Validate Binary Search Tree

```
public boolean isValidBST (TreeNode root){
                Stack<TreeNode> stack = new Stack<TreeNode> ();
                TreeNode cur = root ;
                TreeNode pre = null ;
                while (!stack.isEmpty() || cur != null) {
                        if (cur != null) {
                                stack.push(cur);
                                cur = cur.left ;
                        } else {
                                TreeNode p = stack.pop() ;
                                if (pre != null && p.val <= pre.val) {
                                        return false ;
                                }
                                pre = p ;
                                cur = p.right ;
                        }
                }
                return true ;
        }
```

```
public class Solution {
   public boolean isValidBST(TreeNode root) {
      return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
   }

   public boolean isValidBST(TreeNode root, long minVal, long maxVal) {
      if (root == null) return true;
      if (root.val >= maxVal || root.val <= minVal) return false;
      return isValidBST(root.left, minVal, root.val) && isValidBST(root.right, root.val, maxVal);
   }
}
```

LRU Cache

```
import java.util.LinkedHashMap;
  public class LRUCache {
     private LinkedHashMap<Integer, Integer> map;
     private final int CAPACITY;
     public LRUCache(int capacity) {
        CAPACITY = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true){
           protected boolean removeEldestEntry(Map.Entry eldest) {
              return size() > CAPACITY;
           }
        };
     }
     public int get(int key) {
        return map.getOrDefault(key, -1);
     }
     public void set(int key, int value) {
        map.put(key, value);
     }
  }
```

Add Two Numbers II

```
public class Solution {
   public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
      Stack<Integer> s1 = new Stack<Integer>();
      Stack<Integer> s2 = new Stack<Integer>();
      ListNode newhead = null;
      while(l1 != null){
         s1.push(l1.val);
         l1 = l1.next;
      }
      while(l2 != null){
         s2.push(l2.val);
         l2 = l2.next;
      }
      int f = 0;
      while(!s1.isEmpty() || !s2.isEmpty()){
         int num = f;
         if(!s1.isEmpty()) num += s1.pop();
         if(!s2.isEmpty()) num += s2.pop();
         ListNode node = new ListNode(num % 10);
         f = num / 10;
         node.next = newhead;
         newhead = node;
      }
      if(f == 1){
         ListNode node = new ListNode(1);
         node.next = newhead;
         newhead = node;
      }
      return newhead;
   }
}
```

Best Time to Buy and Sell Stock

```
public class Solution {
   public int maxProfit(int[] prices) {
      if(prices.length < 2) return 0;
      int min = prices[0];
      int max = Integer.MIN_VALUE;
      for(Integer price : prices){
         max = Math.max(price - min, max);
         min = Math.min(price, min);
      }
      return max > 0? max : 0;
   }
}
```

Copy List with Random Pointer

```
public class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        RandomListNode start1 = head;
        RandomListNode start2 = head;
        if(head == null) return head;
        copyNext(start1);
        copyRandom(start2);
        return copyList(head);
    }
    private void copyNext(RandomListNode head){
        while(head != null){
            RandomListNode node = new RandomListNode(head.label);
            node.random = null;
            node.next = head.next;
            head.next = node;
            head = head.next.next;
        }
    }

    private void copyRandom(RandomListNode head){
        while(head != null){
            if(head.random != null){
                head.next.random = head.random.next;
            }
            head = head.next.next;
        }
    }

    private RandomListNode copyList(RandomListNode head){
        RandomListNode newhead = head.next;
        while(head != null){
            RandomListNode temp = head.next;
            head.next = temp.next;
            head = head.next;
            if(temp.next != null) temp.next = temp.next.next;
        }
        return newhead;
    }
}
```

Move Zeroes

```
public class Solution {
    public void moveZeroes(int[] nums) {
        int n = nums.length;
        if(n == 0 || n == 1) return;
        int j = 0;
        for(int i = 0; i < n; i++){
            while(j < n && nums[j] != 0){
                j++;
            }
            if(i > j && nums[i] != 0){
                int tmp = nums[i];
                nums[i] = nums[j];
                nums[j] = tmp;
            }
        }
    }
}
```

Two Sum

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        int[] rst = new int[2];
        for(int i = 0; i < nums.length; i++){
            map.put(nums[i], i);
        }
        for(int i = 0; i < nums.length; i++){
            if(map.containsKey(target - nums[i]) && i != map.get(target - nums[i])){
                rst[0] = i;
                rst[1] = map.get(target - nums[i]);
                return rst;
            }
        }
        return rst;
    }
}
```

First Unique Character in a String

```
public class Solution {
    public int firstUniqChar(String s) {
        int[] alph = new int[26];
        for(int i = 0; i < s.length(); i++){
            alph[s.charAt(i) - 'a']++;
        }
        for(int i = 0; i < s.length(); i++){
            if(alph[s.charAt(i) - 'a'] == 1) return i;
        }
```

```
        return -1;
    }
}
```

Min Stack

```
public class MinStack {

    /** initialize your data structure here. */
    private Stack<Integer> s1, s2;
    public MinStack() {
        s1 = new Stack<Integer>();
        s2 = new Stack<Integer>();
    }

    public void push(int x) {
        s1.push(x);
        if(s2.isEmpty()) s2.push(x);
        else{
            if(s2.peek() < x) s2.push(s2.peek());
            else s2.push(x);
        }
    }

    public void pop() {
        s1.pop();
        s2.pop();
    }

    public int top() {
        return s1.peek();
    }

    public int getMin() {
        return s2.peek();
    }
}
```

Populating Next Right Pointers in Each Node II

```
public class Solution {
    public void connect(TreeLinkNode root) {
        if(root == null) return;
        Queue<TreeLinkNode> q = new LinkedList<TreeLinkNode>();
        q.offer(root);
        while(!q.isEmpty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                TreeLinkNode cur = q.poll();
                if(i != size - 1){
                    cur.next = q.peek();
                }
                if(cur.left != null) q.offer(cur.left);
                if(cur.right != null) q.offer(cur.right);
            }
        }
        return;
    }
}
```

Unique Paths

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

Trapping Rain Water

```
public class Solution {
    public int trap(int[] height) {
        if(height.length == 0) return 0;
        int l = 0;
        int r = height.length - 1;
        int water = 0;
        int leftmax = Integer.MIN_VALUE;
        int rightmax = Integer.MIN_VALUE;
        while(l <= r){
            leftmax = Math.max(height[l], leftmax);
            rightmax = Math.max(height[r], rightmax);
            if(height[l] < height[r]){
                water += (leftmax - height[l]);
                l++;
            }
            else{
                water += (rightmax - height[r]);
                r--;
            }
        }
        return water;
    }
}
```

Reverse Linked List

```
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode newhead = null;
        while(head != null){
            ListNode temp = head.next;
            head.next = newhead;
            newhead = head;
            head = temp;
        }
        return newhead;
    }
}
```

Intersection of Two Linked Lists

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null) return null;
        ListNode node = headA;
        while(node.next != null){
            node = node.next;
        }
        node.next = headB;
        ListNode rst = cycleFirstNode(headA);
        node.next = null;
        return rst;
    }

    private ListNode cycleFirstNode(ListNode head){
        ListNode slow = head, fast = head;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
            if(fast == slow){
                ListNode cur = head;
                while(cur != slow){
                    cur = cur.next;
                    slow = slow.next;
                }
                return cur;
            }
        }
        return null;
    }
}
```

Maximum Subarray

```
public class Solution {
    public int maxSubArray(int[] nums) {
        if(nums.length == 0) return 0;
        int max = nums[0];
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        int min = Math.min(nums[0], 0);
        for(int i = 1; i < nums.length; i++){
            dp[i] = dp[i - 1] + nums[i];
            max = Math.max(max, dp[i] - min);
            min = Math.min(min, dp[i]);
        }
        return max;
    }
}
```

Best Time to Buy and Sell Stock II

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

Sqrt(x)

```
public class Solution {
    public int mySqrt(int x) {
        if(x == 0) return 0;
        int start = 0, end = x;
        while(start + 1 < end){
            int mid = start + (end - start) / 2;
            if(x / mid == mid) return mid;
            else if(x / mid > mid) start = mid;
            else end = mid;
        }
        if(x / end == end) return end;
        else return start;
    }
}
```

Merge Intervals

```
public class Solution {
    public List<Interval> merge(List<Interval> intervals) {
        if(intervals.size() < 2) return intervals;
        List<Interval> list = new ArrayList<Interval>();
        Collections.sort(intervals, new Comparator<Interval>(){
            public int compare(Interval a, Interval b){
                return a.start - b.start;
            }
        });

        int pre_start = intervals.get(0).start, pre_end = intervals.get(0).end;
        for(int i = 1; i < intervals.size(); i++){
            if(intervals.get(i).start > pre_end){
                Interval cur = new Interval(pre_start, pre_end);
                list.add(cur);
                //Notice:
                pre_end = intervals.get(i).end > pre_end? intervals.get(i).end : pre_end;
                pre_start = intervals.get(i).start;
            }
            else{
                pre_end = intervals.get(i).end > pre_end? intervals.get(i).end : pre_end;
            }
            if(i == intervals.size() - 1){
                Interval cur = new Interval(pre_start, pre_end);
                list.add(cur);
            }
        }
        return list;
    }
}
```

Binary Tree Level Order Traversal

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

```
        return rst;
    }
}
```

Implement Queue using Stacks

```
class MyQueue {
    // Push element x to the back of queue.
    private Stack<Integer> s1 = new Stack<Integer>();
    private Stack<Integer> s2 = new Stack<Integer>();
    public void push(int x) {
        s1.push(x);
    }

    // Removes the element from in front of queue.
    public void pop() {
        if(s2.isEmpty()){
            while(!s1.isEmpty()) s2.push(s1.pop());
        }
        s2.pop();
    }

    // Get the front element.
    public int peek() {
        if(s2.isEmpty()){
            while(!s1.isEmpty()) s2.push(s1.pop());
        }
        return s2.peek();
    }

    // Return whether the queue is empty.
    public boolean empty() {
        return (s1.isEmpty() && s2.isEmpty());
    }
}
```

Find the Duplicate Number

```
public class Solution {
    public int findDuplicate(int[] nums) {
        if(nums.length < 2) return -1;
        int slow = nums[0], fast = nums[nums[0]];
        while(slow != fast){
            slow = nums[slow];
            fast = nums[nums[fast]];
        }

        int newfast = 0;
        while(slow != newfast){
            slow = nums[slow];
            newfast = nums[newfast];
        }
        return slow;
    }
```

}

Add Two Numbers

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode head = new ListNode(0);
        ListNode dummyNode = head;
        int f = 0;
        while(l1 != null && l2 != null){
            int sum = l1.val + l2.val + f;
            f = sum / 10;
            ListNode newNode = new ListNode(sum % 10);
            head.next = newNode;
            head = newNode;
            l1 = l1.next;
            l2 = l2.next;
        }
        while(l1 != null){
            int sum = l1.val + f;
            f = sum / 10;
            ListNode newNode = new ListNode(sum % 10);
            head.next = newNode;
            head = newNode;
            l1 = l1.next;
        }
        while(l2 != null){
            int sum = l2.val + f;
            f = sum / 10;
            ListNode newNode = new ListNode(sum % 10);
            head.next = newNode;
            head = newNode;
            l2 = l2.next;
        }
        if(f > 0){
            ListNode newNode = new ListNode(f);
            head.next = newNode;
            head = newNode;
        }
        return dummyNode.next;
    }
}
```

Missing Number

```
public class Solution {
    public int missingNumber(int[] nums) {
        int sum = 0;
        int z = 0;
        int max = 0;
        int n = nums.length;
        for(int i = 0; i < n; i++){
            max = max > nums[i] ? max : nums[i];
            if(nums[i] == 0) z = 1;
            sum += nums[i];
        }
        int sum_all = (max* (max + 1)) / 2;
        if(sum_all == sum){
            if(n == max + 1) return max + 1;
            else return 0;
        }
        else return (sum_all - sum);
    }
}
```

Reverse Integer

```
public class Solution {
    public int reverse(int x) {
        int flag = 1, sum = 0, i = 1;
        if(x < 0){
            flag = -1;
            x = -1 * x;
        }
        while(x > 0){
            int d = x % 10;
            x = x / 10;
            //2147483647,   1534236469
            int temp = sum;
            sum = sum * 10 + d;
            if(temp != sum / 10){
                sum = 0;
                break;
            }
        }
        return sum * flag;
    }
}
```

Reverse Words in a String

```
public class Solution {
    public String reverseWords(String s) {
        if(s.length() == 0) return "";
        String[] array = s.split(" ");
        StringBuilder sb = new StringBuilder();
        for(int i = array.length - 1; i >= 0; i--){
            //Notice equals("")
            if(!array[i].equals("")){
                sb.append(array[i]).append(" ");
            }
        }
        //String str = sb.toString();

        return sb.length() == 0? "" : sb.substring(0, sb.length() - 1);
    }
}
```

Kth Largest Element in an Array

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        //heap
        Queue<Integer> q = new PriorityQueue<>(nums.length, new Comparator<Integer>(){
          public int compare(Integer a, Integer b){
              return b - a;
          }
        });
        for(int i = 0; i < nums.length; i++){
            q.offer(nums[i]);
        }
        while((k-1) != 0){
            q.poll();
            k--;
        }
        return q.poll();
    }
}
```

Merge Sorted Array

```
public class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int index = m + n - 1, i = m - 1, j = n - 1;
        while(i >= 0 && j >= 0){
            //Notice here: no j--; i--;
            nums1[index--] = nums1[i] > nums2[j]? nums1[i--] : nums2[j--];
        }
        while(i >= 0){
            nums1[index--] = nums1[i--];
        }
        while(j >= 0){
            nums1[index--] = nums2[j--];
        }
    }
}
```

Group Anagrams

```
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> rst = new ArrayList<>();
        HashMap<Integer, List<String>> map = new HashMap<>();
        for(String str : strs){
            int hash = hashAnagram(str);
            if(map.containsKey(hash)) map.get(hash).add(str);
            else{
                List<String> list = new ArrayList<String>();
                list.add(str);
                map.put(hash, list);
            }
        }
        for(List<String> list : map.values()){
            rst.add(list);
        }
        return rst;
    }
    private int hashAnagram(String s){
        int[] arr = new int[26];
        for(int i = 0; i < s.length(); i++){
            arr[s.charAt(i) - 'a']++;
        }
        return Arrays.hashCode(arr);
    }
}
```

Longest Substring Without Repeating Characters

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        //hashmap stores the index of character
        if(s.length() < 2) return s.length();
        HashMap<Character, Integer> map = new HashMap<Character, Integer>();
        map.put(s.charAt(0), 0);
        int j = 1, max = 1;
        char c = 'x';
        for(int i = 0; i < s.length(); ){
            while(j < s.length()){
                c = s.charAt(j);
                if(map.containsKey(c)){
                    if(map.get(c) < i){
                        map.put(c, j);
                        j++;
                    }
                    else break;
                }
                else{
                    map.put(c, j);
                    j++;
                }
            }
            max = Math.max(max, j - i);
            if(map.containsKey(c)){
                int index = map.get(c);
                //Notice:
                map.put(c, j);
                i = index + 1;
                j++;
            }
        }
        return max;
    }
}
```

Implement Stack using Queues

```
public class MyStack {

    /** Initialize your data structure here. */
    Queue<Integer> q;
    public MyStack() {
        q = new LinkedList<Integer>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        q.offer(x);
        for(int i = 0; i < q.size() - 1; i++){
            q.offer(q.poll());
        }
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return q.poll();
    }

    /** Get the top element. */
    public int top() {
        return q.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return q.isEmpty();
    }
}
```

Remove Duplicates from Sorted Array

```
public class Solution {
    public int removeDuplicates(int[] nums) {
        if(nums.length == 0) return 0;
        int slow = 0, fast = 1;
        while(slow < fast && fast < nums.length){
            while(slow < fast && fast < nums.length && nums[slow] == nums[fast]) fast++;
            if(slow < fast && fast < nums.length){
                slow++;
                int temp = nums[slow];
                nums[slow] = nums[fast];
                nums[fast] = nums[slow];
                fast++;
            }
        }
        return slow + 1;
```

```
    }
}
```

Binary Tree Zigzag Level Order Traversal

```
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        if(root == null) return rst;
        q.offer(root);
        int n = 0;
        while(!q.isEmpty()){
            int size = q.size();
            List<Integer> list = new ArrayList<Integer>();
            n++;
            for(int i = 0; i < size; i++){
                TreeNode node = q.poll();
                if(n % 2 == 0) list.add(0, node.val);
                else list.add(node.val);
                if(node.left != null) q.offer(node.left);
                if(node.right != null) q.offer(node.right);
            }
            rst.add(list);
        }
        return rst;
    }
}
```

Construct Binary Tree from Preorder and Inorder Traversal

```
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int len = preorder.length;
        return buildHelper(preorder, 0, len - 1, inorder, 0, len - 1);
    }
    private TreeNode buildHelper(int[] preorder, int pre_s, int pre_e, int[] inorder,
                                 int in_s, int in_e){
        if(pre_s > pre_e) return null;
        TreeNode root = new TreeNode(preorder[pre_s]);
        int pos = findPosi(preorder[pre_s], inorder);
        TreeNode left = buildHelper(preorder, pre_s + 1, pre_s + pos - in_s, inorder, in_s, pos - 1);
        TreeNode right = buildHelper(preorder, pre_s + pos - in_s + 1, pre_e, inorder, pos + 1,
                                     in_e);
        root.left = left;
        root.right = right;
        return root;
    }
    private int findPosi(int target, int[] order){
        for(int i = 0; i < order.length; i++){
            if(order[i] == target) return i;
        }
        return -1;
```

```
    }
}
```

Word Search

```
public class Solution {
    public boolean exist(char[][] board, String word) {
        if(board.length == 0) return false;
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(dfs(board, word, i, j, 0)) return true;
            }
        }
        return false;
    }
    private boolean dfs(char[][] board, String word, int x, int y, int n){
        if(n == word.length()) return true;
        if(x < 0 || x >= board.length || y < 0 || y >= board[0].length) return false;
        if(board[x][y] == word.charAt(n)){
            char c = board[x][y];
            board[x][y] = '*';
            //不用全局变量 就用或关系 有返回值
            boolean rst = dfs(board, word, x + 1, y, n + 1) || dfs(board, word, x - 1, y, n + 1) || dfs(board, word, x, y + 1, n + 1) ||
dfs(board, word, x, y - 1, n + 1);
            board[x][y] = c;
            return rst;
        }
        else return false;
    }
}
```

Linked List Cycle

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if(head == null) return false;
        ListNode slow = head, fast = head.next;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
            if(fast == slow) return true;
        }
        return false;
    }
}
```

Longest Palindromic Substring

```
public class Solution {
    public String longestPalindrome(String s) {
        if(s.length() == 0) return s;
        int len = s.length();
        int max = 1;
        String rst = s.substring(0, 1);
        boolean[][] dp = new boolean[len][len];
        for(int i = 0; i < len; i++){
            dp[i][i] = true;
        }
        for(int i = 0; i < len - 1; i++){
            if(s.charAt(i) == s.charAt(i + 1)){
                max = 2;
                rst = s.substring(i, i + 2);
                dp[i][i + 1] = true;
            }
            else dp[i][i + 1] = false;
        }
        for(int l = 2; l < len; l++){
            for(int i = 0; i + l < len; i++){
                if(s.charAt(i) == s.charAt(i + l)){
                    dp[i][i + l] = dp[i + 1][i + l -1];
                    //Notice l + 1
                    if(dp[i][i + l] && (l + 1) > max){
                        max = l + 1;
                        rst = s.substring(i, i + l + 1);
                    }
                }
                else dp[i][i + l] = false;
            }
        }
        return rst;
    }
}
```

Unique Paths II

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        if(m == 0) return 0;
        int[][] path = new int[m][n];
        if(obstacleGrid[0][0] == 1) return 0;
        path[0][0] = 1;
        for(int i = 1; i < m; i++){
            if(obstacleGrid[i][0] == 1) path[i][0] = 0;
            else path[i][0] = path[i - 1][0];
        }
        for(int j = 1; j < n; j++){
            if(obstacleGrid[0][j] == 1) path[0][j] = 0;
            else path[0][j] = path[0][j - 1];
        }
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                if(obstacleGrid[i][j] == 1){
                    path[i][j] = 0;
                }
                else{
                    path[i][j] = path[i - 1][j] + path[i][j - 1];
                }
            }
        }
        return path[m - 1][n - 1];
    }
}
```

Roman to Integer

```
public class Solution {
    public int romanToInt(String s) {
        int pre = Integer.MAX_VALUE, rst = 0;
        for(int i = 0; i < s.length(); i++){
            int d = intNum(s.charAt(i));
            if(d > pre){
                rst += (d - 2 * pre);
            }
            else{
                rst += d;
            }
            pre = d;
        }
        return rst;
    }
    private int intNum(char roman){
        switch(roman){
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            case 'M': return 1000;
        }
        return -1;
    }
}
```

Same Tree

```
public class Solution {
    private boolean isSame;
    public boolean isSameTree(TreeNode p, TreeNode q) {
        isSame = true;
        traverse(p, q);
        return isSame;
    }
    private void traverse(TreeNode p, TreeNode q){
        if(p == null && q == null) return;
        else if(p == null || q == null) {
            isSame = false;
            return;
        }
        else{
            if(p.val != q.val){
                isSame = false;
                return;
            }
```

```
            traverse(p.left, q.left);
            traverse(p.right, q.right);
        }
    }
}
```

Word Break

```
public class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        if(s.length() == 0) return false;
        HashSet<String> set = new HashSet<String>();
        for(String str : wordDict){
            set.add(str);
        }
        int len = s.length();
        boolean[] dp = new boolean[len];
        if(set.contains(s.substring(0, 1))) dp[0] = true;
        else dp[0] = false;
        for(int i = 1; i < len; i++){
            //notice:
            if(set.contains(s.substring(0, i + 1))) dp[i] = true;
            else{
                dp[i] = false;
                for(int j = 0; j < i; j++){
                    if(dp[j] && set.contains(s.substring(j + 1, i + 1))){
                        dp[i] = true;
                        break;
                    }
                }
            }
        }
        return dp[len - 1];
    }
}
```

Rotate Array

```
public class Solution {
    public void rotate(int[] nums, int k) {
        //Notice
        k = k % nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }
    private void reverse(int[] nums, int s, int e){
        while(s <= e){
            int temp = nums[s];
            nums[s] = nums[e];
            nums[e] = temp;
            s++;
            e--;
        }
    }
}
```

3Sum

```
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> rst = new ArrayList<>();
        if(nums.length < 3) return rst;
        for(int i = 0; i < nums.length - 2; i++){
            if(i != 0 && nums[i] == nums[i - 1]) continue;
            int l = i + 1, r = nums.length - 1;
            while(l < r){
                while(l < r && (nums[l] + nums[r] + nums[i]) > 0) r--;
                while(l < r && (nums[l] + nums[r] + nums[i]) < 0) l++;
                if(l < r && (nums[l] + nums[r] + nums[i]) == 0){
                    rst.add(Arrays.asList(nums[i], nums[l], nums[r]));
                    //Notice:
                    r--;
                    l++;
                    while(l < r && nums[r] == nums[r + 1]) r--;
                    while(l < r && nums[l] == nums[l - 1]) l++;
                }
            }
        }
        return rst;
    }
}
```

H-Index

```
public class Solution {
    public int hIndex(int[] citations) {
        if(citations == null || citations.length == 0) return 0;
        Arrays.sort(citations);
        int res = 1;
        for(int i = citations.length - 1; i >= 0; i--){
            if(citations[i] >= res) res++;
            else break;
        }
        return res - 1;
    }
}
```

Search in Rotated Sorted Array

```
public class Solution {
    public int search(int[] nums, int target) {
        int start = 0, end = nums.length - 1;
        while(start + 1 < end){
            int mid = start + (end - start) / 2;
            if(nums[mid] == target) return mid;
            if(nums[start] < nums[mid]){
                if(nums[mid] < target || target < nums[start]) start = mid;
                else end = mid;
            }
            if(nums[end] > nums[mid]){
                if(nums[mid] > target || nums[end] < target) end = mid;
                else start = mid;
            }
        }
        if(nums[start] == target) return start;
        else if(nums[end] == target) return end;
        else return -1;
    }
}
```

Swap Nodes in Pairs

```
public class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode cur = head.next;
        head.next = swapPairs(head.next.next);
        cur.next = head;
        return cur;
    }
}
```

Symmetric Tree

```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;
        return isSymmetricHelper(root.left, root.right);
    }
    public boolean isSymmetricHelper(TreeNode p, TreeNode q){
        if(p == null && q == null) return true;
        else if(p == null || q == null) return false;
        else{
            if(p.val != q.val) return false;
            boolean out = isSymmetricHelper(p.left, q.right);
```

```
            boolean in = isSymmetricHelper(p.right, q.left);
            return out && in;
        }
    }
}
```

3Sum Closest

```
public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int sum3 = nums[0] + nums[1] + nums[2];
        for(int i = 0; i < nums.length - 2; i++){
            int rest = target - nums[i];
            int l = i + 1, r = nums.length - 1;
            int sum2 = nums[i + 1] + nums[i + 2];
            while(l < r){
                sum2 = (Math.abs(sum2 - rest) <= Math.abs(nums[l] + nums[r] - rest)) ? sum2 : (nums[l] + nums[r]);
                if(nums[l] + nums[r] > rest){
                    if(nums[l] > nums[r]) l++;
                    else r--;
                }
                else if(nums[l] + nums[r] < rest){
                    if(nums[l] > nums[r]) r--;
                    else l++;
                }
                else return target;
            }
            sum3 = (Math.abs(sum3 - target) <= Math.abs(sum2 + nums[i] - target)) ? sum3 : (sum2 + nums[i]);
        }
        return sum3;
    }
}
```

Balanced Binary Tree

```
public class Solution {
    public boolean isBalanced(TreeNode root) {
        int result = isBalancedHelper(root);
        if(result == -1) return false;
        else return true;
    }
    private int isBalancedHelper(TreeNode root){
        if(root == null) return 0;
        if(root.left == null && root.right == null) return 1;
        int left = isBalancedHelper(root.left);
        int right = isBalancedHelper(root.right);
        if(Math.abs(left - right) > 1 || left == -1 || right == -1) return -1;
        else return Math.max(left, right) + 1;
    }
}
```

Path Sum II

```
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> rst = new ArrayList<>();
        List<Integer> list = new ArrayList<>();
        pathSumHelper(root, rst, list, 0, sum);
        return rst;
    }
    private void pathSumHelper(TreeNode root, List<List<Integer>> rst,
                        List<Integer> list, int sum, int target){
        if(root == null) return;
        sum += root.val;
        list.add(root.val);
        if(root.left == null && root.right == null){
            if(sum == target) rst.add(new ArrayList<>(list));
            sum -= root.val;
            list.remove(list.size() - 1);
            return;
        }
        pathSumHelper(root.left, rst, list, sum, target);
        pathSumHelper(root.right, rst, list, sum, target);
        sum -= root.val;
        list.remove(list.size() - 1);
    }
}
```

Container With Most Water

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

Read N Characters Given Read4 II - Call multiple times

```
public class Solution extends Reader4 {
    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
     * @return    The number of characters read
     */
    private char[] buff4 = new char[4];
    private int readIndex = 0;
    private int writeIndex = 0;
    public int read(char[] buf, int n) {
        for(int i = 0; i < n; i++){
            if(readIndex == writeIndex){
                writeIndex = read4(buff4);
                readIndex = 0;
                if(writeIndex == 0) return i;
            }
            buf[i] = buff4[readIndex++];
        }
        return n;
    }
}
```

Kth Smallest Element in a BST

```
public class Solution {
    HashMap<TreeNode, Integer> map = new HashMap<>();
    public int kthSmallest(TreeNode root, int k) {
        int x = countChildren(root);
        while(root != null){
            if(map.containsKey(root.left)){
                if(k == map.get(root.left) + 1) return root.val;
                else if(k > map.get(root.left) + 1){
                    //Notice here:
                    k = k - map.get(root.left) - 1;
                    root = root.right;
                }
                else root = root.left;
            }
            //Notice here:
            else if(k == 1){
                return root.val;
            }
            else{
                k = k - 1;
                root = root.right;
            }
        }
        return -1;
    }

    private int countChildren(TreeNode root){
        if(root == null) return 0;
        int left = countChildren(root.left);
        int right = countChildren(root.right);
        if(!map.containsKey(root)){
            map.put(root, left + right + 1);
        }
        return left + right + 1;
    }
}
```

Factorial Trailing Zeroes

```
public class Solution {
    public int trailingZeroes(int n) {
        int num = 0;
        while(n >= 5){
            num += n / 5;
            n = n / 5;
        }
        return num;
    }
}
```

Palindrome Permutation

```
public class Solution {
    public boolean canPermutePalindrome(String s) {
        HashMap<Character, Integer> map = new HashMap<Character, Integer>();
        int odd = 0;
        for(int i = 0; i < s.length(); i++){
            char c = s.charAt(i);
            if(map.containsKey(c)) map.put(c, map.get(c) + 1);
            else map.put(c, 1);
        }
        for(Integer n : map.values()){
            if(n % 2 == 1) odd++;
        }
        if(odd <= 1) return true;
        else return false;
    }
}
```

Palindrome Partitioning

```
public class Solution {
    public List<List<String>> partition(String s) {
        int len = s.length();
        boolean[][] dp = new boolean[len][len];
        isPalindrome(s, dp);
        List<List<String>> rst = new ArrayList<>();
        List<Integer> list = new ArrayList<>();
        partitionHelper(s, dp, list, 0, rst);
        return rst;
    }

    private void isPalindrome(String s, boolean[][] dp){
        int len = s.length();
        for(int i = 0; i < len; i++){
            dp[i][i] = true;
        }
        for(int i = 0; i < len - 1; i++){
            if(s.charAt(i) == s.charAt(i + 1)){
                dp[i][i + 1] = true;
            }
            else dp[i][i + 1] = false;
        }
        for(int l = 2; l < len; l++){
            for(int i = 0; i + l < len; i++){
                if(s.charAt(i) == s.charAt(i + l)){
                    dp[i][i + l] = dp[i + 1][i + l - 1];
                }
                else dp[i][i + l] = false;
            }
        }
        return;
    }

    private void partitionHelper(String s, boolean[][] dp, List<Integer> list, int startIndex, List<List<String>> rst){
        if(startIndex == s.length()){
            int i = 0;
            List<String> solution = new ArrayList<String>();
            for(int j = 0; j < list.size(); j++){
                solution.add(s.substring(i, list.get(j) + 1));
                i = list.get(j) + 1;
            }
            rst.add(solution);
            return;
        }
        for(int i = startIndex; i < s.length(); i++){
            if(dp[startIndex][i]){
                list.add(i);
                partitionHelper(s, dp, list, i + 1, rst);
                list.remove(list.size() - 1);
            }
```

```
        }
        return;
    }
}
```

Implement Trie (Prefix Tree)

```
public class Trie {
    /** Initialize your data structure here. */
    class TrieNode{
        HashMap<Character, TrieNode> subtree;
        boolean isWord;
        public TrieNode(){
            subtree = new HashMap<Character, TrieNode>();
            isWord = false;
        }
    }

    TrieNode root;
    public Trie() {
        root = new TrieNode();
    }

    /** Inserts a word into the trie. */
    public void insert(String word) {
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(!cur.subtree.containsKey(c)){
                cur.subtree.put(c, new TrieNode());
            }
            cur = cur.subtree.get(c);
        }
        cur.isWord = true;
    }

    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(!cur.subtree.containsKey(c)){
                return false;
            }
            cur = cur.subtree.get(c);
        }
        return cur.isWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    public boolean startsWith(String prefix) {
        TrieNode cur = root;
        for(int i = 0; i < prefix.length(); i++){
            char c = prefix.charAt(i);
            if(!cur.subtree.containsKey(c)){
                return false;
            }
```

```
            cur = cur.subtree.get(c);
        }
        return true;
    }
}
```

Serialize and Deserialize Binary Tree

```
public class Codec {
    private static final String spliter = ",";
    private static final String NN = "X";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NN).append(spliter);
        } else {
            sb.append(node.val).append(spliter);
            buildString(node.left, sb);
            buildString(node.right,sb);
        }
    }
    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Deque<String> nodes = new LinkedList<>();
        nodes.addAll(Arrays.asList(data.split(spliter)));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Deque<String> nodes) {
        String val = nodes.remove();
        if (val.equals(NN)) return null;
        else {
            TreeNode node = new TreeNode(Integer.valueOf(val));
            node.left = buildTree(nodes);
            node.right = buildTree(nodes);
            return node;
        }
    }
}
```

Lexicographical Numbers

```
public List<Integer> lexicalOrder(int n) {
        List<Integer> list = new ArrayList<>(n);
        int curr = 1;
        for (int i = 1; i <= n; i++) {
            list.add(curr);
            if (curr * 10 <= n) {
                curr *= 10;
            } else if (curr % 10 != 9 && curr + 1 <= n) {
                curr++;
            } else {
                while ((curr / 10) % 10 == 9) {
                    curr /= 10;
                }
                curr = curr / 10 + 1;
            }
        }
        return list;
    }
```