Easy 1:-

Code:-

```
public class teju LastWordLength {
    public static int lengthOfLastWord(String s) {
        // Trim the input string to remove leading and trailing spaces
        s = s.trim();

        // Split the string into words using space as a delimiter
        String[] words = s.split(" ");

        // Get the length of the last word
        String lastWord = words[words.length - 1];
        return lastWord.length();
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println(lengthOfLastWord("Hello World")); // Output: 5
        System.out.println(lengthOfLastWord("   fly me   to   the moon  ")); // Output: 4
        System.out.println(lengthOfLastWord("luffy is still joyboy")); // Output: 6
    }
}
```

Explanation:-

You can achieve this by splitting the input string into words and then finding the length of the last word. Here's a Java code snippet for that.This code defines a method lengthOfLastWord that takes a string s as input and returns the length of the last word. The main method contains test cases to demonstrate the functionality.

Easy 2 :-

Code:-

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}
```

```java
class SortedArrayToBST {
    public TreeNode sortedArrayToBST(int[] nums) {
        if (nums == null || nums.length == 0) {
            return null;
        }

        return sortedArrayToBST(nums, 0, nums.length - 1);
    }

    private TreeNode sortedArrayToBST(int[] nums, int start, int end) {
        if (start > end) {
            return null;
        }

        int mid = start + (end - start) / 2;
        TreeNode root = new TreeNode(nums[mid]);

        root.left = sortedArrayToBST(nums, start, mid - 1);
        root.right = sortedArrayToBST(nums, mid + 1, end);

        return root;
    }

    // Helper function to print the tree in-order
    public void inOrderTraversal(TreeNode root) {
        if (root != null) {
            inOrderTraversal(root.left);
            System.out.print(root.val + " ");
            inOrderTraversal(root.right);
        }
    }

    public static void main(String[] args) {
        SortedArrayToBST converter = new SortedArrayToBST();

        // Example 1
        int[] nums1 = {-10, -3, 0, 5, 9};
        TreeNode root1 = converter.sortedArrayToBST(nums1);
        converter.inOrderTraversal(root1);
        // Output: -10 -3 0 5 9
        System.out.println(); // Add a newline for clarity


        // Example 2
        int[] nums2 = {1, 3};
        TreeNode root2 = converter.sortedArrayToBST(nums2);
        converter.inOrderTraversal(root2);
        // Output: 1 3
    }
}
```

Explanation:-

To solve this problem, you can use a recursive approach. The idea is to choose the middle element of the sorted array as the root of the binary search tree. Then, recursively build the left and right subtrees using the elements to the left and right of the middle element, respectively This code defines a TreeNode class for the binary tree nodes and a SortedArrayToBST class with the sortedArrayToBST method to convert a sorted array to a height-balanced binary search tree. The inOrderTraversal method is a helper function to print the tree in in-order traversal for verification. The main method includes test cases for the provided examples.

Easy 3:-

Code:-

```
import java.util.ArrayList;
import java.util.List;

class PascalTriangle {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> triangle = new ArrayList<>();

        if (numRows <= 0) {
            return triangle;
        }

        for (int i = 0; i < numRows; i++) {
            List<Integer> row = new ArrayList<>();

            for (int j = 0; j <= i; j++) {
                if (j == 0 || j == i) {
                    row.add(1); // The first and last elements in each row are always 1.
                } else {
                    // Calculate the sum of the two numbers directly above the current element.
                    int sum = triangle.get(i - 1).get(j - 1) + triangle.get(i - 1).get(j);
                    row.add(sum);
                }
            }

            triangle.add(row);
        }

        return triangle;
    }

    public static void main(String[] args) {
        PascalTriangle pascalTriangle = new PascalTriangle();

        // Example 1
        int numRows1 = 5;
        List<List<Integer>> result1 = pascalTriangle.generate(numRows1);
        System.out.println(result1);
        // Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

        // Example 2
        int numRows2 = 1;
        List<List<Integer>> result2 = pascalTriangle.generate(numRows2);
        System.out.println(result2);
        // Output: [[1]]
    }
}
```

Explanation:-

You can generate Pascal's triangle by using a nested loop to iterate through each row and column. Each element in the triangle is the sum of the elements directly above and above to the left. Here's the Java code for this task. This code defines a PascalTriangle class with a generate method that takes the number of rows (numRows) and returns the Pascal's triangle as a list of lists. The main method includes test cases for the provided examples.

Medium 1 :-

Code:-

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
```

```java
        val = x;
    }
}

class LowestCommonAncestorBST {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) {
            return null;
        }

        int rootVal = root.val;
        int pVal = p.val;
        int qVal = q.val;

        if (pVal < rootVal && qVal < rootVal) {
            // Both nodes are in the left subtree
            return lowestCommonAncestor(root.left, p, q);
        } else if (pVal > rootVal && qVal > rootVal) {
            // Both nodes are in the right subtree
            return lowestCommonAncestor(root.right, p, q);
        } else {
            // One node is in the left subtree, and the other is in the right subtree
            // or one of the nodes is the current root
            return root;
        }
    }

    public static void main(String[] args) {
        LowestCommonAncestorBST solution = new LowestCommonAncestorBST();

        // Example 1
        TreeNode root1 = new TreeNode(6);
        root1.left = new TreeNode(2);
        root1.right = new TreeNode(8);
        root1.left.left = new TreeNode(0);
        root1.left.right = new TreeNode(4);
        root1.right.left = new TreeNode(7);
        root1.right.right = new TreeNode(9);
        root1.left.right.left = new TreeNode(3);
        root1.left.right.right = new TreeNode(5);
        TreeNode p1 = new TreeNode(2);

        TreeNode q1 = new TreeNode(8);

        System.out.println(solution.lowestCommonAncestor(root1, p1, q1).val); // Output: 6


        // Example 2

        TreeNode root2 = new TreeNode(6);

        root2.left = new TreeNode(2);

        root2.right = new TreeNode(8);

        root2.left.left = new TreeNode(0);

        root2.left.right = new TreeNode(4);

        root2.right.left = new TreeNode(7);

        root2.right.right = new TreeNode(9);

        root2.left.right.left = new TreeNode(3);

        root2.left.right.right = new TreeNode(5);

        TreeNode p2 = new TreeNode(2);

        TreeNode q2 = new TreeNode(4);

        System.out.println(solution.lowestCommonAncestor(root2, p2, q2).val); // Output: 2


        // Example 3

        TreeNode root3 = new TreeNode(2);

        root3.left = new TreeNode(1);

        TreeNode p3 = new TreeNode(2);

        TreeNode q3 = new TreeNode(1);

        System.out.println(solution.lowestCommonAncestor(root3, p3, q3).val); // Output: 2

    }

}
```

Explanation:-

the lowest common ancestor (LCA) of two nodes in a binary search tree (BST) by traversing the tree from the root to the leaves. The key property of a BST is that the left subtree of a node contains only nodes with values less than the node, and the right subtree contains only nodes with values greater than the node.This code defines a TreeNode class for the binary tree nodes and a LowestCommonAncestorBST class with the lowestCommonAncestor method to find the LCA of two nodes in a BST. The main method includes test cases for the provided examples.

Medium 2:-

Code:-

```
import java.util.ArrayList;

import java.util.List;


class MajorityElement {

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

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

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

int candidate1 = 0, candidate2 = 0;
int count1 = 0, count2 = 0;

// Voting process to find potential candidates
for (int num : nums) {
    if (num == candidate1) {
        count1++;
    } else if (num == candidate2) {
        count2++;
    } else if (count1 == 0) {
        candidate1 = num;
        count1 = 1;
    } else if (count2 == 0) {
        candidate2 = num;
        count2 = 1;
    } else {
        count1--;
        count2--;
    }
}

// Count occurrences of potential candidates
count1 = 0;
count2 = 0;
for (int num : nums) {
    if (num == candidate1) {
        count1++;
    } else if (num == candidate2) {
        count2++;
    }
}

// Check if candidates appear more than ⌊ n/3 ⌋ times
if (count1 > nums.length / 3) {
```

```
            result.add(candidate1);
        }
        if (count2 > nums.length / 3) {
            result.add(candidate2);
        }

        return result;
    }

    public static void main(String[] args) {
        MajorityElement majorityElement = new MajorityElement();

        // Example 1
        int[] nums1 = {3, 2, 3};
        System.out.println(majorityElement.majorityElements(nums1)); // Output: [3]

        // Example 2
        int[] nums2 = {1};
        System.out.println(majorityElement.majorityElements(nums2)); // Output: [1]

        // Example 3
        int[] nums3 = {1, 2};
        System.out.println(majorityElement.majorityElements(nums3)); // Output: [1, 2]
    }
}
```

Explanation:-

To solve this problem, we can use the Boyer-Moore Majority Vote algorithm. This algorithm is designed to find elements that appear more than ⌊ n/2 ⌋ times in an array, but it can be extended to find elements that appear more than ⌊ n/3 ⌋ times.This code defines a MajorityElement class with a majorityElements method to find elements that appear more than ⌊ n/3 ⌋ times in an array. The main method includes test cases for the provided examples.

Medium 3:-

Code:-

```
class MaximalSquare {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
```

```java
        int m = matrix.length;
        int n = matrix[0].length;
        int[][] dp = new int[m][n];
        int maxSide = 0;

        // Initialize the first row and column of the dp array
        for (int i = 0; i < m; i++) {
            dp[i][0] = matrix[i][0] - '0';
            maxSide = Math.max(maxSide, dp[i][0]);
        }
        for (int j = 0; j < n; j++) {
            dp[0][j] = matrix[0][j] - '0';
            maxSide = Math.max(maxSide, dp[0][j]);
        }

        // Fill in the rest of the dp array
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == '1') {
                    // Calculate the maximum side length at position (i, j)
                    dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1;
                    maxSide = Math.max(maxSide, dp[i][j]);
                }
            }
        }

        // Return the area of the largest square
        return maxSide * maxSide;
    }

    public static void main(String[] args) {
        MaximalSquare solution = new MaximalSquare();

        // Example 1
        char[][] matrix1 = {
            {'0'}
        };
        System.out.println(solution.maximalSquare(matrix1)); // Output: 0
        // Example 2

        char[][] matrix2 = {

            {'0', '1'},

            {'1', '0'}

        };

        System.out.println(solution.maximalSquare(matrix2)); // Output: 1


        // Example 3

        char[][] matrix3 = {

            {'1', '0', '1', '0', '0'},

            {'1', '0', '1', '1', '1'},

            {'1', '1', '1', '1', '1'},

            {'1', '0', '0', '1', '0'}

        };

        System.out.println(solution.maximalSquare(matrix3)); // Output: 4

    }

}
```

Explanation:-

To find the largest square containing only 1's in a binary matrix, you can use dynamic programming. The idea is to create a 2D array to store the maximum side length of the square ending at each position in the matrix. The maximum side length at a given position (i, j) is determined by the values in the adjacent positions (i-1, j), (i, j-1), and (i-1, j-1).This code defines a MaximalSquare class with a maximalSquare method to find the area of the largest square containing only 1's in the given binary matrix. The main method includes test cases for the provided examples.

Hard 1:-

Code:-

```
import java.util.ArrayDeque;

import java.util.Deque;

import java.util.LinkedList;


class MaxSlidingWindow {

    public int[] maxSlidingWindow(int[] nums, int k) {

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

            return new int[0];

        }


        int n = nums.length;

        int[] result = new int[n - k + 1];

        int ri = 0;
```

```java
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < nums.length; i++) {
            // Remove elements that are out of the current window
            while (!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }

            // Remove elements that are smaller than the current element
            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }

            // Add the current index to the deque
            deque.offer(i);

            // Add the maximum element to the result array when the window is fully formed
            if (i >= k - 1) {
                result[ri++] = nums[deque.peek()];
            }
        }

        return result;
    }

    public static void main(String[] args) {
        MaxSlidingWindow solution = new MaxSlidingWindow();

        // Example 1
        int[] nums1 = {1};
        int k1 = 1;
        int[] result1 = solution.maxSlidingWindow(nums1, k1);
        for (int num : result1) {
            System.out.print(num + " "); // Output: 1
        }
        System.out.println();

        // Example 2
        int[] nums2 = {1, 3, -1, -3, 5, 3, 6, 7};

        int k2 = 3;

        int[] result2 = solution.maxSlidingWindow(nums2, k2);

        for (int num : result2) {

            System.out.print(num + " "); // Output: 3 3 5 5 6 7

        }

        System.out.println();

    }

}
```

Explanation:-

this problem using a deque (double-ended queue) to efficiently maintain the maximum elements in the sliding window. The deque will store the indices of elements in the current window, and you can maintain it in a way that the front of the deque always represents the maximum element in the current window.This code defines a MaxSlidingWindow class with a maxSlidingWindow method to find the maximum sliding window. The main method includes test cases for the provided examples.

Hard 2:-

Code:-

```
class ShortestPalindrome {

    public String shortestPalindrome(String s) {

        int n = s.length();


        // Find the longest palindrome substring starting from the beginning

        int end = 0;

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

            if (isPalindrome(s, 0, i)) {

                end = i;

                break;

            }

        }


        // Reverse the remaining part and prepend it to the original string

        StringBuilder prefix = new StringBuilder(s.substring(end + 1)).reverse();

        return prefix + s;

    }


    private boolean isPalindrome(String s, int start, int end) {

        while (start < end) {

            if (s.charAt(start) != s.charAt(end)) {
                return false;
            }
            start++;
            end--;
        }
        return true;
    }

    public static void main(String[] args) {
        ShortestPalindrome solution = new ShortestPalindrome();

        // Example 1
        String s1 = "aacecaaa";
        System.out.println(solution.shortestPalindrome(s1)); // Output: "aaacecaaa"

        // Example 2
        String s2 = "abcd";
        System.out.println(solution.shortestPalindrome(s2)); // Output: "dcbabcd"
    }
}
```

Explanation :-

Find the longest palindrome substring that starts from the beginning of the given string s.

Reverse the remaining part of the string (substring after the longest palindrome found) and prepend it to the original string.This code defines a ShortestPalindrome class with a shortestPalindrome method to find the shortest palindrome by adding characters in front of the given string s. The isPalindrome method checks if a given substring is a palindrome. The main method includes test cases for the provided examples.

Hard 3:-

Code:-

```
class CountDigitOne {
    public int countDigitOne(int n) {
        if (n <= 0) {
            return 0;
        }

        int count = 0;
        for (long i = 1; i <= n; i *= 10) {
            long divisor = i * 10;
            count += (n / divisor) * i + Math.min(Math.max(n % divisor - i + 1, 0), i);

        }


        return count;

    }


    public static void main(String[] args) {

        CountDigitOne solution = new CountDigitOne();


        // Example 1

        int n1 = 13;

        System.out.println(solution.countDigitOne(n1)); // Output: 6


        // Example 2

        int n2 = 0;

        System.out.println(solution.countDigitOne(n2)); // Output: 0

    }

}
```

Explaination:-

To count the total number of digit 1 appearing in all non-negative integers less than or equal to n, you can follow a mathematical approach. The idea is to count the occurrences of digit 1 at each place value (ones, tens, hundreds, etc.) and then sum them up.This code defines a CountDigitOne class with a countDigitOne method to count the total number of digit 1 appearing in all non-negative integers less than or equal to n. The main method includes test cases for the provided examples. The algorithm is based on the observation of patterns in the occurrence of digit 1 at each place value.