**/\* Check whether string is palindrome or not\*/**

public class Solution {

public boolean isPalindrome(String s) {

if(s==null || s.length()<=1)

return true;

String original = s.replaceAll("[^A-Za-z0-9]","").toLowerCase();

String reverse = new StringBuffer(original).reverse().toString();

return original.equals(reverse);

}

}

**/\* Reverse the words in sentence / Reverse the words in string \*/**

public class Solution

{

public String reverseWords(String s)

{

String [] words = s.split(" ");

StringBuilder sb = new StringBuilder();

int end = words.length - 1;

for(int i = 0; i<= end; i++)

{

if(!words[i].isEmpty())

{

sb.insert(0, words[i]);

if(i < end)

sb.insert(0, " ");

}

}

return sb.toString();

}

}

**/\* Reverse the words in sentence / Reverse the words in string ALTERNATIVE\*/**

public class Solution {

public String reverseWords(String s) {

StringBuilder sb = new StringBuilder();

if(s==null || s.length()<1)

return sb.toString();

String[] arr = s.split("\\s{1,}");

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

{

sb.append(arr[i] + " ");

}

return sb.toString().trim();

}

}

**/\* get the length of last word in string/ get the length of last word in sentence \*/**

public class Solution {

public int lengthOfLastWord(String s) {

String str = s.trim();

if (str==null) {

return 0;

}

String[] k = str.split(" ");

return k[k.length-1].length();

}

}

**/\*Length of longest substring with non-repetitive / repeating characters \*/**

public class Solution {

public int lengthOfLongestSubstring(String s) {

int i = 0, j = 0, max = 0;

Set<Character> set = new HashSet<>();

while (j < s.length()) {

if (!set.contains(s.charAt(j))) {

set.add(s.charAt(j++));

max = Math.max(max, set.size());

} else {

set.remove(s.charAt(i++));

}

}

return max;

}

}

**/\*find the index of string in another string / substring \*/**

public class Solution {

public int strStr(String haystack, String needle) {

if(haystack == null || needle == null || needle.length() > haystack.length()){

return -1;

}

for(int i = 0; i < haystack.length() - needle.length() + 1; i ++){

if(haystack.substring(i, i + needle.length()).equals(needle)){

return i;

}

}

return -1;

}

}

**/\*Simplify path \*/**

public class Solution {

public String simplifyPath(String path) {

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

String [] str = path.split("/");

for (String s:str)

{

if(s==null || "".equals(s) || ".".equals(s))

continue;

if("..".equals(s))

{

if(!st.isEmpty())

st.pop();

}

else

st.push(s);

}

if(st.isEmpty())

{

return "/";

}

StringBuilder str\_build = new StringBuilder("");

for(String newStr:st)

{

str\_build.append("/"+newStr);

}

return str\_build.toString();

}

}

**/\* Implementation of Minimum stack / min stack\*/**

class MinStack {

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

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

public void push(int x) {

mainStack.push(x);

if (minStack.empty()) {

minStack.push(x);

} else if (minStack.peek() >= x) {

minStack.push(x);

}

}

public void pop() {

int poppedElement = mainStack.pop();

if (poppedElement == minStack.peek()) {

minStack.pop();

}

}

public int top() {

return mainStack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**/\*Largest Rectangle in a histogram \*/**

public class Solution {

public int largestRectangleArea(int[] height) {

if(height == null || height.length==0)

return 0;

int len = height.length;

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

int maxArea = 0;

for(int i = 0; i <= len; i++){

int h = (i == len ? 0 : height[i]);

if(s.isEmpty() || h >= height[s.peek()]){

s.push(i);

}else{

int tp = s.pop();

maxArea = Math.max(maxArea, height[tp] \* (s.isEmpty() ? i : i - 1 - s.peek()));

i--;

}

}

return maxArea;

}

}

**/\*validate parenthesis \*/**

public class Solution {

public boolean isValid(String s) {

if(s==null){ return true; }

Stack<Character> s1=new Stack<Character>();

char[] c = s.toCharArray();

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

if(!s1.empty()){

if((s1.peek()=='(' && c[i]==')' )||(s1.peek()=='{' && c[i]=='}' )||(s1.peek()=='[' && c[i]==']' ) )

s1.pop();

else

s1.push(c[i]);

}else

s1.push(c[i]);

}

return s1.empty();

}

}

**/\* Implement stack using queue\*/**

class MyStack {

Queue<Integer> qu = new LinkedList<Integer>();

// Push element x onto stack.

public void push(int x) {

qu.add(x);

for(int i=0; i<qu.size()-1; i++)

{

qu.add(qu.remove());

}

}

// Removes the element on top of the stack.

public void pop() {

qu.remove();

}

// Get the top element.

public int top() {

return qu.peek();

}

// Return whether the stack is empty.

public boolean empty() {

return qu.isEmpty();

}

}

**/\*Implement queue using stack \*/**

class MyQueue {

// Push element x to the back of queue.

Stack<Integer> s1 = new Stack<Integer>();

Stack<Integer> s2 = new Stack<Integer>();

public void push(int x) {

s1.push(x);

}

// Removes the element from in front of queue.

public void pop() {

if(!s2.isEmpty()){

s2.pop();

return;

}

while(!s1.isEmpty()){

s2.push(s1.pop());

}

s2.pop();

}

// Get the front element.

public int peek() {

if(!s2.isEmpty())

return s2.peek();

while(!s1.isEmpty())

{

s2.push(s1.pop());

}

return s2.peek();

}

// Return whether the queue is empty.

public boolean empty() {

return s1.isEmpty()&&s2.isEmpty();

}

}

**/\* Evaluate Reverse polish notation\*/**

public class Solution {

public int evalRPN(String[] tokens) {

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

int temp;

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

switch (tokens[i]) {

case "+":

temp = stack.pop();

stack.push(temp + stack.pop());

break;

case "-":

temp = stack.pop();

stack.push(stack.pop() - temp);

break;

case "\*":

temp = stack.pop();

stack.push(stack.pop() \* temp);

break;

case "/":

temp = stack.pop();

stack.push(stack.pop() / temp);

break;

default:

stack.push(Integer.parseInt(tokens[i]));

}

}

return stack.peek();

}

}

**/\*Binary tree zig zag level order traversal \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

Stack<TreeNode> s1 = new Stack<TreeNode>();

Stack<TreeNode> s2 = new Stack<TreeNode>();

TreeNode curr = root;

s1.push(curr);

List<List<Integer>> arlist = new ArrayList<List<Integer>>();

if(root==null)

return arlist;

while(!s1.isEmpty() || !s2.isEmpty())

{

List<Integer> li1 = new ArrayList<Integer>();

List<Integer> li2 = new ArrayList<Integer>();

int flag1 = 0;

int flag2 = 0;

while(!s1.isEmpty())

{

flag1=1;

li1.add(s1.peek().val);

if(s1.peek().left!=null)

s2.push(s1.peek().left);

if(s1.peek().right!=null)

s2.push(s1.peek().right);

s1.pop();

}

if(flag1==1)

arlist.add(li1);

while(!s2.isEmpty())

{

flag2=1;

li2.add(s2.peek().val);

if(s2.peek().right!=null)

s1.push(s2.peek().right);

if(s2.peek().left!=null)

s1.push(s2.peek().left);

s2.pop();

}

if(flag2==1)

arlist.add(li2);

}

return arlist;

}

}

**/\* Binary search tree iterator / BST iterator\*/**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class BSTIterator {

private Stack<TreeNode> stack = null;

public BSTIterator(TreeNode root) {

stack = new Stack<>();

TreeNode curr = root;

while(curr!=null){

stack.push(curr);

curr = curr.left;

}

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return !stack.isEmpty();

}

/\*\* @return the next smallest number \*/

public int next() {

if(hasNext()){

int ret = stack.peek().val;

TreeNode curr = stack.pop();

if(curr.right!=null){

curr = curr.right;

while(curr!=null){

stack.push(curr);

curr = curr.left;

}

}

return ret;

}

return -1;

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

**/\* Basic calculator to evaluate expression / contains only ( ) + - / basic calculator to evaluate string \*/**

public class Solution {

public int calculate(String s) {

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

int result = 0;

int number = 0;

int sign = 1;

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

char c = s.charAt(i);

if(Character.isDigit(c)){

number = 10 \* number + (int)(c - '0');

}else if(c == '+'){

result += sign \* number;

number = 0;

sign = 1;

}else if(c == '-'){

result += sign \* number;

number = 0;

sign = -1;

}else if(c == '('){

//we push the result first, then sign;

stack.push(result);

stack.push(sign);

//reset the sign and result for the value in the parenthesis

sign = 1;

result = 0;

}else if(c == ')'){

result += sign \* number;

number = 0;

result \*= stack.pop(); //stack.pop() is the sign before the parenthesis

result += stack.pop(); //stack.pop() now is the result calculated before the parenthesis

}

}

if(number != 0) result += sign \* number;

return result;

}

}

**/\* Binary tree preorder traversal iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> answer = new LinkedList<Integer>();

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

if(root==null)

{

return answer;

}

stack.push(root);

while(stack.isEmpty()==false)

{

TreeNode node = stack.pop();

answer.add(node.val);

if(node.right!=null)

{

stack.push(node.right);

}

if(node.left!=null)

{

stack.push(node.left);

}

}

return answer;

}

}

**/\* Binary tree preorder traversal recursion\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root!=null) {

list.add(root.val);

list.addAll(preorderTraversal(root.left));

list.addAll(preorderTraversal(root.right));

}

return list;

}

}

**/\* Binary tree postorder traversal recursion\*/**

*/\*\**

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root!=null) {

list.addAll(postorderTraversal(root.left));

list.addAll(postorderTraversal(root.right));

list.add(root.val);

}

return list;

}

}

**/\* Binary tree postorder traversal iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root == null) {

return list;

}

HashMap<TreeNode,Boolean> nodesPushedToStack = new HashMap<TreeNode,Boolean>();

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

stack.push(root);

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

if (!nodesPushedToStack.containsKey(node)) {

nodesPushedToStack.put(node,true);

stack.push(node);

if (node.right != null) {

stack.push(node.right);

}

if (node.left != null) {

stack.push(node.left);

}

}

else {

list.add(node.val);

}

}

return list;

}

}

**/\* Binary tree inorder traversal iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> ans = new ArrayList<>();

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

TreeNode cur = root;

while (!stack.isEmpty() || cur != null) {

while (cur != null) {

stack.push(cur);

cur = cur.left;

}

cur = stack.pop();

ans.add(cur.val);

cur = cur.right;

}

return ans;

}

}

**/\* Binary tree inorder traversal recursion \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

List<Integer> ans = new ArrayList<>();

public List<Integer> inorderTraversal(TreeNode root) {

if (root == null) return ans;

inorderTraversal(root.left);

ans.add(root.val);

return inorderTraversal(root.right);

}

}

**/\*Contains Duplicate. Two distinct indices. Difference between i and j is at most/ atmost k\*/**

public class Solution {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

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

if (map.containsKey(nums[i])) {

if (i - map.get(nums[i]) <= k) return true;

}

map.put(nums[i], i);

}

return false;

}

}

**/\* Check if array contains any duplicate / array duplicate\*/**

public class Solution {

public boolean containsDuplicate(int[] nums) {

if (nums.length==0) return false;

HashSet<Integer> hs = new HashSet<>();

for(int n : nums) {

if (!hs.add(n)) return true;

}

return false;

}

}

**/\* Majority element in the array\*/**

public class Solution {

    public int majorityElement(int[] nums) {

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

        int majority = (int)Math.floor(nums.length/2);

        int highest = 0;

        for(int i : nums){

            if(map.containsKey(i))

                map.put(i,map.get(i)+1);

            else

                map.put(i,1);

        }

        for(Map.Entry<Integer,Integer> result : map.entrySet()){

            if(result.getValue()>majority){

                highest = result.getKey();

                majority = result.getValue();

            }

        }

        return highest;

    }

}

**/\* merge two sorted arrays at back of other array/ at back of first array/ without using extra space\*/**

public class Solution {

public void merge(int[] nums1, int m, int[] nums2, int n) {

int num1Tail = m - 1;

int num2Tail = n - 1;

int newTail = m + n - 1;

while(num2Tail >= 0) {

//nums1: {4, }

//nums2: {1}

//so num1Tail >= 0 is very important!!!

if(num1Tail >= 0 && nums1[num1Tail] > nums2[num2Tail]) {

nums1[newTail] = nums1[num1Tail];

num1Tail--;

}

else {

nums1[newTail] = nums2[num2Tail];

num2Tail--;

}

newTail--;

}

}

}

**/\*Move zeros / 0 to one end\*/**

public class Solution {

public void moveZeroes(int[] nums) {

int p = 0;

for(int i : nums)

{

if(i!=0)

{

nums[p]=i;

p++;

}

}

while(p<nums.length)

{

nums[p]=0;

p++;

}

}

}

**/\*Pascal triangle\*/**

public class Solution {

public List<List<Integer>> generate(int numRows) {

List<List<Integer>> ans = new ArrayList<List<Integer>>();

if (numRows == 0)

return ans;

LinkedList<Integer> sub = new LinkedList<Integer>();

sub.add(1);

ans.add(sub);

for (int i = 1; i < numRows; i++) {

LinkedList<Integer> subAns = new LinkedList<Integer>();

List<Integer> list = ans.get(i - 1);

subAns.addFirst(1);

for (int j = 1; j < i; j++)

subAns.addFirst(list.get(j - 1) + list.get(j));

subAns.addFirst(1);

ans.add(subAns);

}

return ans;

}

}

**/\*Plus one to the array/at MSB\*/**

public class Solution {

public int[] plusOne(int[] digits) {

int n = digits.length;

for(int i=n-1; i>=0; i--) {

if(digits[i] < 9) {

digits[i]++;

return digits;

}

digits[i] = 0;

}

int[] newNumber = new int [n+1];

newNumber[0] = 1;

return newNumber;

}

}

**/\* Remove Duplicates from sorted array without using extra space\*/**

public class Solution {

public int removeDuplicates(int[] nums) {

int j = 0;

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

if (i > 0 && nums[i - 1] == nums[i])

continue;

nums[j] = nums[i];

j++;

}

return j;

}

}

**/\* remove duplicates from array with at most/atmost 2 same / 2 duplicates \*/**

public class Solution {

public int removeDuplicates(int[] nums) {

if(nums.length<3)

return nums.length;

int count=2;

for(int i=2 ; i <nums.length; i++)

{

nums[count] = nums[i];

if(!(nums[count]==nums[count-1] && nums[count]==nums[count-2]))

count++;

}

return count;

}

}

**/\*Summary ranges \*/**

public class Solution {

public List<String> summaryRanges(int[] nums) {

StringBuffer sb = new StringBuffer();

List<String> res= new ArrayList<String>();

if(nums == null || nums.length==0)

return res;

if(nums.length==1)

{

sb.append(nums[0]);

res.add(sb.toString());

return res;

}

int p=0;

for (int i = 1; i < nums.length; i++)

{

if (nums[i] == nums[i - 1] + 1)

{

if (i == nums.length - 1)

{

res.add(nums[p] + "->" + nums[i]);

}

}

else

{

if (i == p + 1)

{

res.add(nums[p] + "");

}

else

{

res.add(nums[p] + "->" + nums[i - 1]);

}

if (i == nums.length - 1)

{

res.add(nums[i] + "");

}

p = i;

}

}

return res;

}

}

**/\*3 Sum closest \*/**

public class Solution {

public int threeSumClosest(int[] nums, int target) {

if(nums.length==3)

return (nums[0]+nums[1]+nums[2]) ;

Arrays.sort(nums);

int diff = Integer.MAX\_VALUE;

int output=0;

for(int i=0; i<nums.length-2;i++)

{

int low=i+1;

int high = nums.length-1;

while(low<high)

{

int sum = nums[i] + nums[low] + nums[high];

if(Math.abs(target-sum)<diff)

{

diff = Math.abs(target-sum);

output=sum;

}

if(sum>target)

high--;

else

low++;

}

}

return output;

}

}

**/\*3 Sum zero / 3sum zero / 3 numbers sum to zero\*/**

public class Solution {

public List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> li = new ArrayList<List<Integer>>();

Arrays.sort(nums);

for(int i=0 ; i<=nums.length-3 ; i++)

{

if(i>0 && nums[i]==nums[i-1])

continue;

int start=i+1;

int end= nums.length-1;

while(start<end)

{

if(start>i+1 && nums[start]==nums[start-1])

{

start++;

continue;

}

if(end<nums.length-1 && nums[end]==nums[end+1])

{

end--;

continue;

}

int sum = nums[i]+ nums[start] +nums[end];

if(sum==0)

{

List<Integer> l = new ArrayList<Integer>();

l.add(nums[i]);

l.add(nums[start]);

l.add(nums[end]);

li.add(l);

start++;

end--;

}

else if(sum>0)

end--;

else

start++;

}

}

return li;

}

}

**/\* get the row of a pascal triangle (worst case)\*/**

public class Solution {

public List<Integer> getRow(int k) {

Integer[] arr = new Integer[k + 1];

Arrays.fill(arr, 0);

arr[0] = 1;

for (int i = 1; i <= k; i++)

for (int j = i; j > 0; j--)

arr[j] = arr[j] + arr[j - 1];

return Arrays.asList(arr);

}

}

**/\* get the row of a pascal triangle (best case)\*/**

public class Solution {

public List<Integer> getRow(int rowIndex) {

List<Integer> result = new ArrayList<Integer>();

result.add(1);

long tmp = 1;

for(int i=1;i<=rowIndex ; i++){

tmp = tmp\*(rowIndex-i+1)/i;

result.add((int)(tmp));

}

return result;

}

}

**/\* Replace element from an array and find the new length of array \*/**

public class Solution {

public int removeElement(int[] nums, int val) {

int low=0;

int high = nums.length-1;

while(low<=high)

{

while(high>low && nums[high]==val)

{

high--;

}

if(nums[low]==val)

{

nums[low] = nums[high];

high--;

}

low++;

}

return high+1;

}

}

**/\* 4 Sum to a target value / 4sum target / 4 numbers sum to a given number \*/**

public class Solution {

public List<List<Integer>> fourSum(int[] nums, int target) {

Set<List<Integer>> resultSet = new HashSet<List<Integer>>();

int a, b, c, d, start, end;

Arrays.sort(nums);

for(int i = 0; i < nums.length - 3; i++) {

a = nums[i];

for(int j = i + 1; j < nums.length - 2; j++) {

b = nums[j];

start = j + 1;

end = nums.length - 1;

while(start < end) {

c = nums[start];

d = nums[end];

if(a + b + c + d == target) {

List<Integer> quadruple = new ArrayList<Integer>();

quadruple.add(a);

quadruple.add(b);

quadruple.add(c);

quadruple.add(d);

resultSet.add(quadruple);

start++;

end--;

} else if(a + b + c + d < target)

start++;

else

end--;

}

}

}

List<List<Integer>> result = new ArrayList<List<Integer>>();

result.addAll(resultSet);

return result;

}

}

**/\* index of Peak Element. Peak element is the element who is greater than his neighbours.\*/**

public class Solution {

public int findPeakElement(int[] nums) {

int max = Integer.MIN\_VALUE;

int j=0;

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

{

if(nums[i]>max)

{

max=nums[i];

j=i;

}

}

return j;

}

}

**/\* Minimum element in rotated sorted array\*/**

public class Solution {

public int findMin(int[] nums) {

int l = 0, r = nums.length-1;

while (l < r) {

int mid = (l + r) / 2;

if (nums[mid] < nums[r]) {

//right side in order, and mid is smallest of right side,

// min should be mid or in left.

r = mid;

} else {

//left side in order and it is the larger part,

// min should be in right side.

l = mid + 1;

}

}

return nums[l];

}

}

**/\* First missing positive \*/**

public class Solution {

public int firstMissingPositive(int[] nums) {

if(nums == null || nums.length ==0)

return 1;

int i = 0, n = nums.length;

while (i < n) {

// If the current value is in the range of (0,length) and it's not at its correct position,

// swap it to its correct position.

// Else just continue;

if (nums[i] >= 0 && nums[i] < n && nums[nums[i]] != nums[i])

swap(nums, i, nums[i]);

else

i++;

}

int k = 1;

// Check from k=1 to see whether each index and value can be corresponding.

while (k < n && nums[k] == k)

k++;

// If it breaks because of empty array or reaching the end. K must be the first missing number.

if (n == 0 || k < n)

return k;

else // If k is hiding at position 0, K+1 is the number.

return nums[0] == k ? k + 1 : k;

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**/\*Max / Maximum jump to reach the end of array \*/**

public class Solution {

public boolean canJump(int[] nums) {

int maxIndex = nums.length-1;

int maxJump = nums[0];

for(int i = 0; i <= maxJump; i++)

{

maxJump=Math.max(maxJump,i+nums[i]);

if(maxJump>=maxIndex) return true;

}

return false;

}

}

**/\* Valid / validate parenthesis using stack\*/**

**if** (s.length() == 0) **return** **true**;

Stack<Character> **stack** = **new** Stack<Character>();

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

Character c = s.charAt(i);

**if** (c == '(' || c == '[' || c == '{') **stack**.push(c);

**else** **if** (**stack**.empty()) **return** **false**;

**else**{

Character c2 = **stack**.pop();

**if**(c2 == '(' && c != ')' || c2 == '{' && c != '}' || c2 == '[' && c != ']') **return** **false**;

}

}

**return** **stack**.empty();

**/\* Valid / validate parenthesis using both Hashmap and stack \*/**

public class Solution {

public boolean isValid(String s) {

if(s==null){ return true; }

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

Map<Character, Character> charMap = initCharMap();

for(Character ch : s.toCharArray()){

if(charStack.isEmpty()){

if(charMap.keySet().contains(ch)){

return false;

}

else{

charStack.push(ch);

}

}

else{

if(charStack.peek()==charMap.get(ch)){

charStack.pop();

}

else{

charStack.push(ch);

}

}

}

if(charStack.isEmpty()){

return true;

}

return false;

}

private Map<Character, Character> initCharMap(){

Map<Character, Character> charMap = new HashMap<>();

charMap.put(')','(');

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

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

return charMap;

}

}

**/\* Trapping Rain Water \*/**

public class Solution {

public int trap(int[] heights) {

if(heights == null || heights.length < 2){ return 0; }

int res = 0;

int min = 0;

int max = heights.length - 1;

int minVal = heights[min];

int maxVal = heights[max];

while(max - min > 1){

if(minVal <= maxVal){

if(heights[min+1] > minVal){

minVal = heights[min+1];

min ++;

}

else{

res += minVal - heights[min+1];

min ++;

}

}

else{

if(heights[max-1] > maxVal){

maxVal = heights[max-1];

max --;

}

else{

res += maxVal - heights[max-1];

max --;

}

}

}

return res;

}

}

**/\* Trapping Rain Water using stack\*/**

public class Solution {

public int trap(int[] A) {

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

int sum = 0;

int pre = 0;

int i = -1;

while(++i < A.length){

if(A[i]==0){pre = 0;continue;}

while(!stack.isEmpty() && A[i] >= A[stack.peek()]){

sum += (A[stack.peek()] - pre) \* (i-stack.peek()-1);

pre = A[stack.pop()];

}

if(!stack.isEmpty()){

sum += (A[i] - pre) \* (i-stack.peek()-1);

pre = A[i];

}

stack.push(i);

}

return sum;

}

}

**/\* Min stack / Minimum stack\*/**

class MinStack {

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

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

public void push(int x) {

mainStack.push(x);

if (minStack.empty()) {

minStack.push(x);

} else if (minStack.peek() >= x) {

minStack.push(x);

}

}

public void pop() {

int poppedElement = mainStack.pop();

if (poppedElement == minStack.peek()) {

minStack.pop();

}

}

public int top() {

return mainStack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**/\*Preorder traversal of BST recursion\*/**

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root!=null) {

list.add(root.val);

list.addAll(preorderTraversal(root.left));

list.addAll(preorderTraversal(root.right));

}

return list;

}

}

**/\*Preorder traversal of BST iterative solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> answer = new LinkedList<Integer>();

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

if(root==null)

{

return answer;

}

stack.push(root);

while(stack.isEmpty()==false)

{

TreeNode node = stack.pop();

answer.add(node.val);

if(node.right!=null)

{

stack.push(node.right);

}

if(node.left!=null)

{

stack.push(node.left);

}

}

return answer;

}

}

**/\*Postorder traversal of BST iterative solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

LinkedList<Integer> ans = new LinkedList<>();

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

if (root == null) return ans;

stack.push(root);

while (!stack.isEmpty()) {

TreeNode cur = stack.pop();

ans.addFirst(cur.val);

if (cur.left != null) {

stack.push(cur.left);

}

if (cur.right != null) {

stack.push(cur.right);

}

}

return ans;

}

}

**/\*Postorder traversal of BST recursive solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root!=null) {

list.addAll(postorderTraversal(root.left));

list.addAll(postorderTraversal(root.right));

list.add(root.val);

}

return list;

}

}

**/\*Inorder traversal of BST recursive solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<Integer>();

if (root!=null) {

list.addAll(inorderTraversal(root.left));

list.add(root.val);

list.addAll(inorderTraversal(root.right));

}

return list;

}

}

**/\*Inorder traversal of BST iterative solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> ans = new ArrayList<>();

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

TreeNode cur = root;

while (!stack.isEmpty() || cur != null) {

while (cur != null) {

stack.push(cur);

cur = cur.left;

}

cur = stack.pop();

ans.add(cur.val);

cur = cur.right;

}

return ans;

}

}

**/\*Binary Search tree BST Iterator\*/**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class BSTIterator {

private Stack<TreeNode> stack = null;

public BSTIterator(TreeNode root) {

stack = new Stack<>();

TreeNode curr = root;

while(curr!=null){

stack.push(curr);

curr = curr.left;

}

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return !stack.isEmpty();

}

/\*\* @return the next smallest number \*/

public int next() {

if(hasNext()){

int ret = stack.peek().val;

TreeNode curr = stack.pop();

if(curr.right!=null){

curr = curr.right;

while(curr!=null){

stack.push(curr);

curr = curr.left;

}

}

return ret;

}

return -1;

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

**/\*Basic maths calculator using stack / evaluate arithmetic expression.\*/**

public class Solution {

public int calculate(String s) {

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

int result = 0;

int number = 0;

int sign = 1;

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

char c = s.charAt(i);

if(Character.isDigit(c)){

number = 10 \* number + (int)(c - '0');

}else if(c == '+'){

result += sign \* number;

number = 0;

sign = 1;

}else if(c == '-'){

result += sign \* number;

number = 0;

sign = -1;

}else if(c == '('){

//we push the result first, then sign;

stack.push(result);

stack.push(sign);

//reset the sign and result for the value in the parenthesis

sign = 1;

result = 0;

}else if(c == ')'){

result += sign \* number;

number = 0;

result \*= stack.pop(); //stack.pop() is the sign before the parenthesis

result += stack.pop(); //stack.pop() now is the result calculated before the parenthesis

}

}

if(number != 0) result += sign \* number;

return result;

}

}

**/\* Zig zag traversal of binary search tree BST \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

Stack<TreeNode> s1 = new Stack<TreeNode>();

Stack<TreeNode> s2 = new Stack<TreeNode>();

TreeNode curr = root;

s1.push(curr);

List<List<Integer>> arlist = new ArrayList<List<Integer>>();

if(root==null)

return arlist;

while(!s1.isEmpty() || !s2.isEmpty())

{

List<Integer> li1 = new ArrayList<Integer>();

List<Integer> li2 = new ArrayList<Integer>();

int flag1 = 0;

int flag2 = 0;

while(!s1.isEmpty())

{

flag1=1;

li1.add(s1.peek().val);

if(s1.peek().left!=null)

s2.push(s1.peek().left);

if(s1.peek().right!=null)

s2.push(s1.peek().right);

s1.pop();

}

if(flag1==1)

arlist.add(li1);

while(!s2.isEmpty())

{

flag2=1;

li2.add(s2.peek().val);

if(s2.peek().right!=null)

s1.push(s2.peek().right);

if(s2.peek().left!=null)

s1.push(s2.peek().left);

s2.pop();

}

if(flag2==1)

arlist.add(li2);

}

return arlist;

}

}

**/\* Stack implementation using queue\*/**

class MyStack {

Queue<Integer> qu = new LinkedList<Integer>();

// Push element x onto stack.

public void push(int x) {

qu.add(x);

for(int i=0; i<qu.size()-1; i++)

{

qu.add(qu.remove());

}

}

// Removes the element on top of the stack.

public void pop() {

qu.remove();

}

// Get the top element.

public int top() {

return qu.peek();

}

// Return whether the stack is empty.

public boolean empty() {

return qu.isEmpty();

}

}

**/\* queue implementation using stack\*/**

class MyQueue {

// Push element x to the back of queue.

Stack<Integer> s1 = new Stack<Integer>();

Stack<Integer> s2 = new Stack<Integer>();

public void push(int x) {

s1.push(x);

}

// Removes the element from in front of queue.

public void pop() {

if(!s2.isEmpty()){

s2.pop();

return;

}

while(!s1.isEmpty()){

s2.push(s1.pop());

}

s2.pop();

}

// Get the front element.

public int peek() {

if(!s2.isEmpty())

return s2.peek();

while(!s1.isEmpty())

{

s2.push(s1.pop());

}

return s2.peek();

}

// Return whether the queue is empty.

public boolean empty() {

return s1.isEmpty()&&s2.isEmpty();

}

}

**/\* Reverse Polish Notation \*/**

public class Solution {

public int evalRPN(String[] tokens) {

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

int temp;

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

switch (tokens[i]) {

case "+":

temp = stack.pop();

stack.push(temp + stack.pop());

break;

case "-":

temp = stack.pop();

stack.push(stack.pop() - temp);

break;

case "\*":

temp = stack.pop();

stack.push(stack.pop() \* temp);

break;

case "/":

temp = stack.pop();

stack.push(stack.pop() / temp);

break;

default:

stack.push(Integer.parseInt(tokens[i]));

}

}

return stack.peek();

}

}

**/\* Largest Rectangle in Histogram\*/**

public class Solution {

public int largestRectangleArea(int[] height) {

if(height == null || height.length==0)

return 0;

int len = height.length;

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

int maxArea = 0;

for(int i = 0; i <= len; i++){

int h = (i == len ? 0 : height[i]);

if(s.isEmpty() || h >= height[s.peek()]){

s.push(i);

}else{

int tp = s.pop();

maxArea = Math.max(maxArea, height[tp] \* (s.isEmpty() ? i : i - 1 - s.peek()));

i--;

}

}

return maxArea;

}

}

**/\* Simplify path to get the home directory\*/**

public class Solution {

public String simplifyPath(String path) {

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

String [] str = path.split("/");

for (String s:str)

{

if(s==null || "".equals(s) || ".".equals(s))

continue;

if("..".equals(s))

{

if(!st.isEmpty())

st.pop();

}

else

st.push(s);

}

if(st.isEmpty())

{

return "/";

}

StringBuilder str\_build = new StringBuilder("");

for(String newStr:st)

{

str\_build.append("/"+newStr);

}

return str\_build.toString();

}

}

**/\*Zig – Zag conversion of string\*/**

public class Solution {

public String convert(String s, int n) {

if (s.length()==0 || n==1)

return s;

StringBuilder[] sb = new StringBuilder[n];

for(int m=0; m<sb.length;m++)

sb[m] = new StringBuilder("");

int j=0;

int f=1;

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

{

if(f==1){

sb[j].append(s.charAt(i));

j++;

if(j==n)

{

j--;

f=0;

}

}

else

{

j--;

sb[j].append(s.charAt(i));

if(j==0){

f=1;

j++;

}

}

}

StringBuilder result = new StringBuilder();

for (StringBuilder sr : sb)

result.append(sr);

return result.toString();

}

}

**/\*Reverse an integer number \*/**

public class Solution {

public int reverse(int x) {

int f=0;

if(x<0)

{

x=x\* (-1);

f=1;

}

if(x%10==x)

return x;

int y = x;

int n;

long r=0;

while(y>0)

{

r = r\*10;

n=y%10;

r=n+r;

y=y/10;

}

if(r>Integer.MAX\_VALUE || r<Integer.MIN\_VALUE)

return 0;

if(f==1)

return (int)r\*(-1);

else

return (int)r;

}

}

**/\*String to integer / atoi \*/**

public class Solution {

public int myAtoi(String str) {

int index = 0, sign = 1, total = 0;

//1. Empty string

if(str.length() == 0) return 0;

//2. Remove Spaces

while(str.charAt(index) == ' ' && index < str.length())

index ++;

//3. Handle signs

if(str.charAt(index) == '+' || str.charAt(index) == '-'){

sign = str.charAt(index) == '+' ? 1 : -1;

index ++;

}

//4. Convert number and avoid overflow

while(index < str.length()){

int digit = str.charAt(index) - '0';

if(digit < 0 || digit > 9) break;

//check if total will be overflow after 10 times and add digit

if(Integer.MAX\_VALUE/10 < total || Integer.MAX\_VALUE/10 == total && Integer.MAX\_VALUE %10 < digit)

return sign == 1 ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;

total = 10 \* total + digit;

index ++;

}

return total \* sign;

}

}

**/\*check if integer is palindrome or not\*/**

public class Solution {

public boolean isPalindrome(int x) {

int xx = x;

int reversed = 0;

while(xx > 0) {

reversed = reversed \* 10 + xx % 10;

xx /= 10;

}

return x == reversed;

}

}

**/\*Roman to Integer\*/**

public class Solution {

public int romanToInt(String s) {

int result = 0;

char[] sChar = s.toCharArray();

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

map.put('I',1);

map.put('V',5);

map.put('X',10);

map.put('L',50);

map.put('C',100);

map.put('D',500);

map.put('M',1000);

if(sChar.length == 1) return map.get(sChar[0]);

for(int i = 0; i < sChar.length-1; i++){

if( map.get(sChar[i]) >= map.get(sChar[i+1]) )

result += map.get(sChar[i]);

else

result -= map.get(sChar[i]);

}

result += map.get(sChar[sChar.length-1]);

return result;

}

}

**/\*Prefix string among all strings\*/**

public class Solution {

public String longestCommonPrefix(String[] strs) {

StringBuilder result = new StringBuilder();

if (strs!= null && strs.length > 0){

Arrays.sort(strs);

char [] a = strs[0].toCharArray();

char [] b = strs[strs.length-1].toCharArray();

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

if (b.length > i && b[i] == a[i]){

result.append(b[i]);

}

else {

return result.toString();

}

}

}

return result.toString();

}

}

**/\*Remove Nth node from last in List / Linked List / LinkedList\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

if(head == null)

return head ;

ListNode result= new ListNode(0);

result.next = head;

ListNode fast = result;

ListNode slow = result;

for(int i=0 ; i<n ; i++)

{

if(fast==null)

return null;

fast= fast.next;

}

while(fast.next!=null)

{

fast=fast.next;

slow=slow.next;

}

slow.next = slow.next.next;

return result.next;

}

}

**/\*check for valid parenthesis \*/**

public class Solution {

public boolean isValid(String s) {

if(s==null){ return true; }

Stack<Character> s1=new Stack<Character>();

char[] c = s.toCharArray();

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

if(!s1.empty()){

if((s1.peek()=='(' && c[i]==')' )||(s1.peek()=='{' && c[i]=='}' )||(s1.peek()=='[' && c[i]==']' ) )

s1.pop();

else

s1.push(c[i]);

}else

s1.push(c[i]);

}

return s1.empty();

}

}

**/\*Merge two sorted lists iteratively \*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

if(l1==null)

return l2;

if(l2==null)

return l1;

ListNode l3 = new ListNode(Integer.MIN\_VALUE);

ListNode result= l3;

while(l1!=null && l2!=null)

{

if(l1.val>l2.val)

{

l3.next = new ListNode(l2.val);

l2=l2.next;

l3=l3.next;

}

else

{

l3.next = new ListNode(l1.val);

l1 = l1.next;

l3=l3.next;

}

}

if(l1!=null)

{

while(l1!=null)

{

l3.next = new ListNode(l1.val);

l1=l1.next;

l3=l3.next;

}

}

if(l2!=null)

{

while(l2!=null)

{

l3.next = new ListNode(l2.val);

l2=l2.next;

l3=l3.next;

}

}

return result.next;

}

}

**/\*Merge two sorted lists recursively \*/**

/\*\*

 \* Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode(int x) { val = x; }

 \* }

 \*/

public class Solution {

    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

        if(l1==null && l2==null) return null;

        else if(l1==null) return l2;

        else if(l2==null) return l1;

        if(l1.val<=l2.val){

            l1.next = mergeTwoLists(l1.next,l2);

            return l1;

        }

        else{

            l2.next = mergeTwoLists(l1,l2.next);

            return l2;

        }

    }

}

**/\*Remove duplicates from sorted array \*/**

public class Solution {

public int removeDuplicates(int[] nums) {

int j = 0;

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

if (i > 0 && nums[i - 1] == nums[i])

continue;

nums[j] = nums[i];

j++;

}

return j;

}

}

**/\*remove all instances of that value in place and return the new length \*/**

public class Solution {

public int removeElement(int[] nums, int val) {

int low=0;

int high = nums.length-1;

while(low<=high)

{

while(high>low && nums[high]==val)

{

high--;

}

if(nums[low]==val)

{

nums[low] = nums[high]; // replacing low value with the last because order can be changed

high--;

}

low++;

}

return high+1;

}

}

**/\*check if suduko sudoko is valid or not \*/**

public class Solution {

public boolean isValidSudoku(char[][] board) {

HashSet[] row = new HashSet[9];

HashSet[] column = new HashSet[9];

HashSet[] ceil = new HashSet[9];

for(int i=0; i<9; i++)

{

row[i] = new HashSet<Character>();

column[i] = new HashSet<Character>();

ceil[i] = new HashSet<Character>();

}

for(int i=0 ; i<9 ; i++)

{

for(int j=0 ; j<9; j++)

{

if(board[i][j]!='.'){

if(row[i].contains(board[i][j]) || column[j].contains(board[i][j]) || ceil[3 \*(i/3)+(j/3)].contains(board[i][j]))

{

return false;

}

else

{

row[i].add(board[i][j]);

column[j].add(board[i][j]);

ceil[3\*(i/3)+(j/3)].add(board[i][j]);

}

}

}

}

return true;

}

}

**/\*count and say iterative like 1, 11, 21, 1211… \*/**

public class Solution {

public String countAndSay(int n) {

String result = "1";

for(int i=1; i<n; i++)

{

StringBuilder sb= new StringBuilder();

int j=0;

while(j<result.length())

{

char temp = result.charAt(j);

int count=0;

while(j<result.length()&&result.charAt(j)==temp)

{

j++;

count++;

}

sb.append(count);

sb.append(temp);

}

result=sb.toString();

}

return result;

}

}

**/\*count and say recursive like 1, 11, 21, 1211… \*/**

public class Solution {

public String countAndSay(int n) {

return helper("1", n);

}

public String helper(String step, int count) {

if(count <= 1) {

return step;

}

// process step

char currentChar = step.charAt(0);

int currentCount = 0;

StringBuilder result = new StringBuilder();

for(char ch : step.toCharArray()) {

if(ch != currentChar) {

// hitting a different sequence

// need to add to result

result.append(currentCount).append(currentChar);

currentChar = ch;

currentCount = 1;

} else {

currentCount++;

}

}

result.append(currentCount).append(currentChar);

return helper(result.toString(), count-1);

}

}

**/\* Length of last word in string / sentence\*/**

public class Solution {

public int lengthOfLastWord(String s) {

String str = s.trim();

if (str==null) {

return 0;

}

String[] k = str.split(" ");

return k[k.length-1].length();

}

}

**/\*plus one to the digit \*/**

public class Solution {

public int[] plusOne(int[] digits) {

int n = digits.length;

for(int i=n-1; i>=0; i--) {

if(digits[i] < 9) {

digits[i]++;

return digits;

}

digits[i] = 0;

}

int[] newNumber = new int [n+1];

newNumber[0] = 1;

return newNumber;

}

}

**/\*Add binary numbers / sum of binary numbers \*/**

public class Solution {

public String addBinary(String a, String b) {

int na = a.length()-1;

int nb = b.length()-1;

int sum=0 ;

int carry =0;

StringBuilder sb =new StringBuilder();

while(na>=0 || nb>=0)

{

int x = na>=0 ? a.charAt(na) - '0' : 0;

int y = nb>=0 ? b.charAt(nb) - '0' : 0;

sum=x^y^carry;

carry = x+y+carry>1 ? 1 : 0;

sb.insert(0, sum);

na--;

nb--;

}

if(carry==1)

sb.insert(0, 1);

return sb.toString();

}

}

**/\*climbing stairs in distinct ways \*/**

public class Solution {

public int climbStairs(int n) {

if(n==0 || n==1 || n==2)

return n;

int sum=2, prev=1, current=0;

for(int i=2;i<n;i++)

{

current=sum;

sum=sum+prev;

prev=current;

}

return sum;

}

}

**/\* Remove duplicates from sorted list \*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

if(head==null)

return head;

ListNode current = head;

ListNode comingPointer;

while(current.next!=null)

{

comingPointer = current.next;

if(current.val==comingPointer.val)

{

current.next = comingPointer.next;

}

else

{

current=current.next;

}

}

return head;

}

}

**/\* check if trees are equal\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if(p==null && q==null)

return true;

if(p==null || q==null)

return false;

if(p.val!=q.val)

return false;

return (isSameTree(p.left, q.left) && isSameTree(p.right, q.right));

}

}

**/\*check if tree is symmetric or not / mirror/ mirror image of tree\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSymmetric(TreeNode root) {

if(root==null)

return true;

return (isSymSubtree(root.left, root.right));

}

public boolean isSymSubtree(TreeNode left, TreeNode right)

{

if(left==null)

return (right==null);

if(right==null)

return false;

return (left.val==right.val && isSymSubtree(left.left, right.right) && isSymSubtree(left.right, right.left));

}

}

**/\* Tree level order traversal iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

if(root==null) return result;

ArrayList<TreeNode> level = new ArrayList<TreeNode>();

level.add(root);

while(!level.isEmpty()){

ArrayList<Integer> values = new ArrayList<Integer>();

for(int i=0; i<level.size(); i++){

values.add(level.get(i).val);

}

result.add(values);

ArrayList<TreeNode> nextlevel = new ArrayList<TreeNode>();

for(int j=0 ;j<level.size(); j++){

if(level.get(j).left!=null) nextlevel.add(level.get(j).left);

if(level.get(j).right!=null) nextlevel.add(level.get(j).right);

}

level = nextlevel;

}

return result;

}

}

**/\* Tree level order traversal recursive\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> list = new ArrayList<List<Integer>>();

helper(root, list, 0);

return list;

}

private void helper(TreeNode root,List<List<Integer>> list, int level){

if(root==null) return;

if(list.size()==level) list.add(new ArrayList<Integer>());

list.get(level).add(root.val);

helper(root.left, list, level+1);

helper(root.right, list, level+1);

}

}

**/\* Compare version numbers\*/**

public class Solution {

public int compareVersion(String version1, String version2) {

String [] v1 = version1.split("\\.");

String [] v2 = version2.split("\\.");

for(int i=0; i<Math.max(v1.length, v2.length); i++)

{

int n1 = i<v1.length ? Integer.parseInt(v1[i]) : 0;

int n2 = i<v2.length ? Integer.parseInt(v2[i]) : 0;

if(n1>n2)

return 1;

if(n2>n1)

return -1;

}

return 0;

}

}

**/\*Rotate an array\*/**

public class Solution {

public void rotate(int[] nums, int k) {

if(nums == null || nums.length < 2){

return;

}

k = k % nums.length;

reverse(nums, 0, nums.length - k - 1);

reverse(nums, nums.length - k, nums.length - 1);

reverse(nums, 0, nums.length - 1);

}

private void reverse(int[] nums, int i, int j){

int tmp = 0;

while(i < j){

tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

i++;

j--;

}

}

}

**/\*Excel sheet column title\*/**

public class Solution {

public String convertToTitle(int n) {

StringBuilder sb = new StringBuilder();

while(n!=0)

{

sb.insert(0, (char)('A'+(n-1)%26));

n=(n-1)/26;

}

return sb.toString();

}

}

**/\*first bad version\*/**

/\* The isBadVersion API is defined in the parent class VersionControl.

boolean isBadVersion(int version); \*/

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int l = 1, r = n, mid;

while (l <= r) {

mid = l + (r-l) / 2;

if (isBadVersion(mid))

r = mid - 1;

else

l = mid + 1;

}

return l;

}

}

**/\*Design a stack that supports push, pop, top, and retrieving the minimum element in constant time / Min Stack\*/**

class MinStack {

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

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

public void push(int x) {

mainStack.push(x);

if (minStack.empty()) {

minStack.push(x);

} else if (minStack.peek() >= x) {

minStack.push(x);

}

}

public void pop() {

int poppedElement = mainStack.pop();

if (poppedElement == minStack.peek()) {

minStack.pop();

}

}

public int top() {

return mainStack.peek();

}

public int getMin() {

return minStack.peek();

}

}

**/\*Summary ranges\*/**

public class Solution {

public List<String> summaryRanges(int[] nums) {

StringBuffer sb = new StringBuffer();

List<String> res= new ArrayList<String>();

if(nums == null || nums.length==0)

return res;

if(nums.length==1)

{

sb.append(nums[0]);

res.add(sb.toString());

return res;

}

int p=0;

for (int i = 1; i < nums.length; i++)

{

if (nums[i] == nums[i - 1] + 1)

{

if (i == nums.length - 1)

{

res.add(nums[p] + "->" + nums[i]);

}

}

else

{

if (i == p + 1)

{

res.add(nums[p] + "");

}

else

{

res.add(nums[p] + "->" + nums[i - 1]);

}

if (i == nums.length - 1)

{

res.add(nums[i] + "");

}

p = i;

}

}

return res;

}

}

**/\*All paths in binary tree / binary tree paths\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<String> binaryTreePaths(TreeNode root) {

List<String> list = new ArrayList<String>();

if(root!=null)

{

if(root.left==null && root.right==null)

{

list.add(root.val+"");

}

else

{

if(root.left!=null)

{

list.addAll(binaryTreePaths(root.left));

}

if(root.right!=null)

{

list.addAll(binaryTreePaths(root.right));

}

for(int i=0;i<list.size();i++)

{

list.set(i, root.val + "->" +list.get(i));

}

}

}

return list;

}

}

**/\*Remove a given element / value from a linked list \*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode store = new ListNode(0);

store.next = head;

ListNode current = store;

while(current.next!=null)

{

if(current.next.val == val)

current.next = current.next.next;

else

current=current.next;

}

return store.next;

}

}

**/\* count prime numbers / prime nos exclusive the given number\*/**

public class Solution {

public int countPrimes(int n) {

boolean [] m = new boolean[n];

int count=0;

for(int i=2; i<n ; i++)

{

if(m[i])

continue;

count++;

for(int j=i ; j<n ; j+=i)

m[j] = true;

}

return count;

}

}

**/\* Nim Game\*/**

public class Solution {

public boolean canWinNim(int n) {

return ((n%4)!=0);

}

}

**/\* Maximum depth of binary tree / max depth of binary tree\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int maxDepth(TreeNode root) {

if(root==null)

return 0;

return (Math.max(maxDepth(root.left) + 1,maxDepth(root.right)+1));

}

}

**/\*Delete node in a linkedList / linked list \*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;

}

}

**/\*Invert Binary Tree recursion \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode invertTree(TreeNode root) {

if(root==null)

return root;

TreeNode temp = root.left;

root.left = invertTree(root.right);

root.right = invertTree(temp);

return root;

}

}

**/\*Invert Binary Tree iterative \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode invertTree(TreeNode root) {

if(root == null) return root;

Queue q = new LinkedList<TreeNode>();

q.add(root);

while(!q.isEmpty()){

TreeNode node = (TreeNode)q.remove();

TreeNode tempRight = null, tempLeft = null;

if(node.right != null){

q.add(node.right);

tempRight = node.right;

}

if(node.left != null){

q.add(node.left);

tempLeft = node.left;

}

node.left = tempRight;

node.right = tempLeft;

}

return root;

}

}

**/\*Excel sheet column Number\*/**

public class Solution {

public int titleToNumber(String s) {

{

int sum = 0;

s = s.toUpperCase();

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

{

sum = 26 \* sum + (s.charAt(i) - 'A'+1);

}

return sum;

}

}

}

**/\* Lowest common ancestors in Binary search tree \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root==null || p==root || q==root)

return root;

if((p.val<root.val && q.val>root.val) || (p.val>root.val && q.val<root.val))

return root;

else if(p.val<root.val && q.val<root.val)

return lowestCommonAncestor(root.left, p, q);

else

return lowestCommonAncestor(root.right, p, q);

}

}

**/\* check if two strings are anagram ( less optimize)\*/**

public class Solution {

public boolean isAnagram(String s, String t) {

if(s==null || t==null)

return true;

if(s.length()!=t.length())

return false;

char [] c1 = s.toCharArray();

Arrays.sort(c1);

char [] c2 = t.toCharArray();

Arrays.sort(c2);

String s1 = String.valueOf(c1);

String s2 = String.valueOf(c2);

if(s1.equals(s2))

return true;

else

return false;

}

}

**/\* check if two strings are anagram (more optimize using map)\*/**

public class Solution {

public boolean isAnagram(String s, String t) {

if(s.length() != t.length() ){

return false;

}

if(s == null && t == null){

return false;

}

if(s.equals(t)){

return true;

}

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

for(char a : s.toCharArray()){

map.put(a, map.getOrDefault(a, 0) + 1);

}

for(char b : t.toCharArray()){

if(map.containsKey(b)){

map.put(b, map.get(b) - 1);

if(map.get(b) == 0){

map.remove(b);

}

}else{

map.put(b, map.getOrDefault(b, 0) + 1);

}

}

if(map.isEmpty()){

return true;

}else{

return false;

}

}

}

**/\*Ugly Number \*/**

public class Solution {

public boolean isUgly(int num) {

if(num<=0)

return false;

if(num==1)

return true;

while(num%5==0)

num/=5;

while(num%3==0)

num/=3;

while(num%2==0)

num/=2;

return (num==1);

}

}

**/\*happy number \*/**

public class Solution {

public boolean isHappy(int n) {

int result = 0;

while (n > 9) {

result += Math.pow(n % 10, 2);

n = n / 10;

}

result += Math.pow(n, 2);

if (result > 9) {return isHappy(result);}

if (result == 1) {

return true;

} else {

return false;

}

}

}

**/\* Minimum Depth of binary tree / min depth of binary tree \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int minDepth(TreeNode root) {

if(root==null)

return 0;

if(root.left==null && root.right==null)

return 1;

if(root.right==null)

return 1+minDepth(root.left);

if(root.left==null)

return 1+minDepth(root.right);

return (Math.min(minDepth(root.left) , minDepth(root.right)))+1 ;

}

}

**/\* Intersection of 2 linked lists\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if(headA==null || headB==null) //boundary check

return null;

ListNode a = headA;

ListNode b = headB;

//if a and b have different lengths, then we will stop the loop after the second iteration

while(a!=b)

{

//for the end of first iteration, we just reset the pointer to the head of another linkedlist

a= a==null? headB : a.next;

b= b==null? headA : b.next;

}

return a;

}

}

**/\* Balanced Binary Tree\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isBalanced(TreeNode root) {

if(root==null || (root.left==null&&root.right==null))

return true;

return Math.abs(height(root.left)-height(root.right))<=1 && isBalanced(root.left) && isBalanced(root.right)? true : false;

}

public int height(TreeNode node)

{

if (node==null)

return 0;

return 1+Math.max(height(node.left), height(node.right));

}

}

**/\* check if strings are isomorphic or not \*/**

public class Solution {

public boolean isIsomorphic(String s, String t) {

if(s.length()!=t.length())

return false;

if(s.length()==0)

return true;

Map<Character, Integer> m1 = new HashMap<Character, Integer>();

Map<Character, Integer> m2 = new HashMap<Character, Integer>();

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

{

if(!m1.containsKey(s.charAt(i)))

m1.put(s.charAt(i), i);

if(!m2.containsKey(t.charAt(i)))

m2.put(t.charAt(i), i);

if(m1.size()!=m2.size())

return false;

else

if(m1.get(s.charAt(i))!=m2.get(t.charAt(i)))

return false;

}

return true;

}

}

**/\*check if the number is power of two / power of 2\*/**

public class Solution {

public boolean isPowerOfTwo(int n) {

if(n==1)

return true;

if(n%2 != 0)

return false;

int sum = 1;

for(int i=0 ; i<n/2 ; i++)

{

if(sum>n)

return false;

if(sum==n)

return true;

sum \*= 2;

}

if(sum == n)

return true;

return false;

}

}

**/\* Level order traversal of binary tree in reverse order \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> list1 = new ArrayList<List<Integer>>();

helper(root, list1, 0);

Collections.reverse(list1);

return list1;

//List reversal can also be done like this.

/\*

List<List<Integer>> list2 = new ArrayList<List<Integer>>();

int n= list1.size();

for(int i=0 ; i< n ;i++)

{

list2.add(list1.get(n-1-i));

}

\*/

}

private void helper(TreeNode root,List<List<Integer>> list, int level){

if(root==null) return;

if(list.size()==level) list.add(new ArrayList<Integer>());

list.get(level).add(root.val);

helper(root.left, list, level+1);

helper(root.right, list, level+1);

}

}

**/\* house Robber\*/**

public class Solution {

public int rob(int[] num) {

if(num.length==0)

return 0;

if(num.length==1)

return num[0];

num[1] = Math.max(num[0], num[1]);

for(int i=2 ; i<num.length; i++)

num[i] = Math.max(num[i]+num[i-2], num[i-1]);

return num[num.length-1];

}

}

**/\* Check if sum of node of path in binary tree equals a given number / All paths in Binary tree\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if(root==null)

return false;

if(root.left==null && root.right==null)

return root.val==sum;

int subSum = sum - root.val;

return (hasPathSum(root.left, subSum) || hasPathSum(root.right, subSum));

}

}

**/\*Trailing zeros in factorial of a number \*/**

public class Solution {

public int trailingZeroes(int n) {

int count=0;

while(n>1)

{

count = count+ (n/5);

n=n/5;

}

return count;

}

}

**/\*Rectangle Area – two rectangles are combined to find the total area\*/**

public class Solution {

public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {

int space = 0;

int space1 = (Math.abs(A-C)\*Math.abs(B-D));

int space2 = (Math.abs(E-G)\*Math.abs(F-H));

int left = Math.max(A,E);

int top = Math.min(D,H);

int right = Math.min(C,G);

int bottom = Math.max(B,F);

if(top>=bottom && right>=left){

space = (Math.abs(bottom-top) \* Math.abs(left-right));

}

return space1+space2-space;

}

}

**/\* Reverse a linked list iterative\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

**/\* Reverse a linked list iterative recursive\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode reverseList(ListNode head) {

if(head == null ||head.next == null){

return head;

}

ListNode root = reverseList(head.next);

head.next.next = head;

head.next = null;

return root;

}

}

**/\* check if linked list is palindrome or not iterative\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isPalindrome(ListNode head) {

if (head == null || head.next == null)

return true;

ListNode fast = head, slow = head, prev = null;

while (fast != null && fast.next != null) {

prev = slow;

slow = slow.next;

fast = fast.next.next;

}

prev.next = null;

slow = reverseList(slow);

while (head != null && slow != null) {

if (head.val != slow.val)

return false;

head = head.next;

slow = slow.next;

}

return true;

}

public ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

**/\* check if linked list is palindrome or not recursive \*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

ListNode h;

public boolean isPalindrome(ListNode head) {

if (head == null)

return true;

if (h == null)

h = head;

boolean tmp = true;

if (head.next != null)

tmp &= isPalindrome(head.next);

tmp &= (head.val == h.val);

h = h.next;

return tmp;

}

}

**/\*Merge N sorted arrays / n-sorted array\*/**

public int[] mergeNArrays(int[][] arrays, int n)

{

int result[];

for(int i=0; i<n ; i++)

{

result[] = Merge(result, arrays[i]);

}

return result;

}

**/\*find the single number in the array with all numbers two times except one\*/**

public class Solution {

public int singleNumber(int[] nums) {

int n = 0;

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

{

n = n ^ nums[i];

}

return n;

}

}

**/\*Best time to buy and sell stocks with single transaction \*/**

public class Solution {

public int maxProfit(int[] prices) {

int max = 0;

int min = Integer.MAX\_VALUE;

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

{

min = Math.min(min, prices[i]);

max = Math.max(max, prices[i]-min);

}

return max;

}

}

**/\* Best time to buy and sell stock with multiple transactions\*/**

public class Solution {

public int maxProfit(int[] prices) {

int profit = 0;

for(int i=1; i<prices.length; i++)

{

if(prices[i]-prices[i-1]<=0)

continue;

profit += prices[i] - prices[i-1];

}

return profit;

}

}

**/\* find the single number in the array with all numbers three times except one \*/**

public class Solution {

public int singleNumber(int[] nums) {

if(nums==null || nums.length==0)

return 0;

HashMap <Integer, Integer> hm = new HashMap<Integer, Integer>();

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

hm.put(nums[i], hm.getOrDefault(nums[i], 0)+1);

for(Map.Entry<Integer, Integer> en : hm.entrySet())

{

if(en.getValue()!=3)

return en.getKey();

}

return -1;

}

}

**/\* find the single number / two numbers in the array with all numbers two times except two elements which occur one time \*/**

public class Solution {

public int[] singleNumber(int[] nums) {

Set<Integer> set = new HashSet<Integer>();

for(int i:nums){

if(set.add(i)==false)

set.remove(i);

}

int a[]= new int [set.size()];

int c=0;

for(int b:set){

a[c]=b;

c++;

}

return a;

}

}

**/\*Product of array except itself \*/**

public class Solution {

public int[] productExceptSelf(int[] nums) {

int n = nums.length;

int[] output = new int[n];

int temp = 1;

output[0] = 1;

for(int i = 1; i<n ; i++)

{

output[i] = output[i-1] \* nums[i-1];

}

for(int i=n-1 ; i>=0; i--)

{

output[i]= temp\*output[i];

temp = temp\*nums[i];

}

return output;

}

}

**/\*check if linked list / linkedlist has a cycle \*/**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public boolean hasCycle(ListNode head) {

if(head==null)

return false;

ListNode fast = head;

ListNode slow = head;

while((fast!= null && fast.next!=null))

{

fast = fast.next.next;

slow= slow.next;

if(fast==slow)

break;

}

if(fast== null || fast.next==null)

return false;

else

return true;

}

}

**/\*find the missing value in the array of integers \*/**

public class Solution {

public int missingNumber(int[] nums) {

int n = nums.length;

int expectedOutput = (n\*(n+1))/2;

for(int i=0;i<n;i++)

{

expectedOutput-=nums[i];

}

return expectedOutput;

}

}

**/\*Populating Next Right Pointers in each node of Binary search tree / left node should make an arrow towards right node \*/**

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if(root==null)

{

return;

}

if(root.left!=null)

{

root.left.next = root.right;

if(root.next!=null)

root.right.next = root.next.left;

}

if(root.left!=null)

connect(root.left);

if(root.right!=null)

connect(root.right);

}

}

**/\* Search insert position of a target value in a sorted array, if already present, return its index \*/**

public class Solution {

public int searchInsert(int[] nums, int target) {

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

{

if(nums[i]>=target)

return i;

}

return nums.length;

}

}

**/\* Integer to roman less optimized solution iteratively without using enum\*/**

int [] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

String [] str = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

StringBuilder sb = new StringBuilder();

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

{

while(num>=values[i])

{

num-= values[i];

sb.append(str[i]);

}

}

return sb.toString();

**/\* Integer to roman less optimized solution iteratively using enum\*/**

public class Solution {

public enum Type{

M(1000),CM(900),D(500),CD(400),C(100),XC(90),L(50),XL(40),X(10),IX(9),V(5),IV(4),I(1);

private final int value;

Type(int value) {

this.value = value;

}

};

public String intToRoman(int num) {

StringBuilder output = new StringBuilder();

for (Type t:Type.values()) {

while (num>=t.value) {

output.append(t);

num -= t.value;

}

}

return output.toString();

}

}

**/\* Integer to roman more optimized solution recursively \*/**

public class Solution {

public String intToRoman(int num) {

if (num>=1000) {

return ("M"+intToRoman(num-1000));

} else if (num>=900) {

return ("CM"+intToRoman(num-900));

} else if (num>=500) {

return ("D"+intToRoman(num-500));

} else if (num>=400) {

return ("CD"+intToRoman(num-400));

} else if (num>=100) {

return ("C"+intToRoman(num-100));

} else if (num>=90) {

return ("XC"+intToRoman(num-90));

} else if (num>=50) {

return ("L"+intToRoman(num-50));

} else if (num>=40) {

return ("XL"+intToRoman(num-40));

} else if (num>=10) {

return ("X"+intToRoman(num-10));

} else if (num>=9) {

return ("IX"+intToRoman(num-9));

} else if (num>=5) {

return ("V"+intToRoman(num-5));

} else if (num>=4) {

return ("IV"+intToRoman(num-4));

} else if (num>=1) {

return ("I"+intToRoman(num-1));

}

return ("");

}

}

**/\* Maximum subarray / Maximum sub array/ Maximum sub-array/ sum of sub array which has maximum sum\*/**

public class Solution {

public int maxSubArray(int[] nums) {

int max = nums[0];

int [] arr = new int[nums.length];

arr[0] = nums[0];

for(int i=1 ; i<nums.length; i++)

{

arr[i] = Math.max(nums[i], nums[i]+arr[i-1]);

max= Math.max(max, arr[i]);

}

return max;

}

}

**/\*Convert a sorted array to binary search tree / convert a sorted array to BST recursive\***

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode sortedArrayToBST(int[] nums) {

if(nums == null || nums.length==0)

return null;

return getTreeNode(nums, 0 , nums.length-1);

}

private TreeNode getTreeNode(int [] nums, int start, int end )

{

if(start>end)

return null;

int middle = start + (end-start)/2 ;

TreeNode n = new TreeNode(nums[middle]);

n.left = getTreeNode(nums, start, middle-1);

n.right = getTreeNode(nums, middle+1, end);

return n;

}

}

**/\* Convert a sorted array to binary search tree / convert a sorted array to BST iterative \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode sortedArrayToBST(int[] nums) {

int len = nums.length;

if ( len == 0 ) { return null; }

// 0 as a placeholder

TreeNode head = new TreeNode(0);

Deque<TreeNode> nodeStack = new LinkedList<TreeNode>() {{ push(head); }};

Deque<Integer> leftIndexStack = new LinkedList<Integer>() {{ push(0); }};

Deque<Integer> rightIndexStack = new LinkedList<Integer>() {{ push(len-1); }};

while ( !nodeStack.isEmpty() ) {

TreeNode currNode = nodeStack.pop();

int left = leftIndexStack.pop();

int right = rightIndexStack.pop();

int mid = left + (right-left)/2; // avoid overflow

currNode.val = nums[mid];

if ( left <= mid-1 ) {

currNode.left = new TreeNode(0);

nodeStack.push(currNode.left);

leftIndexStack.push(left);

rightIndexStack.push(mid-1);

}

if ( mid+1 <= right ) {

currNode.right = new TreeNode(0);

nodeStack.push(currNode.right);

leftIndexStack.push(mid+1);

rightIndexStack.push(right);

}

}

return head;

}

}

**/\* Find minimum in a rotated sorted array \*/**

public class Solution {

public int findMin(int[] nums) {

int l = 0, r = nums.length-1;

while (l < r) {

int mid = (l + r) / 2;

if (nums[mid] < nums[r]) {

//right side in order, and mid is smallest of right side,

// min should be mid or in left.

r = mid;

} else {

//left side in order and it is the larger part,

// min should be in right side.

l = mid + 1;

}

}

return nums[l];

}

}

**/\* Generate Parenthesis / generate all combination of parenthesis\*/**

public class Solution {

public List<String> generateParenthesis(int n) {

List<String> li = new ArrayList<String>();

if(n<0)

return li;

generate("", li, n, n);

return li;

}

public void generate(String s, List<String> list, int left, int right)

{

if(left>right)

return;

if(left>0)

generate(s+"(", list, left-1, right);

if(right>0)

generate(s+")", list, left , right-1);

if(left==0&&right==0)

{

list.add(s);

return;

}

}

}

**/\* sort colors / dutch flag problem / sort 0 1 2\*/**

public class Solution {

public void sortColors(int[] nums) {

int r = 0;

int w = 0 ;

int b = 0;

for(int i : nums){

if(i==0)

r++;

if(i==1)

w++;

if(i==2)

b++;

}

int i=0;

while(r>0)

{

nums[i++] = 0;

r--;

}

while(w>0)

{

nums[i++] = 1;

w--;

}

while(b>0)

{

nums[i++] = 2;

b--;

}

}

}

**/\* Swap nodes in pairs in linked list.\*/**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode swapPairs(ListNode head) {

if(head==null || head.next==null)

return head;

ListNode current = head;

ListNode next;

ListNode result = new ListNode(-1);

ListNode prev = result;

while(current!=null && current.next!=null)

{

next = current.next;

prev.next = next;

current.next = next.next;

next.next = current;

prev = current;

current = current.next;

}

return result.next;

}

}

**/\* Find permutations of collection of numbers [first solution]\*/**

public class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

permute(result, nums, 0);

return result;

}

private void permute(List<List<Integer>> result, int[] array, int start) {

if (start >= array.length) {

List<Integer> current = new ArrayList<Integer>();

for (int a : array) {

current.add(a);

}

result.add(current);

} else {

for (int i=start; i<array.length; i++) {

swap(array, start, i);

permute(result, array, start+1);

swap(array, start, i);

}

}

}

private void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

**/\* Find permutations of collection of numbers [second solution]\*/**

public class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> permutations = new ArrayList<>();

if(nums.length==0)

return permutations;

generate(nums, 0 , new ArrayList<>(), permutations);

return permutations;

}

public void generate(int[] nums, int start, List<Integer> permutation, List<List<Integer>> permutations)

{

if(permutation.size()==nums.length){

permutations.add(permutation);

return;

}

for(int i=0 ;i<=permutation.size();i++)

{

List<Integer> newPermutation = new ArrayList<>(permutation);

newPermutation.add(i, nums[start]);

generate(nums, start+1, newPermutation, permutations);

}

}

}

**/\* Kth smallest element in BST recursive\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int kthSmallest(TreeNode root, int k) {

ArrayList<Integer>buffer = new ArrayList<Integer>();

inOrderSearch(root, buffer, k);

return buffer.get(k-1);

}

public void inOrderSearch(TreeNode node, ArrayList<Integer> buffer, int k)

{

if(buffer.size()>k)

return;

if(node.left!=null)

{

inOrderSearch(node.left, buffer, k);

}

buffer.add(node.val);

if(node.right!=null)

{

inOrderSearch(node.right, buffer, k);

}

}

}

**/\* Kth smallest element in BST iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int kthSmallest(TreeNode root, int k) {

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

int c=0;

TreeNode cur=root;

while(cur!=null){

stack.push(cur);

cur=cur.left;

}

while(!stack.isEmpty()){

TreeNode ptr=stack.pop();

c++;

if(c==k)return ptr.val;

TreeNode rt=ptr.right;

while(rt!=null){

stack.push(rt);

rt=rt.left;

}

}

return 0;

}

}

**/\* zig-zag / zig zag traversal of BST using recursion\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<List<Integer>>();

dfs(res, root, 0);

return res;

}

private void dfs(List<List<Integer>> res, TreeNode root, int level){

if (root == null) return;

if (level >= res.size()){

res.add(new LinkedList<Integer>());

}

LinkedList<Integer> list = (LinkedList<Integer>) res.get(level);

if (level % 2 == 0){

list.add(root.val);

} else {

list.addFirst(root.val);

}

dfs(res, root.left, level + 1);

dfs(res, root.right, level + 1);

}

}

**/\* Sum of root to leaf numbers in a binary search tree\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int sumNumbers(TreeNode root) {

if(root==null)

return 0;

if(root.left==null && root.right==null)

return root.val;

List<Integer> li = new ArrayList<Integer>();

StringBuilder sb = new StringBuilder();

binaryTree(root, sb, li);

int sum=0;

for(int i: li)

sum=sum+i;

return sum;

}

public void binaryTree(TreeNode root, StringBuilder sb, List<Integer> li)

{

if(root==null)

return;

StringBuilder result = new StringBuilder(sb);

result.append(root.val);

if(root.left==null && root.right==null)

{

li.add(Integer.parseInt(result.toString()));

}

binaryTree(root.left, result , li);

binaryTree(root.right, result , li);

}

}

**/\* Construct binary tree from inorder and postorder traversal \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode buildTree(int[] inorder, int[] postorder) {

if(inorder.length==0)

return null;

int head = postorder[postorder.length-1];

TreeNode root = new TreeNode(head);

int i = findIndex(inorder, head);

int[] inleft = Arrays.copyOfRange(inorder, 0, i);

int[] postleft = Arrays.copyOfRange(postorder, 0, i);

int[] inright = Arrays.copyOfRange(inorder, i+1, inorder.length);

int[] postright = Arrays.copyOfRange(postorder, i, inorder.length-1);

root.left = buildTree(inleft, postleft);

root.right = buildTree(inright, postright);

return root;

}

public static int findIndex(int[] a, int d)

{

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

{

if(a[i]==d)

return i;

}

return -1;

}

}

**/\*Construct binary tree from preorder and inorder traversal \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

if(inorder.length==0)

return null;

int head = preorder[0];

TreeNode root = new TreeNode(head);

int i = findIndex(inorder, head);

int[] inleft = Arrays.copyOfRange(inorder, 0, i);

int[] inright = Arrays.copyOfRange(inorder, i+1, inorder.length);

int[] preleft = Arrays.copyOfRange(preorder, 1, i+1);

int[] preright = Arrays.copyOfRange(preorder, i+1, inorder.length);

root.left = buildTree(preleft, inleft);

root.right = buildTree(preright, inright);

return root;

}

public static int findIndex(int[] a, int d)

{

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

{

if(a[i]==d)

return i;

}

return -1;

}

}

**/\*count number of nodes in a complete binary tree (Time limit exceeded) / not a good solution\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int countNodes(TreeNode root) {

if(root==null)

return 0;

return 1+countNodes(root.left) + countNodes(root.right);

}

}

**/\* count the number of nodes in a complete binary tree / optimal solution \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int countNodes(TreeNode root) {

if(root==null)

return 0;

int leftDepth = getDepth(root,'l');

int rightDepth = getDepth(root,'r');

if(leftDepth==rightDepth)

return (1<<leftDepth)-1;

else

return 1+countNodes(root.left)+countNodes(root.right);

}

public int getDepth(TreeNode node, char d)

{

int depth =0;

while(node!=null)

{

if(d == 'l')

node=node.left;

else

node=node.right;

depth++;

}

return depth;

}

}

**/\* Check if the given tree is a binary search tree / Validate Binary search Tree / Validate BST (Bad solution) / (Wrong solution)\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isValidBST(TreeNode root) {

if(root==null)

return true;

return isValidateBST(root.left)&&isValidateBST(root.right);

}

}

**/\* Check if the given tree is a binary search tree / Validate Binary search Tree / Validate BST (correct solution) / (perfect solution) \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isValidBST(TreeNode root) {

if(root==null)

return true;

return checkBST(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

public boolean checkBST(TreeNode root, long min, long max)

{

if(root==null)

return true;

if(root.val<=min || root.val>=max)

return false;

return checkBST(root.left, min, root.val)&&checkBST(root.right, root.val, max);

}

}

**/\* Right side view of the tree iterative\*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> rightSideView(TreeNode root) {

List<Integer> result = new LinkedList<Integer>();

if(root==null)

return result;

List<TreeNode> candidate = new LinkedList<TreeNode>();

candidate.add(root);

while(!candidate.isEmpty())

{

List<TreeNode> temp = new LinkedList<TreeNode>();

result.add(candidate.get(0).val);

for(TreeNode n : candidate)

{

if(n.right!=null)

temp.add(n.right);

if(n.left!=null)

temp.add(n.left);

}

candidate = temp;

}

return result;

}

}

**/\* Right side view of the tree recursive \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> rightSideView(TreeNode root) {

ArrayList list = new ArrayList();

rightSideView(root, 0, list);

return list;

}

public void rightSideView(TreeNode root, int level, ArrayList list) {

if(root == null) return;

if(list.size() == level)

list.add(root.val);

rightSideView(root.right, level + 1, list);

rightSideView(root.left, level + 1, list);

}

}

**/\* Lowest common ancestor in a binary tree \*/**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root==null || root == p || root==q)

return root;

TreeNode left = lowestCommonAncestor(root.left, p , q);

TreeNode right = lowestCommonAncestor(root.right, p , q);

if(left!=null && right!=null)

return root;

return left!=null ? left : right;

}

}