Stack & Queue
DATA STRUCTURE

Content

1. . Parenthesis checker

Statement : Given an expression string exp. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.
For example, the program should print 'balanced' for exp = “[()]{}{[()()]()}” and 'not balanced' for exp = “[(])”

```
static boolean ispar(String x){
        Stack<Character> stack = new Stack<>();
        for(int i=0;i<x.length();i++){
                char c = x.charAt(i);
                if(c=='(' || c=='{' || c=='['
                        stack.push(c);
                else{
                        if(!stack.isEmpty()){
                                char top = stack.pop();
                                switch(c){
                                        case ')':
                                                if(top!='(')
                                                        return false;
                                                break;
                                        case '}':
                                                if(top!='{')
                                                        return false;
                                                break;
                                        case ']':
                                                if(top!='[')
                                                        return false;
                                }
                        }else
                                return false;
                }
        }
        if(!stack.isEmpty())
                return false;
        return true;
}
```

2. Get Minimum Element from Stack

Statement: You are given **N** elements and your task is to Implement a Stack in which you can get minimum element in O(1) time.

```
public GfG(){
    Stack<Integer> s;
    int minEle;

    public GfG(){
        s = new Stack<>();
        minEle = -1;
    }
    //returns min element from stack
    int getMin(){
        if(!s.empty()){
            return minEle;
        }
        return -1;
    }
    //returns poped element from stack
    int pop(){
        int res = -1;
        if(!s.empty()){
            int y = s.pop();
            if(y<minEle){
                res = minEle;
                minEle = 2*minEle - y;
            }else{
                res = y;
            }
        }
        return res;
    }

    //push element x into the stack
    void push(int x){
        if(s.empty()){
            s.push(x);
            minEle = x;
        }else{
            if(x>=minEle){
                s.push(x);
```

```
                }else{
                        s.push(2*x-minEle);
                        minEle = x;
                }
            }
        }
}
```

3Next Larger Element

Statement : Given an array **A** of size **N** having distinct elements, the task is to find the next greater element for each element of the array in order of their appearance in the array. If no such element exists, output **-1**

```
class GFG{
    static void nGE(int n,int arr[]){
        Stack<Integer> s = new Stack<>();
        int arr1[] = new int[n];
        for(int i=0;i<n;i++){
            arr1[i] = -1;
        }
        s.push(0);
        // iterating from 0 to n-1
        for(int i=1;i<n;i++) {
            while(!s.isEmpty()){
                int top = s.peek();
                if(arr[i]<arr[top])
                    break;
                else{
                    arr1[top] = arr[i];
                    s.pop();
                }
            }
            s.push(i);
        }

        for (int i = 0; i < n; i++)
            System.out.print(arr1[i]+" ");
        }
    }
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
            int test = sc.nextInt();
            for(int t=0;t<test;t++){
                int n = sc.nextInt();
                int arr[] = new int[n];
                for(int i=0;i<n;i++){
                    arr[i] = sc.nextInt();
                }
                nGE(n,arr);
                System.out.println();
            }
        }
}
```

4. Sorting of Stack

Given a stack, the task is to sort it such that the top of the stack has the greatest element.

```
public Stack<Integer> sort(Stack<Integer> s){
        Stack<Integer> s1 = new Stack<>();
        Stack<Integer> s2 = new Stack<>();
        while(!s.isEmpty()){
            int v = s.pop();
            while(!s1.isEmpty() && s1.peek()>v)
                s2.push(s1.pop());
            s1.push(v);
            while(!s2.isEmpty())
                s1.push(s2.pop());
        }
        return s1;
}
```

5. Evaluation of Postfix Expression

Given a postfix expression, the task is to evaluate the expression and print the final value. Operators will only include the basic arithmetic operators like ***,/,+ and -**.

```
public static int evaluatePostFix(String exp){
        Stack<Integer> stack = new Stack<>();
        for(int i=0;i<exp.length();i++){
            char c = exp.charAt(i);
            if(c-'0'>=0 && c-'0'<10){
                stack.push(c-'0');
```

```
            }else{
                int b = stack.pop();
                int a = stack.pop();
                switch(c){
                    case '*':
                        stack.push(a*b);
                        break;
                    case '/':
                        stack.push(a/b);
                        break;
                    case '+':
                        stack.push(a+b);
                        break;
                    case '-':
                        stack.push(a-b);
                }
            }
        }
        return stack.pop();
}
```

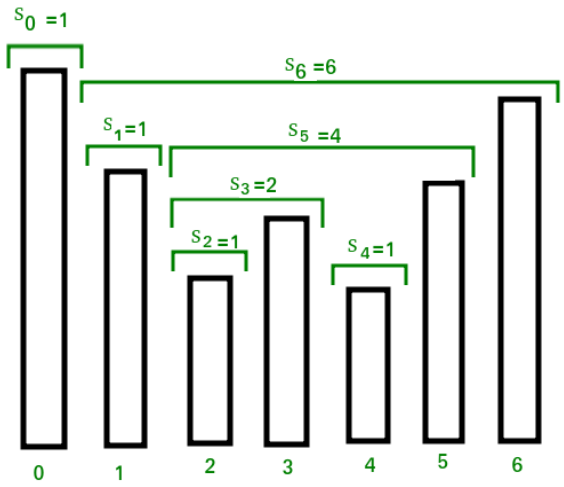
6. Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate the span of stock's price for all **n** days.

The span $\mathbf{S_i}$ of the stock's price on a given day **I** is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}.

Eg. N = 7 100 80 60 70 60 75 85 output: 1 1 1 2 1 4 6




```
public static int[] calculateSpan(int price[], int n){
        Stack<Integer> st = new Stack<>();
        int ans[] = new int[n];
        ans[0] = 1;
        st.push(0);
        for(int i=1;i<n;i++){
                ans[i] = 1;
                while(!st.isEmpty()){
                        int top = st.peek();
                        if(price[top]>price[i]){
                                break;
                        }else{
                                st.pop();
                                ans[i] += ans[top];
                        }
                }
                st.push(i);
        }
        return ans;
}
```

7 Maximum of all subarrays of size k

Statement : Given an array **A** and an integer **K**. Find the maximum for each and every contiguous subarray of size K.

```
static void printMax(int arr[],int n,int k){
        Deque<Integer> Q = new LinkedList<Integer>();
        int i;
        for(i=0;i<k;i++){
```

```
            while(!Q.isEmpty() && arr[i]>=arr[Q.peekLast()])
                Q.removeLast();
            Q.addLast(i);
        }
        for(;i<n;i++){
            System.out.print(arr[Q.peekFirst()]+" ");

            while(!Q.isEmpty() && Q.peekFirst()<=i-k)
                Q.removeFirst();

            while(!Q.isEmpty() && arr[i]>=arr[Q.peekLast()])
                Q.removeLast();

            Q.addLast(i);
        }
        System.out.print(arr[Q.peekFirst()]);
}
public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        int test = sc.nextInt();
        for(int t = 0;t<test;t++){
            int n = sc.nextInt();
            int k = sc.nextInt();
            int arr[] = new int[n];
            for(int i=0;i<n;i++){
                arr[i] = sc.nextInt();
            }
            printMax(arr,n,k);
            System.out.println();
        }
}
```

8 Rotten Oranges

Statement : Given a matrix of dimension **r*c** where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:
0 : Empty cell
1 : Cells have fresh oranges
2 : Cells have rotten oranges

So, we have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can **r**ot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1]

(**up**, **down**, **left** and **right**) in unit time. If it is impossible to rot every orange then simply return -1.

```
class Node{
	int x,y,time;
	public Node(int x,int y,int time){
		this.x = x;
		this.y = y;
		this.time = time;
	}
}
class GFG {
	public static void main (String[] args) {
		Scanner sc = new Scanner(System.in);
		int test = sc.nextInt();
		for(int t=0;t<test;t++){
			int n = sc.nextInt();
			int m = sc.nextInt();
			int arr[][] = new int[n][m];
			for(int i=0;i<n;i++){
				for(int j=0;j<m;j++){
					arr[i][j] = sc.nextInt();
				}
			}
			//processing using BFS
			//create a queue
			Queue<Node> q = new LinkedList<>();
			for(int i=0;i<n;i++){
				for(int j=0;j<m;j++){
					if(arr[i][j]==2){
						q.add(new Node(i,j,0));
					}
				}
			}
			int res=0;
			while(!q.isEmpty()){
				Node node = q.remove();
				res = node.time;
				int x = node.x;
				int y = node.y;
				//top
				if(x-1>=0 && arr[x-1][y]==1){
```

```
                        arr[x-1][y]=2;
                        q.add(new Node(x-1,y,node.time+1));
                    }
                    //down
                    if(x+1<n && arr[x+1][y]==1){
                        arr[x+1][y]=2;
                        q.add(new Node(x+1,y,node.time+1));
                    }
                    //left
                    if(y-1>=0 && arr[x][y-1]==1){
                        arr[x][y-1]=2;
                        q.add(new Node(x,y-1,node.time+1));
                    }
                    //right
                    if(y+1<m && arr[x][y+1]==1){
                        arr[x][y+1]=2;
                        q.add(new Node(x,y+1,node.time+1));
                    }
                }
                //trverse again for any 1 left or not
                for(int i=0;i<n;i++){
                    for(int j=0;j<m;j++){
                        if(arr[i][j]==1){ //not possible to rot every orange
                            res = -1;
                            break;
                        }
                    }
                }
                System.out.println(res);
            }
        }
}
```

9 Circular tour

Statement : Suppose there is a circle. There are **N** petrol pumps on that circle. You will be given two sets of data.

1. The amount of petrol that every petrol pump has.

2. Distance from that petrol pump to the next petrol pump.

Your task is to complete the function **tour**() which returns an integer denoting the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity).

Note : Assume for 1 litre petrol, the truck can go 1 unit of distance.

```
int tour(int petrol[], int distance[]){
        int n = petrol.length;
        if(n==1)
                return 0;
        int start = 0;
        int end = 1;
        int curr_petrol = petrol[start] - distance[start];

        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while(end != start || curr_petrol < 0) {

                // If current amount of petrol in truck becomes less than 0, then
                // remove the starting petrol pump from tour
                while(curr_petrol < 0 && start != end) {
                        // Remove starting petrol pump. Change start
                        curr_petrol -= petrol[start] - distance[start];
                        start = (start + 1) % n;

                        // If 0 is being considered as start again, then there is no
                        // possible solution
                        if(start == 0)
                                return -1;
                        }
                        // Add a petrol pump to current tour
                        curr_petrol += petrol[end] - distance[end];

                        end = (end + 1)%n;
                }

                // Return starting point
                return start;
        }
```

10 LRU Cache

Statement : The task is to design and implement methods of an **LRU cache**. The class has two methods **get**() and **set**() which are defined as follows.
get(x) : Returns the value of the key **x** if the key exists in the cache otherwise returns **-1.**
set(x,y) : inserts the value if the key **x** is not already present. If the cache reaches its capacity it should invalidate the least recently used item before inserting the new item.
In the constructor of the class the size of the cache should be intitialized.

```
class LRUCache {
    Map<Integer,Integer> map;
    Deque<Integer> dq;
    int capacity;

    /*Inititalize an LRU cache with size N */
    public LRUCache(int N) {
        map = new HashMap<>();
        dq = new LinkedList<>();
        capacity = N;
    }

    //Returns the value of the key x if present else returns -1
    public int get(int x) {
        if(map.containsKey(x)){
            return map.get(x);
        }
        return -1;
    }

    //Sets the key x with value y in the LRU cache
    public void set(int x, int y) {
        if(!map.containsKey(x)){//key not present
            if(dq.size()==capacity){
                int f = dq.remove();
                map.remove(f);
            }
        }else{//key present
            int index=0,i=0;
            Iterator<Integer> itr = dq.iterator();
            while(itr.hasNext()){
                if(itr.next()==x){
                    index = i;
                    break;
                }
```

```
                    i++;
                }
                dq.remove(index);
            }
            map.put(x,y);
            dq.add(x);
        }
    }
```