LeetCode Problems

Part I. Algorithm

Single Number

Given an array of integers, every element appears twice except for one. Find that single one.
Note: Your algorithm should have a linear runtime complexity.

```
public class Solution {
    public int singleNumber(int[] A) {
        int x = 0;
        for(int i: A)
            x = x ^ i;
            // XOR algorithm, in binary, same value get 0 and different value get 1.
            // Finally the twice number all become 0 only left the single time number.
        return x;
    }
}
```

Single Number II

Given an array of integers, every element appears three times except for one. Find that single one.

```
public class Solution {
    public int singleNumber(int[] A) {
        HashMap<Integer, Integer> singleNumber = new HashMap<Integer, Integer>();
        for (int a : A){
            if (singleNumber.containsKey(a))
                singleNumber.put(a, singleNumber.get(a) + 1);
            else
            //have to write else, or the statement would compile every time, the value would always be 1
                singleNumber.put(a, 1);
        }
        for (Integer i: singleNumber.keySet()){
            if(singleNumber.get(i) != 3)
                return i;
        }
        return 0;
    }
}
```

Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

```
public class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0)
            return false;
        int diverg = 1;
        while (x / diverg >= 10){
            diverg *= 10;
        }
        while(x > 0){
            int left = x / diverg;
            int right = x % 10;
            if (left != right)
                return false;
            x = (x % diverg) / 10;
            //x lost two digits, so diverg need to divide 100
            diverg = diverg / 100;
        }
        return true;
    }
}
```

Reverse Integer

Reverse digits of an integer. Example1: x = 123, return 321 Example2: x = -123, return -321

```
public class Solution {
    public int reverse(int x) {
        int r = 0;
        while(x != 0){
            //while loop!
            r = r * 10 + x % 10;
            x = x/10;
        }
        return r;
    }
}
```

Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

```
public class Solution {
        public int romanToInt(String s) {
                HashMap<Character, Integer> romanNumber = new HashMap<Character, Integer>();
                romanNumber.put('I',1);
                romanNumber.put('V',5);
                romanNumber.put('X',10);
                romanNumber.put('L',50);
                romanNumber.put('C',100);
                romanNumber.put('D',500);
                romanNumber.put('M',1000);
                //map the roman char with integer
                int romanValue = romanNumber.get(s.charAt(s.length() - 1));
                //wanna get value, use mapVariable.get(key);
                for(int i = s.length() - 2; i >= 0; i--){
                        if(romanNumber.get(s.charAt(i + 1)) <= romanNumber.get(s.charAt(i)))
                                romanValue += romanNumber.get(s.charAt(i));
                        else
                                romanValue -= romanNumber.get(s.charAt(i));
                        //algorithm: right char >= left char, => right - left
                        //                      left char > right char, => left + right
                }
                return romanValue;
        }
}
```

Integer to Roman

Given an integer, convert it to a roman numeral.
Input is guaranteed to be within the range from 1 to 3999.

```
public class Solution {
	public String intToRoman(int num) {
		String roman[] = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
		int integer[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
		//get the max value of each region
		String romanValue = "";
		for (int i = 0; num != 0; i++){
			//every time find the max value then combine the chars.
			while (num >= integer[i]){
				num -= integer[i];
				romanValue += roman[i];
			}
		}
		return romanValue;
	}
}
```

Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

```
public class Solution {
	public int climbStairs(int n) {
		int[] way = new int[n + 1];
		way[0] = 1;
		way[1] = 1;
		for (int i = 2; i <= n; i++){
			way[i] = way[i - 1] + way[i - 2];
		}
		return way[n];
	}
}
//too slow to code as recursion
//code as dynamic processing
```

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:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

```
public class Solution {
    public int candy(int[] ratings) {
        if (ratings == null || ratings.length == 0)
            return 0;
        int[] comp1 = new int[ratings.length];
        int[] comp2 = new int[ratings.length];
        //calculate from left to right
        comp1[0] = 1;
        for (int i = 1; i < ratings.length; i++){
            if (ratings[i] > ratings[i - 1])
                comp1[i] = comp1[i - 1] + 1;
            else
                comp1[i] = 1;
        }
        //calculate from right to left
        comp2[ratings.length - 1] = 1;
        for (int i = ratings.length - 2; i >= 0; i--){
            if (ratings[i] > ratings[i + 1])
                comp2[i] = comp2[i + 1] + 1;
            else
                comp2[i] = 1;
        }
        //get the max value of two arrays
        int[] max = new int[ratings.length];
        int result = 0;
        for (int i = 0; i < ratings.length; i++){
            max[i] = Math.max(comp1[i], comp2[i]);
            result += max[i];
        }
        return result;
    }
}
```

Container With Most Water

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

```
public class Solution {
    public int maxArea(int[] height) {
        int start = 0;
        int end = height.length - 1;
        int maxV = Integer.MIN_VALUE;
        while (start < end){
            //the volumn is the less value times the distance
            int volumn = Math.min(height[start], height[end]) * (end - start);
            maxV = Math.max(maxV, volumn);
            if (height[start] < height[end])
                start ++;
            else
                end --;
        }
        return maxV;
    }
}
```

Count and Say

The count-and-say sequence is the sequence of integers beginning as follows:

1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.

11 is read off as "two 1s" or 21.

21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth sequence.

```
public class Solution {
        public String countAndSay(int n) {
                if (n <= 1)
                        return String.valueOf(1);//Returns the string representation of the int argument.
                else
                        return helper(countAndSay(n - 1));
        }
        public String helper(String s){
                int count = 1;
                int i = 0;
                StringBuffer sb = new StringBuffer();
                //Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
                while (i < s.length()){
                        int j = i + 1;
                        while (j < s.length() && s.charAt(j) == s.charAt(i)){
                                count ++;
                                j++;
                        }
                        sb.append(count);//Appends the string representation of the char argument to this 
sequence.
                        sb.append(s.charAt(i));
                        i = j;
                        count = 1;
                }
                return sb.toString();//Returns a string representing the data in this sequence
        }
}
```

Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

```
public class Solution {
	public int divide(int dividend, int divisor) {
		long a = Math.abs((long)dividend);
		long b = Math.abs((long)divisor);
		long result = 0;
		while (a >= b){
			long c = b;
			int i = 0;
			while (c <= a){
				a = a - c;
				c = c << 1;
				result += 1 << i;
				i ++;
			}
		}
		if ((dividend < 0 && divisor > 0) || (dividend > 0 && divisor < 0))
			result = -result;
		return (int)result;
	}
}
```

Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

```
public class Solution {
    public int evalRPN(String[] tokens) {
        int val = 0;
        if (tokens == null || tokens.length == 0)
            return val;
        Stack<String> num = new Stack<String>();
        String operator = "+-*/";
        for (String t: tokens){
            if (!operator.contains(t))
                num.push(t);
            else{
                int num1 = Integer.valueOf(num.pop());//string integer convert
                int num2 = Integer.valueOf(num.pop());
                switch(t){
                    case "+":
                        num.push(String.valueOf(num1 + num2));
                        break;
                    case "-":
                        num.push(String.valueOf(num2 - num1));
                        break;
                    case "*":
                        num.push(String.valueOf(num1 * num2));
                        break;
                    case "/":
                        num.push(String.valueOf(num2 / num1));
                        break;
                }
            }
        }
        val = Integer.valueOf(num.pop());
        return val;
    }
}
```

Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i]. You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations. Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

```
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        if (gas == null || gas.length == 0 || cost == null || cost.length == 0)
            return -1;
        int start = 0, sum = 0, total = 0;
        for (int i = 0; i < gas.length; i++){
            sum += gas[i] - cost[i];//instead of "="
            total += sum;
            if (sum < 0){
                start = i + 1;
                sum = 0;//total is no need to be 0 since it counts the whole value of gas and cost
            }
        }
        if (total < 0)
            return -1;
        return start;
    }
}
```

Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit. Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0. For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:
00 – 0,   01 – 1,   11 – 3,   10 – 2

```
public class Solution {
    public ArrayList<Integer> grayCode(int n) {
        ArrayList<Integer> intToGray = new ArrayList<Integer>();
        int size = (int)Math.pow(2,n);
        //must add cast in front of Math since pow returns long
        for (int i = 0; i < size; i++){
            intToGray.add(i ^ (i >> 1));
            //integer to gray code algorithm: binary i right move 1 bit, then XOR to the origional i
        }
        return intToGray;
    }
}
```

Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

```
public class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] result = new int[m][n];
        //initial the left and the top
        result[0][0] = grid[0][0];
        for (int i = 1; i < m; i++)
            result[i][0] = grid[i][0] + result[i - 1][0];
        //since it only caculate two blocks, so we have to add the number before that
        for (int j = 1; j < n; j++)
            result[0][j] = grid[0][j] + result[0][j - 1];
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                result[i][j] = grid[i][j] + Math.min(result[i - 1][j], result[i][j - 1]);
            }
        }
        return result[m - 1][n - 1];
    }
}
```

Plus One

Given a non-negative number represented as an array of digits, plus one to the number.
The digits are stored such that the most significant digit is at the head of the list.

```java
public class Solution {
	public int[] plusOne(int[] digits) {
		if (digits == null || digits.length == 0)
			return digits;
		int carry = 1;
		for (int i = digits.length - 1; i >= 0; i--){
			int answer = (digits[i] + carry) % 10;
			carry = (digits[i] + carry) / 10;
			digits[i] = answer;
			if (carry == 0)
				return digits;
		}
		//if the digits are all 9
		int[] result = new int[digits.length + 1];
		result[0] = 1;
		return result;
	}
}
```

Pow(x, n)
Implement pow(x, n).

```
public class Solution {
        public double pow(double x, int n) {
                if (n > 0)
                        return power(x, n);
                else
                        return 1 / power(x, -n);
        }
        public double power(double x, int n){
                if (n == 0)
                        return 1;
                double value = power(x, n / 2);
                if (n % 2 == 0)
                        value = value * value;
                else
                        value = value * value * x;
                return value;
        }
}//runtime O(logN)
/*
  public class Solution {
        public double pow(double x, int n) {
                if (x == 0)
                        return 0;
                if (n == 0)
                        return 1;
                double res = x;
                for (int i = 2; i <= n; i++){
                        res = res * x;
                }
                return res;
        }
  }
  //runtime O(N)
  */
```

Sqrt(x)

Implement int sqrt(int x). Compute and return the square root of x.

```
public class Solution {
	public int sqrt(int x) {
		if (x < 0)
			return -1;
		if (x == 0)
			return 0;
		double result = 0, y = 1;
		//Newton's method: y = y - f(y)/f'(y)
		//f(y) = y * y - x, when f(y) = 0, y = sqrt(x), f'(y) = 2 * y
		while (y != result){
			result = y;
			y = (y + x / y) / 2;
		}
		return (int)y;
	}
}
```

Rotate Image

You are given an n x n 2D matrix representing an image. Rotate the image by 90 degrees (clockwise).
Could you do this in-place?

```
public class Solution {
	public void rotate(int[][] matrix) {
		//in-place：原地分割法
		// a | b
		// c | d
		int n = matrix.length;
		for (int i = 0; i < n / 2; i++){
			for (int j = 0; j < Math.ceil((double)n / 2.0); j++){
				int temp = matrix[i][j];
				matrix[i][j] = matrix[n - 1 - j][i]; //c -> a
				matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j]; //d -> c
				matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i]; //b -> d
				matrix[j][n - 1 - i] = temp; //a -> b
			}
		}
	}
}
```

Unique Paths

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).
How many possible unique paths are there?

```
public class Solution {
        public int uniquePaths(int m, int n) {
                int[][] result = new int[m][n];
                //set the initial left & initial top
                for (int i = 0; i < m; i++)
                        result[i][0] = 1;
                for (int j = 0; j < n; j++)
                        result[0][j] = 1;
                for (int i = 1; i < m; i++){
                        for (int j = 1; j < n; j++){
                                //each block is the sum of two blocks on its left and top
                                result[i][j] = result[i - 1][j] + result[i][j - 1];
                        }
                }
                return result[m - 1][n - 1];
        }
}
/*
  public class Solution {
        public int uniquePaths(int m, int n) {
                if (m == 1 || n == 1)
                        return 1;
                return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
        }
  }
  runtime exceed
  */
```

Unique Paths II

Follow up for "Unique Paths": Now consider if some obstacles are added to the grids. How many unique paths would there be? An obstacle and empty space is marked as 1 and 0 respectively in the grid.
For example, There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[    [0,0,0],
     [0,1,0],
     [0,0,0] ]    The total number of unique paths is 2.
```

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int row = obstacleGrid.length;
        int col = obstacleGrid[0].length;
        int[][] newMatrix = new int[row][col];
        //define the beginning of the matrix
        if (obstacleGrid[0][0] == 1)
            newMatrix[0][0] = 0;
        else
            newMatrix[0][0] = 1;
        //build the most left and top of the matrix
        for (int i = 1; i < row; i++){
            if (obstacleGrid[i][0] == 1)
                newMatrix[i][0] = 0;
            else
                newMatrix[i][0] = newMatrix[i - 1][0];
        }
        for (int j = 1; j < col; j++){
            if (obstacleGrid[0][j] == 1)
                newMatrix[0][j] = 0;
            else
                newMatrix[0][j] = newMatrix[0][j - 1];
        }
        //same method as the unique path
        for (int i = 1; i < row; i++){
            for (int j = 1; j < col; j++){
                if (obstacleGrid[i][j] == 1)
                    newMatrix[i][j] = 0;
                else
                    newMatrix[i][j] = newMatrix[i - 1][j] + newMatrix[i][j - 1];
            }
        }
        return newMatrix[row - 1][col - 1];
    }
}
```

Part II. Tree

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced. 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.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public boolean isBalanced(TreeNode root) {
                if(root == null)
                        return true;
                if(Math.abs(height(root.left) - height(root.right)) > 1)
                        return false;
                else
                        //if not say this, one possible case maybe happen when the most left and most right subtree
are balanced and others are not
                        return isBalanced(root.left) && isBalanced(root.right);
        }
        private int height(TreeNode root){
                if (root == null)
                        return 0;
                else
                        //return the longest subtree value
                        return (Math.max(height(root.left), height(root.right))) + 1;
        }
}
```

Binary Tree Preorder/Inorder/Postoder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *        int val;
 *        TreeNode left;
 *        TreeNode right;
 *        TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> returnNumber = new ArrayList<Integer>();
        if(root == null)
            return returnNumber;
        else{
        // Preorder Code
            returnNumber.add(root.val);
            returnNumber.addAll(preorderTraversal(root.left));
            returnNumber.addAll(preorderTraversal(root.right));
            //recursion, so for the end of recursion there will be a val.
            //addAll method, add until the end of the list
        //Inorder Code
           returnNumber.addAll(inorderTraversal(root.left));
            returnNumber.add(root.val);
            returnNumber.addAll(inorderTraversal(root.right));
        //Postorder Code
           returnNumber.addAll(postorderTraversal(root.left));
            returnNumber.addAll(postorderTraversal(root.right));
            returnNumber.add(root.val);
        }
        return returnNumber;
    }
}
```

Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).
For example, Given binary tree {3,9,20,#,#,15,7},

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
                ArrayList<ArrayList<Integer>> list = new ArrayList<ArrayList<Integer>>();
                //link list to another method
                eachLevel(root, list, 1);
                return list;
        }
        private void eachLevel(TreeNode a, ArrayList<ArrayList<Integer>> list, int depth){
                if(a == null)
                        return;
                ArrayList<Integer> subList = new ArrayList<Integer>();
                subList.add(a.val);
                if(list.size() < depth)
                        //add value in a new depth
                        list.add(subList);
                else
                        // add value in the same depth
                        list.get(depth - 1).add(a.val);
                eachLevel(a.left, list, depth + 1);
                eachLevel(a.right, list, depth + 1);
        }
}
```

Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *       int val;
 *       TreeNode left;
 *       TreeNode right;
 *       TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
        ArrayList<ArrayList<Integer>> levels = new ArrayList<ArrayList<Integer>>();
        reverseLevel(root, 1, levels);
        //reverse all the things in the arraylist
        Collections.reverse(levels);
        return levels;
    }
    private void reverseLevel(TreeNode a, int height, ArrayList<ArrayList<Integer>> levels){
        ArrayList<Integer> subLevel = new ArrayList<Integer>();
        if (a == null)
            return;
        subLevel.add(a.val);
        if(height >    levels.size())
            levels.add(subLevel);
        else
            levels.get(height - 1).add(a.val);
        reverseLevel(a.left, height + 1, levels);
        reverseLevel(a.right, height + 1, levels);
    }
}
```

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between). Given binary tree {3,9,20,#,#,15,7}, returns [[3], [20, 9], [15, 17]].

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *        int val;
 *        TreeNode left;
 *        TreeNode right;
 *        TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (root == null)
            return result;
        ArrayList<TreeNode> visit = new ArrayList<TreeNode>();
        visit.add(root);
        boolean order = true;
        while (!visit.isEmpty()){
            ArrayList<TreeNode> next = new ArrayList<TreeNode>();
            ArrayList<Integer> temp = new ArrayList<Integer>();
            for (TreeNode node: visit){
                temp.add(node.val);
            }
            result.add(temp);
            for (int i = visit.size() - 1; i >= 0; i--){
                TreeNode node = visit.get(i);
                if(order){
                    if(node.right != null)
                        next.add(node.right);
                    if(node.left != null)
                        next.add(node.left);
                }
                else{
                    if(node.left != null)
                        next.add(node.left);
                    if(node.right != null)
                        next.add(node.right);
                }
            }
            //if order is the same with the initialization, then false, if not the same, then true
            order = order? false: true;
            visit = new ArrayList<TreeNode>(next);
        }
        return result;
    }
}
```

Construct Binary Tree from Inorder & Postorder / Preorder & Inorder Traversal

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

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public TreeNode buildTree(int[] inorder, int[] postorder) {
            if (inorder == null || inorder.length == 0)
                return null;
            return buildTree(inorder, 0, (inorder.length - 1), postorder, 0, (postorder.length - 1));
        }
        public TreeNode buildTree(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int
postEnd){
            if (inStart > inEnd)
                return null;
            int rootVal = postorder[postEnd];
            int index = 0;
            for (int i = inStart; i <= inEnd; i++){
                if (inorder[i] == rootVal){
                    index = i;
                    break;
                }
            }
            int length = index - inStart;
            TreeNode root = new TreeNode(rootVal);
            root.left = buildTree(inorder, inStart, index - 1, postorder, postStart, postStart + length - 1);
            root.right = buildTree(inorder, index + 1, inEnd, postorder, postStart + length, postEnd - 1);
            return root;
        }
}
```

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

```
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || preorder.length == 0)
            return null;
        return buildTree(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1);
    }
    public TreeNode buildTree(int[] preorder, int preStart, int preEnd, int[] inorder, int inStart, int
inEnd){
        if (preStart > preEnd)
            return null;
        int rootVal = preorder[preStart];
        int index = 0;
        for (int i = 0; i <= inEnd; i++){
            if (inorder[i] == rootVal){
                index = i;
                break;
            }
        }
        int length = index - inStart;
        TreeNode root = new TreeNode(rootVal);
        root.left = buildTree(preorder, preStart + 1, preStart + length, inorder, inStart, index - 1);
        root.right = buildTree(preorder, preStart + length + 1, preEnd, inorder, index + 1, inEnd);
        return root;
    }
}
```

Convert Sorted Array / List To Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *        int val;
 *        TreeNode left;
 *        TreeNode right;
 *        TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode sortedArrayToBST(int[] number, int begin, int end){
        if (begin > end)
            return null;
        int middle = (begin + end) / 2;
        TreeNode root = new TreeNode(number[middle]);
        root.left = sortedArrayToBST(number, begin, middle - 1);
        //could not replace begin as 0, or time limited exceed
        //since not every time the begin equals to 0
        root.right = sortedArrayToBST(number, middle + 1, end);
        return root;
    }
    public TreeNode sortedArrayToBST(int[] num) {
        return sortedArrayToBST(num, 0, num.length - 1);
    }
}
```

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *        int val;
 *        ListNode next;
 *        ListNode(int x) { val = x; next = null; }
 * }
 */
/**
 * Definition for binary tree
 * public class TreeNode {
 *        int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    static ListNode treeHead;
    public TreeNode sortedListToBST(ListNode head) {
        if (head == null)
            return null;
        treeHead = head;
        int length = 0;
        ListNode temp = head;
        while (temp != null){
            length++;
            temp = temp.next;
        }
        return sortedListToBST(0, length - 1);
    }
    public TreeNode sortedListToBST(int start, int end){
        if (start > end)
            return null;
        int middle = (start + end) / 2;
        //build the left node first
        TreeNode left = sortedListToBST(start, middle - 1);
        //build the root of the tree
        TreeNode root = new TreeNode(treeHead.val);
        root.left = left;
        //build the right of the tree
        treeHead = treeHead.next;
        root.right = sortedListToBST(middle + 1, end);
        return root;
    }
}
```

Maximum / Minimum Depth of Binary Tree

Given a binary tree, find its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public int maxDepth(TreeNode root) {
                if(root == null)
                        return 0;
                int lResult = maxDepth(root.left);
                int rResult = maxDepth(root.right);
                if(lResult > rResult)
                        return lResult + 1;
                else
                        return rResult + 1;
        }
}
```

Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```
public class Solution {
        public int minDepth(TreeNode root) {
                if(root == null)
                        return 0;
                if(root.left == null)
                        return 1 + minDepth(root.right);
                if(root.right == null)
                        return 1 + minDepth(root.left);
                //if no this return, no class return
                return Math.min(minDepth(root.left), minDepth(root.right)) + 1;
        }
}
```

Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public boolean isSameTree(TreeNode p, TreeNode q) {
                if(p == null && q == null)
                // wrong(p.val == null) because integer cannot be null;
                        return true;
                else if(p != null && q != null){
                        if(p.val == q.val){
                                return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
                        }
                        return false;
                }
                else
                        return false;
        }
}
```

Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,

Given n = 3, there are a total of 5 unique BST's.

```
public class Solution {
	public int numTrees(int n) {
		int unique = 1;
		for (int i = 1; i <= n; i++){
			if (n == 1|| n == 0)
				unique = 1;
			else{
				unique = 2 * (2 * i - 1) * unique / (i + 1);
				//Catalan number algorithm: c = (2n)!/((n+1)!n!)
				//Application:
					// # of different n binary search tree
					// # of different paths to n * n square
					//# of triangle in n+2 polygon
			}
		}
		return unique;
	}
}
```

Populating Next Right Pointer in Each Node

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.

```
/**
 * Definition for binary tree with next pointer.
 * public class TreeLinkNode {
 *         int val;
 *         TreeLinkNode left, right, next;
 *         TreeLinkNode(int x) { val = x; }
 * }
 */
public class Solution {
        public void connect(TreeLinkNode root) {
                if(root == null)
                        return;
                if(root.left != null)
                        root.left.next = root.right;
                if(root.right != null && root.next != null)
                        root.right.next = root.next.left;
                connect(root.left);
                connect(root.right);
        }
}//root.next = the right number in binary tree
```

Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

```
public class Solution {
    public void connect(TreeLinkNode root) {
        if(root == null){
            return;
        }
        if(root.right!=null){
            root.right.next = findNext(root.next);
        }
        if(root.left!=null){
            root.left.next = root.right==null?findNext(root.next):root.right;
        }
        connect(root.right);
        connect(root.left);
    }
    public TreeLinkNode findNext(TreeLinkNode root){
        if(root==null){
            return null;
        }else{
            TreeLinkNode iter = root;
            TreeLinkNode result = null;
            while(iter!=null){
                if(iter.left!=null){
                    result = iter.left;
                    break;
                }
                if(iter.right!=null){
                    result = iter.right;
                    break;
                }
                iter = iter.next;
            }
            return result;
        }
    }
}
```

Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *        int val;
 *        TreeNode left;
 *        TreeNode right;
 *        TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null)
            return true;//remember!!
        return isSymmetric(root.left, root.right);
    }
    public boolean isSymmetric(TreeNode a, TreeNode b){
        if(a == null)
            return b == null;
        if(b == null)
            return false;
        if(a.val != b.val)
            return false;
        if(!isSymmetric(a.left, b.right))
            return false;
        if(!isSymmetric(a.right, b.left))
            return false;
        else
            return true;
    }
}
```

Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *       int val;
 *       TreeNode left;
 *       TreeNode right;
 *       TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
    private boolean isValidBST(TreeNode root, int min, int max){
        if (root == null)
            return true;
        if (root.val <= min || root.val >= max)
            return false;
        return isValidBST(root.left, min, root.val) && isValidBST(root.right, root.val, max);
    }
}
```

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and sum = 22, return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *      int val;
 *      TreeNode left;
 *      TreeNode right;
 *      TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null)
            return false;
        if(root.left == null && root.right == null && root.val == sum)
            return true;
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
    }
}
```

Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.
For example: Given the below binary tree and sum = 22, [[5,4,11,2], [5,8,4,5]]

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public ArrayList<ArrayList<Integer>> pathSum(TreeNode root, int sum) {
                ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
                if (root == null)
                        return result;
                ArrayList<Integer> temp = new ArrayList<Integer>();
                pathSum(root, sum, result, temp);
                return result;
        }
        public void pathSum(TreeNode root, int sum, ArrayList<ArrayList<Integer>> result, ArrayList<Integer>
temp){
                if (root == null)
                        return;
                int currentVal = root.val;
                temp.add(currentVal);
                if (root.left == null && root.right == null){
                        if (sum - currentVal == 0){
                                ArrayList<Integer> current = new ArrayList<Integer>(temp);
                                result.add(current);
                        }
                }
                pathSum(root.left, sum - currentVal, result, temp);
                pathSum(root.right, sum - currentVal, result, temp);
                //time limited exceeded if without remove, what exactly did temp remove?
                temp.remove(temp.size() - 1);
        }
}
```

Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path 1->2->3 which represents the number 123. Find the total sum of all root-to-leaf numbers. The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13. Return the sum = 12 + 13 = 25.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *         int val;
 *         TreeNode left;
 *         TreeNode right;
 *         TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
        public int sumNumbers(TreeNode root) {
                if(root == null)
                        return 0;
                return getSum(root, 0, 0);
        }
        private int getSum(TreeNode node, int num, int sum){
                if(node == null)
                        return sum;
                num = num * 10 + node.val;
                if(node.left == null && node.right == null){
                        sum += num;
                        //if not return here, the sum value will be doubled.
                        return sum;
                }
                sum = getSum(node.left, num, sum) + getSum(node.right, num, sum);
                return sum;
        }
}
```

Flatten Binary Tree to Linked List
Given a binary tree, flatten it to a linked list in-place.

```
/**
 * 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> result = new Stack<TreeNode>();
                if (root == null)
                        return;
                result.push(root);
                TreeNode temp = null;
                while (!result.isEmpty()){
                        TreeNode node = result.pop();
                        if (node.right != null)
                                result.push(node.right);
                        if (node.left != null)
                                result.push(node.left);
                        if (temp != null){
                                temp.left = null;
                                temp.right = node;
                        }
                        temp = node;
                }
        }
}
```

Part III. ListNode

Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4) Output: 7 -> 0 -> 8

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *         int val;
 *         ListNode next;
 *         ListNode(int x) {
 *                 val = x;
 *                 next = null;
 *         }
 * }
 */
public class Solution {
        public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
                if (l1 == null)
                        return l2;
                if (l2 == null)
                        return l1;
                int carry = 0;
                ListNode list1 = l1;
                ListNode list2 = l2;
                ListNode result = new ListNode(0);
                ListNode temp = result;
                while (list1 != null || list2 != null){
                        if (list1 != null){
                                carry += list1.val;
                                list1 = list1.next;
                        }
                        if (list2 != null){
                                carry += list2.val;
                                list2 = list2.next;
                        }
                        temp.next = new ListNode(carry % 10);
                        temp = temp.next;
                        carry = carry / 10;
                }
                if (carry == 1){
```

```
            temp.next = new ListNode(1);
        }
        return result.next;
    }
}
```

Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.
For example,
Given 1->2->3->4, you should return the list as 2->1->4->3. Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *      int val;
 *      ListNode next;
 *      ListNode(int x) {
 *          val = x;
 *          next = null;
 *      }
 * }
 */
public class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null|| head.next == null)
            return head;
        return swapTwoInteger(head);
    }
    public ListNode swapTwoInteger(ListNode node){
        if(node == null|| node.next == null)
            return node;
        ListNode next = node.next.next; //in order to make the recursion
        ListNode temp = node;
        node = node.next;
        node.next = temp; //swap function
        //recursion
        node.next.next = swapTwoInteger(next); //recursion
        //return head
        return node; //return head
    }
}
```

Linked List Cycle

Given a linked list, determine if it has a cycle in it.

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *       int val;
 *       ListNode next;
 *       ListNode(int x) {
 *           val = x;
 *           next = null;
 *       }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if(head == null)
            return false;
        ListNode fast = head;
        ListNode slow = head;
        while(fast != null){
            //must confirm current first, then check if next exists.
            if(fast.next != null)
                fast = fast.next.next;
            else
                return false;
            slow = slow.next;
            if(slow == fast)
                return true;
        }
        return false;
    }
}
//two pointer chasing, if fast meet slow, then has a circle.
```

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *      int val;
 *      ListNode next;
 *      ListNode(int x) {
 *          val = x;
 *          next = null;
 *      }
 * }
 */
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null)
            return null;
        ListNode fast = head;
        ListNode slow = head;
        while(fast != null && fast.next != null){
            fast = fast.next.next;
            slow = slow.next;
            if(fast == slow)
                break;
        }
        //no cycle happens
        if(fast != slow)
            return null;
        ListNode answer = head;
        //A = head, B = cycle begin, C = meet point
        //fast = a + b + c + b, slow = (a + b)
        //fast = 2 * slow => a + b + c + b = 2(a + b) => a = c
        while(fast != null && answer != null){
            if(fast == answer)
                break;
            fast = fast.next;
            answer = answer.next;
        }
        return answer;
    }
}
```

Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *         int val;
 *         ListNode next;
 *         ListNode(int x) {
 *                 val = x;
 *                 next = null;
 *         }
 * }
 */
public class Solution {
        public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
                //new ListNode which point to ahead of the head of LinkedList
                ListNode head = new ListNode(0);
                ListNode mergeList = head;
                while(l1 != null && l2 != null){
                        if(l1.val < l2.val){
                                head.next = l1;
                                l1 = l1.next;
                        }
                        else{
                                head.next = l2;
                                l2 = l2.next;
                        }
                        head = head.next;
                }
                if(l1 != null){
                        head.next = l1;
                        l1 = l1.next;
                }
                if(l2 != null){
                        head.next = l2;
                        l2 = l2.next;
                }
                return mergeList.next;
        }
}
```

Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

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

```
/**
 * 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 newList = head;
        while(newList != null && newList.next != null){
            if(newList.val == newList.next.val)
                newList.next = newList.next.next;
            else
                newList = newList.next;
        }
        return head;
    }
}
```

Remove Duplicates from Sorted 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->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.

```
/**
 * 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) {
                ListNode temp = new ListNode(0);
                temp.next = head;
                head = temp;
                ListNode list1 = head;
                while (list1.next != null){
                        ListNode list2 = list1.next;
                        while (list2.next != null && list2.val == list2.next.val){
                                list2 = list2.next;
                        }
                        if (list2 != list1.next)
                                list1.next = list2.next;
                        else
                                list1 = list1.next;
                }
                return head.next;
        }
}
```

Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *       int val;
 *       ListNode next;
 *       ListNode(int x) {
 *           val = x;
 *           next = null;
 *       }
 * }
 */
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if(head == null)
            return null;
        ListNode a = head;
        ListNode b = head;
        //algorithm
        //a going to the nth list, a and b have gaps of n
        //a,b both going till a to the end of the list
        //b next is the list should be removed
        //use temp to delete that list
        for(int i = 0; i < n; i++){
            a = a.next;
        }
        if(a == null){
            head = head.next;
            return head;
        }
        while(a.next != null){
            a = a.next;
            b = b.next;
        }
        ListNode temp = b.next.next;
        b.next = temp;
        return head;
    }
}
```

Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example: Given 1->2->3->4->5->NULL and k = 2, return 4->5->1->2->3->NULL.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *        int val;
 *        ListNode next;
 *        ListNode(int x) {
 *              val = x;
 *              next = null;
 *        }
 * }
 */
public class Solution {
        public ListNode rotateRight(ListNode head, int n) {
                if (head == null)
                        return null;
                ListNode n1 = head;
                ListNode n2 = head;
                while (n > 0){
                        n2 = n2.next;
                        n --;
                        if (n2 == null)
                                n2 = head;//have to write, or n2.next could not have any node
                }
                if (n1 == n2)
                        return n1;
                while (n2.next != null){
                        n1 = n1.next;
                        n2 = n2.next;
                }
                ListNode temp = n1.next;
                n2.next = head;
                n1.next = null;
                return temp;
        }
}
```

Part IV. Array

Anagram

Given an array of strings, return all groups of strings that are anagrams.
Note: All inputs will be in lower-case.

```java
public class Solution {
	public ArrayList<String> anagrams(String[] strs) {
		ArrayList<String> result = new ArrayList<String>();
		if (strs == null || strs.length == 0)
			return result;
		HashMap<String, ArrayList<String>> anagram = new HashMap<String, ArrayList<String>>();
		for (String s: strs){
			char[] temp = s.toCharArray();
			Arrays.sort(temp);
			String sortedS = new String(temp);
			//map all the string which has the same characters
			if (anagram.containsKey(sortedS)){
				anagram.get(sortedS).add(s);
			}
			else{
				ArrayList<String> list = new ArrayList<String>();
				list.add(s);
				anagram.put(sortedS, list);
			}
		}
		Set<String> set = anagram.keySet();
		for (String s: set){
			ArrayList<String> value = anagram.get(s);
			if (value.size() > 1)
			//have to add if statement
				result.addAll(value);
		}
		return result;
	}
}
```

Two Sum

Given an array of integers, find two numbers such that they add up to a specific target number. The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based. You may assume that each input would have exactly one solution.
Input: numbers={2, 7, 11, 15}, target=9 Output: index1=1, index2=2

```
public class Solution {
	public int[] twoSum(int[] numbers, int target) {
		HashMap<Integer, Integer> temp = new HashMap<Integer, Integer>();
		int[] result = new int[2];
		for (int i = 0; i < numbers.length; i++){
			if (!temp.containsKey(numbers[i]))
				temp.put(target - numbers[i], i);
			//Associates the specified value with the specified key in this map.
			else{
				int index = temp.get(numbers[i]);
				result[0] = index + 1;
				result[1] = i + 1;
				Arrays.sort(result);
				break;
			}
		}
		return result;
	}
}
```

3Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

```
public class Solution {
    public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (num == null || num.length < 3)
            return result;
        Arrays.sort(num);
        for (int i = 0; i < num.length - 2; i++){//num.length - 2 since j = i + 1
            if (i == 0 || num[i] > num[i - 1]){//avoid duplicate solution
                int j = i + 1;
                int k = num.length - 1;
                while (j < k){
                    if (num[i] + num[j] + num[k] == 0){
                        ArrayList<Integer> temp = new ArrayList<Integer>();
                        temp.add(num[i]);
                        temp.add(num[j]);
                        temp.add(num[k]);
                        result.add(temp);
                        j ++;
                        k --;
                        while (j < k && num[k] == num[k + 1])
            //avoid two neighborsare the same, which will produce duplicate solution
                            k --;
                        while (j < k && num[j] == num[j - 1])
                            j++;
                    }
                    else if (num[i] + num[j] + num[k] > 0)
                        k --;
                    else
                        j ++;

                }
            }
        }
        return result;
    }
}
```

3Sum Closest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

```
public class Solution {
    public int threeSumClosest(int[] num, int target) {
        int result = 0;
        int min = Integer.MAX_VALUE;
        Arrays.sort(num);
        for (int i = 0; i < num.length; i++){
            int j = i + 1;
            int k = num.length - 1;
            while (j < k){
                int sum = num[i] + num[j] + num[k];
                int diff = Math.abs(sum - target);
                if (diff <= min){
                    min = diff;
                    result = sum;
                }
                //could not write as:
                //min = Math.min(min, diff);
                //result = sum;
                //since the result could only equals to sum if diff is smaller than min
                if (sum <= target)
                    j ++;
                else
                    k --;
            }
        }
        return result;
    }
}
```

4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target?

```
public class Solution {
    public ArrayList<ArrayList<Integer>> fourSum(int[] num, int target) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (num == null || num.length < 4)
            return result;
        Arrays.sort(num);
        for (int i = 0; i < num.length - 3; i++){
            if (i != 0 && num[i] == num[i - 1])
                continue;
            for (int j = i + 1; j < num.length - 2; j++){
                if (j != i + 1 && num[j] == num[j - 1])
                    continue;
                int k = j + 1;
                int l = num.length - 1;
                while (k < l){
                    int sum = num[i] + num[j] + num[k] + num[l];
                    if (sum == target){
                        ArrayList<Integer> temp = new ArrayList<Integer>();
                        temp.add(num[i]);
                        temp.add(num[j]);
                        temp.add(num[k]);
                        temp.add(num[l]);
                        result.add(temp);
                        k ++;
                        l --;
                        while (k < l && num[k] == num[k - 1])
                            k ++;
                        while (k < l && num[l] == num[l + 1])
                            l --;
                    }
                    else if (sum < target)
                        k ++;
                    else
                        l --;
                }
            }
        }
        return result;
    }
}
```

Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. The same repeated number may be chosen from C unlimited number of times. Note: All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
The solution set must not contain duplicate combinations.
For example, given candidate set 2,3,6,7 and target 7, A solution set is: [7] [2, 2, 3]

```
public class Solution {
    public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int target) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (candidates == null || candidates.length == 0)
            return result;
        Arrays.sort(candidates);
        ArrayList<Integer> temp = new ArrayList<Integer>();
        combinationSum(result, temp, candidates, target, 0, 0);
        return result;
    }
    public void combinationSum(ArrayList<ArrayList<Integer>> result, ArrayList<Integer> temp, int[]
candidates, int target, int step, int sum){
        if (sum == target){
            if (!result.contains(temp))
                result.add(new ArrayList<Integer>(temp));
            return;
        }
        if (sum > target)
            return;
        for (int i = step; i < candidates.length; i++){
            temp.add(candidates[i]);
            combinationSum(result, temp, candidates, target, i, sum + candidates[i]);
            temp.remove(temp.size() - 1);//what's that for?
        }
    }
}
```

Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. Each number in C may only be used once in the combination. Note: All numbers (including target) will be positive integers. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak). The solution set must not contain duplicate combinations. For example, given candidate set 10,1,2,7,6,1,5 and target 8, A solution set is: [1, 7] [1, 2, 5] [2, 6] [1, 1, 6]

```
public class Solution {
    public ArrayList<ArrayList<Integer>> combinationSum2(int[] num, int target) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (num == null || num.length == 0)
            return result;
        Arrays.sort(num);
        ArrayList<Integer> temp = new ArrayList<Integer>();
        combinationSum2(num, target, 0, result, temp);
        return result;
    }
    public void combinationSum2(int[] num, int target, int step, ArrayList<ArrayList<Integer>> result,
ArrayList<Integer> temp){
        if (target == 0){
            result.add(new ArrayList<Integer>(temp));
            return;
        }
        if (target < 0 || step >= num.length)
            return;
        for (int i = step; i < num.length; i++){
            //avoid the duplicate solution of array
            if (i > step && num[i] == num[i - 1])
                continue;
            temp.add(num[i]);
            combinationSum2(num, target - num[i], i + 1, result, temp);
            temp.remove(temp.size() - 1);
        }
    }
}
```

First Missing Positive

Given an unsorted integer array, find the first missing positive integer. Given [1,2,0] return 3, and [3,4,-1,1] return 2.

```
public class Solution {
    public int firstMissingPositive(int[] A) {
        Arrays.sort(A);
        int index = 0;
        int n = 1;
        while (index < A.length && A[index] <= 0){
        //have to set the index < A.length to make the boundary
            index ++;
        }
        for (int i = index; i < A.length; i++){
            if (i > 0 && A[i] == A[i - 1])//have to set i > 0 to make the boundary
                continue;//consider the duplicate solution in the array
            else if (A[i] != n)
                return n;
            else
                n++;
        }
        return n;
    }
}
```

Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
For example, given n = 3, a solution set is: "((()))", "(()())", "(())()", "()(())", "()()()"

```
public class Solution {
    public ArrayList<String> generateParenthesis(int n) {
        ArrayList<String> set = new ArrayList<String>();
        generateParenthesis(n, 0, 0, "", set);
        return set;
    }
    public void generateParenthesis(int n, int left, int right, String s, ArrayList<String> set){
        //the left parenthesis has to be large or equal to the right
        if (left < right)
            return;
        //return set when left and right equal to n
        if (left == n && right == n){
            set.add(s);
            return;
        }
        //when the left equals to n, then only add right
        if (left == n){
            generateParenthesis(n,left, right + 1, s + ")", set);
            return;
            //have to return, or it will continue running
        }
        generateParenthesis(n, left + 1, right, s + "(", set);
        generateParenthesis(n, left, right + 1, s + ")", set);
    }
}
```

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.

```
public class Solution {
    public int longestValidParentheses(String s) {
        if (s == null || s.length() == 0)
            return 0;
        int begin = -1;
        int maxLen = 0;
        Stack<Integer> temp = new Stack<Integer>();
        for (int i = 0; i < s.length(); i++){
            if (s.charAt(i) == '(')
                temp.push(i);
            else{
                if (temp.isEmpty())
                    begin = i;
                else{
                    temp.pop();
                    if (temp.isEmpty())
                        maxLen = Math.max(maxLen, i - begin);
                    else
                        maxLen = Math.max(maxLen,i - temp.peek());
                    //Looks at the object at the top of this stack without removing it from the stack.
similar to pop()
                }
            }

        }
        return maxLen;
    }
}
```

Valid Parentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. The brackets must close in the correct order, "()" and "()[]{}" are all valid but "()" and "([])" are not.

```
public class Solution {
        public boolean isValid(String s) {
                HashMap<Character, Character> valid = new HashMap<Character, Character>();
                valid.put('(',')');
                valid.put('{','}');
                valid.put('[',']');
                //Stack class represents a last-in-first-out (LIFO) stack of objects.
                char[] sToArray = s.toCharArray();
                Stack<Character> stack = new Stack<Character>();
                for (char i: sToArray){
                        if (valid.keySet().contains(i))//Returns a Set view of the keys contained in this map.
                                stack.push(i);//Pushes an item onto the top of this stack.
                        else if(valid.values().contains(i)){//Returns a Collection view of the values contained in this
map.
                                if(!stack.isEmpty() && valid.get(stack.peek()) == i)
//Returns the value to which the specified key is mapped, or null if this map contains no mapping for the
key.
//Looks at the object at the top of this stack without removing it from the stack.
                                        stack.pop();
//Removes the object at the top of this stack and returns that object as the value of this function.
                                else
                                        return false;
                        }
                }
                return stack.isEmpty();
        }
}
```

Insert Interval

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).
You may assume that the intervals were initially sorted according to their start times.
Example 1:
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].
Example 2:
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].
This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

```
/**
 * Definition for an interval.
 * public class Interval {
 *         int start;
 *         int end;
 *         Interval() { start = 0; end = 0; }
 *         Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
        public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {
                ArrayList<Interval> result = new ArrayList<Interval>();
                for (Interval i: intervals){
                        if (i.end < newInterval.start)
                                result.add(i);
                        else if (i.start > newInterval.end){
                                result.add(newInterval);
                                newInterval = i;
                        }
                        else if(i.end >= newInterval.start || i.start <= newInterval.end){
                                newInterval = new Interval(Math.min(i.start, newInterval.start), Math.max(i.end,
newInterval.end));
                        }
                }
                result.add(newInterval);//if intervals is empty
                return result;
        }
}
```

Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

Given [1,3],[2,6],[8,10],[15,18],

return [1,6],[8,10],[15,18].

```
/**
 * Definition for an interval.
 * public class Interval {
 *         int start;
 *         int end;
 *         Interval() { start = 0; end = 0; }
 *         Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
        public ArrayList<Interval> merge(ArrayList<Interval> intervals) {
                ArrayList<Interval> result = new ArrayList<Interval>();
                if (intervals == null || intervals.size() <= 1)
                        return intervals;
                Collections.sort(intervals, new IntervalComparator());//sort intervals
                Interval a = intervals.get(0);
                for (int i = 1; i < intervals.size(); i++){
                        Interval b = intervals.get(i);
                        if (a.end >= b.start){
                                a = new Interval(Math.min(a.start, b.start), Math.max(a.end, b.end));
                        }
                        else{
                                result.add(a);
                                a = b;
                        }
                }
                result.add(a);
                return result;
        }
}
class IntervalComparator implements Comparator<Interval>{
        public int compare(Interval a, Interval b){
                return a.start - b.start;
        }
        //int compare(T o1, T o2), Compares its two arguments for order. Returns a negative integer, zero, or
a positive integer as the first argument is less than, equal to, or greater than the second.
}
```

Letter Combinations of a Phone Number

Given a digit string, 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.
Input:Digit string "23" Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

```
public class Solution {
        public ArrayList<String> letterCombinations(String digits) {
                char[][] letter = {{}, {'a', 'b', 'c'}, {'d', 'e', 'f'}, {'g', 'h', 'i'}, {'j', 'k', 'l'}, {'m','n','o'}, {'p', 'q', 'r', 's'}, {'t', 'u', 
'v'}, {'w', 'x', 'y', 'z'}};
                ArrayList<String> res = new ArrayList<String>();
                letterCombinations(digits, letter, res, "");
                return res;
        }
        public void letterCombinations(String digits, char[][] letter, ArrayList<String> res, String temp){
                if (digits.length() == 0){
                        res.add(temp);
                        return;
                }
                for (int i = 0; i < letter[digits.charAt(0) - '0' - 1].length; i++){
                        letterCombinations(digits.substring(1), letter, res, temp + letter[digits.charAt(0) - '0' - 1][i]);
                }
        }
}
```

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. For example, Given height = [2,1,5,6,2,3], return 10.

```
public class Solution {
    public int largestRectangleArea(int[] height) {
        int area = 0;
        Stack<Integer> stack = new Stack<Integer>();
        for (int i = 0; i < height.length; i++){
            if (stack.isEmpty() || height[stack.peek()] < height[i])
                stack.push(i);
            else{
                int start = stack.pop();
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
                area = Math.max(area, height[start] * width);
                i--;
            }
        }
        //for the last index case
        while(!stack.isEmpty()){
            int start = stack.pop();
            int width = stack.isEmpty() ? height.length : height.length - stack.peek() - 1;
            area = Math.max(area, height[start] * width);
        }
        return area;
    }
}
//runtime: O(N)
//find the height which is smaller than the left one, then calculate the area before this height to find the max area
```

Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

```
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        int row = matrix.length;
        if (row == 0)
            return 0;
        int col = matrix[0].length;//have to write here not with the row since the boundary case should
be defined first
        int[][] ones = new int[row][col];
        //the ones matrix is used to count the continuous number of 1 in each row
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                if (matrix[i][j] == '1'){
                    if (j == 0)
                        ones[i][j] = 1;
                    else
                        ones[i][j] = ones[i][j - 1] + 1;
                }
            }
        }
        int max = 0;
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                int minHeight = i;
                int minWidth = ones[i][j];
                while(minHeight >= 0){
                    //have to consider the upper row value
                    minWidth = Math.min(minWidth, ones[minHeight][j]);
                    int area = minWidth * (i - minHeight + 1);
                    max = Math.max(max, area);
                    minHeight--;
                }
            }
        }
        return max;
    }
}
```

N Queens

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

```
public class Solution {
    public ArrayList<String[]> solveNQueens(int n) {
        ArrayList<String[]> result = new ArrayList<String[]>();
        int[] location = new int[n]; //use array to present the location of Q
        solveNQueens(n, result, location, 0);
        return result;
    }
    public void solveNQueens(int n, ArrayList<String[]> result, int[] location, int current){
        if (current == n)
            printBoard(location, result, n);
        else{
            for (int i = 0; i < n; i++){
                location[current] = i;
                if (isValid(location, current))
                    solveNQueens(n, result, location, current + 1);
            }
        }
    }
    // N queens rule: no two queens share the same row, column, or diagonal
    public boolean isValid(int[] location, int current){
        for (int i = 0; i < current; i++){
            if (location[i] == location[current] || Math.abs(location[current] - location[i]) == (current - i))
                return false;
        }
        return true;
    }
    public void printBoard(int[] location, ArrayList<String[]> result, int n){
        String[] ans = new String[n];
        for (int i = 0; i < n; i++){
            String row = new String();
            for (int j = 0; j < n; j++){
                if (location[i] == j)
                    row += "Q";
                else
                    row += ".";
            }
            ans[i] = row;
        }
        result.add(ans);
    }
}
```

N Queens II

Follow up for N-Queens problem. Return the total number of distinct solutions.

```
public class Solution {
    int count;
    public int totalNQueens(int n) {
        count = 0;
        int[] location = new int[n];
        totalNQueens(n, location, 0);
        return count;
    }
    public void totalNQueens(int n, int[] location, int current){
        if (current == n){
            count ++;
            return;
        }
        else{
            for (int i = 0; i < n; i++){
                location[current] = i;
                if (isValid(location, current))
                    totalNQueens(n, location, current + 1);
            }
        }
    }
    public boolean isValid(int[] location, int current){
        for (int i = 0; i < current; i++){
            if (location[i] == location[current] || Math.abs(location[i] - location[current]) == (current - i))
                return false;
        }
        return true;
    }
}
```

Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers. If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order). The replacement must be in-place, do not allocate 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

```
public class Solution {
    public void nextPermutation(int[] num) {
        if (num.length <= 1)
            return;
        for (int i = num.length - 2; i >= 0; i--){
            if (num[i] < num[i + 1]){
                int j;
                for (j = num.length - 1; j >= i + 1; j--){
                    if (num[i] < num[j])
                        break;
                }
                //swap two integers in binary, using XOR
                num[i] = num[i] ^ num[j];
                num[j] = num[i] ^ num[j];
                num[i] = num[i] ^ num[j];
                Arrays.sort(num, i + 1, num.length);
        //Sorts the specified range of the array into ascending order.(int fromIndex, int toIndex)
                return;
            }
        }
        for (int i = 0; i < num.length / 2; i++){
            int temp = num[i];
            num[i] = num[num.length - i - 1];
            num[num.length - i - 1] = temp;
        }
        return;
    }
}
```

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. For example, given s = "aab", Return [["aa","b"], ["a","a","b"]]

```
public class Solution {
	public ArrayList<ArrayList<String>> partition(String s) {
		ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();
		if (s == null || s.length() == 0)
			return result;
		ArrayList<String> temp = new ArrayList<String>();
		partition(s, result, temp, 0);
		return result;
	}
	public void partition(String s, ArrayList<ArrayList<String>> result, ArrayList<String> temp, int start){
		if (start == s.length()){
			result.add(new ArrayList<String>(temp));
			return;
		}
		for (int i = start + 1; i <= s.length(); i++){
			String subString = s.substring(start, i);
			if (isPalindrome(subString)){
				temp.add(subString);
				partition(s.substring(i), result, temp, start);
				temp.remove(temp.size() - 1);
			}
		}
	}
	public boolean isPalindrome(String s){
		int left = 0, right = s.length() - 1;
		while (left < right){
			if (s.charAt(left) != s.charAt(right))
				return false;
			left ++;
			right --;
		}
		return true;
	}
}
```

Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome.Return the minimum cuts needed for a palindrome partitioning of s. For example, given s = "aab", Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

```
public class Solution {
	public int minCut(String s) {
		int min = 0;
		if (s == null || s.length() == 0)
			return min;
		int[] cuts = new int[s.length() + 1];
		//s.substring(i, j) is palindrome if matrix[i][j] == true
		boolean[][] matrix = new boolean[s.length()][s.length()];
		for (int i = 0; i < s.length(); i++)
			//the max number of cuts for each s.substring(i)
			cuts[i] = s.length() - i;
		for (int i = s.length() - 1; i >= 0; --i){
			for (int j = i; j < s.length(); ++j){
				if ((s.charAt(i) == s.charAt(j) && (j - i < 2)) || (s.charAt(i) == s.charAt(j) && matrix[i +
1][j - 1])){
					//two situations
					//1: i and j are neighbors or the same
					//2: the chars between i and j are also palindrome
					matrix[i][j] = true;
					cuts[i] = Math.min(cuts[i], cuts[j + 1] + 1);
				}
			}
		}
		min = cuts[0];
		return min - 1;
	}
}
```

Pascal's Triangle

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5,

Return

```
[           [1],
           [1,1],
          [1,2,1],
        [1,3,3,1],
      [1,4,6,4,1] ]
```

```
public class Solution {
        public ArrayList<ArrayList<Integer>> generate(int numRows) {
                ArrayList<ArrayList<Integer>> list = new ArrayList<ArrayList<Integer>>();
                for (int i = 0; i < numRows; i++){
                        ArrayList<Integer> subList = new ArrayList<Integer>();
                        // the most left one is always 1
                        subList.add(1);
                        if (i > 0){
                                for (int j = 0; j < list.get(i - 1).size() - 1; j++){
                                        //add the upper subList's two integers
                                        subList.add(list.get(i - 1).get(j) + list.get(i - 1).get(j + 1));
                                }
                                // the most right one is always 1
                                subList.add(1);
                        }
                        list.add(subList);
                }
                return list;
        }
}
```

Pascal’s Triangle II

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3,

Return [1,3,3,1].

```
public class Solution {
        public ArrayList<Integer> getRow(int rowIndex) {
                ArrayList<Integer> result = new ArrayList<Integer>();
                if (rowIndex < 0)
                        return result;
                result.add(1);
                for (int i = 1; i <= rowIndex; i++){
                        for (int j = i - 1; j > 0; j--){
                                //.set(int index, e element)
                                //Replaces the element at the specified position in this list with the specified element.
                                result.set(j, result.get(j) + result.get(j - 1));
                        }
                        result.add(1);
                }
                return result;
        }
}
```

Permutations

Given a collection of numbers, return all possible permutations.

For example, [1,2,3] have the following permutations: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

```
public class Solution {
    public ArrayList<ArrayList<Integer>> permute(int[] num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        permute(num, 0, result);
        return result;
    }
    private void permute(int[] num, int start, ArrayList<ArrayList<Integer>> result){
        if (start >= num.length){
            ArrayList<Integer> item = temp(num);
            result.add(item);
        }
        for (int j = start; j < num.length; j++){
            swap(num, start, j);
            permute(num, start + 1, result);
            swap(num, start, j);
        }
    }
    private ArrayList<Integer> temp(int[] num){
        ArrayList<Integer> test = new ArrayList<Integer>();
        for (int n: num)
            test.add(n);
        return test;
    }
    private void swap(int[] array, int start, int end){
        int temp = array[start];
        array[start] = array[end];
        array[end] = temp;
    }
}
```

Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.
For example, [1,1,2] have the following unique permutations: [1,1,2], [1,2,1], and [2,1,1].

```
public class Solution {
    public ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        permuteUnique(num, 0, result);
        return result;
    }
    private void permuteUnique(int[] num, int start, ArrayList<ArrayList<Integer>> result){
        if (start >= num.length){
            ArrayList<Integer> item = temp(num);
            result.add(item);
        }
        for (int j = start; j < num.length; j++){
            if (containDuplicate(num, start, j)){
                swap(num, start, j);
                permuteUnique(num, start + 1, result);
                swap(num, start, j);
            }
        }
    }
    private ArrayList<Integer> temp(int[] num){
        ArrayList<Integer> test = new ArrayList<Integer>();
        for (int n: num){
            test.add(n);
        }
        return test;
    }
    private boolean containDuplicate(int[] array, int start, int end){
        for (int i = start; i < end; i++){
            if (array[i] == array[end])
                return false;
        }
        return true;
    }
    private void swap(int[] array, int start, int end){
        int temp = array[start];
        array[start] = array[end];
        array[end] = temp;
    }
}
```

Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear 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 class Solution {
	public int removeDuplicates(int[] A) {
		if (A.length <= 1)
			return A.length;
		int i = 0;
		for (int j = 1; j < A.length; j++){
			if(A[i] != A[j])
				A[++i] = A[j];
		}
		//i is the ith in the array, i + 1 is the length
		return i + 1;
	}
}
```

Remove Duplicates from Sorted Array II

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].

```
public class Solution {
	public int removeDuplicates(int[] A) {
		if(A.length <= 2)
			return A.length;
		int current = 2;
		int previous = 1;
		while(current < A.length){
			if(!(A[current] == A[previous] && A[current] == A[previous - 1]))
				A[++ previous] = A[current ++];
			else
				current ++;
		}
		return previous + 1;
	}
}
```

Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties: Integers in each row are sorted from left to right. The first integer of each row is greater than the last integer of the previous row.

For example, Consider the following matrix:

```
[   [1,   3,  5,  7],
    [10, 11, 16, 20],
    [23, 30, 34, 50] ]
```

Given target = 3, return true.

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return false;
        int start = 0, end = matrix.length * matrix[0].length - 1;
        while (start <= end){
            int middle = (start + end) / 2;
            //find the position of middle in the matrix
            int midRow = middle / matrix[0].length;
            int midCol = middle % matrix[0].length;
            //check if the target is less / equal / larger to middle
            if (matrix[midRow][midCol] == target)
                return true;
            else if (matrix[midRow][midCol] < target)
                start = middle + 1;
            else
                end = middle - 1;
        }
        //if there is no target value in the matrix, return false
        return false;
    }
}
```

Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.
For 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 {
        public ArrayList<Integer> spiralOrder(int[][] matrix) {
                ArrayList<Integer> spiral = new ArrayList<Integer>();
                if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
                        return spiral;
                int row = matrix.length, col = matrix[0].length;
                int x = 0, y = 0;
                while (row > 0 && col > 0){
                        if (row == 1){
                                for (int i = 0; i < col; i++)
                                        spiral.add(matrix[x][y++]);
                                break;
                        }
                        if (col == 1){
                                for (int i = 0; i < row; i++)
                                        spiral.add(matrix[x++][y]);
                                break;
                        }
                        for (int i = 0; i < col - 1; i++)
                                spiral.add(matrix[x][y++]);
                        for (int i = 0; i < row - 1; i++)
                                spiral.add(matrix[x++][y]);
                        for (int i = 0; i < col - 1; i++)
                                spiral.add(matrix[x][y--]);
                        for (int i = 0; i < row - 1; i++)
                                spiral.add(matrix[x--][y]);
                        x ++;
                        y ++;
                        row -= 2;
                        col -= 2;
                }
                return spiral;
        }
}
```

Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

For example,

Given n = 3,

You should return the following matrix:

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

```
public class Solution {
        public int[][] generateMatrix(int n) {
                if (n < 0)
                        return null;
                int[][] result = new int[n][n];
                int start = 1, x = 0, y = 0;
                for (int i = n; i >= 0; i -= 2){
                        if (i == 1)
                                result[x][y] = start;
                        else{
                                for (int j = 0; j < i - 1; j++)
                                        result[x][y++] = start++;
                                for (int j = 0; j < i - 1; j++)
                                        result[x++][y] = start++;
                                for (int j = i - 1; j > 0; j--)
                                        result[x][y--] = start++;
                                for (int j = i - 1; j > 0; j--)
                                        result[x--][y] = start++;
                        }
                        x++;
                        y++;
                }
                return result;
        }
}
```

Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of O(log n). If the target is not found in the array, return [-1, -1].
For example, Given [5, 7, 7, 8, 8, 10] and target value 8, return [3, 4].

```
public class Solution {
	public int[] searchRange(int[] A, int target) {
		if (A == null || A.length == 0)
			return null;
		int[] result = new int[2];
		result[0] = searchRange(A, target, 0, A.length - 1);
		result[1] = searchRange(A, target + 1, 0, A.length - 1);
		if (result[0] == result[1]){
			result[0] = -1;
			result[1] = -1;
		}
		else
			result[0] ++;
		return result;
	}
	public int searchRange(int[] A, int target, int start, int end){
		if (start == end)
			return A[start] < target ? start : start - 1;
		if (start == end - 1)
			return A[end] < target ? end : (A[start] < target ? start : start - 1);
		int middle = (start + end) / 2;
		if (A[middle] >= target)
			end = middle - 1;
		else
			start = middle;
		return searchRange(A, target, start, end);
	}
}
```

Search In Rotated Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). You are given a target value to search. If found in the array return its index, otherwise return -1. No duplicate exists in the array.

```
public class Solution {
	public int search(int[] A, int target) {
		int result = -1;
		if(A == null || A.length == 0)
			return result;
		for(int i = 0; i < A.length; i++){
			if (A[i] == target){
				result = i;
				break;
			}
		}
		return result;
	}
}//run-time: O(N)

public class Solution {
	public int search(int[] A, int target) {
		int left = 0, right = A.length - 1;
		while (left <= right){
			int middle = (left + right) / 2;
			if (target == A[middle])
				return middle;
			if (A[middle] < A[right]){
				if (target > A[middle] && target <= A[right])
					left = middle + 1;
				else
					right = middle - 1;
			}
			else{
				if (target < A[middle] && target >= A[left])
					right = middle - 1;
				else
					left = middle + 1;
			}
		}
		return -1;
	}
}//run-time: O(logN)
```

Search In Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed? Would this affect the run-time complexity? How and why? Write a function to determine if a given target is in the array.

```
public class Solution {
	public boolean search(int[] A, int target) {
		if (A == null || A.length == 0)
			return false;
		for (int i = 0; i < A.length; i++){
			if (A[i] == target){
				return true;
			}
		}
		return false;
	}
}//run-time: O(N)

public class Solution {
	public boolean search(int[] A, int target) {
		int left = 0, right = A.length - 1;
		while (left <= right){
			int middle = (left + right) / 2;
			if (target == A[middle])
				return true;
			if (A[middle] > A[left]){
				if (A[middle] > target && A[left] <= target)
					right = middle - 1;
				else
					left = middle + 1;
			}
			else if (A[middle] < A[left]){
				if (A[middle] < target && A[right] >= target)
					left = middle + 1;
				else
					right = middle - 1;
			}
			else
				left ++;
		}
		return false;
	}
}//run-time: O(N)
```

Subsets

Given a set of distinct integers, S, return all possible subsets.

Note: Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

```
public class Solution {
    public ArrayList<ArrayList<Integer>> subsets(int[] S) {
        ArrayList<ArrayList<Integer>> subset = new ArrayList<ArrayList<Integer>>();
        if (S == null || S.length == 0)
            return subset;
        Arrays.sort(S);//Sorts the specified array into ascending numerical order.
        for (int i = 0; i < S.length; i++){
            ArrayList<ArrayList<Integer>> temp = new ArrayList<ArrayList<Integer>>();
            for (ArrayList<Integer> a: subset){
                temp.add(new ArrayList<Integer>(a));
            }
            for (ArrayList<Integer> a: temp){
                a.add(S[i]);
            }
            ArrayList<Integer> single = new ArrayList<Integer>();
            single.add(S[i]);
            temp.add(single);
            //could not just add S[i] since S[i] is int, but temp is ArrayList<Integer>
            subset.addAll(temp);
        }
        subset.add(new ArrayList<Integer>());
        return subset;
    }
}
```

Subsets II

Given a collection of integers that might contain duplicates, S, return all possible subsets.

```
public class Solution {
	public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
		ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
		if (num == null || num.length == 0)
			return result;
		Arrays.sort(num);
		ArrayList<ArrayList<Integer>> temp = new ArrayList<ArrayList<Integer>>();
		for (int i = num.length - 1; i >= 0; i--){
			//get the existing array in result
			if (i == num.length - 1 || num[i] != num[i + 1] || temp.size() == 0){
				temp = new ArrayList<ArrayList<Integer>>();
				for (int j = 0; j < result.size(); j++){
					temp.add(new ArrayList<Integer>(result.get(j)));
				}
			}
			//add one element to temp
			for (ArrayList<Integer> a: temp)
				a.add(0, num[i]);
			//add the new single number to the temp
			if (i == num.length - 1 || num[i] != num[i + 1]){
			//have to write "i == num.length - 1" at first, or the runtime will be exceeded
				ArrayList<Integer> test = new ArrayList<Integer>();
				test.add(num[i]);
				temp.add(test);
			}
			//add all the arrays in the temp into result
			for (ArrayList<Integer> a: temp){
				result.add(new ArrayList(a));
			}
		}
		//add the empty block at the end of the arrayList
		result.add(new ArrayList<Integer>());
		return result;
	}
}
```

Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below. For example, given the following triangle

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

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

```java
public class Solution {
        public int minimumTotal(List<List<Integer>> triangle) {
                int row = triangle.size();
                int[] min = new int[row];
                //get the buttom array
                for (int i = 0; i < triangle.get(row - 1).size(); i++){
                        min[i] = triangle.get(row - 1).get(i);
                }
                //compare the neighbor two integer and pick the smaller one then add to the top level
                for (int i = row - 2; i >= 0; i--){
                        for (int j = 0; j < triangle.get(i).size(); j++){
                                min[j] = triangle.get(i).get(j) + Math.min(min[j], min[j + 1]);
                        }
                }
                return min[0];
        }
}
```

Valid Sudoko

Determine if a Sudoku is valid. The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

```
public class Solution {
    public boolean isValidSudoku(char[][] board) {
        //test whether each row is valid or not
        for (int i = 0; i < board.length; i++){
            HashSet<Character> test = new HashSet<Character>();
            for (int j = 0; j < board[0].length; j++){
                if (board[i][j] != '.' && !test.add(board[i][j]))
         //Adds the specified element to this set if it is not already present.return boolean
         //could not use contains, which could just check if it is already present, but could not add into it
                    return false;
            }
        }
        //test whether each column is valid or not
        for (int i = 0; i < board[0].length; i++){
            HashSet<Character> test = new HashSet<Character>();
            for (int j = 0; j < board.length; j++){
                if (board[j][i] != '.' && !test.add(board[j][i]))
                    return false;
            }
        }
        //test whether each block is valid or not
        for (int i = 0; i < 3; i ++){//could not use i += 3
            for (int j = 0; j < 3; j ++){
                HashSet<Character> test = new HashSet<Character>();
                for (int k = i * 3; k < i * 3 + 3; k++){
                    for (int l = j * 3; l < j * 3 + 3; l++){
                        if (board[k][l] != '.' && !test.add(board[k][l]))
                            return false;
                    }
                }
            }
        }
        return true;
    }
}
```

Sudoko Solver

Write a program to solve a Sudoku puzzle by filling the empty cells. Empty cells are indicated by the character '.'.

```
public class Solution {
    public void solveSudoku(char[][] board) {
        solveSudoku(board, 0, 0);
    }
    public boolean solveSudoku(char[][] board, int i, int j){
        if (j >= 9)
            return solveSudoku(board, i + 1, 0);
        if (i == 9)
            return true;
        if (board[i][j] == '.'){
            for (int k = 1; k <= 9; k++){
                board[i][j] = (char)(k + '0');
                if (isValid(board, i, j)){
                    if (solveSudoku(board, i, j + 1))
                        return true;
                }
                board[i][j] = '.';
            }
        }
        else{
            return solveSudoku(board, i, j + 1);
        }
        return false;
    }
    public boolean isValid(char[][] board, int i, int j){
        for (int k = 0; k < 9; k++){
            if (k != j && board[i][k] == board[i][j])
                return false;
        }
        for (int k = 0; k < 9; k++){
            if (k != i && board[k][j] == board[i][j])
                return false;
        }
        for (int row = i / 3 * 3; row < i / 3 * 3 + 3; row++){
            for (int col = j / 3 * 3; col < j / 3 * 3 + 3; col++){
                if (row != i && col != j && board[row][col] == board[i][j])
                    return false;
            }
        }
        return true;
    }
}
```

Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum. For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6.

```
public class Solution {
	public int maxSubArray(int[] A) {
		int sum = 0;
		int maxSum = Integer.MIN_VALUE;
		for(int i = 0; i < A.length; i++) {
			sum += A[i];
			maxSum = Math.max(maxSum, sum);
			// make sure the begin of the subArray is positive
			if(sum < 0) sum = 0;
		}
		return maxSum;
	}
}
```

Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

```
public class Solution {
	public void merge(int A[], int m, int B[], int n) {
		int i = m - 1;
		int j = n - 1;
		int k = m + n - 1;
		while(k >= 0){
			//A has enough space, so j may become 0
			if(j < 0 || i >= 0 && A[i] > B[j])
				A[k--] = A[i--];
			else
				A[k--] = B[j--];
		}
	}
}
```

Remove Element

Given an array and a value, remove all instances of that value in place and return the new length. The order of elements can be changed. It doesn't matter what you leave beyond the new length.

```
public class Solution {
    public int removeElement(int[] A, int elem) {
        if(A.length == 0)
            return 0;
        int j = 0;
        for(int i: A){
            if(i != elem){
                A[j] = i;
                j++;
            }
        }
        return j;
    }
}
```

Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue. Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

```
public class Solution {
    public void sortColors(int[] A) {
        if(A == null)
            return;
        for(int i = 1; i < A.length; i++){
            for(int j = 0; j < i; j++){
                if(A[i] < A[j]){
                    int temp = A[i];
                    A[i] = A[j];
                    A[j] = temp;
                    continue;
                }
            }
        }
    }
}
```

Search Insert Position

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.
Here are few examples.
[1,3,5,6], 5 → 2   [1,3,5,6], 2 → 1   [1,3,5,6], 7 → 4   [1,3,5,6], 0 → 0

```
public class Solution {
    public int searchInsert(int[] A, int target) {
        int index = A.length;
        for(int i = 0; i < A.length; i++){
            if ((A[i] == target) || (target < A[i])){
                index = i;
                break;
            }
        }
        return index;
    }
}
//runtime:N

    public int searchInsert(int[] A, int target) {
        if(A == null)
            return -1;
        int low = 0, high = A.length - 1;
        while(low <= high){
            int m = (low + high) / 2;
            if(A[m] == target)
                return m;
            if(A[m] > target)
                high = m - 1;
            else low = m + 1;
        }
        return low;
    }
//runtime:logN
```

Set Matrix Zeroes

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

```
public class Solution {
    public void setZeroes(int[][] matrix) {
        //check if the first row or column has 0
        boolean firstRow = false;
        boolean firstCol = false;
        for (int i = 0; i < matrix.length; i++){
            if (matrix[i][0] == 0){
                firstRow = true;
                break;
            }
        }
        for (int j = 0; j < matrix[0].length; j++){
            if (matrix[0][j] == 0){
                firstCol = true;
                break;
            }
        }
        //mark all the 0 in the matrix to the 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;
                }
            }
        }
        //set all the marked elements into 0
        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 the whole first row or column to 0 if they has 0 at original
        if (firstRow){
            for (int i = 0; i < matrix.length; i++){
                matrix[i][0] = 0;
            }
        }
        if (firstCol){
            for (int i = 0; i < matrix[0].length; i++){
                matrix[0][i] = 0;
            }
        }
    }
}
```

Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.
If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null)
            return 0;
        //choose the max value, any number compare to it is smaller
        int min = Integer.MAX_VALUE;
        int max = 0;
        for (int i: prices){
            min = Math.min(min, i);
            max = Math.max(max, (i - min));
        }
        return max;
    }
}
```

Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

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

Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.
Design an algorithm to find the maximum profit. You may complete at most two transactions.

```
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0)
            return 0;
        int[] left = new int[prices.length];
        int[] right = new int[prices.length];
        int max = 0;
        maxProfit(prices, left, right);
        for (int i = 0; i < prices.length; i++){
            max = Math.max(max, left[i] + right[i]);
        }
        return max;
    }
    public void maxProfit(int[] prices, int[] left, int[] right){
        //for the left part
        left[0] = 0;
        int min = prices[0];
        for (int i = 1; i < prices.length; i++){
            left[i] = Math.max(left[i - 1], prices[i] - min);
            min = Math.min(min, prices[i]);
        }
        //for the right part
        right[prices.length - 1] = 0;
        int max = prices[right.length - 1];
        for (int i = prices.length - 2; i >= 0; i--){
            right[i] = Math.max(right[i + 1], max - prices[i]);
            max = Math.max(max, prices[i]);
        }
    }
}
//找寻一个点i，将原来的price[0..n-1]分割为price[0..i]和price[i..n-1]，分别求两段的最大profit
```

Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index.

For example:

A = [2,3,1,1,4], return true. A = [3,2,1,0,4], return false.

```
public class Solution {
        public boolean canJump(int[] A) {
                //base case: length <= 1
                //base case: A[0] >= length
                if(A.length <= 1 || A[0] >= A.length)
                        return true;
                //base case: A[0] == 0
                else if(A[0] == 0)
                        return false;
                //set the maxLength see if can get to the final index
                int maxLength = A[0];
                for(int i = 1; i < A.length; i++){
                        //have to be >=, or [2 0 0] is wrong
                        if(maxLength >= i && (i + A[i]) >= (A.length - 1))
                                return true;
                        if((i + A[i] > maxLength))
                                maxLength = i + A[i];
                        //have to be <=, or [2 0 1] is wrong
                        if(maxLength <= i && A[i] == 0)
                                return false;
                }
                return true;
        }
}
```

Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
For example: Given array A = [2,3,1,1,4]
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

```
public class Solution {
    public int jump(int[] A) {
        if(A == null || A.length == 0 || A.length == 1){
            return 0;
        }
        int start = 0, end = 0, step = 0, max = 0;
        while (end < A.length){
            max = 0;
            step ++;
            for (int i = start; i <= end; i++){
                if ((A[i] + i) >= (A.length - 1)){
                    return step;
                }
                if((A[i] + i) > max){
                    max = A[i] + i;
                }
            }
            start = end + 1;
            end = max;
        }
        return step;
    }
}
/*
 public class Solution {
    public int jump(int[] A) {
        if (A== null || A.length == 0)
            return 0;
        int[] max = new int[A.length];
        for (int i = 0; i < A.length; i++){
            max[i] = A[i] + i;
        }
        int start = 0, end = A.length - 1, step = 0;
        while(end > 0) {
            step ++;
```

```
                for (int i = start; i <= end; i++) {
                        if(max[i] >= (A.length-1)) {
                                end     = i;
                                break;
                        }
                }
        }
        return step;
    }
}
*/
```

Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2], The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

```
public class Solution {
    public int longestConsecutive(int[] num) {
        //not increase the runtime, just increase the space when using the HashSet
        HashSet<Integer> numSeq = new HashSet<Integer>();
        for(int i: num){
            numSeq.add(i);
        }
        int max = 1;
        for(int i: num){
            int less = i - 1;
            int more = i + 1;
            int count = 1;
            while(numSeq.contains(less)){
                count++;
                numSeq.remove(less);
                less--;
            }
            while(numSeq.contains(more)){
                count++;
                numSeq.remove(more);
                more++;
            }
            max = Math.max(max, count);
        }
        return max;
    }
}
//cannot use sort function since runtime is O(NlogN)
```

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.

For example,   Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

```
public class Solution {
        public int trap(int[] A) {
                if(A == null || A.length <= 2)
                        return 0;
                //left to right to fill the blank
                int max = A[0];
                int index = 0;
                int[] fixTrap = new int[A.length];
                for (int i = 0; i < A.length; i++){
                        //i must be 0 instead of 1, or fixTrap will lost the first value
                        if(A[i] > max){
                                max = A[i];
                                index = i;
                        }
                        fixTrap[i] = max;
                }
                //max to right to delete the over calucated space
                int maxRight = A[A.length - 1];
                for (int j = A.length - 1; j > index; j--){
                        if (A[j] > maxRight)
                                maxRight = A[j];
                        fixTrap[j] = maxRight;
                }
                //difference between original and fixed figure
                int sum = 0;
                for (int k = 0; k < A.length; k++){
                        sum += fixTrap[k] - A[k];
                }
                return sum;
        }
}
```

Word Search

Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once. For example,

```
Given board = [    ["ABCE"],
                   ["SFCS"],
                   ["ADEE"] ]
```

word = "ABCCED", -> returns true, word = "SEE", -> returns true, word = "ABCB", -> returns false.

```
public class Solution {
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || word == null || word.length() == 0)
            return false;
        boolean [][] check=new boolean[board.length][board[0].length];
        for (int row = 0; row < board.length; row++){
            for (int col = 0; col < board[0].length; col++){
                if (exist(board, word, check, 0, row, col))
                    return true;
            }
        }
        return false;
    }
    public boolean exist(char[][] board, String word, boolean[][] check, int i, int row, int col){
        if (board[row][col] != word.charAt(i) || check[row][col])
            return false;
        check[row][col] = true;
        if (i == word.length() - 1)
            return true;

        if (row - 1 >= 0 && exist(board, word, check, i + 1, row - 1, col))
            return true;
        else if (row + 1 <= board.length - 1 && exist(board, word, check, i + 1, row + 1, col))
            return true;
        else if (col - 1 >= 0 && exist(board, word, check, i + 1, row, col - 1))
            return true;
        else if (col + 1 <= board[0].length - 1 && exist(board, word, check, i + 1, row, col + 1))
            return true;
        else{
            check[row][col] = false;
        }
        return false;
    }
}
```

Part IV. String

Add Binary

Given two binary strings, return their sum (also a binary string).

For example,

a = "11" b = "1" Return "100".

```
public class Solution {
	public String addBinary(String a, String b) {
		if (a == null)
			return b;
		if (b == null)
			return a;
		else{
			StringBuilder result = new StringBuilder();
			//StringBuilder function; could insert char at any place
			int aLength = a.length() - 1;
			int bLength = b.length() - 1;
			int carry = 0;

			while (aLength >= 0 || bLength >= 0 || carry > 0){
				//solve the different length problem, just add 0
				int aNumber = (aLength >= 0) ? a.charAt(aLength--) - '0': 0;
				int bNumber = (bLength >= 0) ? b.charAt(bLength--) - '0': 0;
				int current = (aNumber + bNumber + carry) % 2;
				carry = (aNumber + bNumber + carry)/2;
				result.insert(0, current);
			}
			//convert character sequence into string
			return result.toString();
		}
	}
}
```

Length of Last Word

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string. If the last word does not exist, return 0. For example,   Given s = "Hello World", return 5.

```
public class Solution {
	public int lengthOfLastWord(String s) {
		if(s == null || s.length() == 0)
			return 0;
		int count = 0;
		for (int i = s.length() - 1; i >= 0; i--){
			if (s.charAt(i) == ' ' && count != 0){
				break;
			}
			//if use count++ instead of stating if statement, there is one situation that string s = " " which
will return 1 instead of 0
			if(s.charAt(i) != ' ')
				count ++;
		}
		return count;
	}
}
//cannot use split function since the number of words in s cannot be known
//String[] ss = new String[N];    N cannot be blanked
```

Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:
Given an encoded message containing digits, determine the total number of ways to decode it.
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).
The number of ways decoding "12" is 2.

```java
public class Solution {
	public int numDecodings(String s) {
		//1D dynamic programming
		if (s == null || s.length() == 0)
			return 0;
		if (s.charAt(0) == '0')
			return 0;
		int[] temp = new int[s.length() + 1];
		temp[0] = 1;
		temp[1] = 1;
		int result;
		for (int i = 2; i <= s.length(); i++){
			result = Integer.parseInt(s.substring(i - 1, i));
			//Parses the string argument as a signed decimal integer.
			if (result != 0)
				temp[i] = temp[i - 1];
			if (s.charAt(i - 2) != '0'){
				result = Integer.parseInt(s.substring(i - 2, i));
				if (result >= 1 && result <= 26)
					temp[i] += temp[i - 2];
			}
		}
		return temp[s.length()];
	}
}
```

Distinct Subsequences

Given a string S and a string T, count the number of distinct subsequences of T in S. A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not). Here is an example: S = "rabbbit", T = "rabbit" Return 3.

```
public class Solution {
    public int numDistinct(String S, String T) {
        int[][] temp = new int[S.length() + 1][T.length() + 1];
        temp[0][0] = 1;// if S & T are all empty
        for (int i = 1; i <= S.length(); i++){
            temp[i][0] = 1;// if T is empty
        }
        for (int j = 1; j <= T.length(); j++){
            temp[0][j] = 0;// if S is empty
        }
        for (int i = 1; i <= S.length(); i++){
            for (int j = 1; j <= T.length(); j++){
                temp[i][j] = temp[i - 1][j];
                if (S.charAt(i - 1) == T.charAt(j - 1))
                    temp[i][j] += temp[i - 1][j - 1];
            }
        }
        return temp[S.length()][T.length()];
    }
}
```

Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.) You have the following 3 operations permitted on a word:

a) Insert a character
b) Delete a character
c) Replace a character

```
public class Solution {
	public int minDistance(String word1, String word2) {
		int[][] temp = new int[word1.length() + 1][word2.length() + 1];
		for (int i = 0; i <= word1.length(); i++){
			temp[i][0] = i;
		}
		for (int j = 0; j <= word2.length(); j++){
			temp[0][j] = j;
		}
		for (int i = 1; i <= word1.length(); i++){
			char letter1 = word1.charAt(i - 1);
			for (int j = 1; j <= word2.length(); j++){
				char letter2 = word2.charAt(j - 1);
				if (letter1 == letter2)
					temp[i][j] = temp[i - 1][j - 1];
				else{
					//temp[i - 1][j - 1] is the replace spot
					int replace = temp[i - 1][j - 1] + 1;
					//temp[i][j - 1] is the delete spot
					int delete = temp[i][j - 1] + 1;
					//temp[i - 1][j] is the insert spot
					int insert = temp[i - 1][j] + 1;
					temp[i][j] = Math.min(replace, Math.min(delete, insert));
				}
			}
		}
		return temp[word1.length()][word2.length()];
	}
}
```

Implement strStr()

Implement strStr(). Returns a pointer to the first occurrence of needle in haystack, or null if needle is not part of haystack.

```
public class Solution {
    public String strStr(String haystack, String needle) {
        int haystackLen = haystack.length();
        int needleLen = needle.length();
        if (haystackLen == needleLen && haystackLen == 0)
            return "";
        if (needleLen == 0)
            return haystack;
        for (int i = 0; i < haystackLen; i++){
            if (haystackLen - i < needleLen)
                return null;
            int k = i, j = 0;
            while (needle.charAt(j) == haystack.charAt(k) && k < haystackLen && j < needleLen){
                j++;
                k++;
                if (j ==needleLen)
                    return haystack.substring(i);
            }
        }
        return null;
    }
}
```

Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.
For example, Given:
s1 = "aabcc", s2 = "dbbca",
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

```
public class Solution {
	public boolean isInterleave(String s1, String s2, String s3) {
		//dynamic programming 2D
		if (s3.length() != s1.length() + s2.length())
			return false;
		boolean[][] check = new boolean[s1.length() + 1][s2.length() + 1];
		check[0][0] = true;
		int i = 1;
		while (i <= s1.length() && s1.charAt(i - 1) == s3.charAt(i - 1)){
			check[i][0] = true;
			i ++;
		}
		i = 1;
		while (i <= s2.length() && s2.charAt(i - 1) == s3.charAt(i - 1)){
			check[0][i] = true;
			i ++;
		}
		for (i = 1; i <= s1.length(); i++){
			for (int j = 1; j <= s2.length(); j++){
				if (s3.charAt(i + j - 1) == s1.charAt(i - 1) && check[i - 1][j])
					check[i][j] = true;
				if (s3.charAt(i + j - 1) == s2.charAt(j - 1) && check[i][j - 1])
					check[i][j] = true;
			}
		}
		return check[s1.length()][s2.length()];
	}
}
```

Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

```
public class Solution {
	public String longestCommonPrefix(String[] strs) {
		int n = strs.length;
		if (strs == null || n == 0)
			return "";
		int i = 0;//or the return could not find i
		for (i = 0; i < strs[0].length(); i++){
			char c = strs[0].charAt(i);
			for (int j = 1; j < n; j++){
				if (i >= strs[j].length() || strs[j].charAt(i) != c)
					return strs[0].substring(0, i);
			}
		}
		return strs[0].substring(0, i);
	}
}
```

Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        boolean[] check = new boolean[256];//ASCII Initialize character translation and type tables
        for (int i = 0; i < 256; i++)
            check[i] = false;
        int i = 0, j = 0, len = s.length();
        int max = 0;
        while (j < len){
            if (!check[s.charAt(j)]){
                check[s.charAt(j)] = true;
                j++;
            }
            else{
                max = Math.max(max, j - i);
                while (s.charAt(i) != s.charAt(j)){
                    check[s.charAt(i)] = false;//why?
                    i++;
                }
                i ++;
                j ++;
            }
        }
        return max = Math.max(max, len - i);
    }
}
```

Longest Palindromic Substing

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

```
public class Solution {
        public String longestPalindrome(String s) {
                if (s.isEmpty())
                        return null;
                if (s.length() == 1)
                        return s;
                String maxLen = s.substring(0, 1);
                for (int i = 0; i < s.length(); i++){
                        //the case that the string is palindrome with i
                        String temp = longestPalindrome(s, i, i);
                        if (temp.length() > maxLen.length())
                                maxLen = temp;
                        //the case that the string is palindrome with i & i + 1
                        temp = longestPalindrome(s, i, i + 1);
                        if (temp.length() > maxLen.length())
                                maxLen = temp;
                }
                return maxLen;
        }
        public String longestPalindrome(String s, int begin, int end){
                while (begin >= 0 && end < s.length() && s.charAt(begin) == s.charAt(end)){
                        begin--;
                        end++;
                }
                return s.substring(begin + 1, end);
        }
}
```

Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example, S = "ADOBECODEBANC"   T = "ABC"

Minimum window is "BANC".

If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

```
public class Solution {
    public String minWindow(String S, String T) {
        HashMap<Character, Integer> hasFound = new HashMap<Character, Integer>();
        HashMap<Character, Integer> needFind = new HashMap<Character, Integer>();
        for (int i = 0; i < T.length(); i++){
            hasFound.put(T.charAt(i), 0);
            if (needFind.containsKey(T.charAt(i)))
                needFind.put(T.charAt(i), needFind.get(T.charAt(i)) + 1);
            else
                needFind.put(T.charAt(i), 1);
        }
        int begin = 0, minSize = S.length(), count = 0;
        String result = "";
        for (int end = 0; end < S.length(); end++){
            char c = S.charAt(end);
            if (needFind.containsKey(c)){
                hasFound.put(c, hasFound.get(c) + 1);
                if (hasFound.get(c) <= needFind.get(c))
                    count ++;
                if (count == T.length()){
                    while ((!needFind.containsKey(S.charAt(begin))) ||
(hasFound.get(S.charAt(begin)) > needFind.get(S.charAt(begin)))){
                        if (needFind.containsKey(S.charAt(begin))){
                            hasFound.put(S.charAt(begin), hasFound.get(S.charAt(begin)) - 1);
                        }
                        begin ++;
                    }
                    if (end - begin + 1 <= minSize){
                        minSize = end - begin + 1;
                        result = S.substring(begin, end + 1);
                    }
                }
            }
        }
        return result;
    }
}
```

Multiply Strings

Given two numbers represented as strings, return multiplication of the numbers as a string.

```
public class Solution {
    public String multiply(String num1, String num2) {
        //reverse the string which easier to calculate
        num1 = new StringBuilder(num1).reverse().toString();
        num2 = new StringBuilder(num2).reverse().toString();
        //Causes this character sequence to be replaced by the reverse of the sequence
        int[] num = new int[num1.length() + num2.length()];
        for (int i = 0; i < num1.length(); i++){
            for (int j = 0; j < num2.length(); j++){
                int a = num1.charAt(i) - '0';
                int b = num2.charAt(j) - '0';
                num[i + j] += a * b;
            }
        }
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < num.length; i++){
            int digit = num[i] % 10;
            int carry = num[i] / 10;
            result.insert(0, digit);//insert(offset, int)在第0位加digit
            //Inserts the string representation of the second int argument into this sequence.
            if (i < num.length - 1)
                num[i + 1] += carry;
        }
        while (result.length() > 0 && result.charAt(0) == '0')
            result.deleteCharAt(0);
        //Removes the char at the specified position in this sequence.
        return result.length() == 0? "0" : result.toString();
    }
}
```

Permutation Sequence

The set [1,2,3,…,n] contains a total of n! unique permutations. By listing and labeling all of the permutations in order, We get the following sequence (ie, for n = 3):"123""132""213""231""312""321"
Given n and k, return the kth permutation sequence. Note: Given n will be between 1 and 9 inclusive.

```
public class Solution {
        public String getPermutation(int n, int k) {
                //the total number of sequence for n is n * (n - 1)!
                ArrayList<Integer> num = new ArrayList<Integer>();
                int sum = 1;
                for (int i = 1; i <= n; i++){
                        num.add(i);
                        sum *= i;
                }
                sum /= n;
                k--;
                StringBuffer temp = new StringBuffer();
                for (int i = 1; i <= n; i++){
                        //use step to know the specific group that k is located
                        int step = k / sum;
                        temp.append(num.get(step));//Returns the element at the specified position in this list.
                        num.remove(num.get(step));
                        if (i == n)
                                break;
                        k %= sum;
                        sum /= (n - i);
                }
                return temp.toString();
        }
}
```

Regular Expression Matching

Implement regular expression matching with support for '.' and '*'.

'.' Matches any single character.

'*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be: bool isMatch(const char *s, const char *p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "a*") → true

isMatch("aa", ".*") → true

isMatch("ab", ".*") → true

isMatch("aab", "c*a*b") → true

```
public class Solution {
    public boolean isMatch(String s, String p) {
        if (p.length() == 0)
            return s.length() == 0;
        //second char is not *
        if (p.length() == 1 || p.charAt(1) != '*'){
            if (s.length() < 1 || (p.charAt(0) != '.' && p.charAt(0) != s.charAt(0)))
                return false;
            return isMatch(s.substring(1), p.substring(1));//beginIndex
        }
        //second char is *
        else{
            int i = -1;//what is i for?
            //first char is . or equal to s
            while (i < s.length() && (i < 0 || p.charAt(0) == '.'|| p.charAt(0) == s.charAt(i))){
                if (isMatch(s.substring(i + 1), p.substring(2)))
                    return true;
                i++;
            }
            return false;
        }
    }
}
```

Simplify Path

Given an absolute path for a file (Unix-style), simplify it.
For example,
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"

```
public class Solution {
    public String simplifyPath(String path) {
        if (path == null || path.length() == 0)
            return path;
        String[] temp = path.split("/");
        //Splits this string around matches of the given regular expression.
        LinkedList<String> result = new LinkedList<String>();
        for (String s: temp){
            if (s.length() == 0 || s.equals("."))
                continue;
            else if (s.equals("..")){
                if (!result.isEmpty())
                    result.pop();
            }
            else
                result.push(s);
        }
        //has to consider the case of empty, or if path = "/", then the return is [] rather than [/]
        if (result.isEmpty())
            result.push("");
        String ans = "";
        while (!result.isEmpty()){
            ans += "/" + result.removeLast(); //Removes and returns the last element from this list.
        }
        return ans;
    }
}
```

Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
For example,
"A man, a plan, a canal: Panama" is a palindrome.
"race a car" is not a palindrome.
Have you consider that the string might be empty? This is a good question to ask during an interview.
For the purpose of this problem, we define empty string as valid palindrome.

```
public class Solution {
        public boolean isPalindrome(String s) {
                s = s.replaceAll("[^a-zA-Z0-9]","").toLowerCase();
                //replaceAll:Replaces each substring of this string that matches the given regular expression with
the given replacement.
                //a-zA-Z:a through z or A through Z, inclusive (range)
                //^a-zA-Z0-9:exclusive a through z and 0 to 9
                if(s == null)
                        return false;
                if (s.length() == 0)
                        return true;
                Stack<Character> result = new Stack<Character>();
                int index = 0;
                while (index < s.length() / 2){
                        result.push(s.charAt(index));
                        index ++;
                }
                if (s.length() % 2 == 1)
                        index ++;
                while (index < s.length()){
                        if (result.empty())
                                return false;
                        char temp = result.pop();
                        if (temp != s.charAt(index))
                                return false;
                        else
                                index ++;
                }
                return true;
        }
}
```

Word Ladder

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that: Only one letter can be changed at a time. Each intermediate word must exist in the dictionary. For example, Given:

start = "hit"   end = "cog"   dict = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",

return its length 5.

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

```
public class Solution {
	public int ladderLength(String start, String end, Set<String> dict) {
		if (dict.size() == 0)
			return 0;
		LinkedList<String> tempString = new LinkedList<String>();
		LinkedList<Integer> intString = new LinkedList<Integer>();
		tempString.add(start);
		intString.add(1);
		while (!tempString.isEmpty()){
			int result = intString.pop();
			String tempResult = tempString.pop();
			if (tempResult.equals(end))
				return result;
			for (int i = 0; i < tempResult.length(); i++){
				char[] tempChar = tempResult.toCharArray();
				for (char a = 'a'; a <= 'z'; a++){
					tempChar[i] = a;
					String newString = new String(tempChar);
					if (dict.contains(newString)){
						tempString.add(newString);
						intString.add(result + 1);
						dict.remove(newString);
					}
				}
			}
		}
		return 0;
	}
}
```

ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P     A     H    N
A  P  L  S  I  I  G
Y     I      R
```

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

```
public class Solution {
    public String convert(String s, int nRows) {
        if (s == null || s.length() == 0 || nRows <= 0)
            return "";
        if (nRows == 1)
            return s;
        StringBuilder result = new StringBuilder();
        int size = 2 * nRows - 2;
        for (int i = 0; i < nRows; i++){
            for (int j = i; j < s.length(); j += size){
                result.append(s.charAt(j));
                if (i != 0 && i != nRows - 1 && j + size -i * 2< s.length())
                    result.append(s.charAt(j + size - 2 *i));
            }
        }
        return result.toString();//stringBuilder, not string
    }
}
```