**Q1. Balanced Paranthesis**

Given an expression string **A**, examine whether the pairs and the orders of “{“,”}”, ”(“,”)”, ”[“,”]” are correct in **A**.

Refer to the examples for more clarity.

**Logic -**

1. We traverse the given string from the left. If the i-th character is an opening bracket, we push it onto the stack.
2. If it is a closing bracket, we check for the impossible case 2 and case 3. If they are being violated, then we can simply return 0.
3. Otherwise, we can pop the topmost bracket from the stack.  
   To check for case 1, if our stack is not empty at the end of our traversal, then we can say that the brackets are not correctly matched.
4. If all the conditions are fulfilled, then we can return 1.

**Code -**

public class Solution {

public int solve(String A) {

HashMap<Character, Character> map = new HashMap<Character, Character>();

map.put('}', '{');

map.put(']', '[');

map.put(')', '(');

Stack<Character> st = new Stack<Character>();

for(int i=0;i<A.length();i++) {

if(map.containsKey(A.charAt(i))) {

if(st.isEmpty()) return 0;

Character temp = st.pop();

if(map.get(A.charAt(i)) != temp) return 0;

} else {

st.push(A.charAt(i));

}

}

if(!st.isEmpty()) return 0;

return 1;

}

}

**Q2. Double Character Trouble**

You are given a string **A**.

An operation on the string is defined as follows:

Remove the first occurrence of the same consecutive characters. eg for a string "abbcd", the first occurrence of same consecutive characters is "bb".

Therefore the string after this operation will be "acd".

Keep performing this operation on the string until there are no more occurrences of the same consecutive characters and return the final string.

**Logic -**

Consider an example string abba.  
When we remove the “bb”, the remaining string is “aa” which has to be removed as well.  
So we need to keep track of the characters before the first occurrence of similar consecutive characters.  
We can do this using a stack.  
We keep pushing the characters in a stack, if the current character is equal to the top of the stack,  
we pop from the stack since they represent  
a pair of similar characters.  
Finally, we print the stack in reverse.

**Code -**

public class Solution {

public String solve(String A) {

if(A.length() <= 1) return A;

Stack<Character> st = new Stack<Character>();

StringBuilder ans = new StringBuilder();

for(int i=0;i<A.length();i++) {

if(st.isEmpty() || st.peek() != A.charAt(i)) {

st.push(A.charAt(i));

} else {

st.pop();

}

}

while(!st.isEmpty()) {

ans.append(st.pop());

}

return ans.reverse().toString();

}

}

**Q3. Evaluate Expression**

An arithmetic expression is given by a string array **A** of size **N**. Evaluate the value of an arithmetic expression in **Reverse Polish Notation**.

Valid operators are +, -, \*, /. Each string may be an integer or an operator.

**Logic -**

When you encounter an operator, that is when you need the top 2 numbers on the stack, perform the operation on them, and put them on the stack.

**Code -**

public class Solution {  
 public int evalRPN(ArrayList < String > A) {  
 Stack < Integer > values = new Stack < Integer > ();  
 int first;  
 int second;  
 for (String str: A) {  
 // on encountering an operator, pop the top two elements from the stack,   
 // perform the operation and push that back into the stack  
 if (equal(str, "+")) {  
 second = values.pop();  
 first = values.pop();  
 values.push(first + second);  
 } else if (equal(str, "\*")) {  
 second = values.pop();  
 first = values.pop();  
 values.push(first \* second);  
 } else if (equal(str, "/")) {  
 second = values.pop();  
 first = values.pop();  
 values.push(first / second);  
 } else if (equal(str, "-")) {  
 second = values.pop();  
 first = values.pop();  
 values.push(first - second);  
 } else {  
 first = Integer.parseInt(str);  
 values.push(first);  
 }  
 }  
 return values.peek();  
 }  
 public boolean equal(String str1, String str2) {  
 return str1.equalsIgnoreCase(str2);  
 }  
}

**Q4. Check two bracket expressions**

Given two strings **A** and **B**. Each string represents an expression consisting of **lowercase English alphabets, '+', '-', '(' and ')'**.

The task is to compare them and check if they are similar. If they are identical, return 1 else, return 0.

**NOTE:** It may be assumed that there are at most 26 operands from ‘a’ to ‘z’, and every operand appears only once.

**Logic -**

1. We will evaluate each expression one by one.
2. Calculate the sign on each operand present from ‘a’ to ‘z’ for first string A.
3. Now, repeat the same process on string B but with the opposite sign.
4. If the total sign on each operand is 0, return 1.
5. Else return 0.

**Code -**

class Solution:

# @param A : string

# @param B : string

# @return an integer

def removeBrackets(self, A):

flag = False

hashA = {}

stack = []

i = 0

while(i<len(A)):

if A[i] == '-':

if A[i+1] == '(':

stack.append('-')

flag = not flag

i+=2

else:

temp2 = '+' if flag else '-'

temp = temp2+A[i+1]

if temp not in hashA:

hashA[temp] = 0

hashA[temp] += 1

i+=2

elif A[i] == '+':

if A[i+1] == '(':

stack.append('+')

i+=2

else:

temp2 = '-' if flag else '+'

temp = temp2+A[i+1]

if temp not in hashA:

hashA[temp] = 0

hashA[temp] += 1

i+=2

elif A[i] == '(':

stack.append('+')

i+=1

elif A[i] == ')':

if stack[-1] == '-':

flag = not flag

stack.pop()

i+=1

else:

temp2 = '-' if flag else '+'

temp = temp2+A[i]

if temp not in hashA:

hashA[temp] = 0

hashA[temp] += 1

i+=1

return hashA

def solve(self, A, B):

hashA = self.removeBrackets(A)

hashB = self.removeBrackets(B)

# Comparing the 2 hashmaps

if (len(hashA) != len(hashB)): return 0

for i in hashA.keys():

if i not in hashB or hashA[i] != hashB[i]:

return 0

return 1

**Q5. Redundant Braces**

Given a string **A** denoting an expression. It contains the following operators '+', '-', '\*', '/'.

Check whether A has redundant braces or not.

**NOTE:** A will be always a valid expression and will not contain any white spaces.

**Logic -**

1. We keep pushing elements onto the stack till we encounter ')'. When we encounter ')', if we get the ‘)’ and the top of the stack it means we have found a redundant bracket, else we start popping elements until we find a matching '('.

**Code -**

class Solution:

# @param A : string

# @return an integer

def braces(self, A):

stack = []

for i in range(len(A)):

if (A[i] == ')'):

if(stack[-1] == '('):

return 1

while(len(stack) != 0 and stack[-1] != '('):

stack.pop()

if len(stack) != 0: stack.pop()

elif (A[i] == '+' or A[i] == '-' or A[i] == '\*' or A[i] == '/' or A[i] == '('):

stack.append(A[i])

return 0

**Q6. Infix to Postfix**

Given string **A** denoting an infix expression. Convert the infix expression into a postfix expression.

String A consists of **^, /, \*, +, -, (, )** and **lowercase English alphabets** where lowercase English alphabets are operands and ^, /, \*, +, - are operators.

Find and return the postfix expression of A.

**Logic -**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,  
   3.1 If the precedence of the scanned operator is greater than that of the operator in the stack(or the stack is empty, or the stack contains a ‘(‘ ), push it.  
   3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that, Push the scanned operator to the stack. (If you encounter parenthesis while popping, stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

**Code -**

public class Solution {

public String solve(String A) {

HashMap<Character, Integer> operators = new HashMap<Character, Integer>();

operators.put('-', 1);

operators.put('+', 1);

operators.put('\*', 2);

operators.put('/', 2);

operators.put('^', 4);

Stack<Character> st = new Stack<Character>();

StringBuilder ans = new StringBuilder();

for(int i=0;i<A.length();i++) {

// If a operator is found

if(operators.containsKey(A.charAt(i))) {

while(!st.isEmpty()) {

if(st.peek() == '(') break;

if(operators.get(A.charAt(i)) <= operators.get(st.peek())) {

ans.append(st.pop());

continue;

}

break;

}

st.push(A.charAt(i));

}

else if(A.charAt(i) == '(') {

st.push('(');

}

else if(A.charAt(i) == ')') {

while(!st.isEmpty() && st.peek() != '(') {

ans.append(st.pop());

}

if(!st.isEmpty()) st.pop();

}

else {

ans.append(A.charAt(i));

}

}

while(!st.isEmpty()) {

ans.append(st.pop());

}

return ans.toString();

}

}

**Q7. Largest Rectangle in Histogram**

Given an array of integers **A**.

**A** represents a histogram i.e **A[i]** denotes the height of the **ith** histogram's bar. Width of each bar is **1**.

Find the area of the largest rectangle formed by the histogram.

**Logic – Consider every bar to be height of the rectangle**

1. Calculate Nearest Min index (r) from Right and Nearest Min (l) from Left.
2. Area of the rectangle with that height = r-l-1 \* A[i]

**Code -**

public class Solution {

public class ListNode {

int val;

ListNode next;

public ListNode(int d) {

val = d;

next = null;

}

}

public int[] smallestFromLeft(int[] A) {

int[] ans = new int[A.length];

ListNode st = null;

for (int i = 0; i < A.length; i++) {

while (st != null && A[st.val] >= A[i]) {

st = st.next;

}

if (st == null) {

ans[i] = -1;

} else {

ans[i] = st.val;

}

ListNode newNode = new ListNode(i);

newNode.next = st;

st = newNode;

}

return ans;

}

public int[] smallestFromRight(int[] A) {

int[] ans = new int[A.length];

ListNode st = null;

for (int i = A.length - 1; i >= 0; i--) {

while (st != null && A[st.val] >= A[i]) {

st = st.next;

}

if (st == null) {

ans[i] = A.length;

} else {

ans[i] = st.val;

}

ListNode newNode = new ListNode(i);

newNode.next = st;

st = newNode;

}

return ans;

}

public int largestRectangleArea(int[] A) {

int[] l = smallestFromLeft(A);

int[] r = smallestFromRight(A);

int ans = 0;

for (int i = 0; i < A.length; i++) {

ans = Math.max(ans, (r[i] - l[i] - 1) \* A[i]);

}

return ans;

}

}

**Q8. MAX and MIN**

Given an array of integers **A**.

value of a array = max(array) - min(array).

Calculate and return the sum of values of all possible subarrays of A **modulo 109+7**.

**Logic –** Consider every element of the array to be max and min and calculate contribution in max in min

1. Calculate Nearest Right Max and Nearest Right Min, Nearest Left Max and Nearest Left Min.
2. Mx = (rmax[i] - I) \* (I-lmax[i]) \* A[i]. Mn = (rmin[i]-i)\*(I-lmin[i])

**Code -**

public class Solution {

public int[] prevGreatest(int[] A) {

int [] ans = new int[A.length];

Stack<Integer> stack = new Stack<>();

for (int i=0;i<A.length;i++) {

while(!stack.empty() && A[stack.peek()] <= A[i]) stack.pop();

if(stack.empty()) ans[i] = -1;

else ans[i] = stack.peek();

stack.push(i);

}

return ans;

}

public int[] prevSmallest(int[] A) {

int [] ans = new int[A.length];

Stack<Integer> stack = new Stack<>();

for (int i=0;i<A.length;i++) {

while(!stack.empty() && A[stack.peek()] >= A[i]) stack.pop();

if(stack.empty()) ans[i] = -1;

else ans[i] = stack.peek();

stack.push(i);

}

return ans;

}

public int[] nextGreatest(int[] A) {

int [] ans = new int[A.length];

Stack<Integer> stack = new Stack<>();

for (int i=A.length-1;i>=0;i--) {

while(!stack.empty() && A[stack.peek()] <= A[i]) stack.pop();

if(stack.empty()) ans[i] = A.length;

else ans[i] = stack.peek();

stack.push(i);

}

return ans;

}

public int[] nextSmallest(int[] A) {

int [] ans = new int[A.length];

Stack<Integer> stack = new Stack<>();

for (int i=A.length-1;i>=0;i--) {

while(!stack.empty() && A[stack.peek()] >= A[i]) stack.pop();

if(stack.empty()) ans[i] = A.length;

else ans[i] = stack.peek();

stack.push(i);

}

return ans;

}

public int solve(int[] A) {

long mx = 0;

long mn = 0;

int mod = ((1000 \* 1000 \* 1000) + 7);

int [] prevGreat = prevGreatest(A);

int [] nextGreat = nextGreatest(A);

int [] prevSmall = prevSmallest(A);

int [] nextSmall = nextSmallest(A);

for (int i=0;i<A.length;i++) {

mx += (long)((long)(i-prevGreat[i]) \* (long)(nextGreat[i]-i) \* (long)(A[i]));

mn += (long)((long)(i-prevSmall[i]) \* (long)(nextSmall[i]-i) \* (long)(A[i]));

}

return (int)((mx - mn) % mod);

}

}

**Q9. Maximum Rectangle**

Given a 2D binary matrix of integers **A** containing 0's and 1's of size **N x M**.

Find the largest rectangle containing only 1's and return its area.

**Note:** Rows are numbered from top to bottom and columns are numbered from left to right.

**Logic -**

1. Build a histogram of the matrix.
2. If(A[row][col] != 0) A[row][col] += A[row-1][col], else A[row][col] == 0
3. Consider each row as a histogram and solve it (Refer Q7).

**Code -**

class Solution:

# @param A : list of list of integers

# @return an integer

def prevNearest(self, A):

stack = []

ans = []

for i in range(len(A)):

while(len(stack)!=0 and A[stack[-1]] >= A[i]):

stack.pop()

if len(stack) == 0: ans.append(-1)

else: ans.append(stack[-1])

stack.append(i)

return ans

def nextNearest(self, A):

stack = []

ans = []

for i in range(len(A)-1, -1, -1):

while(len(stack)!=0 and A[stack[-1]] >= A[i]):

stack.pop()

if len(stack) == 0: ans.append(len(A))

else: ans.append(stack[-1])

stack.append(i)

return ans[::-1]

def histogram(self, A):

l = self.prevNearest(A)

r = self.nextNearest(A)

ans = 0

for i in range(len(A)):

# print(r[i], l[i], A[i],end=" ")

ans = max(ans, (r[i]-l[i]-1) \* A[i])

return ans

def solve(self, A):

for i in range(1, len(A)):

for j in range(len(A[0])):

if(A[i][j] == 0):

continue

A[i][j] += A[i-1][j]

# print(A, end=" ")

ans = 0

for i in range(len(A)):

ans = max(ans, self.histogram(A[i]))

return ans

**Q10. All Subarrays**

Given an integer array **A** of size **N**. You have to generate it's all subarrays having a size greater than 1.

Then for each subarray, find **Bitwise XOR** of its maximum and second maximum element.

Find and return the **maximum value of XOR** among all subarrays.

**Logic –** If we find a bigger element than the peek element, it means that the peek element is useless. So, we pop it.

1. Maintain a **monotone-decreasing-stack.**
2. While a new element came into the view, pop the top element in the stack, and check the corresponding interval, until the new element is greater than the top element in the stack.
3. We can easily see it is correct since we won’t lost the answer as long as it exists.

**Code -**

public class Solution {

public int solve(ArrayList<Integer> A) {

Stack<Integer> st = new Stack<Integer>();

int ans = 0;

st.push(A.get(0));

for(int i=1;i<A.size();i++) {

while(!st.isEmpty() && A.get(i) >= st.peek()) {

ans = Math.max(ans, (st.peek()^A.get(i)));

st.pop();

}

if(!st.isEmpty()) {

ans = Math.max(ans, (st.peek()^A.get(i)));

}

st.push(A.get(i));

}

return ans;

}

}