to find for each index *i* of this array G(F(i)), where 1≤i≤N . If such a number does not exist, for a particular index *i*, output *1* as its answer. If such a number does exist, output A[G(F(i))]

   Input :

   The first line contains a single integer *N* denoting the size of array *A*. Each of the next *N* lines contains a single integer, where the integer on the ith line denotes A[i].

   Output :

   Print *N* space separated integers on a single line, where the ith integer denotes A[G(F(i))] or *1*, if G(F(i)) does not exist.

   Constraints:

   1≤N≤30000
   0≤A[i]≤1018

   Sample Input
   8
   3
   7
   1
   7
   8
   4
   5
   2

   Understand the problem, write the output, and give the explanation for the output sequence.

Solution:

```java
import java.io.*;
import java.util.*;

public class Main {
     public static void main(String[] args) throws IOException
{
          Reader R=new Reader();
          int n=R.nextInt(); // Size of array
          long a[]=new long[n+1];
          a[0]=-1;

          for(int i=1;i<=n;i++) // Get array data
               a[i]=R.nextLong();
          int f[]=new int[n+1];
          int g[]=new int[n+1];

          ArrayDeque<Integer> dq=new ArrayDeque<Integer>();
          dq.offerLast(1);
          for(int i=2;i<=n;i++)
          {
               while(!dq.isEmpty() && a[dq.peekLast()]<a[i])
                    f[dq.pollLast()]=i;
               dq.offerLast(i);
          }
          while(!dq.isEmpty())
               f[dq.pollLast()]=-1;
          dq.offerLast(1);
          for(int i=2;i<=n;i++)
          {
               while(!dq.isEmpty() && a[dq.peekLast()]>a[i])
                    g[dq.pollLast()]=i;
               dq.offerLast(i);
          }
          while(!dq.isEmpty())
               g[dq.pollLast()]=-1;

    StringBuilder sb = new StringBuilder();
          for(int i=1; i<=n; i++){
               if(f[i]==-1 || g[f[i]]==-1 )
          sb.append(-1 + " ");
               else
                    sb.append(a[g[f[i]]] + " ");
          }
          System.out.println(sb.toString());
     }

     static class Reader {
        final private int BUFFER_SIZE = 1 << 16;
        private DataInputStream din;
        private byte[] buffer;
```

```
        private int bufferPointer, bytesRead;

        public Reader() {
            din = new DataInputStream(System.in);
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public Reader(String file_name) throws IOException {
            din = new DataInputStream(new
FileInputStream(file_name));
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public String readLine() throws IOException {
            byte[] buf = new byte[64]; // line length
            int cnt = 0, c;
            while ((c = read()) != -1) {
                if (c == '\n')
                    break;
                buf[cnt++] = (byte) c;
            }
            return new String(buf, 0, cnt);
        }

        public int nextInt() throws IOException {
            int ret = 0;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();
            do {
                ret = ret * 10 + c - '0';
            } while ((c = read()) >= '0' && c <= '9');

            if (neg)
                return -ret;
            return ret;
        }

        public char nextChar() throws IOException {
            int ret = 0;
            byte c = read();
            while (c <= ' ')
                c = read();

            return (char) c;
        }
```

```
        public long nextLong() throws IOException {
            long ret = 0;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();
            do {
                ret = ret * 10 + c - '0';
            }
            while ((c = read()) >= '0' && c <= '9');
            if (neg)
                return -ret;
            return ret;
        }

        public double nextDouble() throws IOException {
            double ret = 0, div = 1;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();

            do {
                ret = ret * 10 + c - '0';
            }
            while ((c = read()) >= '0' && c <= '9');

            if (c == '.') {
                while ((c = read()) >= '0' && c <= '9') {
                    ret += (c - '0') / (div *= 10);
                }
            }

            if (neg)
                return -ret;
            return ret;
        }

        private void fillBuffer() throws IOException {
            bytesRead = din.read(buffer, bufferPointer = 0,
BUFFER_SIZE);
            if (bytesRead == -1)
                buffer[0] = -1;
        }

        private byte read() throws IOException {
```

```
            if (bufferPointer == bytesRead)
                fillBuffer();
            return buffer[bufferPointer++];
        }

        public void close() throws IOException {
            if (din == null)
                return;
            din.close();
        }
    }
}
```

2. Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
    1. Only one disk can be moved at a time.
    2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
    3. No disk may be placed on top of a smaller disk

    Sample Input:

    - 3

    Sample Output:

    - Disk 1 moved from A to C
    - Disk 2 moved from A to B
    - Disk 1 moved from C to B
    - Disk 3 moved from A to C
    - Disk 1 moved from B to A
    - Disk 2 moved from B to C
    - Disk 1 moved from A to C

Solution:

```
import java.util.*;
import java.io.*;
import java.math.*;
class Tower
```

```
{
    static void Hanoi(int n, char n1,char n2, char n3)
    {
        if (n == 0)
        {
            return;
        }
        Hanoi(n - 1, n1,n3, n2);
        System.out.println("Disk " + n + " moved from  " +
                           n1 +" to " + n2);
        Hanoi(n - 1, n3,n2, n1);
    }
    public static void  main(String args[])
    {
        int n = 3;
        Hanoi(n, 'A', 'C', 'B');
    }
}
```

In-Lab:

1. Postfix notation (sometimes called "Reverse Polish Notation" or "RPN") explicitly specifies the operation order of a mathematical expression. Here are some examples of normal (infix) expressions followed by their postfix equivalents:

   - "1 + 2" == "1 2 +"
   - "3 - 1" == "3 1 -"
   - "2 * 3" == "2 3 *"
   - "3 + (4 * 5)" == "4 5 * 3 +"
   - "5 + ((1 + 2) * 4) - 3" == "5 1 2 + 4 * + 3 -"

   Write a Java program to convert and evaluate simple postfix expressions using a stack

Solution:

```
import java.io.*;
import java.lang.reflect.Array;
import java.util.*;
import java.lang.*;
public class favtutor{

    public static int precedence(char x){

        if(x=='^'){
            return 2;
        }
        else if(x=='*' || x=='/'){
            return 1;
```

```
        }
        else if(x=='+' || x=='-'){
            return 0;
        }
        return -1;
    }

    public static String InfixToPostfix(String str){

        Stack<Character>stk= new Stack<>();
        String ans="";

        int n= str.length();

        for (int i = 0; i <n ; i++) {
            char x= str.charAt(i);

            if(x>='0' && x<='9'){
                ans+=x;
            }

            else if(x=='('){
                stk.push('(');
            }
            else if(x==')'){

                while(!stk.isEmpty() && stk.peek()!='('){
                    ans+=stk.pop();
                }
                if(!stk.isEmpty()){
                    stk.pop();
                }

            }
            else{

                while(!stk.isEmpty() &&
precedence(stk.peek())>=precedence(x)){
                    ans+=stk.pop();
                }
                stk.push(x);

            }
        }
        while(!stk.isEmpty()){
            ans+=stk.pop();
        }
        return ans;
    }
}
```

2. A letter means push and an asterisk means pop in the following sequence. Give the sequence of values returned by the pop operations when this sequence of operations is performed on an initially empty LIFO stack.

Sample Input:
E A S * Y * Q U E * * * S T * * * I O * N * * *HI

Sample Output:

HI

Solution:

```
class Stack
{
    private int arr[];
    private int top;
    private int capacity;
    Stack(int size)
    {
        arr = new int[size];
        capacity = size;
        top = -1;
    }
    public void push(int x)
    {
        if (isFull())
        {
            System.out.println("Overflow\nProgram
Terminated\n");
            System.exit(-1);
        }
        System.out.println("Inserting " + x);
        arr[++top] = x;
    }
    public int pop()
    {
        if (isEmpty())
        {
            System.out.println("Underflow\nProgram
Terminated");
            System.exit(-1);
        }
        System.out.println("Removing " + peek());
        return arr[top--];
    }
    public int peek()
    {
        if (!isEmpty()) {
            return arr[top];
```

```
        }
        else {
            System.exit(-1);
        }

        return -1;
    }
    public int size() {
        return top + 1;
    }
    public boolean isEmpty() {
        return top == -1;
    }
    public boolean isFull() {
        return top == capacity - 1;
    }
}
class Main
{
    public static void main (String[] args)
    {
        Stack stack = new Stack(3);
        stack.push('E');
        stack.push('A');
        stack.push('S');
        stack.pop();
        stack.push('Y');
        stack.pop();
        stack.push('Q');
        stack.push('U');
        stack.push('E');
        stack.pop();
        stack.pop();
        stack.pop();
        stack.push('S');
        stack.push('T');
        stack.pop();
        stack.pop();
        stack.pop();
        stack.push('I');
        stack.push('O');
        stack.pop();
        stack.push('N');
        stack.pop();
        stack.pop();
        stack.pop();
        stack.push('H');
        stack.push('I');
        System.out.println("The top element is " +
stack.peek());
```

```
        System.out.println("The stack size is " +
stack.size());
        if (stack.isEmpty()) {
            System.out.println("The stack is empty");
        }
        else {
            System.out.println("The stack is not empty");
        }
    }
}
```

3. There are n gas stations along a circular route, where the amount of gas at the i^{th} station is gas[i].

   You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the i^{th} station to its next $(i + 1)^{th}$ station. You begin the journey with an empty tank at one of the gas stations.

   Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**

   Example 1:

   Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
   Output: 3
   Explanation:
   Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
   Travel to station 4. Your tank = 4 - 1 + 5 = 8
   Travel to station 0. Your tank = 8 - 2 + 1 = 7
   Travel to station 1. Your tank = 7 - 3 + 2 = 6
   Travel to station 2. Your tank = 6 - 4 + 3 = 5
   Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
   Therefore, return 3 as the starting index.

   Example 2:
   Input: gas = [2,3,4], cost = [3,4,3]
   Output: -1
   Explanation:
   You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
   Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
   Travel to station 0. Your tank = 4 - 3 + 2 = 3
   Travel to station 1. Your tank = 3 - 3 + 3 = 3
   You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
   Therefore, you can't travel around the circuit once no matter where you start.
   Constraints:

1 n == gas.length == cost.length
2 1 <= n <= 10^5
3 0 <= gas[i], cost[i] <= 10^4

Solution:

```
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int currRemaining = 0;
        int totalRemaining = 0;
        int start = 0;
        for (int i = 0; i < gas.length; i ++){
            int remaining = gas[i] - cost[i];
            if (currRemaining < 0){
                start = i;
                currRemaining = remaining;
            }
            else{
                currRemaining += remaining;
            }
            totalRemaining += remaining;
        }
        if (totalRemaining < 0)
            return -1;
        else
            return start;
    }
}
```

Post-Lab:

1. You are given two arrays each of size n, a and b consisting of the first n positive integers each exactly once, that is, they are permutations.

Your task is to find the minimum time required to make both the arrays empty. The following two types of operations can be performed any number of times each taking 1 second:

- In the first operation, you are allowed to rotate the first array clockwise.
- In the second operation, when the first element of both the arrays is the same, they are removed from both the arrays and the process continues.

Input format

- The first line contains an integer n, denoting the size of the array.
- The second line contains the elements of array a.
- The third line contains the elements of array b.

Output format

Print the total time taken required to empty both the array.

Constraints

1≤n≤100

Sample Input

3
1 3 2
2 3 1

Sample Output

6

Explanation

Perform operation 1 to make a = 3, 2, 1
Perform operation 1 to make a = 2, 1, 3
Now perform operation 2 to make a = 1, 3 and b = 3, 1
Perform operation 1 to make a = 3, 1
Now perform operation 2 to make a = 1 and b = 1
Now perform operation 2 to make a = {} and b = {}

Solution:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.*;
import java.io.*;
class TestClass {
    public static void main(String args[] ) throws Exception {

     BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine());
        LinkedList<Integer> a = new LinkedList<Integer>();
        LinkedList<Integer> b = new LinkedList<Integer>();
        insert(a,n,br.readLine());
        insert(b,n,br.readLine());
        int count = 0;
        while(a.size()!=0 && b.size()!=0) {
            while(a.peek()!=b.peek()){
                rotate(a);
                count++;
            }
            if(a.peek()==b.peek()){
                remove(a,b);
                count++;
            }
        }
        System.out.println(count);

    }
```

```
    public static void remove(LinkedList<Integer> a,
LinkedList<Integer> b){
     a.remove();
     b.remove();
    }
    public static void rotate(LinkedList<Integer> arr){
        int t = arr.remove();
        arr.addLast(t);
    }
    public static void insert(LinkedList<Integer> arr,int n,
String in) throws NumberFormatException, IOException{
        for(String i:in.split(" ")){
            arr.add(Integer.parseInt(i));
        }
    }
```

2. Given a string S of parentheses '(' or ')'. The task is to find a minimum number of parentheses '(' or ')' (at any positions) we must add to make the resulting parentheses string is valid.
   Examples:

   Input: str = "())"
   Output: 1
   One '(' is required at beginning.

   Give your explanation for the above problem using Stack or Queue structure.
   Find out the minimum number of parentheses needed to complete the input

   1. ()[(){()}]
   2. []{()()}
   3. ((()))()()
   4. ())((()

Solution:

```
public class GFG
{
    static int minParentheses(String p)
    {
        int bal = 0;
        int ans = 0;
        for (int i = 0; i < p.length(); ++i) {
            bal += p.charAt(i) == '(' ? 1 : -1;

            if (bal == -1) {
                ans += 1;
```

```
                bal += 1;
            }
        }
        return bal + ans;
    }
    public static void main(String args[])
    {
        String p = "())";
        System.out.println(minParentheses(p));
    }
}
```