Tutu’rself
Crack the Coding Interview

Arpan Das

Volume - I

Coding Interview Questions for Top Companies

1. Valid Parentheses:

```java
public boolean isValid(String s) {
    if (s == null || s.length() == 0) return true;
    char[] arr = s.toCharArray();
    Stack<Character> stack = new Stack<>();
    for (char current: arr) {
        if (current == '(' || current == '{' || current == '[') {
            stack.push(current);
        } else if (stack.isEmpty()) {
            return false;
        } else {
            char pop = stack.pop();
            if (pop == '(' && current != ')') return false;
            if (pop == '{' && current != '}') return false;
            if (pop == '[' && current != ']') return false;
        }
    }
    if (!stack.isEmpty()) return false;
    return true;
}
```

2. Longest Substring without repeating Characters:

```java
public int lengthOfLongestSubstring(String s) {
    if (s == null)  return 0;
    Set<Character> set = new HashSet<>();
    int left = 0, right = 0, maxLength = 0;
    while (right < s.length()) {
        char current = s.charAt(right);
        if (!set.contains(current)) {
            set.add(current);
            right++;
            maxLength = Math.max(maxLength, (right - left));
        } else {
            char toRemove = s.charAt(left);
            set.remove(toRemove);
            left++;
        }
    }
    return maxLength;
}
```

3. Merge K sorted List:

```java
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;
    if (lists.length == 1) return lists[0];
    ListNode head = null;
    for (int i = 1; i < lists.length; i++) {
        if (i == 1) head = merge(lists[0], lists[1]);
        else head = merge(head, lists[i]);
    }
    return head;
}

private ListNode merge(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    ListNode temp = new ListNode(-1);
    ListNode current = temp;

    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }
    current.next = l1 == null ? l2 : l1;
    return temp.next;
}
```

4. Two Sum | Find target sum from array:

```java
public int[] twoSum(int[] nums, int target) {
    if (nums == null || nums.length < 2) return new int[0];
    int[] result = new int[2];
    Map<Integer,Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
      if (map.get(nums[i]) != null) {
        result[0] = i;
        result[1] = map.get(nums[i]);
        break;
      } else {
        int diff = (target - nums[i]);
        map.put(diff, i);
      }
    }
    return result;
}
```

5. Binary Tree Level Order Traversal:

```java
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> nodeList = new ArrayList<>();
    if (root == null) return nodeList;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        List<Integer> level = new ArrayList<>();
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        nodeList.add(level);
    }
    return nodeList;
}
```

6. Maximum Subarray Sum:

```java
public int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    boolean isAllNegative = true;
    int minNegative = Integer.MIN_VALUE;
    int currentSum = 0;
    int maxSum = 0;
    for (int current: nums) {
        if (current > 0) isAllNegative = false;
        if (current > minNegative && isAllNegative){
          minNegative = current;
        }
        currentSum = currentSum + current;
        if (currentSum < 0) currentSum = 0;
        if (currentSum > maxSum) maxSum = currentSum;
    }
    return isAllNegative ? minNegative : maxSum;
}
```

7. Climbing Stairs:

```java
int destination = 0;
int[] memo = null;
public int climbStairs(int n) {
    memo = new int[n + 1];
    destination = n;
    return climb(0);
}
private int climb(int currentStairNo) {

    if (currentStairNo > destination) return 0;
    if (currentStairNo == destination) return 1;
    if (memo[currentStairNo] > 0) return memo[currentStairNo];

    memo[currentStairNo] = climb(currentStairNo + 1) +
                           climb(currentStairNo + 2);

    return memo[currentStairNo];
}
```

8. Number of Island:

```java
public int numOfIslands(char[][] grid) {
    if (grid == null || grid.length == 0) return 0;
    int height = grid.length;
    int width = grid[0].length;
    int number = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (grid[i][j] == '1') {
                number++;
                DFS(grid, i, j);
            }
        }
    }
    return number;
}

private void DFS(char[][] grid, int i, int j) {
    int h = grid.length , w = grid[0].length;
    if (i < 0 || j < 0 || i >= h || j >= w || grid[i][j] == '0'){
       return;
    }
    grid[i][j] = '0';
    DFS(grid, (i - 1), j);
    DFS(grid, (i + 1), j);
    DFS(grid, i, (j - 1));
    DFS(grid, i, (j + 1));
}
```

9. Generate Parentheses:

```java
public List<String> generateParentheses(int n) {
    List<String> result = new ArrayList<>();
    char[] res = new char[2 * n];
    generate(result, res, 0, 0, 0, n);
    return result;
}

private void generate(List<String> result, char[] res, int pos,int open,
                      int close, int max) {

    if (res.length == pos) {
        if (isValid(res)){
           result.add(String.valueOf(res));
        }
    }
    else {
        if (open <= max) {
            res[pos] = '(';
            generate(result, res, pos + 1, open + 1, close, max);
            res[pos] = ')';
        }
        generate(result, res, pos + 1, open, close + 1, max);
    }
}

private boolean isValid(char[] res) {
    int balance = 0;
    for (char c: res) {
        if (c == '(') balance ++;
        if (c == ')') balance --;
        if (balance < 0) return false;
    }
    return balance == 0;
}
```

10. Add Two Numbers:

```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

    if (l1 == null) return l2;
    if (l2 == null) return l1;
    ListNode temp = new ListNode(0);
    ListNode current = temp;
    int carry = 0;

    while (l1 != null && l2 != null) {
        int sum = l1.val + l2.val + carry;
        carry = sum / 10;
        sum = sum % 10;
        current.next = new ListNode(sum);
        l1 = l1.next;
        l2 = l2.next;
        current = current.next;
    }

    if (l1 != null) {
        if (carry > 0) {
            current.next = addTwoNumbers(l1, new ListNode(carry));
        } else current.next = l1;
    }

    if (l2 != null) {
        if (carry > 0) {
            current.next = addTwoNumbers(l2, new ListNode(carry));
        } else current.next = l2;
    }

    if (l1 == null && l2 == null && carry > 0)
        current.next = new ListNode(carry);

    return temp.next;
}
```

11. Remove Nth last node of a Linked List:

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    if (head == null || n < 0) return new ListNode(0);
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode fast = dummy, slow = dummy;
    while (fast != null && n > 0) {
        fast = fast.next;
        n--;
    }
    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }
    slow.next = slow.next.next;
    return dummy.next;
}
```

12. Merge Two Sorted List:

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    ListNode temp = new ListNode(-1);
    ListNode current = temp;
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }
    current.next = l1 == null ? l2 : l1;
    return temp.next;
}
```

13. Rotate List:

```
public ListNode rotateRight(ListNode head, int k) {
    if (head == null) return null;
    if (head.next == null) return head;
    ListNode temp = head;
    int len = 1;
    while (temp.next != null) {
        temp = temp.next;
        len++;
    }
    temp.next = head;
    if (k >= len) k = k % len;
    k = (len - k - 1);
    ListNode newTail = head;
    while (k > 0) {
        newTail = newTail.next;
        k--;
    }
    ListNode newHead = newTail.next;
    newTail.next = null;
    return newHead;
}
```

14. Copy List with Random Pointer:

```
private Map<Node,Node> visited = new HashMap<>();
public Node copyRandomList(Node head) {
    if (head == null) return null;
    if (visited.containsKey(head)) {
        return visited.get(head);
    }
    Node node = new Node(head.val);
    visited.put(head, node);
    node.next = copyRandomList(head.next);
    node.random = copyRandomList(head.random);
    return node;
}
```

15. Reorder List:

```java
public void reorderList(ListNode head) {
    if (head == null || head.next == null) return;
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    while (fastPtr != null && fastPtr.next != null) {
        fastPtr = fastPtr.next.next;
        slowPtr = slowPtr.next;
    }
    ListNode middle = slowPtr.next;
    slowPtr.next = null;
    middle = reverse(middle);
    head = merge(head, middle);
}
public ListNode reverse(ListNode head) {
    if (head == null)  return null;
    ListNode current = head;
    ListNode previous = null;
    while (current != null) {
        ListNode next = current.next;
        current.next = previous;
        previous = current;
        current = next;
    }
    return previous;
}
private ListNode merge(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    ListNode temp = new ListNode(0);
    ListNode current = temp;
    int i = 1;
    while (l1 != null && l2 != null) {
        if (i % 2 != 0) {
            current.next = l1;
            l1 = l1.next;
        }else {
            current.next = l2;
            l2 = l2.next;
        }
        i++;
        current = current.next;
    }
    if (l1 != null || l2 != null) current.next = l1 == null ? l2 : l1;
    return temp.next;
}
```

16. Group Anagrams:

```
public List<List<String>> groupAnagrams(String[] strs) {
  List<List<String>> result = new ArrayList<>();
  if(strs == null || strs.length == 0) return result;
  Map<String,List<String>> anagramGroup = new HashMap<>();
  for(String str : strs){
      char[] arr = str.toCharArray();
      Arrays.sort(arr);
      String key = String.valueOf(arr);
      List<String> anagramList = anagramGroup.get(key);
      if(anagramList == null){
        anagramList = new ArrayList<>();
      }
      anagramList.add(str);
      anagramGroup.put(key,anagramList);
  }
  for(String key:anagramGroup.keySet()){
      result.add(anagramGroup.get(key));
  }
  return result;
}
```

17. Lowest Common Ancestor of Binary Tree:

```
public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p, TreeNode q)
{
    if (p == null) return q;
    if (q == null) return p;

    Queue<TreeNode> queue = new LinkedList<>();
    Map<TreeNode,TreeNode> map = new HashMap<>();
    queue.offer(root);
    map.put(root, null);

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (node.left != null) {
                map.put(node.left, node);
                queue.offer(node.left);
            }
            if (node.right != null) {
                map.put(node.right, node);
                queue.offer(node.right);
            }
        }
    }
    Set<TreeNode> ancestorOfP = new HashSet<>();
    while (p != null) {
        ancestorOfP.add(p);
        p = map.get(p);
    }
    while (q != null) {
        if (ancestorOfP.contains(q)) {
            return q;
        }
        ancestorOfP.add(q);
        q = map.get(q);
    }
    return null;
}
```

18. Binary Tree Maximum Path Sum:

```
int maximumSum = Integer.MIN_VALUE;
public int maxPathSum(TreeNode root) {
    sum(root);
    return maximumSum;
}

private int sum(TreeNode node) {
    if (node == null) return 0;
    int leftSum = Math.max(sum(node.left), 0);
    int rightSum = Math.max(sum(node.right), 0);
    int currentSum = node.val + leftSum + rightSum;
    maximumSum = Math.max(maximumSum, currentSum);
    return Math.max(leftSum, rightSum) + node.val;
}
```

19. Binary Tree Zigzag Level Traversal:

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
      List<List<Integer>> result = new ArrayList<>();
      if(root == null) return result;
      Queue<TreeNode> queue = new LinkedList<>();
      queue.offer(root);
      boolean leftToRight = true;
      while(!queue.isEmpty()){
            int size = queue.size();
            Integer[] current = new Integer[size];
            int index = leftToRight ? 0 : (size-1);
            for(int i = 0 ; i < size ; i++){
              TreeNode node = queue.poll();
              current[index] = node.val;
              index = leftToRight ? index+1 : index-1;
              if(node.left != null) queue.offer(node.left);
              if(node.right != null) queue.offer(node.right);
            }
            result.add(Arrays.asList(current));
            leftToRight = leftToRight ? false : true;
      }
      return result;
}
```

20. Serialize and Deserialize Binary Tree:

```java
public String serialize(TreeNode root) {
    StringBuffer sb = new StringBuffer();
    _serialize(root, sb);
    return sb.toString();
}

private void _serialize(TreeNode node, StringBuffer sb) {
    if (node == null) {
        sb.append("null,");
        return;
    }
    sb.append(node.val).append(",");
    _serialize(node.left, sb);
    _serialize(node.right, sb);
}

public TreeNode deserialize(String data) {
    if (data == null || data.length() == 0) return new TreeNode(0);
    String[] nodes = data.split(",");
    LinkedList<String> nodeList = new LinkedList<>(Arrays.asList(nodes));
    TreeNode root = _deserialize(nodeList);
    return root;
}

private TreeNode _deserialize(LinkedList < String > nodeList) {
    if (nodeList.getFirst().equals("null")) {
        nodeList.removeFirst();
        return null;
    }
    TreeNode node = new TreeNode(Integer.parseInt(nodeList.getFirst()));
    nodeList.removeFirst();
    node.left = _deserialize(nodeList);
    node.right = _deserialize(nodeList);
    return node;
}
```

21. Symmetric Binary Tree:

```java
public boolean isSymmetric(TreeNode root) {
    if (root == null) return true;
    return isSymmetric(root, root);
}

private boolean isSymmetric(TreeNode node1, TreeNode node2) {
    Queue<TreeNode> q1 = new LinkedList<>();
    Queue<TreeNode> q2 = new LinkedList<>();
    q1.offer(node1);
    q2.offer(node2);
    while (!q1.isEmpty()) {
        TreeNode n1 = q1.poll();
        TreeNode n2 = q2.poll();
        if (n1 == null && n2 == null) continue;
        if (n1 == null || n2 == null) return false;
        if (n1.val != n2.val) return false;
        q1.offer(n1.left);
        q1.offer(n1.right);
        q2.offer(n2.right);
        q2.offer(n2.left);
    }
    return true;
}
```

22. Count Complete Binary Tree Nodes:

```java
public int countNodes(TreeNode root) {

    if (root == null) return 0;
    int leftHeight = leftHeight(root);
    int rightHeight = rightHeight(root);
    if (leftHeight == rightHeight) {
        return (int) Math.pow(2, leftHeight) - 1;
    }
    return countNodes(root.left) + countNodes(root.right) + 1;
}

private int leftHeight(TreeNode node) {
    int height = 0;
    while (node != null) {
        height++;
        node = node.left;
    }
    return height;
}

private int rightHeight(TreeNode node) {
    int height = 0;
    while (node != null) {
        height++;
        node = node.right;
    }
    return height;
}
```

23. Convert Sorted Array to Binary Search Tree:

```java
public TreeNode sortedArrayToBST(int[] nums) {
    if (nums == null || nums.length == 0) {
        return null;
    }
    TreeNode root = convert(nums, 0, nums.length - 1);
    return root;
}
private TreeNode convert(int[] nums, int left, int right) {
    if (left > right) return null;
    int pivot = (left + right) / 2;
    TreeNode node = new TreeNode(nums[pivot]);
    node.left = convert(nums, left, pivot - 1);
    node.right = convert(nums, pivot + 1, right);
    return node;
}
```

24. Subtree of another Binary Tree:

```java
public boolean isSubtree(TreeNode s, TreeNode t) {
    if (s == null && t == null) return true;
    if (s == null || t == null) return false;
    StringBuilder sBuilder = new StringBuilder();
    preOrder(s, sBuilder);
    StringBuilder tBuilder = new StringBuilder();
    preOrder(t, tBuilder);
    return sBuilder.toString().indexOf(tBuilder.toString()) >= 0;
}
private void preOrder(TreeNode node, StringBuilder sb) {
    if (node == null) {
        sb.append("null");
        return;
    }
    sb.append("#").append(String.valueOf(node.val));
    preOrder(node.left, sb);
    preOrder(node.right, sb);
}
```

25. Maximum Level Sum of Binary Tree:

```
public int maxLevelSum(TreeNode root) {
    if (root == null) return -1;
    Queue<TreeNode> queue = new LinkedList<>();
    int maxSum = Integer.MIN_VALUE;
    queue.add(root);
    int currentLevel = 1;
    int maxLevel = 0;
    while (!queue.isEmpty()) {
        int no_of_nodes = queue.size();
        int currentSum = 0;
        for (int i = 0; i < no_of_nodes; i++) {
            TreeNode node = queue.remove();
            currentSum = currentSum + node.val;
            if (node.left != null) queue.add(node.left);
            if (node.right != null) queue.add(node.right);
        }
        if (currentSum > maxSum) {
            maxSum = currentSum;
            maxLevel = currentLevel;
        }
        currentLevel++;
    }
    return maxLevel;
}
```

26. Maximum Depth of Binary Tree:

```
public int maxDepth(TreeNode root) {
    if (root == null) return 0;
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    return Math.max(left, right) + 1;
}

public int maxDepth(TreeNode root) {

    if (root == null) return 0;
    int maxDepth = Integer.MIN_VALUE;
    Queue<TreeNode> queue = new LinkedList<>();
    Queue<Integer> depthQ = new LinkedList<>();
    queue.offer(root); depthQ.offer(1);

    while (!queue.isEmpty()) {

        int currentDepth = depthQ.poll();
        TreeNode curNode = queue.poll();
        maxDepth = Math.max(maxDepth, currentDepth);

        if (curNode.left != null) {
            queue.offer(curNode.left);
            depthQ.offer(currentDepth + 1);
        }

        if (curNode.right != null) {
            queue.offer(curNode.right);
            depthQ.offer(currentDepth + 1);
        }
    }
    return maxDepth;
}
```

27. Binary Search Tree (BST) Iterator:

```
class BSTIterator {

    Stack<TreeNode> stack;

    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        leftInorder(root);
    }

    private void leftInorder(TreeNode node){
        while(node != null){
            stack.push(node);
            node = node.left;
        }
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode node = stack.pop();
        if(node.right != null){
            leftInorder(node.right);
        }
        return node.val;
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return stack.size() > 0;
    }
}
```

28. Boundary of a Binary Tree:

```java
public class Solution {

    public boolean isLeaf(TreeNode t) {
        return t.left == null && t.right == null;
    }

    public void addLeaves(List<Integer> res, TreeNode root) {
        if (isLeaf(root)) {
            res.add(root.val);
        } else {
            if (root.left != null) {
                addLeaves(res, root.left);
            }
            if (root.right != null) {
                addLeaves(res, root.right);
            }
        }
    }

    public List<Integer> boundaryOfBinaryTree(TreeNode root) {
        ArrayList<Integer> res = new ArrayList<>();
        if (root == null) {
            return res;
        }
        if (!isLeaf(root)) {
            res.add(root.val);
        }
        TreeNode t = root.left;
        while (t != null) {
            if (!isLeaf(t)) {
                res.add(t.val);
            }
            if (t.left != null) {
                t = t.left;
            } else {
                t = t.right;
            }

        }
        addLeaves(res, root);
        Stack<Integer> s = new Stack<>();
        t = root.right;
        while (t != null) {
            if (!isLeaf(t)) {
                s.push(t.val);
            }
            if (t.right != null) {
                t = t.right;
            } else {
                t = t.left;
            }
        }
        while (!s.empty()) {
            res.add(s.pop());
        }
        return res;
    }
}
```

29. Lowest Common Ancestor of a Binary Search Tree:

```
public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p,TreeNode q)
{
    if (root == null) return null;
    if (p == null) return q;
    if (q == null) return p;
    TreeNode lca = find(root, p, q);
    return lca;
}

private TreeNode find(TreeNode node, TreeNode p, TreeNode q) {
    if (p.val <= node.val && q.val >= node.val) return node;
    if (q.val <= node.val && p.val >= node.val) return node;
    TreeNode lca = null;
    if (q.val < node.val && p.val < node.val) {
        lca = find(node.left, p, q);
    } else if (q.val > node.val && p.val > node.val) {
        lca = find(node.right, p, q);
    }
    return lca;
}
```

30. Diameter of a Binary Tree:

```java
int maxDiameter = 0;
public int diameterOfBinaryTree(TreeNode root) {
    if (root == null)  return 0;
    calculateDiameter(root);
    return maxDiameter;
}
private int calculateDiameter(TreeNode node) {
    if (node == null) return 0;
    int left = calculateDiameter(node.left);
    int right = calculateDiameter(node.right);
    int currentDiameter = left + right;
    maxDiameter = Math.max(maxDiameter, currentDiameter);
    return Math.max(left, right) + 1;
}
```

31. Kth smallest element of a Binary Search Tree:

```java
private Stack<TreeNode> stack = new Stack<>();
private void populate(TreeNode node) {
    while (node != null) {
        stack.push(node);
        node = node.left;
    }
}
private int next() {
    TreeNode node = stack.pop();
    if (node.right != null) populate(node.right);
    return node.val;
}
public int kthSmallest(TreeNode root, int k) {
    populate(root);
    int kth = 0;
    while (k > 0) {
        kth = next();
        k--;
    }
    return kth;
}
```

32.Cousins in a Binary Tree:

```java
int levelX = 0, levelY = 0;
TreeNode parentX = null, parentY = null;
boolean foundX = false, foundY = false;

public boolean isCousins(TreeNode root, int x, int y) {
    if (root == null || x == y) return false;
    find(root, null, 1, x, y);
    return levelX == levelY && parentX.val != parentY.val;
}

private void find(TreeNode node, TreeNode parent, int level, int x, int y)
{
    if (node == null) return;
    if (foundX && foundY) return;
    if (node.val == x) {
        foundX = true;
        parentX = parent;
        levelX = level;
    } else if (node.val == y) {
        foundY = true;
        parentY = parent;
        levelY = level;
    }
    find(node.left, node, (level + 1), x, y);
    find(node.right, node, (level + 1), x, y);
}
```

33. Binary Tree Right Side View:

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if(root == null) return result;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        TreeNode node = null;
        for (int i = 0; i < size; i++) {
            node = queue.poll();
            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        result.add(node.val);
    }
    return result;
}
```

34. Find Duplicate Subtrees:

```
private Map<String,Integer> subTreeCount = new HashMap<>();
private List<TreeNode> result = new ArrayList<>();

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    if (root == null) return result;
    collect(root);
    return result;
}
private String collect(TreeNode node) {
    if (node == null)
        return "*";
    }
    String code = node.val + "," + collect(node.left) + "," +
                  collect(node.right);
    int count = subTreeCount.getOrDefault(code, 0);
    subTreeCount.put(code, (count + 1));
    if (count == 2) result.add(node);
    return code;
}
```

35.Flip Equivalent Binary Trees:

```
public boolean flipEquiv(TreeNode root1, TreeNode root2) {

    if (root1 == null && root2 == null) return true;
    if (root1 == null || root2 == null) return false;

    List<Integer> aList = new ArrayList<>();
    DFS(root1, aList);

    List<Integer> bList = new ArrayList<>();
    DFS(root2, bList);

    return aList.equals(bList);
}


private void DFS(TreeNode node, List<Integer> list) {
    if (node == null) return;
    list.add(node.val);

    int left = node.left == null ? -1 : node.left.val;
    int right = node.right == null ? -1 : node.right.val;
    if (left <= right) {
        DFS(node.left, list);
        DFS(node.right, list);
    } else {
        DFS(node.right, list);
        DFS(node.left, list);
    }
}
```

36.Populate next right pointer of each node of a Binary Trees:

```
// Every node has a right pointer as next
public Node connect(Node root) {
    if (root == null) {
        return root;
    }
    Queue<Node> Q = new LinkedList<Node>();
    Q.add(root);
    while (Q.size() > 0) {
        int size = Q.size();
        for (int i = 0; i < size; i++) {
            Node node = Q.poll();
            if (i < size - 1) {
                node.next = Q.peek();
            }
            if (node.left != null) {
                Q.add(node.left);
            }
            if (node.right != null) {
                Q.add(node.right);
            }
        }
    }
    return root;
}
```

37.Binary Tree Paths:

```
public List<String> binaryTreePaths(TreeNode root) {
    List<String> paths = new ArrayList<>();
    if (root == null) return paths;
    LinkedList<TreeNode> nodeStack = new LinkedList<>();
    LinkedList<String> pathStack = new LinkedList<>();
    nodeStack.add(root);
    pathStack.add(String.valueOf(root.val));
    String path = null;
    while (!nodeStack.isEmpty()) {
        TreeNode node = nodeStack.removeLast();
        path = pathStack.removeLast();
        if (node.left == null && node.right == null) {
            paths.add(path);
        }
        if (node.left != null) {
            nodeStack.add(node.left);
            pathStack.add(path + "->" + String.valueOf(node.left.val));
        }
        if (node.right != null) {
            nodeStack.add(node.right);
            pathStack.add(path + "->" + String.valueOf(node.right.val));
        }
    }
    return paths;
}
```

38. Find root to leaf paths with target sum in a Binary Tree:

```java
public List<List<Integer>> pathSum(TreeNode root, int sum) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;
    Queue<TreeNode> nodeQ = new LinkedList<>();
    nodeQ.offer(root);
    Queue<List<Integer>> pathQ = new LinkedList<>();
    List<Integer> pathList = new ArrayList<>();
    pathList.add(root.val);
    pathQ.offer(pathList);
    while (!nodeQ.isEmpty()) {
        TreeNode current = nodeQ.poll();
        List<Integer> path = pathQ.poll();
        if (current.left == null && current.right == null) {
            if (isPathEqualToSum(path, sum)) {
                result.add(path);
            }
        }
        if (current.left != null) {
            nodeQ.offer(current.left);
            List<Integer> leftList = new ArrayList<>(path);
            leftList.add(current.left.val);
            pathQ.offer(leftList);
        }
        if (current.right != null) {
            nodeQ.offer(current.right);
            List<Integer> rightList = new ArrayList<>(path);
            rightList.add(current.right.val);
            pathQ.offer(rightList);
        }
    }
    return result;
}

private boolean isPathEqualToSum(List<Integer> path, int sum) {
    int total = 0;
    for (int num: path) {
        total += num;
    }
    return sum == total;
}
```

39.Maximum depth of a N-ary Tree:

```java
// Definition of a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
}
public int maxDepth(Node root) {
    if (root == null) return 0;
    Queue<Node> nodeQueue = new LinkedList<>();
    Queue<Integer> depthQueue = new LinkedList<>();
    nodeQueue.offer(root);
    depthQueue.offer(1);
    int maxDepth = 0;
    while (!nodeQueue.isEmpty()) {
        Node node = nodeQueue.poll();
        Integer depth = depthQueue.poll();
        maxDepth = Math.max(maxDepth, depth);
        List < Node > childrens = node.children;
        if (childrens != null && childrens.size() > 0) {
            depth = depth + 1;
            for (Node node1: childrens) {
                nodeQueue.offer(node1);
                depthQueue.offer(depth);
            }
        }
    }
    return maxDepth;
}
```

40.N-ary Tree Level Order Traversal:

```
// Definition of a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
        Queue<Node> levelQueue = new LinkedList<>();
        levelQueue.offer(root);
        while (!levelQueue.isEmpty()) {
            List<Integer> level = new ArrayList<>();
            int size = levelQueue.size();
            for (int i = 0; i < size; i++) {
                Node node = levelQueue.poll();
                level.add(node.val);
                levelQueue.addAll(node.children);
            }
            result.add(level);
        }
        return result;
    }
}
```

41. Serialize and Deserialize N-ary Tree:

```
// Encodes a tree to a single string.
public String serialize(Node root) {
    if (root == null) return "";
    List<String> tree = new ArrayList<>();
    _serialize(root, tree);
    return String.join(",", tree);
}

private void _serialize(Node node, List<String> tree) {
    if (node == null) return;
    tree.add(String.valueOf(node.val));
    int size = node.children != null ? node.children.size() : 0;
    tree.add(String.valueOf(size));
    for (Node n: node.children) {
        _serialize(n, tree);
    }
}

// Decodes your encoded data to a tree.
public Node deserialize(String data) {
    if (data == null || data.isEmpty()) return null;
    Queue<String> queue = new
LinkedList<>(Arrays.asList(data.split(",")));
    Node node = _deserialize(queue);
    return node;
}

private Node _deserialize(Queue<String> queue) {
    int val = Integer.parseInt(queue.poll());
    Node current = new Node(val);
    int size = Integer.parseInt(queue.poll());
    List<Node> children = new ArrayList<>(size);
    for (int i = 0; i < size; i++) {
        Node child = _deserialize(queue);
        children.add(child);
    }
    current.children = children;
    return current;
}
```

42. All Nodes in Distance K in Binary Tree:

```
private Map<TreeNode, TreeNode> parent;

public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {
      parent = new HashMap();
      dfs(root, null);
      Queue<TreeNode> queue = new LinkedList();
      queue.add(null);
      queue.add(target);
      Set<TreeNode> seen = new HashSet();
      seen.add(target);
      seen.add(null);
      int dist = 0;
      while (!queue.isEmpty()) {
         TreeNode node = queue.poll();
         if (node == null) {
            if (dist == K) {
            List<Integer> ans = new ArrayList();
            for (TreeNode n: queue)
              ans.add(n.val);
            return ans;
          }
          queue.offer(null);
          dist++;
        } else {
          if (!seen.contains(node.left)) {
            seen.add(node.left);
            queue.offer(node.left);
          }
          if (!seen.contains(node.right)) {
            seen.add(node.right);
            queue.offer(node.right);
          }
          TreeNode par = parent.get(node);
          if (!seen.contains(par)) {
            seen.add(par);
            queue.offer(par);
          }
        }
     }
     return new ArrayList<Integer>();
}

public void dfs(TreeNode node, TreeNode par) {
      if (node != null) {
        parent.put(node, par);
        dfs(node.left, node);
        dfs(node.right, node);
      }
}
```

43. Plus One to Array:

```
public int[] plusOne(int[] digits) {
    List<Integer> result = new ArrayList<>();
    int carry = 1;
    int len = digits.length - 1;
    for (int i = len; i >= 0; i--) {
        int sum = digits[i] + carry;
        if (sum < 10) {
            result.add(sum);
            carry = 0;
        } else {
            carry = sum / 10;
            result.add(sum % 10);
        }
    }
    if (carry > 0)
        result.add(carry);
    Collections.reverse(result);
    int[] res = result.stream().mapToInt(i -> i).toArray();
    return res;
}
```

44. Unique Paths 1:

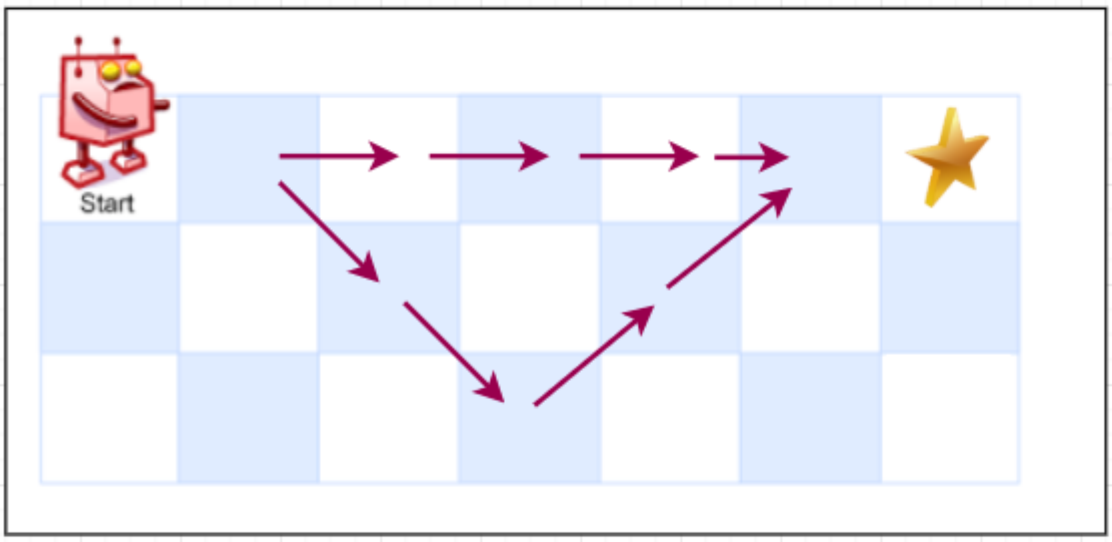
Backtracking:

```
public int uniquePaths(int m, int n) {
    int[][] board = new int[m][n];
    board[m - 1][n - 1] = 1;
    int count = backTrack(board, m, n, 0, 0);
    return count;
}

private int backTrack(int[][] board, int m, int n, int i, int j) {
    if (i == m || j == n) return 0;
    if (board[i][j] != 0) return board[i][j];
    int c1 = backTrack(board, m, n, (i + 1), j);
    int c2 = backTrack(board, m, n, i, (j + 1));
    board[i][j] = (c1 + c2);
    return board[i][j];
}
```

Dynamic Programming:

```
public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];
    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int i = 0; i < n; i++) dp[0][i] = 1;
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

45. Unique Paths 2:



```java
int height, width;
public int countPaths(int h, int w) {
  this.height = h;
  this.width = w;
  int[][] board = new int[h][w];
  board[0][w - 1] = 1;
  int count = backTrack(board, 0, 0);
  return count;
}

private int backTrack(int[][] board, int i, int j) {
  if (i >= height || j >= width || i < 0) return 0;
  if (board[i][j] != 0) return board[i][j];
  int c1 = backTrack(board, i, (j + 1));
  int c2 = backTrack(board, (i + 1), (j + 1));
  int c3 = backTrack(board, (i - 1), (j + 1));
  board[i][j] = (c1 + c2 + c3);
  return board[i][j];
}
```

46.Unique Paths 3 | Obstacle Grid:

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid[0][0] == 1) {
            return 0;
        }
        int rows = obstacleGrid.length;
        int columns = obstacleGrid[0].length;
        boolean obstacleFound = false;
        for (int i = 0; i < rows; i++) {
            if (obstacleGrid[i][0] == 0 && !obstacleFound) {
                obstacleGrid[i][0] = 1;
            } else {
                obstacleGrid[i][0] = 0;
                obstacleFound = true;
            }
        }
        obstacleFound = false;
        for (int i = 1; i < columns; i++) {
            if (obstacleGrid[0][i] == 0 && !obstacleFound) {
                obstacleGrid[0][i] = 1;
            } else {
                obstacleGrid[0][i] = 0;
                obstacleFound = true;
            }
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < columns; j++) {
                if (obstacleGrid[i][j] == 0) {
                    obstacleGrid[i][j] = obstacleGrid[i - 1][j] +
                                         obstacleGrid[i][j - 1];
                } else {
                    obstacleGrid[i][j] = 0;
                }
            }
        }
        return obstacleGrid[rows - 1][columns - 1];
    }
}
```

47.Knight Movement in a Chess Board:

Given any source point, (C, D) and destination point, (E, F) on a chessboard, we need to find whether Knight can move to the destination or not.

```
public class Solution {
     public int knight(int N, int M, int x1, int y1, int x2, int y2) {
        // The 8 positions a knight could move to for the current position.
        int[] dx = {-1, -2, -1, -2, 1, 2, 1, 2};
        int[] dy = {-2, -1, 2, 1, -2, -1, 2, 1};

        boolean[][] isVisited = new boolean[N+1][M+1];
        Queue<Coordinate> queue = new LinkedList<Coordinate>();
        queue.add(new Coordinate(x1, y1));
        isVisited[x1][y1] = true;
        int moveCount = 0;
        // BFS to find the minimum number of steps.
        while (!queue.isEmpty()) {
            int nodesAtCurrentBreadth = queue.size();
            // Iterate over the coordinates at current breadth, as
            // moveCount would be increased by 1 per breadth level.
            for (int count = 0; count < nodesAtCurrentBreadth; count++) {
                Coordinate currPos = queue.remove();
                if (currPos.x == x2 && currPos.y == y2) {
                    return moveCount;
                }
                for (int i = 0; i < dx.length; i++) {
                  if (isValid(currPos.x + dx[i], currPos.y + dy[i], N, M)
                  && isVisited[currPos.x + dx[i]][currPos.y + dy[i]]==
                  false){
                    queue.add(
                     new Coordinate(currPos.x + dx[i],currPos.y + dy[i]));
                     isVisited[currPos.x + dx[i]][currPos.y + dy[i]] = true;
                    }
                }
            }
            moveCount++;
        }
        return -1;
    }
    private boolean isValid(int x, int y, int N, int M) {
        if (x <= 0 || y <= 0 || x > N || y > M) {
            return false;
        }
        return true;
    }
 }
```

```
class Coordinate {
    int x, y;
    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

48.Peak Index of a Mountain Array

```
public int peakIndexInMountainArray(int[] A) {
    int[] nums = A;
    if (nums == null || nums.length == 0) return -1;
    int low = 0, high = nums.length - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] > nums[mid + 1]) high = mid;
        else low = (mid + 1);
    }
    return low;
}
```

49. Rotate Image

```
public void rotate(int[][] matrix) {
    transpose(matrix);
    reverse(matrix);
}
private void transpose(int[][] matrix) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = (i + 1); j < matrix[0].length; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}
private void reverse(int[][] matrix) {
    for (int[] row: matrix) {
        for (int i = 0, j = row.length - 1; i < j; i++, j--) {
            int temp = row[j];
            row[j] = row[i];
            row[i] = temp;
        }
    }
}
```

50. Rotate Array

```
public void rotate(int[] nums, int k) {
    int n = nums.length;
    k = k % n;
    reverse(nums, 0, n - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, n - 1);
}
private void reverse(int[] nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;
        end--;
    }
}
```

51. Isomorphic String:

```
public boolean isIsomorphic(String s, String t) {
    if (s == null && t == null) return true;
    if (s == null || t == null) return false;
    if (s.length() != t.length()) return false;
    Map<Character,Character> mapping = new HashMap<>();
    Set<Character> uniqueT = new HashSet<>();
    for (int i = 0; i < s.length(); i++) {
        char fromS = s.charAt(i);
        char fromT = t.charAt(i);
        Character map = mapping.get(fromS);
        if (map == null) {
            if (uniqueT.contains(fromT)) return false;
            mapping.put(fromS, fromT);
            uniqueT.add(fromT);
        } else {
            if (map.equals(fromT)) continue;
            else return false;
        }
    }
    return true;
}
```

52. First unique Character of a String:

```
public int firstUniqChar(String s) {
    if (s == null || s.length() == 0) return -1;
    char[] arr = s.toCharArray();
    Map<Character,Integer> map = new LinkedHashMap<>();
    for (char c: arr) {
        int count = map.getOrDefault(c, 0);
        map.put(c, count + 1);
    }
    for (int i = 0; i < arr.length; i++) {
        int count = map.get(arr[i]);
        if (count == 1) return i;
    }
    return -1;
}
```

53. Product of array except self:

```
public int[] productExceptSelf(int[] nums) {
    int len = nums.length;
    int[] L = new int[len];
    int[] R = new int[len];

    L[0] = 1;
    for (int i = 1; i < len; i++) {
        L[i] = L[i - 1] * nums[i - 1];
    }
    R[len - 1] = 1;
    for (int i = (len - 2); i >= 0; i--) {
        R[i] = R[i + 1] * nums[i + 1];
    }
    for (int i = 0; i < len; i++) {
        L[i] = L[i] * R[i];
    }
    return L;
}
```

54. Container with most water:

```
public int maxArea(int[] height) {
  if(height == null || height.length == 0) return 0;
  int maxArea = 0;
  int l = 0, r = height.length -1;
  while(l < r){
      int h = Math.min(height[l],height[r]);
      maxArea = Math.max(maxArea, h * (r-l));
      if(height[l]> height[r]) r--;
      else l++;
  }
  return maxArea;
}
```

55. Word Search:

```
int rows = 0;
int columns = 0;
boolean[][] visited = null;

public boolean exist(char[][] board, String word) {
    if (board == null || board.length == 0 || board[0].length == 0
     || word == null || word.length() == 0) {
        return false;
    }
    rows = board.length;
    columns = board[0].length;
    visited = new boolean[rows][columns];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            if (board[i][j] == word.charAt(0)) {
                if (canFormWord(board, i, j, word, 0)) {
                    return true;
                }
            }
        }
    }
    return false;
}


private boolean canFormWord(char[][] board,int i,int j,String word,int len)
{
    if (word.length() == len) return true;
    if (i < 0 || i >= rows || j < 0 || j >= columns
        || visited[i][j] || board[i][j] != word.charAt(len)) {
        return false;
    }
    visited[i][j] = true;
    if (canFormWord(board, i + 1, j, word, len + 1) ||
        canFormWord(board, i - 1, j, word, len + 1) ||
        canFormWord(board, i, j + 1, word, len + 1) ||
        canFormWord(board, i, j - 1, word, len + 1)) {
        return true;
    }
    visited[i][j] = false;
    return false;
}
```

56. Minimum number of meeting rooms required:

```java
class Meeting implements Comparable<Meeting> {
    int start,end;
    public Meeting(int s, int e) {
        this.start = s;
        this.end = e;
    }
    public int compareTo(Meeting m) {
        if (this.start == m.start) return (this.end - m.end);
        return this.start - m.start;
    }
}


class Solution {
    public int minMeetingRooms(int[][] intervals) {
        if (intervals == null || intervals.length == 0) return 0;
        List<Meeting> meetingList = new ArrayList<>();
        for (int[] m: intervals) {
            Meeting meeting = new Meeting(m[0], m[1]);
            meetingList.add(meeting);
        }
        Collections.sort(meetingList);
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>
            (new Comparator<Integer>() {
            public int compare(Integer i1, Integer i2) {
                return i1 - i2;
            }
        });
        for (Meeting current: meetingList) {
            if (minHeap.isEmpty()) minHeap.offer(current.end);
            else {
                if (current.start < minHeap.peek()) {
                    minHeap.offer(current.end);
                } else {
                    minHeap.poll();
                    minHeap.offer(current.end);
                }
            }
        }
        return minHeap.size();
    }
}
```

57. Implement Queue using Stacks:

```
class MyQueue {
    private Stack<Integer> enQueue = null;
    private Stack<Integer> deQueue = null;
    /** Initialize your data structure here. */
    public MyQueue() {
        enQueue = new Stack<>();
        deQueue = new Stack<>();
    }

    /** Push element x to the back of the queue. */
    public void push(int x) {
        enQueue.push(x);
    }

    /** Removes from front of the queue and returns that element.*/
    public int pop() {
        if (!deQueue.isEmpty()) {
            return deQueue.pop();
        }
        fillDeQueue();
        return deQueue.isEmpty() ? -1 : deQueue.pop();
    }

    /** Get the front element. */
    public int peek() {
        if (!deQueue.isEmpty()) {
            return deQueue.peek();
        }
        fillDeQueue();
        return deQueue.isEmpty() ? -1 : deQueue.peek();
    }
    private void fillDeQueue() {
        while (!enQueue.isEmpty()) {
            deQueue.push(enQueue.pop());
        }
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return enQueue.isEmpty() && deQueue.isEmpty();
    }
}
```

58. Interleaving Strings:

```
public boolean isInterleave(String s1, String s2, String s3) {

    if (s3.length() != s1.length() + s2.length())
      return false;
    boolean[][] dp = new boolean[s1.length() + 1][s2.length() + 1];
    for (int i = 0; i <= s1.length(); i++) {
        for (int j = 0; j <= s2.length(); j++) {
            if (i == 0 && j == 0) dp[i][j] = true;
            else if (i == 0) {
              dp[i][j] = dp[i][j - 1] && s2.charAt(j-1) == s3.charAt(j-1);
            } else if (j == 0) {
              dp[i][j] = dp[i - 1][j] && s1.charAt(i-1) == s3.charAt(i-1);
            } else {
                dp[i][j] =
                (dp[i-1][j] && s1.charAt(i-1) == s3.charAt(i+j-1))
                ||
                (dp[i][j-1] && s2.charAt(j-1) == s3.charAt(i+j-1));
            }
        }
    }
    return dp[s1.length()][s2.length()];
}
```

59. Longest Substring with At Most K Distinct Characters:

```
public int lengthOfLongestSubstringKDistinct(String s, int k) {
    if (s == null || s.length() == 0) return 0;
    if (k == 0) return 0;
    Map<Character,Integer> map = new HashMap<>();
    int l = 0, r = 0, max = 1;
    while (r < s.length()) {
        char cur = s.charAt(r);
        map.put(cur, r);
        r++;
        if (map.size() > k) {
            int indexToDelete = Collections.min(map.values());
            char charToDelete = s.charAt(indexToDelete);
            map.remove(charToDelete);
            l = indexToDelete + 1;

        }
        max = Math.max(max, r - l);
    }
    return max;
}
```

60. Maximum Product Subarray:

```
public int maxProduct(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    if (nums.length == 1) return nums[0];
    int maxEnding = 0, minEnding = 0, maxSoFar = 0;
    for (int num: nums) {
       int temp = maxEnding;
       maxEnding = Math.max(num,Math.max(num * maxEnding,num * minEnding));
       minEnding = Math.min(num, Math.min(num * temp, num * minEnding));
       maxSoFar = Math.max(maxEnding, maxSoFar);
    }
    return maxSoFar;
}
```

61. First missing positive in an array:

```
class Solution {

    public int firstMissingPositive(int[] nums) {

        int n = nums.length;
        boolean is_1_present = false;

        for (int i = 0; i < n; i++) {
            if (nums[i] == 1) {
                is_1_present = true;
                break;
            }
        }

        if (!is_1_present) return 1;
        if (n == 1) return 2;

        for (int i = 0; i < n; i++) {
            if (nums[i] <= 0 || nums[i] > n) {
                nums[i] = 1;
            }
        }

        for (int i = 0; i < n; i++) {
            int a = Math.abs(nums[i]);
            if (n == a) nums[0] = -Math.abs(nums[0]);
            else nums[a] = -Math.abs(nums[a]);
        }

        for (int i = 1; i < n; i++) {
            if (nums[i] > 0) return i;
        }

        if (nums[0] > 0) return n;
        else return n + 1;
    }
}
```

62. Minimum window Substring:

```java
public static String minWindow(String s, String t) {
      if ((s == null && t == null) || (s.length() < t.length())) return "";

      Map<Character,Integer> dictT = new HashMap<>();
      for (int i = 0; i < t.length(); i++) {
        char ch = t.charAt(i);
        int count = dictT.getOrDefault(ch, 0);
        dictT.put(ch, (count + 1));
      }
      int required = dictT.size();
      int formed = 0, l = 0, r = 0;
      int[] ans = {-1,  0, 0};
      Map<Character,Integer> currentWindow = new HashMap<>();

      while (r < s.length()) {
            char ch = s.charAt(r);
            int count = currentWindow.getOrDefault(ch, 0);
            currentWindow.put(ch, (count + 1));
            if (dictT.containsKey(ch) && dictT.get(ch).intValue() ==
                  currentWindow.get(ch).intValue()) {
                  formed++;
            }
            while (l <= r && formed == required) {
                  ch = s.charAt(l);
                  if (ans[0] == -1 || (r - l + 1) < ans[0]) {
                        ans[0] = (r - l + 1);
                        ans[1] = l;
                        ans[2] = r;
                  }
                  count = currentWindow.get(ch);
                  currentWindow.put(ch, (count - 1));
                  if (dictT.containsKey(ch) &&
                      dictT.get(ch).intValue() >
                        currentWindow.get(ch).intValue()) {
                        formed--;
                  }
                  l++;
            }
            r++;
      }
      return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);
}
```

63. K closest points to origin:

```java
class Point implements Comparable<Point> {
    int x,y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public int compareTo(Point point) {
        return ((x * x) + (y * y)) -
             ((point.x * point.x) + (point.y * point.y));
    }
}
class MaxHeap extends PriorityQueue<Point> {
    public MaxHeap(int size) {
        super(size, new Comparator<Point>() {
            public int compare(Point o1, Point o2) {
                return o2.compareTo(o1);
            }
        });
    }
}

class Solution {
    public int[][] kClosest(int[][] points, int K) {
        MaxHeap maxHeap = new MaxHeap(K);
        for (int[] point: points) {
            Point currentPoint = new Point(point[0], point[1]);
            if (maxHeap.size() < K) {
                maxHeap.add(currentPoint);
            } else {
                Point top = maxHeap.peek();
                if (currentPoint.compareTo(top) < 0) {
                    maxHeap.add(currentPoint);
                    maxHeap.remove();
                }
            }
        }
        int[][] result = new int[K][2];
        int i = 0;
        for (Point point: maxHeap) {
            result[i][0] = point.x;
            result[i][1] = point.y;
            i++;
        }
        return result;
    }
}
```

64. Insert Delete and Get Random in O(1):

```java
class RandomizedSet {
    private Random random = null;
    private Map<Integer,Integer> map = null;
    private List<Integer> valueList = null;
    public RandomizedSet() {
        random = new Random();
        map = new HashMap<>();
        valueList = new ArrayList<>();
    }
    /** Inserts and returns true if the set did not have the value */
    public boolean insert(int val) {
        if(map.containsKey(val)){
            return false;
        }
        valueList.add(val);
        map.put(val,valueList.size()-1);
        return true;
    }
    /** Removes and returns true if the set contained have the value. */
    public boolean remove(int val) {
        if(!map.containsKey(val))  return false;
        int lastIndex = valueList.size() - 1;
        int lastElement = valueList.get(lastIndex);
        if(lastElement != val){
            int index = map.get(val);
            valueList.set(index,lastElement);
            map.put(lastElement,index);
        }
        map.remove(val);
        valueList.remove(lastIndex);
        return true;

    }
    /** Get a random element from the set. */
    public int getRandom() {
        return valueList.get(random.nextInt(valueList.size()));
    }
}
```

65. Gas Station:

```
public int canCompleteCircuit(int[] gas, int[] cost) {
    int len = gas.length;
    int totalTank = 0,currentTank = 0, startStation = 0;
    for (int i = 0; i < len; i++) {
        totalTank = totalTank + (gas[i] - cost[i]);
        currentTank = currentTank + (gas[i] - cost[i]);
        if (currentTank < 0) {
            currentTank = 0;
            startStation = i + 1;
        }
    }
    return totalTank >= 0 ? startStation : -1;
}
```

66. Sorted array with each value occurring twice except one:

```
public int singleNonDuplicate(int[] nums) {
    int low = 0;
    int high = nums.length - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (mid % 2 == 1) mid--;
        if (nums[mid] != nums[mid + 1]) {
            high = mid;
        } else {
            low = mid + 2;
        }
    }
    return nums[low];
}
```

67. Next Greater Element 1:

```
public int[] nextGreaterElement(int[] nums1, int[] nums2) {
    Stack<Integer> stack = new Stack<>();
    Map<Integer,Integer> map = new HashMap<>();
    int[] result = new int[nums1.length];
    for (int i = 0; i < nums2.length; i++) {
        if (stack.isEmpty()) {
            stack.push(nums2[i]);
            continue;
        }
        while (!stack.isEmpty() && nums2[i] > stack.peek()) {
            int val = stack.pop();
            map.put(val, nums2[i]);
        }
        stack.push(nums2[i]);
    }
    while (!stack.isEmpty()) map.put(stack.pop(), -1);
    for (int i = 0; i < nums1.length; i++) {
        result[i] = map.get(nums1[i]);
    }
    return result;
}
```

68. Reverse Integer:

```
public int reverse(int x) {
    if (x == 0) return 0;
    boolean isNegative = x < 0 ? true : false;
    if (isNegative) x = -x;
    int reverse = 0,remainder = 0;
    while (x > 0) {
        remainder = x % 10;
        x = x / 10;
        if (reverse > Integer.MAX_VALUE / 10) return 0;
        reverse = reverse * 10 + remainder;
    }
    if (isNegative)
        reverse = -reverse;
    return reverse;
}
```

69. Minimum area Rectangle:

```java
public int minAreaRect(int[][] points) {
    if (points == null || points.length == 0) return 0;
    Map<Integer,List<Integer>> yListByX = new TreeMap<>();
    for (int[] point: points) {
        int x = point[0];
        int y = point[1];
        List<Integer> yList = yListByX.get(x);
        if (yList == null) {
            yList = new ArrayList < > ();
            yListByX.put(x, yList);
        }
        yList.add(y);
    }

    int minArea = Integer.MAX_VALUE;
    Map<String,Integer> seenX = new HashMap<>();
    for (int x: yListByX.keySet()) {
        List<Integer> yList = yListByX.get(x);
        Collections.sort(yList);
        for (int i = 0; i < yList.size(); i++) {
            int y1 = yList.get(i);
            for (int j = (i + 1); j < yList.size(); j++) {
                int y2 = yList.get(j);
                String key = y1 + "_" + y2;
                if (seenX.containsKey(key)) {
                    int prevX = seenX.get(key);
                    int currentArea = (x - prevX) * (y2 - y1);
                    minArea = Math.min(minArea, currentArea);
                }
                seenX.put(key, x);
            }
        }
    }
    return minArea < Integer.MAX_VALUE ? minArea : 0;
}
```

70.Reverse word in String:

```java
public String reverseWords(String s) {

    if (s == null || s.length() == 0) return "";
    int left = 0, right = s.length() - 1;

    while (left <= right) {
        if (s.charAt(left) == ' ') left++;
        else break;
    }

    while (left <= right) {
        if (s.charAt(right) == ' ') right--;
        else break;
    }

    Stack<String> wordStack = new Stack<String>();
    StringBuffer buffer = new StringBuffer();
    while (left <= right) {
        char current = s.charAt(left);
        if (current != ' ') buffer.append(current);
        else if (buffer.length() > 0) {
            wordStack.push(buffer.toString());
            buffer = new StringBuffer();
        }
        left++;
    }

    if (buffer.length() > 0) {
        wordStack.push(buffer.toString());
    }
    buffer = new StringBuffer();
    while (!wordStack.isEmpty()) {
        buffer.append(wordStack.pop());
        if (wordStack.size() > 0) buffer.append(" ");
    }
    return buffer.toString();
}
```

71. Find First and Last Position of Element in Sorted Array:

```java
// returns the leftmost (or rightmost) index at which `target` should be
// inserted in sorted array `nums` via binary search.

private int extremeInsertionIndex(int[] nums, int target,boolean left) {
    int lo = 0;
    int hi = nums.length;

    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (nums[mid] > target || (left && target == nums[mid])) {
            hi = mid;
        } else {
            lo = mid + 1;
        }
    }

    return lo;
}

public int[] searchRange(int[] nums, int target) {
    int[] targetRange = {
        -1,
        -1
    };
    int leftIdx = extremeInsertionIndex(nums, target, true);

    // assert that `leftIdx` is within the array bounds and that `target`
    // is actually in `nums`.
    if (leftIdx == nums.length || nums[leftIdx] != target) {
        return targetRange;
    }
    targetRange[0] = leftIdx;
    targetRange[1] = extremeInsertionIndex(nums, target, false) - 1;
    return targetRange;
}
```

72. Decode String:

```java
public String decodeString(String s) {
    String EMPTY_STRING = "";
    if (s == null || s.length() == 0) return EMPTY_STRING;
    Stack<String> letterStack = new Stack<>();
    Stack<Integer> frequencyStack = new Stack<>();
    int length = s.length();
    String currentFrequency = "";
    for (int i = 0; i < length; i++) {
        char current = s.charAt(i);
        if (Character.isDigit(current)) {
            currentFrequency = currentFrequency + current;
        } else {
            if (currentFrequency.length() > 0) {
                int frequency = Integer.parseInt(currentFrequency);
                frequencyStack.push(frequency);
                currentFrequency = "";
            }
            if (current != ']') {
                letterStack.push(String.valueOf(current));
            } else {
                String str = EMPTY_STRING;
                while (!letterStack.peek().equals("[")) {
                    str = letterStack.pop() + str;
                }
                // Lastly remove the corresponding "["
                letterStack.pop();
                int frequency = frequencyStack.size() > 0 ?
                                frequencyStack.pop() : 1;
                StringBuffer sb = new StringBuffer();
                while (frequency > 0) {
                    sb.append(str);
                    frequency--;
                }
                letterStack.push(sb.toString());
            }
        }
    }
    String result = EMPTY_STRING;
    while (!letterStack.isEmpty()) {
        result = letterStack.pop() + result;
    }
    return result;
}
```

73. Next closest time:

```java
public String nextClosestTime(String time) {

    int hours = Integer.parseInt(time.substring(0, 2));
    int minutes = Integer.parseInt(time.substring(3));
    int timeInMinutes = (hours * 60) + minutes;
    Set<Integer> allowed = new HashSet<>();
    for (int i = 0; i < time.length(); i++) {
        char current = time.charAt(i);
        if (current == ':') continue;
        int digit = current - '0';
        allowed.add(digit);
    }
    StringBuilder result = new StringBuilder();
    while (true) {
        timeInMinutes = (timeInMinutes + 1) % (24 * 60);
        int[] digits = {
            (timeInMinutes / 60 / 10),
            (timeInMinutes / 60 % 10),
            (timeInMinutes % 60 / 10),
            (timeInMinutes % 60 % 10)
        };
        if (isAllowed(allowed, digits)) {
            result.append(digits[0]);
            result.append(digits[1]);
            result.append(":");
            result.append(digits[2]);
            result.append(digits[3]);
            break;
        }
    }
    return result.toString();
}

private boolean isAllowed(Set<Integer> allowed, int[] digits) {
    for (int cur: digits) {
        if (!allowed.contains(cur)) {
            return false;
        }
    }
    return true;
}
```

74.Palindrome Linked List:

```
public boolean isPalindrome(ListNode head) {
    boolean isPalindrome = true;
    if (head == null) {
        return isPalindrome;
    }
    // Find the middle node of the linked list
    ListNode fastPtr = head, slowPtr = head;
    while (fastPtr != null && fastPtr.next != null) {
        fastPtr = fastPtr.next.next;
        slowPtr = slowPtr.next;
    }
    // Reverse the linked list from Middle
    ListNode tempHead = reverseList(slowPtr);
    // preserver to reconstruct the list
    ListNode preserveMiddle = tempHead;
    // Set fastPtr to head
    fastPtr = head;
    /**
     * Compare nodes from head and reverse nodes from
     * middle to check if the linked list is palindrome
     */
    while (tempHead != null) {
        if (fastPtr.val != tempHead.val) {
            isPalindrome = false;
            break;
        }
        fastPtr = fastPtr.next;
        tempHead = tempHead.next;
    }

    // reconstruct the LinkedList
    ListNode middle = reverseList(preserveMiddle);
    slowPtr = head;
    while (slowPtr != null) {
        slowPtr = slowPtr.next;
    }
    slowPtr = middle;
    return isPalindrome;
}
```

```java
private ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```

75. Implement strstr:

```java
public int strStr(String haystack, String needle) {

    if (needle == null || needle.length() == 0) {
        return 0;
    }

    if (haystack == null || haystack.length() == 0) {
        return -1;
    }
    int hLength = haystack.length();
    int nLength = needle.length();

    for (int i = 0; i < hLength; i++) {
        if ((i + nLength) > hLength) {
            break;
        }

        for (int j = 0; j < nLength; j++) {
            char curH = haystack.charAt(i + j);
            char curL = needle.charAt(j);
            if (curH != curL) break;
            if (j == nLength - 1) {
                return i;
            }
        }
    }
    return -1;
}
```

76. Vertical Order Traversal of a Binary Tree:

```
class Loc implements Comparable<Loc>{
    int x, y , val;
    public Loc(int x, int y, int val){
        this.x = x;
        this.y = y;
        this.val = val;
    }
    public int compareTo(Loc loc){
        if(this.x != loc.x) return Integer.compare(this.x, loc.x);
        if(this.y != loc.y) return Integer.compare(this.y, loc.y);
        return Integer.compare(this.val, loc.val);
    }
}

public List<List<Integer>> verticalTraversal(TreeNode root) {
      List<List<Integer>> result = new ArrayList<>();
      if(root == null) return result;
      List<Loc> locationList = new ArrayList<>();
      dfs(locationList,root,0,0);
      Collections.sort(locationList);
      Map<Integer,List<Integer>> valuesByX = new TreeMap<>();
      for(Loc loc : locationList){
            List<Integer> valList = valuesByX.get(loc.x);
            if(valList == null){
                  valList = new ArrayList<>();
                  valuesByX.put(loc.x,valList);
            }
            valList.add(loc.val);
      }
      return valuesByX.values().stream().collect(Collectors.toList());
}

private void dfs(List<Loc> locationList, TreeNode node, int x, int y){
      if(node == null) return;
      Loc loc = new Loc(x,y,node.val);
      locationList.add(loc);
      dfs(locationList,node.left,(x-1),(y+1));
      dfs(locationList,node.right,(x+1),(y+1));
}
```

77.Jump Game 1:

```
public boolean canJump(int[] nums) {
    if (nums == null || nums.length == 0)
        return false;
    int curEnding = 0, maxEnding = 0;
    for (int i = 0; i < nums.length; i++) {
        maxEnding = Math.max(maxEnding, (i + nums[i]));
        if (i == curEnding) {
            curEnding = maxEnding;
            if (curEnding >= nums.length - 1) {
                return true;
            }
        }
    }
    return false;
}
```

78.Jump Game 2:

```
public int jump(int[] nums) {
    if (nums.length <= 1) {
        return 0;
    }
    int curEnding = 0, maxEnding = 0, jump = 0;
    int len = nums.length;
    for (int i = 0; i < len; i++) {
        maxEnding = Math.max(maxEnding, (i + nums[i]));
        if (i == curEnding) {
            jump++;
            curEnding = maxEnding;
            if (curEnding >= len - 1) break;
        }
    }
    return jump;
}
```

79. Search in a 2D Matrix 1:

```java
public boolean searchMatrix(int[][] matrix, int target) {
    int height = matrix.length;
    if (height == 0) return false;
    int width = matrix[0].length;

    int left = 0;
    int right = (height * width - 1);
    while (left <= right) {
        int pivot = left + (right - left) / 2;
        int pivotElement = matrix[pivot / width][pivot % width];
        if (target == pivotElement) {
            return true;
        } else {
            if (target > pivotElement) left = pivot + 1;
            else right = pivot - 1;
        }
    }
    return false;
}
```

80. Search in a 2D Matrix 2:

```java
public boolean searchMatrix(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
       return false;
    int rows = matrix.length;
    int columns = matrix[0].length;
    int i = rows - 1;
    int j = 0;
    while (i >= 0 && j <= columns - 1) {
        if (target == matrix[i][j]) return true;
        if (target > matrix[i][j]) j++;
        else i--;
    }
    return false;
}
```

81. Rotting Oranges:

```
public int orangesRotting(int[][] grid) {
      Queue<int[]> queue = new LinkedList();
      int rows = grid.length;
      int columns = grid[0].length;
      // All the rotten oranges
      for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
           if (grid[i][j] == 2) {
             queue.offer(new int[]{i, j});
           }
        }
      }

      int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
      int level = 0;
      while (!queue.isEmpty()) {
        level++;
        int size = queue.size();
        for (int i = 0; i < size; i++) {
          int[] rotten = queue.poll();
          for(int[] dir : directions){
             int x = rotten[0] + dir[0];
             int y = rotten[1] + dir[1];
             if(x >= rows || y >= columns || x < 0 || y < 0
                          || grid[x][y] != 1){
               continue;
             }
             queue.offer(new int[]{x,y});
             grid[x][y] = 2;
           }
        }
      }
      for(int[] row : grid){ // unreachable oranges
        for (int a : row){
          if(a == 1) return -1;
        }
      }
      return level == 0 ? 0 : level-1;
}
```

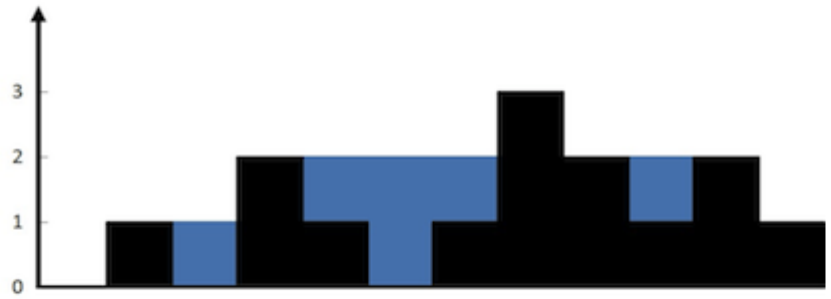
82. Word Break:

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
```

```
public boolean wordBreak(String s, List<String> wordDict) {
    if (s == null) return false;
    Set<String> dict = new HashSet<>(wordDict);
    Boolean[] memo = new Boolean[s.length()];
    return canForm(s, 0, dict, memo);
}

private boolean canForm(String s, int start, Set<String> dict,
                        Boolean[] memo) {
    if (s.length() == start) return true;
    if (memo[start] != null) return memo[start];

    for (int end = start + 1; end <= s.length(); end++) {
        if (dict.contains(s.substring(start, end))
                && canForm(s, end, dict, memo)) {
            memo[start] = true;
        }
    }
    if (memo[start] == null) memo[start] = false;
    return memo[start];
}
```

83. Trapping Rain Water:

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Example:
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

```java
public int trap(int[] height) {
    int left = 0, right = height.length - 1;
    int leftMax = 0, rightMax = 0, ans = 0;
    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] > leftMax) leftMax = height[left];
            else ans += leftMax - height[left];
            left++;
        } else {
            if (height[right] > rightMax) rightMax = height[right];
            else ans += rightMax - height[right];
            right--;
        }
    }
    return ans;
}
```

84.Concatenated Words:

```
Input:["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat",
 "ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
```

```
public static List<String> findAllConcatenatedWordsInADict(String[] words)
{

    List<String> result = new ArrayList<>();
    Set<String> preWords = new HashSet<>();
    Arrays.sort(words, new Comparator < String > () {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    });
    for (int i = 0; i < words.length; i++) {
        if (canForm(words[i], preWords)) {
            result.add(words[i]);
        }
        preWords.add(words[i]);
    }
    return result;
}

private static boolean canForm(String word, Set<String> dict) {
    if (dict.isEmpty()) return false;
    boolean[] dp = new boolean[word.length() + 1];
    dp[0] = true;
    for (int i = 1; i <= word.length(); i++) {
        for (int j = 0; j < i; j++) {
            if (!dp[j]) continue;
            if (dict.contains(word.substring(j, i))) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[word.length()];
}
```

85. Sum Root To Leaf Numbers:

```java
public int sumNumbers(TreeNode root) {
    if (root == null) return 0;
    List<String> pathList = new ArrayList<>();
    Queue<TreeNode> nodeQueue = new LinkedList<>();
    Queue<String> pathQueue = new LinkedList<>();
    nodeQueue.offer(root);
    pathQueue.offer(String.valueOf(root.val));
    while (!nodeQueue.isEmpty()) {
        TreeNode node = nodeQueue.poll();
        String value = pathQueue.poll();
        if (node.left == null && node.right == null) {
            pathList.add(value);
        }
        if (node.left != null) {
            nodeQueue.offer(node.left);
            pathQueue.offer(value + node.left.val);
        }
        if (node.right != null) {
            nodeQueue.offer(node.right);
            pathQueue.offer(value + node.right.val);
        }
    }
    int sum = 0;
    for (String num: pathList){
        sum += Integer.parseInt(num);
    }
    return sum;
}
```

86.Top K Frequent Elements:

```
public List<Integer> topKFrequent(int[] nums, int k) {

    List<Integer> result = new ArrayList<>();
    Map<Integer,Integer> map = new HashMap<>();
    for (int cur: nums) {
        int count = map.getOrDefault(cur, 0);
        count = count + 1;
        map.put(cur, count);
    }

    PriorityQueue<Map.Entry<Integer,Integer>> minHeap = new PriorityQueue<>
        (k, new Comparator<Map.Entry<Integer,Integer>> () {
            @Override
            public int compare(Map.Entry<Integer,Integer> entry_1,
               Map.Entry <Integer,Integer> entry_2) {
                return entry_1.getValue() - entry_2.getValue();
            }
        });

    for (Map.Entry<Integer,Integer> currentEntry: map.entrySet()) {
        if (minHeap.size() < k) {
          minHeap.add(currentEntry);
        } else {
          if (minHeap.peek().getValue() < currentEntry.getValue()) {
            minHeap.offer(currentEntry);
            minHeap.remove();
          }
        }
    }

    for(Map.Entry<Integer,Integer> entry: minHeap){
        result.add(entry.getKey());
    }
    return result;
}
```

87.Find Minimum in Rotated Sorted Array:

```java
public int findMin(int[] nums) {
    if (nums == null || nums.length == 0) return -1;
    int len = nums.length - 1;
    if (nums[0] <= nums[len]) return nums[0];
    int low = 0;
    int high = len;
    while (high > low) {
        int mid = low + (high - low) / 2;
        if (nums[mid] > nums[mid + 1]) return nums[mid + 1];
        else if (nums[mid] > nums[low]) low = mid + 1;
        else high = mid;
    }
    return -1;
}
```

88.Redundant Braces:

```java
//A = "((a + b))" redundant ||  A = "(a + (a + b))" not redundant
public int braces(String a) {
    Stack<Character> stack = new Stack<>();
    boolean lastPopped = false;
    for (int i = 0; i < a.length(); i++) {
        if (a.charAt(i) == '(' || a.charAt(i) == '+' ||
            a.charAt(i) == '-' || a.charAt(i) == '*' ||
            a.charAt(i) == '/') {
            stack.push(a.charAt(i));
        } else if (a.charAt(i) == ')') {
            boolean didPopSymbol = false;
            while (stack.peek() != '(') {
                didPopSymbol = true;
                stack.pop();
            }
            if (!didPopSymbol) return 1;
            stack.pop();
        }
    }
    return 0;
}
```

89.Find Kth Largest element from an Array:

```
public int findKthLargest(int[] nums, int k) {

  Map<Integer,Integer> map = new TreeMap<>(new TreeMapComparator());
  if(nums == null || nums.length == 0) return 0;
  for(int i = 0 ; i < nums.length ; i++){
      int cur = nums[i];
      int count = map.getOrDefault(cur,0);
      map.put(cur,(count+1));
  }
  int kthLargest = 0;
  int index = 0;
  Set<Map.Entry<Integer,Integer>> entrySet = map.entrySet();
  for(Map.Entry<Integer,Integer> entry : entrySet){
      index = index + entry.getValue();
      if(index == k){
        kthLargest = entry.getKey();
        break;
      }
      if( k < index){
        kthLargest = entry.getKey();
        break;
      }
  }
  return kthLargest;
}
```

90.Minesweeper:

```java
class Solution {
    int[] dx = {-1, 0, 1, -1, 1, 0, 1, -1};
    int[] dy = {-1, 1, 1, 0, -1, -1, 0, 1};
    public char[][] updateBoard(char[][] board, int[] click) {
        int x = click[0], y = click[1];
        if (board[x][y] == 'M') {
            board[x][y] = 'X';
            return board;
        }
        dfs(board, x, y);
        return board;
    }
    private void dfs(char[][] board, int x, int y) {
        if (x < 0 || x >= board.length || y < 0 ||
            y >= board[0].length || board[x][y] != 'E') {
            return;
        }
        int num = getNumberOfAdjacentMines(board, x, y);
        if (num == 0) {
            board[x][y] = 'B';
            for (int i = 0; i < 8; i++) {
                int ax = x + dx[i];
                int ay = y + dy[i];
                dfs(board, ax, ay);
            }
        } else {
            board[x][y] = (char)('0' + num);;
        }
    }
    private int getNumberOfAdjacentMines(char[][] board, int x, int y) {
        int num = 0;
        for (int i = 0; i < 8; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];
            if (nx < 0 || nx >= board.length || ny < 0 ||
                ny >= board[0].length) continue;
            if (board[nx][ny] == 'M' || board[nx][ny] == 'X') {
                num++;
            }
        }
        return num;
    }
}
```

91. Reorder Data in Log Files:

```
public String[] reorderLogFiles(String[] logs) {
    if (logs == null || logs.length == 0) return null;
    List<String> letterLogs = new ArrayList<>();
    List<String> digitsLogs = new ArrayList<>();
    for (String logLine: logs) {
        String[] token = logLine.split(" ");
        if (Character.isDigit(token[1].charAt(0))) {
            digitsLogs.add(logLine);
        } else {
            letterLogs.add(logLine);
        }
    }
    Collections.sort(letterLogs, new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            String[] token1 = o1.split(" ");
            String[] token2 = o2.split(" ");
            int cmp = 0;
            int len = Math.min(token1.length, token2.length);
            for (int i = 1; i < len; i++) {
                cmp = token1[i].compareTo(token2[i]);
                if (cmp != 0) break;
            }
            if (cmp == 0)  cmp = token1[0].compareTo(token2[0]);
            return cmp;
        }
    });
    letterLogs.addAll(digitsLogs);
    return letterLogs.toArray(new String[letterLogs.size()]);
}
```

92. Best time to buy and sell stock:

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0) return 0;
    int minPrice = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int i = 0; i < prices.length; i++) {
        minPrice = Math.min(minPrice, prices[i]);
        maxProfit = Math.max(maxProfit, prices[i] - minPrice);
    }
    return maxProfit;
}
```

93. Best time to buy and sell stock II:

Input: [7,1,5,3,6,4]
Output: 7

Explanation:
Buy on day 2 (price = 1) and Sell on day 3 (price = 5).
Profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6),
Profit = 6-3 = 3.

```
public int maxProfit(int[] prices) {
      int maxprofit = 0;
      for (int i = 1; i < prices.length; i++) {
         if (prices[i] > prices[i - 1])
           maxprofit += prices[i] - prices[i - 1];
      }
      return maxprofit;
}
```

94.Next Greater prime palindrome:

```java
public int primePalindrome(int N) {
    if (N == 1) return 2;
    while (true) {
        if (N == reverse(N) && isPrime(N)) return N;
        N++;
        if (10_000_000 < N && N < 100_000_000) N = 100_000_000;
    }
}
public boolean isPrime(int N) {
    boolean flag = true;
    for (int i = 2; i <= N / 2; i++) {
        if (N % i == 0) {
            flag = false;
            break;
        }
    }
    return flag;
}
public int reverse(int N) {
    int ans = 0, remainder = 0;
    while (N > 0) {
        remainder = N % 10;
        ans = 10 * ans + remainder;
        N /= 10;
    }
    return ans;
}
```

95. Most frequent Stack:

```java
int maxFrequency = 0;
Map<Integer,Stack<Integer>> elementListByFrequency;
Map<Integer,Integer> frequencyByElement;

public FreqStack() {
    elementListByFrequency = new HashMap<>();
    frequencyByElement = new HashMap<>();
}

public void push(int x) {
    int frequency = frequencyByElement.getOrDefault(x, 0);
    frequency = frequency + 1;
    frequencyByElement.put(x, frequency);
    maxFrequency = Math.max(maxFrequency, frequency);
    Stack<Integer> elementList = elementListByFrequency.get(frequency);
    if (elementList == null) {
        elementList = new Stack<>();
        elementListByFrequency.put(frequency, elementList);
    }
    elementList.push(x);
}

public int pop() {
    Stack<Integer> elementList = elementListByFrequency.get(maxFrequency);
    int x = elementList.pop();
    if (elementList.isEmpty()) {
        maxFrequency--;
    }
    int frequency = frequencyByElement.get(x);
    if (frequency == 1) {
        frequencyByElement.remove(x);
    } else {
        frequencyByElement.put(x, frequency - 1);
    }
    return x;
}
```

96. Top K Frequent Words:

```java
public List<String> topKFrequent(String[] words, int k) {
    List<String> result = new ArrayList<>();
    if (words == null || words.length == 0) return result;
    Map<String,Integer> map = new HashMap<>();
    for (String word: words) {
        int count = map.getOrDefault(word, 0);
        map.put(word, (count + 1));
    }

    PriorityQueue<Map.Entry<String,Integer>> minHeap = new PriorityQueue<> (k,
        new Comparator<Map.Entry<String,Integer>> () {
            public int compare(Map.Entry<String,Integer> entry_1,
                               Map.Entry<String,Integer> entry_2) {
             if (entry_1.getValue().intValue() == entry_2.getValue().intValue()){
               return entry_2.getKey().compareTo(entry_1.getKey());
             }
             return Integer.compare(entry_1.getValue(), entry_2.getValue());
            }
        });

    Set<Map.Entry<String,Integer>> entrySet = map.entrySet();
    for (Map.Entry<String,Integer> entry: entrySet) {
        minHeap.offer(entry);
        if (minHeap.size() > k) minHeap.poll();
    }

    while (minHeap.size() > 0){
      result.add(minHeap.poll().getKey());
    }
    Collections.reverse(result);
    return result;
}
```

97. Most Common Words:

```
public String mostCommonWord(String paragraph, String[] banned) {
      Map<String,Integer> map = new HashMap<>();
      for(String ban : banned) map.put(ban,-1);
      String[] tokens = paragraph.replaceAll("\\p{P}", " ")
                        .toLowerCase().split(" ");
      for(String token : tokens){
        if(token.length() == 0) continue;
        int count = map.getOrDefault(token,0);
        if(count >= 0){
          count = count + 1;
          map.put(token,count);
        }
      }
      List<Map.Entry<String,Integer>> entryList= new
                                             ArrayList<>(map.entrySet());
      Collections.sort(entryList,new
                     Comparator<Map.Entry<String,Integer>>(){
            public int compare(Map.Entry<String,Integer> e1,
                               Map.Entry<String,Integer> e2){
                  return Integer.compare(e2.getValue() , e1.getValue());
            }
      });
      return entryList.get(0).getKey();
}
```

98.3 SUM:

```java
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums == null || nums.length == 0) return result;
    for (int i = 0; i < nums.length; i++) {
        int a = nums[i];
        Set<Integer> set = new HashSet<>();
        for (int j = (i + 1); j < nums.length; j++) {
            int b = nums[j];
            int c = -(a + b);
            if (set.contains(c)) {
                List < Integer > list = new ArrayList < > ();
                list.add(a);
                list.add(b);
                list.add(c);
                Collections.sort(list);
                if (!result.contains(list)) {
                    result.add(list);
                }
            } else {
                set.add(b);
            }
        }
    }
    return result;
}
```

99. Merge Intervals:

```java
class Interval implements Comparable<Interval> {
    int start, end;
    public Interval(int s, int e) {
        this.start = s;
        this.end = e;
    }
    public int compareTo(Interval in ) {
        if (this.start == in .start) return (this.end - in.end);
        return (this.start - in.start);
    }
}
class Solution {
    public int[][] merge(int[][] intervals) {
        List<Interval> intervalList = new ArrayList<>();
        for (int[] in: intervals) {
            Interval interval = new Interval( in [0], in [1]);
            intervalList.add(interval);
        }
        Collections.sort(intervalList);
        LinkedList<Interval> merged = new LinkedList<>();
        for (Interval current: intervalList) {
            if (merged.isEmpty()) {
                merged.add(current);
            } else {
                Interval last = merged.getLast();
                if (last.end < current.start) {
                    merged.add(current);
                } else {
                    last.end = Math.max(last.end, current.end);
                }
            }
        }
        int[][] result = new int[merged.size()][2];
        int i = 0;
        for (Interval interval: merged) {
            result[i][0] = interval.start;
            result[i][1] = interval.end;
            i++;
        }
        return result;
    }
}
```

100. Search in Rotated Sorted Array:

```java
public int search(int[] nums, int target) {
    if (nums == null || nums.length == 0) return -1;
    int rotation = findRotation(nums);
    int len = nums.length - 1;
    if (rotation == 0)
        return search(nums, 0, len, target);
    if (target >= nums[0])
        return search(nums, 0, rotation - 1, target);
    else return search(nums, rotation, len, target);
}

private int findRotation(int[] nums) {
    int low = 0;
    int high = nums.length - 1;
    if (nums[high] > nums[low]) return 0;
    while (high > low) {
        int mid = low + (high - low) / 2;
        if (nums[mid] > nums[mid + 1]) return (mid + 1);
        else if (nums[mid] > nums[low]) low = (mid + 1);
        else high = mid;
    }
    return 0;
}

private int search(int[] nums, int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

101. Min Cost to Climb Stairs:

```
public int minCostClimbingStairs(int[] cost) {
    if (cost == null || cost.length == 0) {
        return 0;
    }
    int len = cost.length;
    if (len == 1) return cost[0];
    if (len == 2) return Math.min(cost[0], cost[1]);
    int[] dp = new int[len + 1];
    dp[0] = cost[0];
    dp[1] = cost[1];
    for (int i = 2; i < cost.length; i++) {
        dp[i] = cost[i] + Math.min(dp[i - 1], dp[i - 2]);
    }
    return Math.min(dp[len - 2], dp[len - 1]);
}
```

102. Moving Average from Data Stream:

```
class MovingAverage {
    private int size;
    private int sum = 0;
    private LinkedList<Integer> data;
    public MovingAverage(int size) {
        this.size = size;
        data = new LinkedList<>();
    }
    public double next(int val) {
        sum += val;
        data.add(val);
        if (data.size() > size) {
            int first = data.removeFirst();
            sum -= first;
        }
        return sum * 1.0 / data.size();
    }
}
```

103. Longest Increasing Path in a Matrix:

```
int ans = 0;
int[][] cache = null;
int[][] directions = {{1,0},{0,1},{-1,0},{0,-1}};
int rows = 0, columns = 0;

public int longestIncreasingPath(int[][] matrix) {
    if (matrix == null || matrix.length == 0 ||
        matrix[0].length == 0) {
        return 0;
    }
    rows = matrix.length;
    columns = matrix[0].length;
    cache = new int[rows][columns];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            ans = Math.max(ans, DFS(matrix, i, j));
        }
    }
    return ans;
}

private int DFS(int[][] matrix, int i, int j) {
    if (cache[i][j] > 0) return cache[i][j];
    for (int[] dir: directions) {
        int x = i + dir[0];
        int y = j + dir[1];
        if (x >= 0 && x < rows && y >= 0 && y < columns &&
            matrix[x][y] > matrix[i][j]) {
            cache[i][j] = Math.max(cache[i][j], DFS(matrix, x, y));
        }
    }
    return ++cache[i][j];
}
```

104. Maximum points in a line:

```
public int maxPoints(int[][] points) {
    if (points == null || points.length == 0) {
        return 0;
    }
    if (points.length < 3) {
        return points.length;
    }
    Map<Integer,Map<Integer,Integer>> pointCountBySlopeYBySlopeX = new
                                 HashMap<>();
    int result = 0;
    for (int i = 0; i < points.length; i++) {
        //Reset map when considering for every point
        pointCountBySlopeYBySlopeX.clear();
        int x_1 = points[i][0];
        int y_1 = points[i][1];

        //Reset overlap and max declaration to 0 for every fresh point
        int max = 0;
        int overlap = 0;

        for (int j = i + 1; j < points.length; j++) {

            int x_2 = points[j][0];
            int y_2 = points[j][1];

            //compute the slope numerator and denominator
            int slope_x = (x_2 - x_1);
            int slope_y = (y_2 - y_1);

           /***************************************************************
             If both x and y == 0 then both the points in consideration are
             The same.These can be duplicate or overlapping points.So
             Increment overlap and continue the loop since there is no
             slope For overlapping or duplicate points
           ***************************************************************/
            if (slope_x == 0 && slope_y == 0) {
                overlap ++;
                continue;
            }

            // Compute the gcd of x and y; so that 2/4 is considered the
            // same as 1/2
            int gcd = generateGCD(slope_x, slope_y);
            if (gcd != 0) {
                slope_x = slope_x / gcd;
                slope_y = slope_y / gcd;
            }

            Map<Integer,Integer> pointCountBySlopeY =
                  pointCountBySlopeYBySlopeX.get(slope_x);
            if (pointCountBySlopeY == null) {
              pointCountBySlopeY = new HashMap<>();
              pointCountBySlopeYBySlopeX.put(slope_x, pointCountBySlopeY);
            }
            Integer count = pointCountBySlopeY.get(slope_y);
            if (count == null) {
                count = 1;
                pointCountBySlopeY.put(slope_y, count);
            } else {
                count = count + 1;
                pointCountBySlopeY.put(slope_y, count);
            }
            //Local max
            max = Math.max(max, count);
        }
        //global maximum
        result = Math.max(max + overlap + 1, result);
    }
    return result;
}

private int generateGCD(int a, int b) {
    if (b == 0) return a;
    else return generateGCD(b, a % b);
}
```

105. Sliding Window Maximum:

```
public int[] maxSlidingWindow(int[] nums, int k) {
    int len = nums.length;
    int[] result = new int[len - k + 1];
    if (nums.length == 0) return new int[0];

    Queue<Integer> queue = new PriorityQueue<Integer> (new
        Comparator<Integer>() {
        @Override
        public int compare(Integer i1, Integer i2) {
            return Integer.compare(i2, i1);
        }
    });

    for (int i = 0; i < k; i++) {
        queue.add(nums[i]);
    }
    result[0] = queue.peek();
    int j = 1;
    for (int i = k; i < len; i++) {
        queue.remove(nums[i - k]);
        queue.add(nums[i]);
        result[j++] = queue.peek();
    }
    return result;
}
```

106. Longest Palindromic Substring:

```
private int lo, maxLen;

public String longestPalindrome(String s) {
    int len = s.length();
    if (len < 2)
        return s;
    for (int i = 0; i < len - 1; i++) {
        //assume odd length, try to extend Palindrome as possible
        extendPalindrome(s, i, i);
        //assume even length.
        extendPalindrome(s, i, i + 1);
    }
    return s.substring(lo, lo + maxLen);
}

private void extendPalindrome(String s, int left, int right) {

    while (left >= 0 && right < s.length() &&
        s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
    }

    if (maxLen < right - left - 1) {
        lo = left + 1;
        maxLen = (right - left - 1);
    }
}
```

107. Reverse nodes in k group in a Linked List:

```
public ListNode reverseKGroup(ListNode head, int k) {
    ListNode curr = head;
    int count = 0;
    while (curr != null && count != k) { // find the k+1 node
        curr = curr.next;
        count++;
    }
    if (count == k) { // if k+1 node is found
        // head - head-pointer to direct part,
        // curr - head-pointer to reversed part;
        curr = reverseKGroup(curr, k);

        // reverse current k-group:
        while (count-- > 0) {
            // tmp - next head in direct part
            ListNode tmp = head.next;
            // pre appending "direct" head to the reversed list
            head.next = curr;
            // move head of reversed part to a new node
            curr = head;
            // move "direct" head to the next node in direct part
            head = tmp;
        }
        head = curr;
    }
    return head;
}
```

108. Flatten Nested List Iterator:

```java
public class NestedIterator implements Iterator<Integer> {

    private Stack<NestedInteger> data = null;

    public NestedIterator(List<NestedInteger> nestedList) {
        this.data = new Stack<>();
        populateStack(nestedList);
    }

    private void populateStack(List<NestedInteger> list) {
        if (list == null || list.isEmpty()) return;
        int size = list.size() - 1;
        for (int i = size; i >= 0; i--) {
            NestedInteger nestedInteger = list.get(i);
            if (nestedInteger.isInteger()) data.push(nestedInteger);
            else populateStack(nestedInteger.getList());
        }
    }

    @Override
    public Integer next() {
        return data.pop().getInteger();
    }

    @Override
    public boolean hasNext() {
        return data.size() > 0;
    }
}
```

109. Course Schedule II:

```
private static final int WHITE = 0;
private static final int GRAY = 1;
private static final int BLACK = 2;
boolean possible = true;

Stack<Integer> topologicalOrder;
Map<Integer,List<Integer>> adjacencyMatrix;
Map<Integer,Integer> color;

private void init(int num) {
    topologicalOrder = new Stack<>();
    adjacencyMatrix = new HashMap<>();
    color = new HashMap<>();
    for (int i = 0; i < num; i++) {
        color.put(i, WHITE);
    }
}

private void dfs(int node) {
  if (!possible) return;
  color.put(node, GRAY);
  List<Integer> adjList = adjacencyMatrix.getOrDefault(node,new ArrayList<>());
  for (int n: adjList) {
    int col = color.get(n);
    if (col == WHITE) dfs(n);
    if (col == GRAY) {
      possible = false;
      break;
    }
  }
  color.put(node, BLACK);
  topologicalOrder.push(node);
}








public int[] findOrder(int numCourses, int[][] prerequisites) {
    this.init(numCourses);
    for (int i = 0; i < prerequisites.length; i++) {
        int src = prerequisites[i][1];
        int dest = prerequisites[i][0];
        List<Integer> adjList = adjacencyMatrix.get(src);
        if (adjList == null) {
            adjList = new ArrayList < > ();
            adjacencyMatrix.put(src, adjList);
        }
        adjList.add(dest);
    }

    for (int i = 0; i < numCourses; i++) {
        if (color.get(i) == WHITE) {
            dfs(i);
        }
    }

    if (!possible) return new int[0];
    int[] arr = new int[topologicalOrder.size()];
    for (int i = 0; i < arr.length; i++) {
        arr[i] = topologicalOrder.pop();
    }
    return arr;
}
```

110. Path with maximum and minimum value:

```
public int maximumMinimumPath(int[][] A) {
    final int[][] DIRS = {{0,1},{1,0},{0,-1},{-1,0}};
    Queue<int[]> pq = new PriorityQueue<>((a, b)-> Integer.compare(b[0], a[0]));
    pq.add(new int[] {A[0][0], 0, 0 });
    int maxscore = A[0][0];
    A[0][0] = -1; // visited
    while (!pq.isEmpty()) {
        int[] top = pq.poll();
        int i = top[1], j = top[2], n = A.length, m = A[0].length;
        maxscore = Math.min(maxscore, top[0]);
        if (i == n - 1 && j == m - 1)
            return maxscore;
        for (int[] d: DIRS) {
            int newi = d[0] + i, newj = d[1] + j;
            if (newi >= 0 && newi < n && newj >= 0
                          && newj < m && A[newi][newj] >= 0) {
                pq.add(new int[] {
                    A[newi][newj], newi, newj
                });
                A[newi][newj] = -1;
            }
        }
    }
    return -1; // shouldn't get here
}
```

111. LFU Cache:

```
public class LFUCache {
    HashMap<Integer, Integer> vals;
    HashMap<Integer, Integer> counts;
    HashMap<Integer, LinkedHashSet<Integer>> lists;
    int cap;
    int min = -1;
    public LFUCache(int capacity) {
        cap = capacity;
        vals = new HashMap<>();
        counts = new HashMap<>();
        lists = new HashMap<>();
        lists.put(1, new LinkedHashSet<>());
    }
    public int get(int key) {
        if(!vals.containsKey(key))
            return -1;
        int count = counts.get(key);
        counts.put(key, count+1);
        lists.get(count).remove(key);
        if(count==min && lists.get(count).size()==0)
            min++;
        if(!lists.containsKey(count+1))
            lists.put(count+1, new LinkedHashSet<>());
        lists.get(count+1).add(key);
        return vals.get(key);
    }
    public void set(int key, int value) {
        if(cap <= 0)
            return;
        if(vals.containsKey(key)) {
            vals.put(key, value);
            get(key);
            return;
        }
        if(vals.size() >= cap) {
            int evit = lists.get(min).iterator().next();
            lists.get(min).remove(evit);
            vals.remove(evit);
        }
        vals.put(key, value);
        counts.put(key, 1);
        min = 1;
        lists.get(1).add(key);
    }
}
```

112.Longest increasing Subsequence Length:

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 0;
        for (int num : nums) {
            int i = Arrays.binarySearch(dp, 0, len, num);
            if (i < 0) {
                i = -(i + 1);
            }
            dp[i] = num;
            if (i == len) {
                len++;
            }
        }
        return len;
    }
}
```

113. Match regular Expression:

```
class Solution {
    public boolean isMatch(String text, String pattern) {
        if (pattern.isEmpty()) return text.isEmpty();
        boolean first_match = (!text.isEmpty() &&
          (pattern.charAt(0) == text.charAt(0)
           || pattern.charAt(0) == '.'));

        if (pattern.length() >= 2 && pattern.charAt(1) == '*'){
            return (isMatch(text, pattern.substring(2)) ||
            (first_match && isMatch(text.substring(1), pattern)));
        } else {
            return first_match && isMatch(text.substring(1),
                   pattern.substring(1));
        }
    }
}
```

114. Spiral Order Matrix 1:

```
Given a matrix of m * n elements (m rows, n columns), return all
elements of the matrix in spiral order.
Example: Given the following matrix:
[
    [ 1, 2, 3 ],
    [ 4, 5, 6 ],
    [ 7, 8, 9 ]
]
You should return : [1, 2, 3, 6, 9, 8, 7, 4, 5]
```

```
public class Solution {
  // DO NOT MODIFY THE LIST
  public ArrayList<Integer> spiralOrder(final List<ArrayList<Integer>> A) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        // Populate result;
        int m, n;
        m = A.size();
        n = A.get(0).size();
        if (m == 0)
            return result;

        int len;
        int dir = 0; // right
        int row, col, layer;
        row = col = layer = 0;
        result.add(A.get(0).get(0));

        for (int step = 1; step < m * n; step++) {

            switch (dir) {

                // Go right
                case 0:
                    if (col == n - layer - 1) {
                        dir = 1;
                        row++;
                    } else {
                        col++;
                    }
                    break;

                // Go down
                case 1:
                    if (row == m - layer - 1) {
                        dir = 2;
                        col--;
                    } else {
                        row++;
                    }
                    break;





                // Go left
                case 2:
                    if (col == layer) {
                        dir = 3;
                        row--;
                    } else {
                        col--;
                    }
                    break;

                // Go up
                case 3:
                    if (row == layer + 1) {
                        dir = 0;
                        col++;
                        layer++;
                    } else {
                        row--;
                    }
                    break;
            }
            //System.out.println(row + " " + col);
            result.add(A.get(row).get(col));
        }
        return result;
    }
}
```

115. Remove Duplicate from sorted array 1:

Given a sorted array, remove the duplicates in place such that each element appears only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory. For example, given input array `A=[1,1,2]`, your function should return `length = 2`, and A is now `[1,2]`.

```
public static int removeDuplicates(int[] A) {
    if (A.length < 2)
        return A.length;
    int j = 0;
    int i = 1;
    while (i < A.length) {
        if (A[i] != A[j]) {
            j++;
            A[j] = A[i];
        }
        i++;
    }
    return j + 1;
}
```

Note that we only care about the first unique part of the original array. So it is ok if the input array is 1, 2, 2, 3, 3, the array is changed to 1, 2, 3, 3, 3.

116. Remove Duplicate from sorted array 2:

Follow up for "**Remove Duplicates**": *What if duplicates are allowed at most twice?* For example, given sorted array `A = [1,1,1,2,2,3]`, your function should return `length = 5`, and A is now `[1,1,2,2,3]`. So this problem also requires in-place array manipulation.

```
public int removeDuplicates(int[] nums) {
    if (nums == null) {
        return 0;
    }
    if (nums.length < 3) {
        return nums.length;
    }
    int i = 0;
    int j = 1;
    /*
    i, j 1 1 1 2 2 3
    step_1 0 1 i j
    step_2 1 2 i j
    step_3 1 3 i j
    step_4 2 4 i j
    */
    while (j < nums.length) {
        if (nums[j] == nums[i]) {
            if (i == 0) {
                i++;
                j++;
            } else if (nums[i] == nums[i - 1]) {
                j++;
            } else {
                i++;
                nums[i] = nums[j];
                j++;
            }
        } else {
            i++;
            nums[i] = nums[j];
            j++;
        }
    }
    return i + 1;
}
```

The problem with this solution is that there are 4 cases to handle. If we shift our two points to right by 1 element, the solution can be simplified as the **Solution 2**.

Java Solution 2

```java
public int removeDuplicates(int[] nums) {
    if (nums == null) {
        return 0;
    }
    if (nums.length <= 2) {
        return nums.length;
    }
    /*
    ---------------------
    1,1,1,2,2,3
    i j
    ---------------------
    */
    int i = 1; // point to previous
    int j = 2; // point to current
    while (j < nums.length) {
        if (nums[j] == nums[i] && nums[j] == nums[i - 1]) {
            j++;
        } else {
            i++;
            nums[i] = nums[j];
            j++;
        }
    }
    return i + 1;
}
```

117. 3 Sum Closest:

Given an array nums of *n* integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example:

```
Given array nums = [-1, 2, 1, -4], and target = 1.
The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

```java
public int threeSumClosest(int[] nums, int target) {
    // When the array have only 3 elements
    int result = nums[0] + nums[1] + nums[2];
    Arrays.sort(nums);

    for (int start = 0; start < nums.length - 2; start++) {
        int middle = start + 1;
        int end = nums.length - 1;
        while (middle < end) {
            int sum = nums[start] + nums[middle] + nums[end];
            if (sum == target) {
                return target;
            } else if (sum > target) {
                end--;
            } else {
                middle ++;
            }
            if (Math.abs(sum - target) < Math.abs(result - target)) {
                result = sum;
            }
        }
    }
    return result;
}
```

118. Find and Replace String:

To some string S, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same size). Each replacement operation has 3 parameters: a starting index i, a source word x and a target word y. The rule is that if x starts at position i in the original string S, then we will replace that occurrence of x with y. If not, we do nothing.

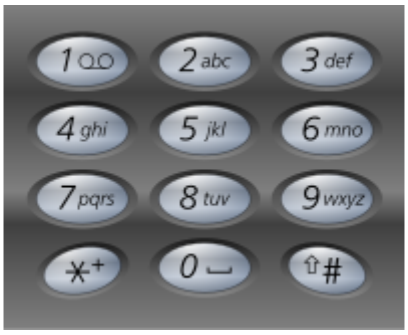
For example, if we have S = "abcd" and we have some replacement operation i = 2, x = "cd", y = "ffff", then because "cd" starts at position 2 in the original string S, we will replace it with "ffff".

```
Input: S = "abcd", indexes = [0,2],
sources = ["a","cd"],targets = ["eee","ffff"]
Output: "eeebffff"
Explanation: "a" starts at index 0 in S, so it's replaced by "eee".
"cd" starts at index 2 in S, so it's replaced by "ffff".
```

```
class Replace {
      String source;
      String target;
      int start;
      public Replace(String st, String t, int s){
            this.source = st;
            this.target = t;
            this.start = s;
  }
}

class Solution {

  public String findReplaceString(String S, int[] indexes,
                                String[] sources,String[] targets) {

    Map<Integer,Replace> replaceMap = new HashMap<>();
    for(int i = 0 ; i < indexes.length ; i++){
      Replace replace = new Replace(sources[i],targets[i],indexes[i]);
      replaceMap.put(indexes[i],replace);
    }

    StringBuffer result = new StringBuffer();
    int i = 0;
    while( i < S.length()){
      Replace replace = replaceMap.get(i);
      if(replace == null){ // no replace found
        result.append(S.charAt(i));
        i++;
      }else{
          String fromS = S.substring(i , ( i + replace.source.length()));
          if(fromS.equals(replace.source)){
            result.append(replace.target);
            i = i + replace.source.length();
          }else{
            result.append(S.charAt(i));
            i++;
          }
      }
    }
    return result.toString();
  }
}
```

119. Letter Combinations of a Phone Number:

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



Example:

```
Input: "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

This problem can be solved by a typical **DFS** algorithm. DFS problems are very similar and can be solved by using a simple recursion.

```
public List<String> letterCombinations(String digits) {
    HashMap<Character, char[]> dict = new HashMap<Character, char[]>();
    dict.put('2',new char[]{'a','b','c'});
    dict.put('3',new char[]{'d','e','f'});
    dict.put('4',new char[]{'g','h','i'});
    dict.put('5',new char[]{'j','k','l'});
    dict.put('6',new char[]{'m','n','o'});
    dict.put('7',new char[]{'p','q','r','s'});
    dict.put('8',new char[]{'t','u','v'});
    dict.put('9',new char[]{'w','x','y','z'});

    List<String> result = new ArrayList<String>();
    if(digits==null||digits.length()==0){
        return result;
    }
    char[] arr = new char[digits.length()];
    helper(digits, 0, dict, result, arr);
    return result;
}

private void helper(String digits,int index,HashMap<Character,char[]> dict,
                        List<String> result, char[] arr){
    if(index==digits.length()){
        result.add(new String(arr));
        return;
    }
    char number = digits.charAt(index);
    char[] candidates = dict.get(number);
    for(int i=0; i<candidates.length; i++){
        arr[index]=candidates[i];
        helper(digits, index+1, dict, result, arr);
    }
}
```

Time complexity is **O(k^n)**, where k is the biggest number of letters a digit can map (k=4) and n is the length of the digit string.

120. Construct Binary Tree from Inorder and Postorder Traversal:

Given **inorder** and **postorder** traversal of a tree, construct the binary tree.

```
in-order:   4 2 5  (1)  6 7 3 8
post-order: 4 5 2  6 7 8 3  (1)
```

From the post-order array, we know that the last element is the root. We can find the root in in-order array. Then we can identify the left and right subtrees of the root from in-order array. Using the length of left sub-tree, we can identify left and right subtrees in post-order array. Recursively, we can build up the tree.

```
public TreeNode buildTree(int[] inorder, int[] postorder) {
  int inStart = 0;
  int inEnd = inorder.length - 1;
  int postStart = 0;
  int postEnd = postorder.length - 1;
  return buildTree(inorder, inStart, inEnd, postorder, postStart,
postEnd);
}
public TreeNode buildTree(int[] inorder, int inStart, int inEnd,
    int[] postorder, int postStart, int postEnd) {
    if (inStart > inEnd || postStart > postEnd) return null;
    int rootValue = postorder[postEnd];
    TreeNode root = new TreeNode(rootValue);
    int k = 0;
    for (int i = 0; i < inorder.length; i++) {
        if (inorder[i] == rootValue) {
            k = i;
            break;
        }
    }
    root.left = buildTree(inorder, inStart, k - 1, postorder, postStart,
        (postStart + k - (inStart + 1)));
    // Because k is not the length, it need to -(inStart+1) to get the length
    root.right = buildTree(inorder, k + 1, inEnd, postorder,
              (postStart + k - inStart) , postEnd - 1);
    // postStart+k-inStart = postStart+k-(inStart+1) +1
    return root;
}
```

121.Distribute Candy:

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:

1. Each child must have at least one candy.

2. Children with a higher rating get more candies than their neighbors.

What are the minimum candies you must give? This problem can be solved in **O(n)** time. We can always assign a neighbor with 1 more if the neighbor has a higher rating value. However, to get the minimum total number, we should always start adding 1s in the ascending order. We can solve this problem by scanning the array from both sides. First, scan the array from left to right, and assign values for all the ascending pairs. Then scan from right to left and assign values to descending pairs. This problem is similar to Trapping Rain Water.

```java
public int candy(int[] ratings) {
    if (ratings == null || ratings.length == 0) {
        return 0;
    }
    int[] candies = new int[ratings.length];
    candies[0] = 1;
    //from let to right
    for (int i = 1; i < ratings.length; i++) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        } else {
            // if not ascending, assign 1
            candies[i] = 1;
        }
    }
    int result = candies[ratings.length - 1];
    //from right to left
    for (int i = ratings.length - 2; i >= 0; i--) {
        int cur = 1;
        if (ratings[i] > ratings[i + 1]) {
            cur = candies[i + 1] + 1;
        }
        result += Math.max(cur, candies[i]);
        candies[i] = cur;
    }
    return result;
}
```

122. Summary Of Ranges:

Given a sorted integer array without duplicates, return the summary of its ranges for consecutive numbers. For example, given `[0,1,2,4,5,7]`, return `["0->2","4->5","7"]`.

Analysis: When iterating over the array, two values need to be tracked:

1. The first value of a new range and,
2. The previous value in the range.

```
public List < String > summaryRanges(int[] nums) {
    List < String > result = new ArrayList < String > ();
    if (nums == null || nums.length == 0)
        return result;
    if (nums.length == 1) {
        result.add(nums[0] + "");
    }
    int pre = nums[0]; // previous element
    int first = pre; // first element of each range
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] == pre + 1) {
            if (i == nums.length - 1) {
                result.add(first + "->" + nums[i]);
            }
        } else {
            if (first == pre) {
                result.add(first + "");
            } else {
                result.add(first + "->" + pre);
            }
            if (i == nums.length - 1) {
                result.add(nums[i] + "");
            }
            first = nums[i];
        }
        pre = nums[i];
    }
    return result;
}
```

123. Missing Ranges:

Given a sorted integer array nums, where the range of elements are in the inclusive range `[lower, upper]`, return its missing ranges.

Example:

```
Input: nums = [0, 1, 3, 50, 75], lower = 0 and upper = 99,
Output: ["2", "4->49", "51->74", "76->99"]
```

```
public List<String> findMissingRanges(int[] nums, int lower, int upper) {
    List<String> result = new ArrayList<>();
    int start = lower;
    if (lower == Integer.MAX_VALUE) {
        return result;
    }
    for (int i = 0; i < nums.length; i++) {
        //handle duplicates, e.g., [1,1,1] lower=1 upper=1
        if (i < nums.length - 1 && nums[i] == nums[i + 1]) {
            continue;
        }
        if (nums[i] == start) {
            start++;
        } else {
            result.add(getRange(start, nums[i] - 1));
            if (nums[i] == Integer.MAX_VALUE) {
                return result;
            }
            start = nums[i] + 1;
        }
    }
    if (start <= upper) {
        result.add(getRange(start, upper));
    }
    return result;
}
private String getRange(int n1, int n2) {
    return n1 == n2 ? String.valueOf(n1) : String.format("%d->%d", n1, n2);
}
```

124. Word Ladder | Find the length of the required transformation:

```
start = "hit" end = "cog" dict = ["hot","dot","dog","lot","log"]
One shortest transformation is "hit" ->"hot" ->"dot" ->"dog" ->"cog"
```

```
class WordNode {
    String word;
    int numSteps;
    public WordNode(String word, int numSteps) {
        this.word = word;
        this.numSteps = numSteps;
    }
}

public class Solution {
    public int ladderLength(String beginWord,String endWord,
                            Set<String> wordDict) {
        LinkedList<WordNode> queue = new LinkedList<WordNode>();
        queue.add(new WordNode(beginWord, 1));
        wordDict.add(endWord);
        while (!queue.isEmpty()) {
            WordNode top = queue.remove();
            String word = top.word;
            if (word.equals(endWord)) {
               return top.numSteps;
            }
            char[] arr = word.toCharArray();
            for (int i = 0; i < arr.length; i++) {
                for (char c = 'a'; c <= 'z'; c++) {
                    char temp = arr[i];
                    if (arr[i] != c) {
                        arr[i] = c;
                    }
                    String newWord = new String(arr);
                    if (wordDict.contains(newWord)) {
                        queue.add(new WordNode(newWord, top.numSteps + 1));
                        wordDict.remove(newWord);
                    }
                    arr[i] = temp;
                }
            }
        }
        return 0;
    }
}
```

125. House Robber I:

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

The key is to find the relation `dp[i] = Math.max(dp[i-1], dp[i-2]+nums[i])`.

```java
public int rob(int[] nums) {
    if (nums == null || nums.length == 0)
        return 0;
    if (nums.length == 1)
        return nums[0];
    int[] dp = new int[nums.length];
    dp[0] = nums[0];
    dp[1] = Math.max(nums[0], nums[1]);
    for (int i = 2; i < nums.length; i++) {
        dp[i] = Math.max(dp[i - 2] + nums[i], dp[i - 1]);
    }
    return dp[nums.length - 1];
}
```

126. House Robber II:

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Analysis This is an extension of House Robber. There are two cases here

1. 1st element is included and last is not included.
2. 1st is not included and last is included.

Therefore, we can use the similar dynamic programming approach to scan the array twice and get the larger value.

```
public int rob(int[] nums) {
    if (nums == null || nums.length == 0)
        return 0;
    if (nums.length == 1)
        return nums[0];
    int max1 = robHelper(nums, 0, nums.length - 2);
    int max2 = robHelper(nums, 1, nums.length - 1);
    return Math.max(max1, max2);
}

public int robHelper(int[] nums, int i, int j) {
    if (i == j) {
        return nums[i];
    }
    int[] dp = new int[nums.length];
    dp[i] = nums[i];
    dp[i + 1] = Math.max(nums[i + 1], dp[i]);
    for (int k = i + 2; k <= j; k++) {
        dp[k] = Math.max(dp[k - 1], dp[k - 2] + nums[k]);
    }
    return dp[j];
}
```

127. Paint House I:

There are a row of n houses, each house can be painted with one of the three colors: **red**, **blue** or **green**. The cost of painting each house with a certain color is different. You have to paint all the houses so that no two adjacent houses have the same color. The cost of painting each house with a certain color is represented by a `[nx3]` cost matrix. **For example:** `costs[0][0]` is the cost of painting house 0 with **Color Red**, `costs[1][2]` is the cost of painting house 1 with **Color Green**, and so on. Find the minimum cost to paint all houses.

```java
public int minCost(int[][] costs) {
    if (costs == null || costs.length == 0) return 0;
    for (int i = 1; i < costs.length; i++) {
        costs[i][0] += Math.min(costs[i - 1][1], costs[i - 1][2]);
        costs[i][1] += Math.min(costs[i - 1][0], costs[i - 1][2]);
        costs[i][2] += Math.min(costs[i - 1][0], costs[i - 1][1]);
    }
    int m = costs.length - 1;
    return Math.min(Math.min(costs[m][0], costs[m][1]), costs[m][2]);
}
```

```java
public int minCost(int[][] costs) {
    if (costs == null || costs.length == 0) return 0;
    int[][] matrix = new int[3][costs.length];
    for (int j = 0; j < costs.length; j++) {
      if (j == 0) {
        matrix[0][j] = costs[j][0];
        matrix[1][j] = costs[j][1];
        matrix[2][j] = costs[j][2];
      }
      else {
        matrix[0][j] =Math.min(matrix[1][j-1],matrix[2][j-1])+ costs[j][0];
        matrix[1][j] =Math.min(matrix[0][j-1],matrix[2][j-1])+ costs[j][1];
        matrix[2][j] =Math.min(matrix[0][j-1],matrix[1][j-1])+ costs[j][2];
      }
    }
    int result = Math.min(matrix[0][costs.length- 1],matrix[1][costs.length-1]);
    result = Math.min(result, matrix[2][costs.length - 1]);
    return result;
}
```

128. Number of Island II:

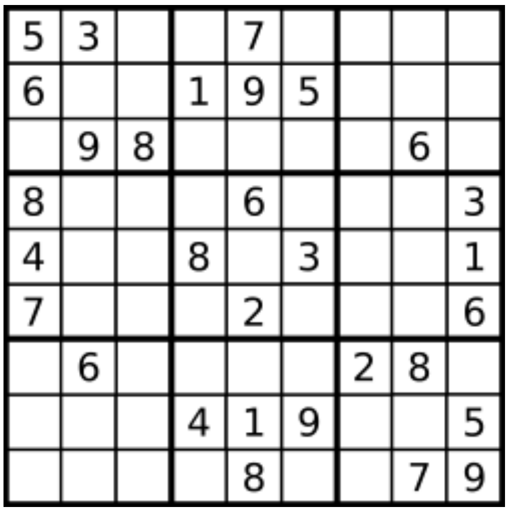
A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

```java
public List<Integer> numIslands2(int m, int n, int[][] positions) {
    int[] rootArray = new int[m * n];
    Arrays.fill(rootArray, -1);
    ArrayList<Integer> result = new ArrayList<Integer>();
    int[][] directions = {{-1,0},{0,1},{1,0},{0,-1}};
    int count = 0;
    for (int k = 0; k < positions.length; k++) {
        count++;
        int[] p = positions[k];
        int index = p[0] * n + p[1];
        rootArray[index] = index; //set root to be itself for each node
        for (int r = 0; r < 4; r++) {
          int i = p[0] + directions[r][0];
          int j = p[1] + directions[r][1];
          if (i >= 0 && j >= 0 && i < m && j < n
                   && rootArray[i * n + j] != -1) {
            int thisRoot = getRoot(rootArray, i * n + j);//get neighbor's root
            if (thisRoot != index) {
               rootArray[thisRoot] = index; //set previous root's root
               count--;
            }
          }
        }
        result.add(count);
    }
    return result;
}
public int getRoot(int[] arr, int i) {
    while (i != arr[i]) {
        i = arr[i];
    }
    return i;
}
```

129. Validate Sudoku:

Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the 9 `3x3` sub-boxes of the grid must contain the digits `1-9` without repetition.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A partially filled sudoku which is valid.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

```
public boolean isValidSudoku(char[][] board) {
    if (board == null || board.length != 9 || board[0].length != 9) return false;
    // check each column
    for (int i = 0; i < 9; i++) {
        boolean[] m = new boolean[9];
        for (int j = 0; j < 9; j++) {
            if (board[i][j] != '.') {
                if (m[(int)(board[i][j] - '1')]) {
                    return false;
                }
                m[(int)(board[i][j] - '1')] = true;
            }
        }
    }
    //check each row
    for (int j = 0; j < 9; j++) {
        boolean[] m = new boolean[9];
        for (int i = 0; i < 9; i++) {
            if (board[i][j] != '.') {
                if (m[(int)(board[i][j] - '1')]) {
                    return false;
                }
                m[(int)(board[i][j] - '1')] = true;
            }
        }
    }
    //check each 3*3 matrix
    for (int block = 0; block < 9; block++) {
        boolean[] m = new boolean[9];
        for (int i = block / 3 * 3; i < block / 3 * 3 + 3; i++) {
            for (int j = block % 3 * 3; j < block % 3 * 3 + 3; j++) {
                if (board[i][j] != '.') {
                    if (m[(int)(board[i][j] - '1')]) {
                        return false;
                    }
                    m[(int)(board[i][j] - '1')] = true;
                }
            }
        }
    }
    return true;
}
```

130. Set Matrix Zeros:

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in-place.

Example 1:

```
Input:
[
  [1,1,1],
  [1,0,1],
  [1,1,1]
]
Output:
[
  [1,0,1],
  [0,0,0],
  [1,0,1]
]
```

```
public class Solution {

    public void setZeroes(int[][] matrix) {
        boolean firstRowZero = false;
        boolean firstColumnZero = false;

        //set first row and column zero or not
        for (int i = 0; i < matrix.length; i++) {
            if (matrix[i][0] == 0) {
                firstColumnZero = true;
                break;
            }
        }

        for (int i = 0; i < matrix[0].length; i++) {
            if (matrix[0][i] == 0) {
                firstRowZero = true;
                break;
            }
        }




        //mark zeros on first row and column
        for (int i = 1; i < matrix.length; i++) {
            for (int j = 1; j < matrix[0].length; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }
        //use mark to set elements
        for (int i = 1; i < matrix.length; i++) {
            for (int j = 1; j < matrix[0].length; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        //set first column and row
        if (firstColumnZero) {
            for (int i = 0; i < matrix.length; i++)
                matrix[i][0] = 0;
        }
        if (firstRowZero) {
            for (int i = 0; i < matrix[0].length; i++)
                matrix[0][i] = 0;
        }
    }
}
```

131. Odd Even Linked List:

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes. The program should run in `O(1)` space complexity and `O(nodes)` time complexity.

Example:

```
Given 1-> 2-> 3-> 4-> 5-> NULL,
return 1-> 3-> 5-> 2-> 4-> NULL.
```

Analysis This problem can be solved by using two pointers. We iterate over the link and move the two pointers.

```
public ListNode oddEvenList(ListNode head) {
    if (head == null)
        return head;
    ListNode result = head;
    ListNode p1 = head;
    ListNode p2 = head.next;
    ListNode connectNode = head.next;
    while (p1 != null && p2 != null) {
        ListNode t = p2.next;
        if (t == null)
            break;
        p1.next = p2.next;
        p1 = p1.next;
        p2.next = p1.next;
        p2 = p2.next;
    }
    p1.next = connectNode;
    return result;
}
```

132. Remove duplicates from sorted Linked List I:

Given a sorted linked list, delete all duplicates such that each element appears only once. **For Example:**

```
Given 1-> 1-> 2, return 1-> 2.
Given 1-> 1-> 2-> 3-> 3, return 1-> 2-> 3.
```

The key of this problem is using the right loop condition. And change what is necessary in each loop. You can use different iteration conditions like the following 2 solutions.

```
/** Definition for singly-linked list.
 *  public class ListNode {
 *    int val;
 *    ListNode next;
 *    ListNode(int x) {
 *      val = x;
 *      next = null;
 *    }
 *  }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode prev = head;
        ListNode p = head.next;
        while (p != null) {
            if (p.val == prev.val) {
                prev.next = p.next;
                p = p.next;
                //no change prev
            } else {
                prev = p;
                p = p.next;
            }
        }
        return head;
    }
}
```

2nd Solution:

```java
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode p = head;
        while (p != null && p.next != null) {
            if (p.val == p.next.val) {
                p.next = p.next.next;
            } else {
                p = p.next;
            }
        }
        return head;
    }
}
```

133. Remove duplicates from sorted Linked List II:

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. For example, given `1 -> 1-> 1 ->2 ->3`, return `2 ->3`.

```java
public ListNode deleteDuplicates(ListNode head) {
    ListNode t = new ListNode(0);
    t.next = head;
    ListNode p = t;
    while (p.next != null && p.next.next != null) {
        if (p.next.val == p.next.next.val) {
            int dup = p.next.val;
            while (p.next != null && p.next.val == dup) {
                p.next = p.next.next;
            }
        } else {
            p = p.next;
        }
    }
    return t.next;
}
```

134. Partition of Linked List:

Given a linked list and a value **x**, partition it such that all nodes less than **x** come before nodes greater than or equal to **x**. You should preserve the original relative order of the nodes in each of the two partitions.

For Example:

```
Given 1-> 4-> 3-> 2-> 5-> 2 and x = 3,
Return 1-> 2-> 2-> 4-> 3-> 5.
```

```java
public class Solution {
    public ListNode partition(ListNode head, int x) {
        if (head == null) return null;
        ListNode fakeHead1 = new ListNode(0);
        ListNode fakeHead2 = new ListNode(0);
        fakeHead1.next = head;
        ListNode p = head;
        ListNode prev = fakeHead1;
        ListNode p2 = fakeHead2;
        while (p != null) {
            if (p.val < x) {
                p = p.next;
                prev = prev.next;
            } else {
                p2.next = p;
                prev.next = p.next;
                p = prev.next;
                p2 = p2.next;
            }
        }
        // close the list
        p2.next = null;
        prev.next = fakeHead2.next;
        return fakeHead1.next;
    }
}
```

135. Intersection of Linked List:

Write a program to find the node at which the intersection of two singly linked lists begins.

For Example, The following two Linked Lists:

```
A:  a1 -> a2 ->
                   c1 -> c2 -> c3 -> NULL
B: b1 -> b2 -> b3 ->
```

The Beginning of the intersect of the two lists are at node c1.

First calculate the length of two lists and find the difference. Then start from the longer list at the diff offset, iterate through 2 lists and find the node.

```
/**
 * 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)
{
        int len1 = 0;
        int len2 = 0;
        ListNode p1 = headA, p2 = headB;
        if (p1 == null || p2 == null)
            return null;

        while (p1 != null) {
            len1++;
            p1 = p1.next;
        }



        while (p2 != null) {
            len2++;
            p2 = p2.next;
        }

        int diff = 0;
        p1 = headA;
        p2 = headB;
        if (len1 > len2) {
            diff = len1 - len2;
            int i = 0;
            while (i < diff) {
                p1 = p1.next;
                i++;
            }
        } else {
            diff = len2 - len1;
            int i = 0;
            while (i < diff) {
                p2 = p2.next;
                i++;
            }
        }
        while (p1 != null && p2 != null) {
            if (p1.val == p2.val) {
                return p1;
            } else {}
            p1 = p1.next;
            p2 = p2.next;
        }
        return null;
    }
}
```

136. Min Stack:

```
class Node{
  int value;
  int min;
  public Node(int v,int m){
    value = v;
    min = m;
  }
}
class MinStack {

    private LinkedList<Node> data;

    /** initialize your data structure here. */
    public MinStack() {
       data = new LinkedList<>();
    }

    public void push(int x) {
      Node node = null;
      if(data.isEmpty()){
        node = new Node(x,x);
      }else{
        Node prev = data.getLast();
        node = new Node(x, Math.min(x,prev.min));
      }
      data.add(node);
    }

    public void pop() {
       if(!data.isEmpty()) data.removeLast();
    }

    public int top() {
      return data.isEmpty() ? -1 : data.getLast().value;

    }

    public int getMin() {
       return data.isEmpty() ? -1 : data.getLast().min;
    }
}
```

137. Longest Valid Parentheses:

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring. For `"(()"`, the longest valid parentheses substring is `"()"`, which has **length = 2**. Another example is `")()())"`, where the longest valid parentheses substring is `"()()"`, which has **length = 4**.

A `stack` can be used to track and reduce pairs. Since the problem asks for length, we can put the index into the stack along with the character. For coding simplicity purposes, we can use 0 to represent `(` and 1 to represent `)`. This problem is similar to Valid Parentheses, which can also be solved by using a stack.

```java
public int longestValidParentheses(String s) {
    Stack<int[]> stack = new Stack<>();
    int result = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == ')') {
            if (!stack.isEmpty() && stack.peek()[0] == 0) {
                stack.pop();
                result = Math.max(result, i - (stack.isEmpty()? -1 :
                  stack.peek()[1]));
            } else {
                stack.push(new int[] {
                    1,
                    i
                });
            }
        } else {
            stack.push(new int[] {
                0,
                i
            });
        }
    }
    return result;
}
```

138. Permutations:

Given a collection of distinct integers, return all possible permutations.

Example:

```
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

The basic idea is to permute `n` numbers into the resulting `List<List<Integer>>` in every possible position.

For example, if the input `num[]` is `{1,2,3}`:

First, add `1` into the resulting `List<List<Integer>>` named `answer`.

- Then, 2 can be added either in front of 1 or after 1. Now we have an answer of `{{2,1},{1,2}}`. There are 2 lists in the current answer.
- Then we have to add 3. first copy `{2,1}` and `{1,2}`, add 3 in position 0.
- Then copy `{2,1}` and `{1,2}`, and add 3 into position 1.
- then do the same thing for position 3.
- Finally we have `2*3=6` lists in answer, which is what we want.
- Please check the implementation on the next page.

```
public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> ans = new ArrayList<List<Integer>>();
    if (num.length ==0) return ans;
    List<Integer> l0 = new ArrayList<Integer>();
    l0.add(num[0]);
    ans.add(l0);
    for (int i = 1; i< num.length; ++i){
        List<List<Integer>> new_ans = new ArrayList<List<Integer>>();
        for (int j = 0; j<=i; ++j){
           for (List<Integer> l : ans){
               List<Integer> new_l = new ArrayList<Integer>(l);
               new_l.add(j,num[i]);
               new_ans.add(new_l);
           }
        }
        ans = new_ans;
    }
    return ans;
}
```

```
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> list = new ArrayList<>();
   // Arrays.sort(nums); // not necessary
   backtrack(list, new ArrayList<>(), nums);
   return list;
}
private void backtrack(List<List<Integer>> list, List<Integer> tempList,
int [] nums){
   if(tempList.size() == nums.length){
      list.add(new ArrayList<>(tempList));
   } else{
      for(int i = 0; i < nums.length; i++){
         // element already exists, skip
         if(tempList.contains(nums[i])) continue;
         tempList.add(nums[i]);
         backtrack(list, tempList, nums);
         tempList.remove(tempList.size() - 1);
      }
   }
}
```

139. Expressive Word:

```
Example: Input: S = "heeellooo" words = ["hello", "hi", "helo"] Output: 1
Explanation: We can extend "e" and "o" in the word "hello" to get "heeellooo".
We can't extend "helo" to get "heeellooo" because the group "ll" is not size 3 or
more.
```

```
class Pair{
      char ch;
      int count;
      public Pair(char c, int cn){
            this.ch = c;
            this.count = cn;
  }
}

class Solution {
    public int expressiveWords(String S, String[] words) {
      if(S == null || words == null || words.length == 0) return 0;
      int count = 0;
      List<Pair> pairList = getPairList(S);
      for(String word : words){
        if(isExpressiveWord(pairList,word)) count ++;
      }
      return count;
    }
    private boolean isExpressiveWord(List<Pair> sPairList,String word){
        List<Pair> wPairList = getPairList(word);
        if(sPairList.size() != wPairList.size()) return false;
        for( int i = 0 ; i < sPairList.size() ; i++){
            Pair fromS = sPairList.get(i);
            Pair fromW = wPairList.get(i);
            if(fromS.ch != fromW.ch) return false;
            if(fromS.count < fromW.count) return false;
            if(fromS.count == fromW.count) continue;
            if(fromS.count >= 3) continue;
            else return false;
        }
        return true;
    }




    private List<Pair> getPairList(String word){
        LinkedList<Pair> pairList = new LinkedList<>();
        for(int i = 0 ; i < word.length() ; i++){
            char current = word.charAt(i);
            if(i == 0 ){
              Pair pair = new Pair(current,1);
              pairList.add(pair);
            }else{
              Pair last = pairList.getLast();
              if(last.ch == current) last.count = last.count + 1;
              else{
                  Pair pair = new Pair(current,1);
                  pairList.add(pair);
              }
            }
        }
        return pairList;
    }
}
```

140. Remove invalid parentheses:

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results. Note: The input string may contain letters other than the parentheses (and).

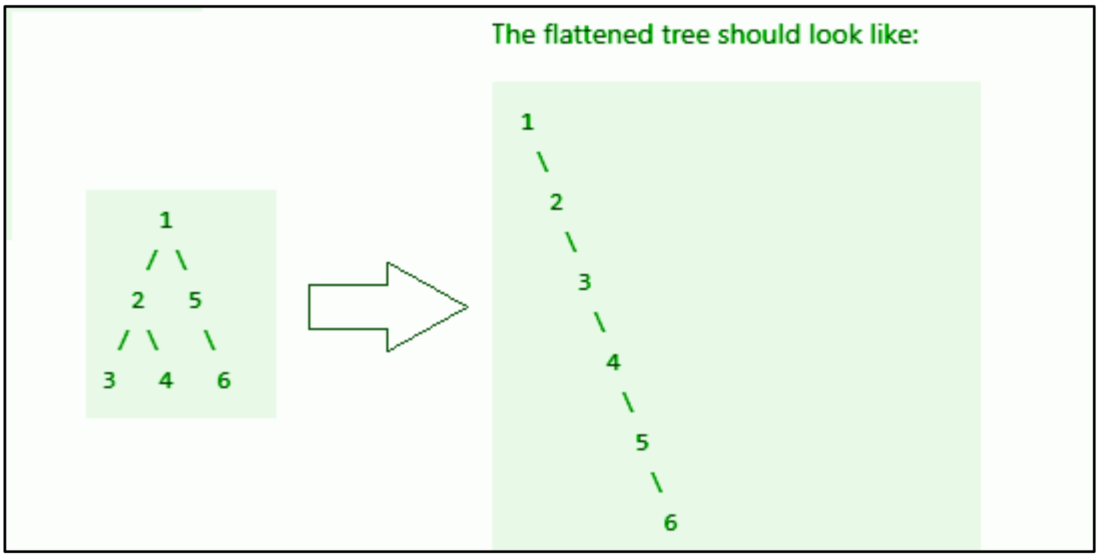
Examples:

```
"()())()"  -> ["()()()", "(())()"]

"(a)())()" -> ["(a)()()", "(a())()"]

")("       -> [""]
```

```
public class Solution {
    ArrayList<String> result = new ArrayList<>();
    int max = 0;
    public List<String> removeInvalidParentheses(String s) {
        if (s == null)
            return result;
        dfs(s, "", 0, 0);
        if (result.size() == 0) {
            result.add("");
        }
        return result;
    }

    public void dfs(String left,String right, int countLeft, int maxLeft)
    {

        if (left.length() == 0) {
            if (countLeft == 0 && right.length() != 0) {
                if (maxLeft > max) {
                    max = maxLeft;
                }
                if (maxLeft == max && !result.contains(right)) {
                    result.add(right);
                }
            }
            return;
        }

        if (left.charAt(0) == '(') {
          //keep (
          dfs(left.substring(1), right + "(", countLeft + 1, maxLeft + 1);
          //drop (
          dfs(left.substring(1), right, countLeft, maxLeft);
        } else if (left.charAt(0) == ')') {
            if (countLeft > 0) {
              dfs(left.substring(1), right + ")", countLeft - 1, maxLeft);
            }
            dfs(left.substring(1), right, countLeft, maxLeft);
        } else {
            dfs(left.substring(1), right + String.valueOf(left.charAt(0)),
                    countLeft, maxLeft);
        }
    }
}
```

141. Flatten Binary Tree to Linked List:

Given a binary tree, flatten it to a linked list in-place. For example, given the following tree:



Go down the tree to the left, when the right child is not null, push the right child to the stack.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void flatten(TreeNode root) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode p = root;
        while(p != null || !stack.empty()){
            if(p.right != null){
                stack.push(p.right);
            }

            if(p.left != null){
                p.right = p.left;
                p.left = null;
            }else if(!stack.empty()){
                TreeNode temp = stack.pop();
                p.right=temp;
            }

            p = p.right;
        }
    }
}
```

142. Binary Tree From Preorder And Postorder

Return any binary tree that matches the given preorder and postorder traversals. Values in the traversals `pre` and `post` are distinct positive integers.

Example 1:

```
Input: pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
Output: [1,2,3,4,5,6,7]
```

Recursive Solution :

Create a node `TreeNode(pre[preIndex])` as the root. Because root node will be lastly iterated in post order, `if root.val == post[posIndex]`, it means we have constructed the whole tree, If we haven't completed constructed the whole tree, So we recursively `constructFromPrePost` for left subtree and right subtree. And finally, we'll reach the `posIndex` that `root.val == post[posIndex]`. We increment `posIndex` and return our `root` node.

```
int preIndex = 0, posIndex = 0;
public TreeNode constructFromPrePost(int[]pre, int[]post) {
  TreeNode root = new TreeNode(pre[preIndex++]);
  if (root.val != post[posIndex])
    root.left = constructFromPrePost(pre, post);
  if (root.val != post[posIndex])
    root.right = constructFromPrePost(pre, post);
  posIndex++;
  return root;
}
```

Iterative Solution:

We will use preorder traversal to generate `TreeNodes`, push them into stack and pop them out by post order.

1. Iterate on preorder array and construct `TreeNode` one by one.
2. Save the current path of the tree into a `Stack`.
3. `node = new TreeNode(pre[i])`, if not left child, add node to the left. otherwise add it to the right.
4. If we meet the same value in the pre and post array, that means we have completed the construction for the current subtree. We pop it from the `Stack`.

```java
public TreeNode constructFromPrePost(int[] pre, int[] post) {
      Deque<TreeNode> s = new ArrayDeque<>();
      s.offer(new TreeNode(pre[0]));
      for (int i = 1, j = 0; i < pre.length; ++i) {
            TreeNode node = new TreeNode(pre[i]);
            while (s.getLast().val == post[j]) {
                  s.pollLast(); j++;
            }
            if (s.getLast().left == null) s.getLast().left = node;
            else s.getLast().right = node;
            s.offer(node);
      }
      return s.getFirst();
}
```

143. Convert BST to Greater Tree:

Given a **Binary Search Tree** (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Example:

```
Input: The root of a Binary Search Tree like this:
              5
            /   \
           2     13

Output: The root of a Greater Tree like this:
             18
            /   \
          20     13
```

```java
class Solution {
    private int sum = 0;
    public TreeNode convertBST(TreeNode root) {
        if (root != null) {
            convertBST(root.right);
            sum += root.val;
            root.val = sum;
            convertBST(root.left);
        }
        return root;
    }
}
```

Intuition

If we don't want to use recursion, we can also perform a reverse in-order traversal via iteration and a literal stack to emulate the call stack.

Algorithm

One way to describe the iterative stack method is in terms of the intuitive recursive solution. First, we initialize an empty stack and set the current node to the root. Then, so long as there are unvisited nodes in the stack or `node` does not point to `null`, we push all of the nodes along the path to the rightmost leaf onto the stack. This is equivalent to always processing the right subtree first in the recursive solution, and is crucial for the guarantee of visiting nodes in order of decreasing value. Next, we visit the node on the top of our stack, and consider its left subtree. This is just like visiting the current node before recursing on the left subtree in the recursive solution. Eventually, our stack is empty and `node` points to the left `null` child of the tree's minimum value node, so the loop terminates.

```java
public TreeNode convertBST(TreeNode root) {
    int sum = 0;
    TreeNode node = root;
    Stack<TreeNode> stack = new Stack<TreeNode>();

    while (!stack.isEmpty() || node != null) {
        /* push all nodes up to (and including) this subtree's maximum on
         * the stack. */
        while (node != null) {
            stack.add(node);
            node = node.right;
        }

        node = stack.pop();
        sum += node.val;
        node.val = sum;

        /* all nodes with values between the current and its parent lie in
         * the left subtree. */
        node = node.left;
    }
    return root;
}
```

144. Construct String from Binary Tree:

You need to construct a string consisting of parentheses and integers from a binary tree with the preorder traversing way.

The `null` node needs to be represented by an empty parenthesis `pair()`. And you need to omit all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the original binary tree.

Example 1:

```
Input: Binary tree: [1,2,3,4]
       1
     /   \
    2     3
   /
  4

Output: "1(2(4))(3)"

Explanation: Originally it needs to be "1(2(4)())(3()())",but you need to omit
all the unnecessary empty parenthesis pairs.And it will be "1(2(4))(3)".
```

Example 2:

```
Input:                 Binary                 tree:                 [1,2,3,null,4]
      1
    /                                                                            \
   2                                                                             3
    \
     4

Output:                                                                "1(2()(4))(3)"
Explanation: Almost the same as the first example, except we can't omit the first
parenthesis pair to break the one-to-one mapping relationship between the input
and the output.
```

```java
public class Solution {
    public String tree2str(TreeNode t) {
        if (t == null)
            return "";
        Stack<TreeNode> stack = new Stack<>();
        stack.push(t);
        Set<TreeNode> visited = new HashSet<>();
        StringBuilder s = new StringBuilder();
        while (!stack.isEmpty()) {
            t = stack.peek();
            if (visited.contains(t)) {
                stack.pop();
                s.append(")");
            } else {
                visited.add(t);
                s.append("(" + t.val);
                if (t.left == null && t.right != null)
                    s.append("()");
                if (t.right != null)
                    stack.push(t.right);
                if (t.left != null)
                    stack.push(t.left);
            }
        }
        return s.substring(1, s.length() - 1);
    }
}
```

```java
public class Solution {
    public String tree2str(TreeNode t) {
        if(t == null)
            return "";
        if(t.left == null && t.right == null)
            return t.val + "";
        if(t.right == null)
            return t.val  + "(" + tree2str(t.left) + ")";
        return t.val +
           "(" + tree2str(t.left) + ")(" + tree2str(t.right) + ")";
    }
}
```

145. Largest Number:

Given a list of non negative integers, arrange them such that they form the largest number.

Example 1:

```
Input: [10,2]  Output: "210"
Input: [3,30,34,5,9] Output: "9534330"
```

Intuition:

To construct the largest number, we want to ensure that the most significant digits are occupied by the largest digits.

Algorithm:

First, we convert each integer to a string. Then, we sort the array of strings.

While it might be tempting to simply sort the numbers in descending order, this causes problems for sets of numbers with the same leading digit. For example, sorting the problem example in descending order would produce the number **9534303**, while the correct answer can be achieved by transposing the **3** and the **30**. Therefore, for each pairwise comparison during the sort, we compare the numbers achieved by concatenating the pair in both orders. We can prove that this sorts into the proper order as follows:

Assume that (Without loss of generality), for some pair of integers **a** and **b**, our comparator dictates that **a** should precede **b** in sorted order. This means that $a \frown b > b \frown a$ (where $\frown$ represents concatenation). For the sort to produce an incorrect ordering, there must be some **c** for which **b** precedes **c** and **c** precedes **a**. This is a contradiction because $a \frown b > b \frown a$ and $b \frown c > c \frown b$ implies $a \frown c > c \frown a$. In other words, our custom comparator preserves transitivity, so the sort is correct.

Once the array is sorted, the most "**significant**" number will be at the front. There is a minor edge case that comes up when the array consists of only zeroes, so if the most significant number is 0, we can simply return 0. Otherwise, we build a string out of the sorted array and return it.

```
class Solution {
    private class LargerNumberComparator implements Comparator<String> {
        @Override
        public int compare(String a, String b) {
            String order1 = a + b;
            String order2 = b + a;
           return order2.compareTo(order1);
        }
    }

    public String largestNumber(int[] nums) {
        // Get input integers as strings.
        String[] asStrs = new String[nums.length];
        for (int i = 0; i < nums.length; i++) {
            asStrs[i] = String.valueOf(nums[i]);
        }

        // Sort strings according to custom comparator.
        Arrays.sort(asStrs, new LargerNumberComparator());

         // If, after being sorted, the largest number is `0`, the entire
         // number is zero.
        if (asStrs[0].equals("0")) {
            return "0";
        }

        // Build the largest number from the sorted array.
        String largestNumberStr = new String();
        for (String numAsStr : asStrs) {
            largestNumberStr += numAsStr;
        }
        return largestNumberStr;
    }
}
```

146. Swap List Nodes in pairs:

Given a `Linked List`, swap every two adjacent nodes and return its head. You may not modify the values in the list's nodes, only nodes itself may be changed.

Example:

`Given 1-> 2-> 3-> 4, you should return the list as 2-> 1-> 4-> 3.`

```java
public class Solution {
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode cur = head;
        ListNode newHead = head.next;
        while (cur != null && cur.next != null) {
            ListNode tmp = cur;
            cur = cur.next;
            tmp.next = cur.next;
            cur.next = tmp;
            cur = tmp.next;
            if (cur != null && cur.next != null) tmp.next = cur.next;
        }
        return newHead;
    }
}
```

147. Anti Diagonals:

Give a **`N*N square matrix`**, return an array of its anti-diagonals. Look at the example for more details.

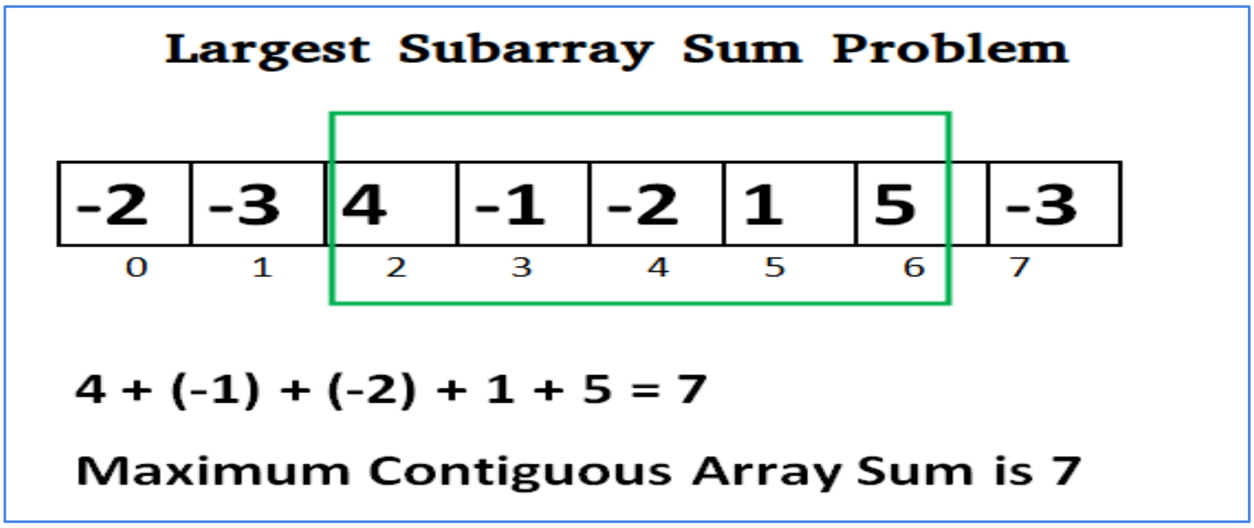
Example:

```
Input:                     Return the following :

[ 1 2 3 ]                        [ [1],
[ 4 5 6 ]                          [2, 4],
[ 7 8 9 ]                          [3, 5, 7],
                                   [6, 8],
                                   [9]
                                 ]
```

```java
public class Solution {
    public List<List<Integer>> diagonal(List<List<Integer>> a) {

        List<List<Integer>> result = new ArrayList<>();
        int dimension = a.size();
        for (int i = 0; i < dimension*2-1; i++) {
            result.add(getDiagonal(i, a));
        }
        return result;
    }

    public List<Integer> getDiagonal(int layer, List<List<Integer>> a) {
        List<Integer> result = new ArrayList<>();
        for (int i = 0; i <= layer; i++) {
            int j = (layer - i);
            if (i < a.size() && j < a.size()) {
                 result.add(a.get(i).get(j));
            }
        }
        return result;
    }
}
```

148. Max Sum Contiguous Subarray

Write an efficient program to find the sum of contiguous subarray within a one dimensional array of numbers which has the largest sum.



```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_ending_here < 0)
          max_ending_here = 0
  (c) if(max_so_far < max_ending_here)
          max_so_far = max_ending_here
return max_so_far
```

Explanation:

Simple idea of **Kadane's algorithm** is to look for all positive contiguous segments of the array (`max_ending_here` is used for this). And keep track of maximum sum contiguous segment among all positive segments (`max_so_far` is used for this). Each time we get a positive sum compare it with `max_so_far` and update `max_so_far` if it is greater than `max_so_far`.

Let's take the example:

```
    int[] a = {-2, -3, 4, -1, -2, 1, 5, -3};
    max_so_far = max_ending_here = 0

    for i=0,  a[0] =  -2
    max_ending_here = max_ending_here + (-2)
    Set max_ending_here = 0 because max_ending_here < 0

    for i=1,  a[1] =  -3
    max_ending_here = max_ending_here + (-3)
    Set max_ending_here = 0 because max_ending_here < 0

    for i=2,  a[2] =  4
    max_ending_here = max_ending_here + (4)
    max_ending_here = 4
    max_so_far is updated to 4 because max_ending_here greater
    than max_so_far which was 0 till now

    for i=3,  a[3] =  -1
    max_ending_here = max_ending_here + (-1)
    max_ending_here = 3

    for i=4,  a[4] =  -2
    max_ending_here = max_ending_here + (-2)
    max_ending_here = 1

    for i=5,  a[5] =  1
    max_ending_here = max_ending_here + (1)
    max_ending_here = 2

    for i=6,  a[6] =  5
    max_ending_here = max_ending_here + (5)
    max_ending_here = 7
    max_so_far is updated to 7 because max_ending_here is
    greater than max_so_far

    for i=7,  a[7] =  -3
    max_ending_here = max_ending_here + (-3)
    max_ending_here = 4
```

```
class Kadane {
    public static void main(String[] args) {
        int [] a = {-2, -3, 4, -1, -2, 1, 5, -3};
        System.out.println("Maximum contiguous sum is " +
            maxSubArraySum(a));
    }

    static int maxSubArraySum(int a[]) {
        int size = a.length;
        int max_so_far = Integer.MIN_VALUE, max_ending_here = 0;
        for (int i = 0; i < size; i++) {
            max_ending_here = max_ending_here + a[i];
            if (max_so_far < max_ending_here)
                max_so_far = max_ending_here;
            if (max_ending_here < 0)
                max_ending_here = 0;
        }
        return max_so_far;
    }
}
```

Above program can be optimized further, if we compare `max_so_far` with `max_ending_here` only if `max_ending_here` is greater than 0.

```
static int maxSubArraySum(int a[], int size) {
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++) {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        /* Do not compare for all elements.
           Compare only when max_ending_here > 0
        */
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

    }
    return max_so_far;
}
```

149. Next Permutation:

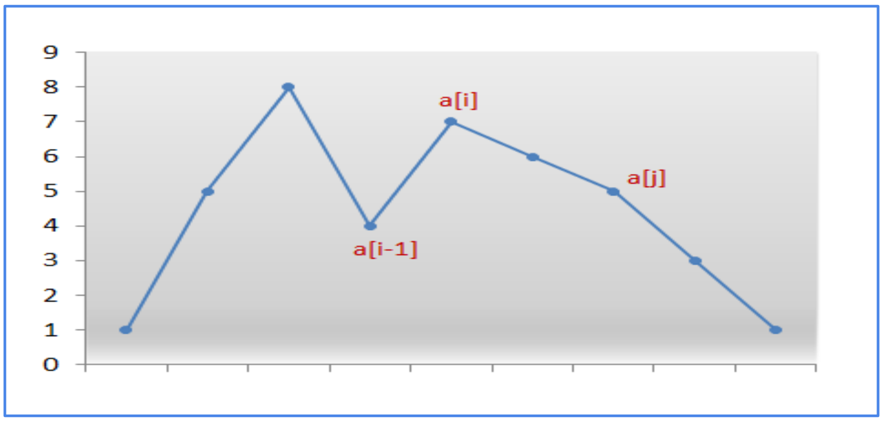
Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers. If such an arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order). The replacement must be in-place and use only constant extra memory. Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

First, we observe that for any given sequence that is in descending order, no next larger permutation is possible. For example, no next permutation is possible for the following array: `[9, 5, 4, 3, 1]`

We need to find the first pair of two successive numbers `a[i]` and *a[i−1]*, from the right, which satisfy *a[i] > a[i−1]*. Now, no rearrangements to the right of *a[i−1]* can create a larger permutation since that subarray consists of numbers in descending order. Thus, we need to rearrange the numbers to the right of *a[i−1]* including itself.

Now, what kind of rearrangement will produce the next larger number? We want to create the permutation just larger than the current one. Therefore, we need to replace the number *a[i−1]* with the number which is just larger than itself among the numbers lying to its right section, say `a[j]`.



We swap the numbers *a[i−1]* and `a[j]`. We now have the correct number at index *i−1*. But still the current permutation isn't the permutation that we are looking for. We need the smallest permutation that can be formed by using the numbers only to the right of *a[i−1]*. Therefore, we need to place those numbers in ascending order to get their smallest permutation.

But, recall that while scanning the numbers from the right, we simply kept decreasing the index until we found the pair `a[i]` and *a[i−1]* where, *a[i]>a[i−1]*. Thus, all numbers to the right of *a[i−1]* were already sorted in descending order. Furthermore, swapping *a[i−1]* and `a[j]` didn't change that order. Therefore, we simply need to reverse the numbers following *a[i−1]* to get the next smallest lexicographic permutation.

```
public class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i + 1] <= nums[i]) {
            i--;
        }
        if (i >= 0) {
            int j = nums.length - 1;
            while (j >= 0 && nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }
    private void reverse(int[] nums, int start) {
        int i = start, j = nums.length - 1;
        while (i < j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

150. Sum of pairwise Hamming Distance:

Hamming Distance between two non-negative integers is defined as the number of positions at which the corresponding bits are different.

For Example: HammingDistance(2,7) = 2, as only the *first* and the *third* bit differs in the binary representation of 2 (010) and 7 (111). Given an array of N non-negative integers, find the sum of hamming distances of all pairs of integers in the array. Return the answer modulo 1000000007.

```
Let f(x,y) be the hamming distance defined above. The input array is A=[2, 4, 6]

The pairwise Hamming Distance is:
f(2, 2) + f(2, 4) + f(2, 6) = 0 + 2 + 1 = 3
f(4, 2) + f(4, 4) + f(4, 6) = 2 + 0 + 1 = 3
f(6, 2) + f(6, 4) + f(6, 6) = 1 + 1 + 0 = 2

Answer: (3 + 3 + 2) = 8
```

```
public int hammingDistance(final List<Integer> A) {
    int n = A.size();
    int dist = 0;
    for (int i = 0; i < 31; i++) {
        int oneCount = 0;
        for (int j = 0; j < n; j++) {
            int num = A.get(j);
            oneCount += (num & 1 << i) != 0 ? 1 : 0;
        }
        int zeroCount = n - oneCount;
        dist += (2 L * oneCount * zeroCount) % 1000000007;
        dist = dist % 1000000007;
    }
    return dist;
}
```

151. Implement Atoi:

```java
public static int myAtoi(String inNumberStr) {
    if (inNumberStr == null || inNumberStr.trim().length() == 0) {
        return 0;
    }
    int result = 0;
    inNumberStr = inNumberStr.trim();
    boolean negate = false;
    char sign = inNumberStr.charAt(0);
    if (sign == '+' || sign == '-') {
        if (sign == '-') {
            negate = true;
        }
        inNumberStr = inNumberStr.substring(1);
    }
    int length = inNumberStr.length();
    int start = -1;
    int index = 0;
    for (; index < length; index++) {
        char a = inNumberStr.charAt(index);
        if (a >= '0' && a <= '9') {
            if (start == -1)
                start = index;
        } else {
            break;
        }
    }
    int end = index - 1;
    if (start == -1) return 0;
    for (int i = start; i <= end; ++i) {
        char a = inNumberStr.charAt(i);
        int digit = a - '0';
        if (!negate) {
            if (result + digit * (int) Math.pow(10, end - i) >= result)
                result += digit * (int) Math.pow(10, end - i);
            else
                result = Integer.MAX_VALUE;
        } else {
            if (result - digit * (int) Math.pow(10, end - i) <= result)
                result -= digit * (int) Math.pow(10, end - i);
            else
                result = Integer.MIN_VALUE;
        }
    }
    return result;
}
```

152. Combination Sum:

Given a set of candidate numbers (candidates) (without duplicates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target. The same repeated number may be chosen from candidates unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]
```

```java
public List<List<Integer>> combinationSum(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}


private void backtrack(List<List<Integer>> list, List<Integer> tempList,
                       int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            // not i + 1 because we can reuse same elements
            backtrack(list, tempList, nums, remain - nums[i], i);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

153. Combination Sum II:

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target. Each number in candidates may only be used once in the combination.

Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8,
A solution set is:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

```java
public List<List<Integer>> combinationSum2(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}
private void backtrack(List<List<Integer>> list, List<Integer> tempList,
                       int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            // skip duplicates
            if(i > start && nums[i] == nums[i-1]) continue;
            templist.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Java solution using DFS:

```java
public List<List<Integer>> combinationSum2(int[] cand, int target) {
    Arrays.sort(cand);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs_com(cand, 0, target, path, res);
    return res;
}

void dfs_com(int[] cand, int cur, int target, List<Integer> path,
             List<List<Integer>> res) {
    if (target == 0) {
        res.add(new ArrayList(path));
        return ;
    }
    if (target < 0){
      return;
    }
    for (int i = cur; i < cand.length; i++){
        if (i > cur && cand[i] == cand[i-1]){
            continue;
        }
        path.add(path.size(), cand[i]);
        dfs_com(cand, i+1, target - cand[i], path, res);
        path.remove(path.size()-1);
    }
}
```

154. Remove Duplicates from Sorted Array II

Given a sorted array *nums*, remove the duplicates in-place such that duplicates appeared at most *twice* and return the new length. Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

Example 1:

```
Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums
being 1, 1, 2, 2 and 3 respectively.

It doesn't matter what you leave beyond the returned length.
```

```
public int removeDuplicates(int[] nums) {
    int i = 0;
    for (int n : nums)
        if (i < 2 || n > nums[i-2])
            nums[i++] = n;
    return i;
}
```

155. Nearest Smaller Element on left of array:

```
//Java implementation of efficient algorithm to find
// smaller element on left side
class Solution {

    // Prints smaller elements on left side of every element
    static void printPrevSmaller(int arr[], int n) {
        // Create an empty stack
        Stack<Integer> S = new Stack<>();

        // Traverse all array elements
        for (int i = 0; i < n; i++) {
            // Keep removing top element from S while the top
            // element is greater than or equal to arr[i]
            while (!S.empty() && S.peek() >= arr[i]) {
                S.pop();
            }

            // If all elements in S were greater than arr[i]
            if (S.empty()) {
                System.out.print("_, ");
            } else //Else print the nearest smaller element
            {
                System.out.print(S.peek() + ", ");
            }

            // Push this element
            S.push(arr[i]);
        }
    }

    /* Driver program to test insertion sort */
    public static void main(String[] args) {
        int arr[] = {1, 3, 0, 2, 5};
        int n = arr.length;
        printPrevSmaller(arr, n);
    }
}
```

156. Max Non Negative SubArray

Given an array, return length of the longest subarray of non- negative integers.

Examples :

```
Input : {2, 3, 4, -1, -2, 1, 5, 6, 3}
Output : 4
The subarray [ 1, 5, 6, 3] has length 4 and contains no negative
integers
```

```java
public ArrayList<Integer> maxset(ArrayList<Integer> a) {
      long maxSum = 0;
      long newSum = 0;
      ArrayList<Integer> maxArray = new ArrayList<Integer>();
      ArrayList<Integer> newArray = new ArrayList<Integer>();
      for (Integer i : a) {
            if (i >= 0) {
                  newSum += i;
                  newArray.add(i);
            } else {
                  newSum = 0;
                  newArray = new ArrayList<Integer>();
            }
            if ((maxSum < newSum) || ((maxSum == newSum)
            && (newArray.size() > maxArray.size()))) {
                  maxSum = newSum;
                  maxArray = newArray;
            }
      }
      return maxArray;
}
```

157. Convert Sorted List to BST

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST. For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

Approach : Inorder Simulation

The inorder traversal on a binary search tree leads to a very interesting outcome.

```
Elements processed in the inorder fashion on a binary search tree turn out
to be sorted in ascending order.
```

The approach listed here makes use of this idea to formulate the construction of a binary search tree. The reason we are able to use this idea in this problem is because we are given a `sorted` linked list initially. Before looking at the algorithm, let us look at how the inorder traversal actually leads to a sorted order of nodes' values. The critical idea based on the inorder traversal that we will exploit to solve this problem, is:

```
We know that the leftmost element in the inorder traversal has to be the
head of our given linked list. Similarly, the next element in the inorder
traversal will be the second element in the linked list and so on. This is
made possible because the initial list given to us is sorted in ascending
order.
```

Now that we have an idea about the relationship between the inorder traversal of a binary search tree and the numbers being sorted in ascending order, let's get to the algorithm.

Algorithm

Let's quickly look at a pseudo-code to make the algorithm simple to understand.

```
➔ function formBst(start, end)
➔      mid = (start + end) / 2
➔      formBst(start, mid - 1)
➔      TreeNode(head.val)
➔      head = head.next
➔      formBst(mid + 1, end)
```

A. Iterate over the linked list to find out it's length. We will make use of two different pointer variables here to mark the beginning and the end of the list. Let's call them `start` and `end` with their initial values being `0` and `length - 1` respectively.

B. Remember, we have to simulate the inorder traversal here. We can find out the middle element by using `(start + end) / 2`. Note that we don't really find out the middle node of the linked list. We just have a variable telling us the index of the middle element. We simply need this to make recursive calls on the two halves.

C. Recurse on the left half by using `start, mid - 1` as the starting and ending points.

D. The invariance that we maintain in this algorithm is that whenever we are done building the left half of the BST, the head pointer in the linked list will point to the root node or the middle node (which becomes the root). So, we simply use the current value pointed to by `head` as the root node and progress the head node by once i.e. `head = head.next.`

E. We recurse on the right hand side using `mid + 1, end` as the starting and ending points.

The Implementation

```
class Solution {

  private ListNode head;

  private int findSize(ListNode head) {
    ListNode ptr = head;
    int c = 0;
    while (ptr != null) {
      ptr = ptr.next;
      c += 1;
    }
    return c;
  }

  private TreeNode convertListToBST(int l, int r) {
    // Invalid case
    if (l > r) {
      return null;
    }
    int mid = (l + r) / 2;
    // First step of simulated inorder traversal. Recursively form
    // the left half
    TreeNode left = this.convertListToBST(l, mid - 1);

    // Once left half is traversed, process the current node
    TreeNode node = new TreeNode(this.head.val);
    node.left = left;
    // Maintain the invariance mentioned in the algorithm
    this.head = this.head.next;
    // Recurse on the right hand side and form BST out of them
    node.right = this.convertListToBST(mid + 1, r);
    return node;
  }

  public TreeNode sortedListToBST(ListNode head) {
    // Get the size of the linked list first
    int size = this.findSize(head);
    this.head = head;
    // Form the BST now that we know the size
    return convertListToBST(0, size - 1);
  }
}
```

158. Largest Distance between nodes of a Tree

Given an arbitrary unweighted rooted tree which consists of `N(2<=N<=40000)` nodes. The goal of the problem is to find the largest distance between two nodes in a tree.

Distance between two nodes is a number of edges on a path between the nodes (there will be a unique path between any pair of nodes since it is a tree). The nodes will be numbered `0` through `N-1`.

The tree is given as an array `P`, there is an edge between nodes `P[i]` and `i (0 <= i < N)`. Exactly one of the i's will have `P[i]` equal to `-1`, it will be root node.

Example:

If given P is `[-1, 0, 0, 0, 3]`, then node `0` is the root and the whole tree looks like this:

```
         0
      /  |  \
     1   2   3
              \
               4
```

One of the longest paths is `1 -> 0 -> 3 -> 4` and its length is `3`, thus the answer is `3`. Note that there are other paths with maximal distance.

```java
public class Solution {
    public int solve(ArrayList<Integer> A) {
        int n=A.size();
        List<List<Integer>> graph = new ArrayList<>();
        for(int i=0; i<n; i++) {
            graph.add(new ArrayList<Integer>());
        }
        int root = -1;
        for(int i=0; i<n; i++) {
            int num = A.get(i);
            if(num == -1) {
                root = i;
                continue;
            }
            graph.get(i).add(num);
            graph.get(num).add(i);
        }
        // Find the node which is farthest from root node using BFS
        int node = bfs(graph, n, root);
        // Find the maximum distance from farthest node using modified DFS
        return dfs(graph, n, node);
    }

    public int bfs(ArrayList<ArrayList<Integer>> graph, int n, int u) {
        boolean[] vis = new boolean[n];
        Queue<Integer> q = new LinkedList<Integer>();
        q.add(u);
        while(!q.isEmpty()) {
            // stores the farthest node from root node
            u = q.remove();
            if(!vis[u]) {
                vis[u] = true;
                for(Integer v : graph.get(u)) {
                    if(!vis[v]) {
                        q.add(v);
                    }
                }
            }
        }
        return u;
    }


    public int dfs(ArrayList<ArrayList<Integer>> graph, int n, int u) {
        int max = 0;
        Stack<Integer> node = new Stack<Integer>();
        Stack<Integer> dist = new Stack<Integer>();
        boolean[] vis = new boolean[n];
        node.push(u);
        dist.push(0);
        while(!node.isEmpty()) {
            u = node.pop();
            int d = dist.pop();
            if(!vis[u]) {
                vis[u] = true;
                for(Integer v : graph.get(u)) {
                    if(!vis[v]) {
                        max = Math.max(max, d+1);
                        node.push(v);
                        dist.push(d+1);
                    }
                }
            }
        }
        return max;
    }
}
```

```java
public int solve(ArrayList<Integer> A) {
        List<int[]> arr = new ArrayList<>();
        for (int i = 0; i < A.size(); ++i) {
                arr.add(new int[2]);
        }
        int maxDistance = 0;
        for (int i = A.size() - 1; i > 0; --i) {
                int element = A.get(i);
                int[] parent = arr.get(element);
                int currentLength = arr.get(i)[0] + 1;
                parent[1] = Math.max(parent[1], currentLength + parent[0]);
                parent[0] = Math.max(parent[0], currentLength);
                maxDistance = Math.max(maxDistance, parent[1]);
        }
        return maxDistance == 0 ? 0 : maxDistance;
}
```

159. Spiral Order Matrix II

Given a positive integer *n*, generate a square matrix filled with elements from 1 to n^2 in spiral order.

```
public int[][] generateMatrix(int n) {
    int[][] res = new int[n][n];
    int cur = 1;
    int rowBegin = 0;
    int rowEnd = n - 1;
    int colBegin = 0;
    int colEnd = n - 1;

    while (cur <= n * n) {
        int i = rowBegin;
        int j = colBegin;
        //left to right
        for (j = colBegin; j <= colEnd; j++) {
            res[rowBegin][j] = cur++;
        }
        rowBegin++;
        //top to bottom
        for (i = rowBegin; i <= rowEnd; i++) {
            res[i][colEnd] = cur++;
        }
        colEnd--;
        //right to left
        for (j = colEnd; j >= colBegin; j--) {
            res[rowEnd][j] = cur++;
        }
        rowEnd--;
        //bottom to top
        for (i = rowEnd; i >= rowBegin; i--) {
            res[i][colBegin] = cur++;
        }
        colBegin++;
    }
    return res;
}
```

160. Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s. The problem is the same as Combination Sum.

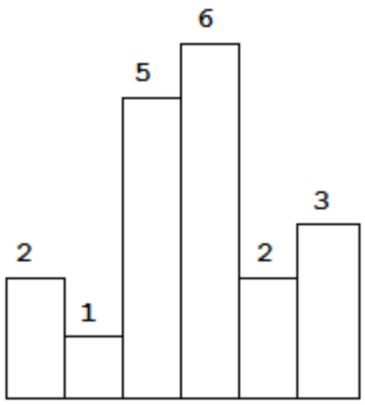
Example:

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

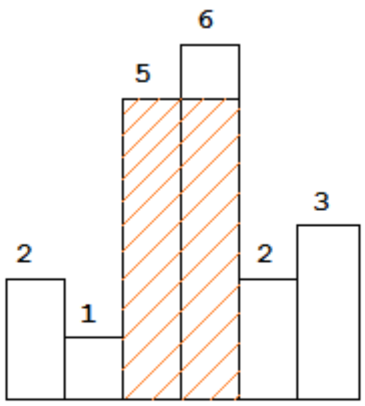
```
public List<List<String>> partition(String s) {
   List<List<String>> list = new ArrayList<>();
   backtrack(list, new ArrayList<>(), s, 0);
   return list;
}

public void backtrack(List<List<String>> list, List<String> tempList,
                      String s, int start){
   if(start == s.length())
      list.add(new ArrayList<>(tempList));
   else{
      for(int i = start; i < s.length(); i++){
         if(isPalindrome(s, start, i)){
            tempList.add(s.substring(start, i + 1));
            backtrack(list, tempList, s, i + 1);
            tempList.remove(tempList.size() - 1);
         }
      }
   }
}

public boolean isPalindrome(String s, int low, int high){
   while(low < high)
      if(s.charAt(low++) != s.charAt(high--)) return false;
   return true;
}
```

161. Largest Rectangle in Histogram

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where the width of each bar is 1, given height = `[2,1,5,6,2,3]`.



The largest rectangle is shown in the shaded area, which has an area = `10` unit.

For any bar `i` the maximum rectangle is of width `r - l - 1` where r - is the last coordinate of the bar to the right with height `h[r] >= h[i]` and l - is the last coordinate of the bar to the left which height `h[l] >= h[i]`

So if for any `i` coordinate we know his utmost higher (or of the same height) neighbors to the right and to the left, we can easily find the largest rectangle:

```
int maxArea = 0;
for (int i = 0; i < height.length; i++) {
     maxArea = Math.max(maxArea, height[i] *
                       (lessFromRight[i] - lessFromLeft[i] - 1));
}
```

The main trick is how to effectively calculate `lessFromRight` and `lessFromLeft` arrays. The trivial solution is to use O(n^2) solution and for each `i` element first find his left/right heighbour in the second inner loop just iterating back or forward:

```
for (int i = 1; i < height.length; i++) {
    int p = i - 1;
    while (p >= 0 && height[p] >= height[i]) {
        p--;
    }
    lessFromLeft[i] = p;
}
```

The only line change shifts this algorithm from O(n^2) to O(n) complexity: we don't need to rescan each item to the left - we can reuse results of previous calculations and "jump" through indices in quick manner:

```
while (p >= 0 && height[p] >= height[i]) {
      p = lessFromLeft[p];
}
```

Here is the whole solution:

```java
public static int largestRectangleArea(int[] height) {
    if (height == null || height.length == 0) {
        return 0;
    }
    // idx of the first bar the left that is lower than current
    int[] lessFromLeft = new int[height.length];

    // idx of the first bar the right that is lower than current
    int[] lessFromRight = new int[height.length];

    lessFromRight[height.length - 1] = height.length;
    lessFromLeft[0] = -1;

    for (int i = 1; i < height.length; i++) {
        int p = i - 1;

        while (p >= 0 && height[p] >= height[i]) {
            p = lessFromLeft[p];
        }
        lessFromLeft[i] = p;
    }

    for (int i = height.length - 2; i >= 0; i--) {
        int p = i + 1;

        while (p < height.length && height[p] >= height[i]) {
            p = lessFromRight[p];
        }
        lessFromRight[i] = p;
    }
    int maxArea = 0;
    for (int i = 0; i < height.length; i++) {
        maxArea = Math.max(maxArea, height[i] *
                           (lessFromRight[i] - lessFromLeft[i] - 1));
    }
    return maxArea;
}
```

162. Multiply Strings

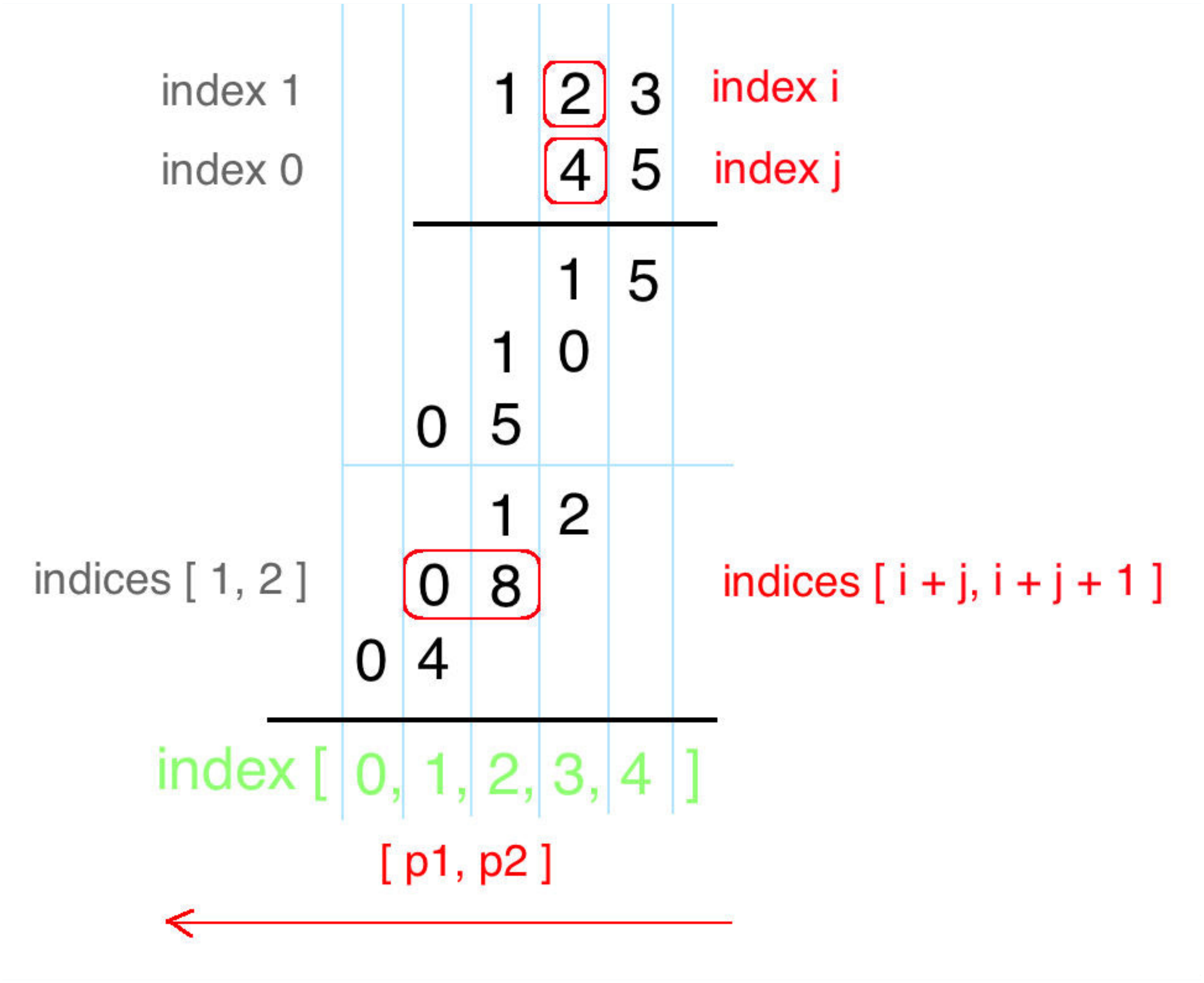
Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`, also represented as a string.

Example 1:

```
Input: num1 = "2", num2 = "3"
Output: "6"
```

Start from right to left, perform multiplication on every pair of digits, and add them together. Let's draw the process! From the following draft, we can immediately conclude:

``num1[i] * num2[j]` will be placed at indices `[i + j`, `i + j + 1]``



Here is the implementation:

```
public String multiply(String num1, String num2) {
    int m = num1.length(), n = num2.length();
    int[] pos = new int[m + n];

    for(int i = m - 1; i >= 0; i--) {
        for(int j = n - 1; j >= 0; j--) {
            int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
            int p1 = i + j, p2 = i + j + 1;
            int sum = mul + pos[p2];

            pos[p1] += sum / 10;
            pos[p2] = (sum) % 10;
        }
    }

    StringBuilder sb = new StringBuilder();
    for(int p : pos) if(!(sb.length() == 0 && p == 0)) sb.append(p);
    return sb.length() == 0 ? "0" : sb.toString();
}
```

163. Clone Graph

Given a reference of a node in a connected undirected graph. Return a deep copy (clone) of the graph. Each node in the graph contains a val (int) and a list (List[Node]) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

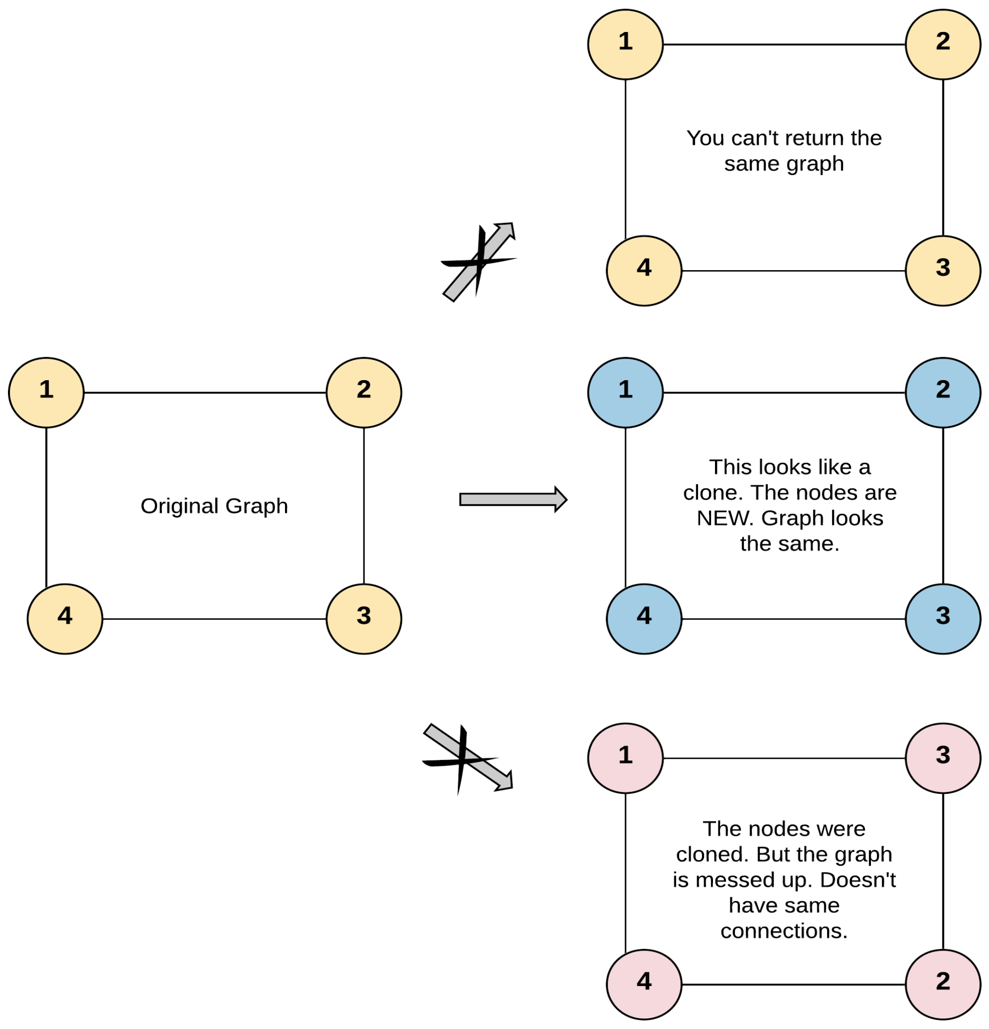
Test case format:

For simplicity sake, each node's value is the same as the node's index (1-indexed). For example, the first node with `val=1`, the second node with `val=2`, and so on. The graph is represented in the test case using an adjacency list.

Adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val=1`. You must return the copy of the given node as a reference to the cloned graph.

Example :



You can't return the same graph
Original Graph
This looks like a clone. The nodes are NEW. Graph looks the same.
The nodes were cloned. But the graph is messed up. Doesn't have same connections.

```
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]

Explanation: There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
```

Constraints:

- 1 <= Node.val <= 100
- Node.val is unique for each node.
- Number of Nodes will not exceed 100.
- There are no repeated edges and no self-loops in the graph.
- The Graph is connected and all nodes can be visited starting from the given node.

```java
class Solution {
    public HashMap<Integer, Node> map = new HashMap<>();

    public Node cloneGraph(Node node) {
        return clone(node);
    }

    public Node clone(Node node) {
        if (node == null) return null;

        if (map.containsKey(node.val))
            return map.get(node.val);

        Node newNode = new Node(node.val, new ArrayList<Node>());
        map.put(newNode.val, newNode);
        for (Node neighbor : node.neighbors)
            newNode.neighbors.add(clone(neighbor));
        return newNode;
    }
}
```

164. Bulls & Cows

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows.

Please note that both secret number and friend's guess may contain duplicate digits.

Example 1:

```
Input: secret = "1807", guess = "7810"
Output: "1A3B"
Explanation: 1 bull and 3 cows. The bull is 8, the cows are 0, 1 and 7.
```

Example 2:

```
Input: secret = "1123", guess = "0111"
Output: "1A1B"
Explanation: The 1st 1 in friend's guess is a bull, the 2nd or 3rd 1 is
a cow.
```

Note: You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

The idea is to iterate over the numbers in secret and in guess and count all bulls right away. For cows maintain an array that stores count of the number appearances in secret and in guess. Increment cows when either number from secret was already seen in guest or vice versa.

```
public String getHint(String secret, String guess) {
    int bulls = 0;
    int cows = 0;
    int[] numbers = new int[10];
    for (int i = 0; i<secret.length(); i++) {
        int s = Character.getNumericValue(secret.charAt(i));
        int g = Character.getNumericValue(guess.charAt(i));
        if (s == g) bulls ++;
        else {
            if (numbers[s] < 0) cows ++;
            if (numbers[g] > 0) cows ++;
            numbers[s] ++;
            numbers[g] --;
        }
    }
    return bulls + "A" + cows + "B";
}
```

A slightly more concise version:

```
public String getHint(String secret, String guess) {
    int bulls = 0;
    int cows = 0;
    int[] numbers = new int[10];
    for (int i = 0; i<secret.length(); i++) {
        if (secret.charAt(i) == guess.charAt(i)) bulls ++;
        else {
            if (numbers[secret.charAt(i)-'0']++ < 0) cows ++;
            if (numbers[guess.charAt(i)-'0']-- > 0) cows ++;
        }
    }
    return bulls + "A" + cows + "B";
}
```

165. Count And Say

The count-and-say sequence is the sequence of integers with the first five terms as following:

`1. 1`	`1` is read off as `"one 1"` or `11`.
`2. 11`	`11` is read off as `"two 1s"` or `21`.
`3. 21`	`21` is read off as `"one 2`, then `one 1"` or `1211`.
`4. 1211`	
`5. 111221`	

Given an integer *n* `where 1 ≤` *n* `≤ 30, generate the` *n*th term of the count-and-say sequence. You can do so recursively, in other words from the previous member read off the digits, counting the number of digits in groups of the same digit.

```java
public String countAndSay(int n) {
    StringBuilder curr = new StringBuilder("1");
    StringBuilder prev;
    int count;
    char say;
    for (int i = 1; i < n; i++) {
        prev = curr;
        curr = new StringBuilder();
        count = 1;
        say = prev.charAt(0);

        for (int j = 1, len = prev.length(); j < len; j++) {
            if (prev.charAt(j) != say) {
                curr.append(count).append(say);
                count = 1;
                say = prev.charAt(j);
            } else count++;
        }
        curr.append(count).append(say);
    }
    return curr.toString();
}
```

166. Edit Distance

Given two words ***word1*** and ***word2***, find the minimum number of operations required to convert ***word1*** to ***word2***.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

Let following be the function Definition :-

`f(i,j):=` minimum cost (or steps) required to convert first `i` characters of *`word1`* to first `j` characters of *`word2`*.

Case 1: `word1[i] == word2[j]`, i.e. the `i`th the jth character matches.

`f(i, j) = f(i - 1, j - 1)`

Case 2: `word1[i] != word2[j]`, then we must either insert, delete or replace, whichever is cheaper

`f(i, j) = 1 + min {f(i, j - 1), f(i - 1, j), f(i - 1, j - 1)}`

1. **`f(i, j - 1)`** represents insert operation.
2. **`f(i - 1, j)`** represents delete operation.
3. **`f(i - 1, j - 1)`** represents replace operation.

Here, we consider any operation from ***`word1`*** to ***`word2`***. It means, when we say insert operation, we insert a new character after ***`word1`*** that matches the jth character of ***`word2`***. So, now I have to match i characters of ***`word1`*** to **`(j-1)`** characters of ***`word2`***. Same goes for the other 2 operations as well.

Note that the problem is symmetric. The insert operation in one direction (i.e. from ***`word1`*** to ***`word2`***) is the same as delete operation in the other. So, we could choose any direction. Above equations become the recursive definitions for DP.

Base Case:

`f(0, k) = f(k, 0) = k`

Below is the direct bottom-up translation of this recurrence relation. It is only important to take care of 0-based index with actual code :-

```java
public class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();

        int[][] cost = new int[m + 1][n + 1];
        for(int i = 0; i <= m; i++)
            cost[i][0] = i;
        for(int i = 1; i <= n; i++)
            cost[0][i] = i;

        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(word1.charAt(i) == word2.charAt(j))
                    cost[i + 1][j + 1] = cost[i][j];
                else {
                    int a = cost[i][j];
                    int b = cost[i][j + 1];
                    int c = cost[i + 1][j];
                    cost[i + 1][j + 1] = a < b ? (a < c ? a : c) :
                                                 (b < c ? b : c);
                    cost[i + 1][j + 1]++;
                }
            }
        }
        return cost[m][n];
    }
}
```

Time complexity : If n is the length of word1, m of word2, because of the two indented loops, it is **O(nm)**