**1. Anagrams**

import java.util.Arrays;

public class Anagrams {

    public static void main(String[] args) {

        String a = "listen";

        String b = "silent";

        char[] c = a.toCharArray();

        char[] d = b.toCharArray();

        Arrays.sort(c);

        Arrays.sort(d);

        if (Arrays.equals(c, d)) {

            System.out.println("Anagrams");

        } else {

            System.out.println("Not anagrams");

        }

    }

}

**Output**



**Time Complexity**

**O(n log n)**

**2.** **Row with max 1's**

public class Rowswithmax1 {

    public static void main(String[] args) {

        int[][] arr1 = {

                { 0, 1, 1, 1 },

                { 0, 0, 1, 1 },

                { 1, 1, 1, 1 },

                { 0, 0, 0, 0 }

        };

        int[][] arr2 = {

                { 0, 0 },

                { 1, 1 }

        };

        System.out.println("Row with max 1s: " + rowWithMax1s(arr1));

        System.out.println("Row with max 1s: " + rowWithMax1s(arr2));

    }

    public static int rowWithMax1s(int[][] arr) {

        int n = arr.length;

        int m = arr[0].length;

        int maxRowIndex = -1;

        int j = m - 1;

        for (int i = 0; i < n; i++) {

            while (j >= 0 && arr[i][j] == 1) {

                j--;

                maxRowIndex = i;

            }

        }

        return maxRowIndex;

    }

}

**Time Complexity**

**O(n + m)**

**3.** **Longest consecutive Subsequence**

import java.util.HashSet;

public class Longestconsecutivesequence {

    public static void main(String[] args) {

        int[] arr = { 2, 6, 1, 9, 4, 5, 3 };

        System.out.println("Longest consecutive subsequence length: " + findLongestConsecutiveSubsequence(arr));

    }

    public static int findLongestConsecutiveSubsequence(int[] arr) {

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

        for (int num : arr) {

            set.add(num);

        }

        int longestStreak = 0;

        for (int num : arr) {

            if (!set.contains(num - 1)) {

                int currentNum = num;

                int currentStreak = 1;

                while (set.contains(currentNum + 1)) {

                    currentNum += 1;

                    currentStreak += 1;

                }

                longestStreak = Math.max(longestStreak, currentStreak);

            }

        }

        return longestStreak;

    }

}

**Time Complexity**

**O(n)**

**4.** **Longest palindrome in a string**

class ListNode {

    int val;

    ListNode next;

    ListNode(int val) {

        this.val = val;

        this.next = null;

    }

}

public class Palindromelinkedlist {

    public static void main(String[] args) {

        ListNode head1 = new ListNode(1);

        head1.next = new ListNode(2);

        head1.next.next = new ListNode(1);

        head1.next.next.next = new ListNode(1);

        head1.next.next.next.next = new ListNode(2);

        head1.next.next.next.next.next = new ListNode(1);

        System.out.println(isPalindrome(head1));

        ListNode head2 = new ListNode(1);

        head2.next = new ListNode(2);

        head2.next.next = new ListNode(3);

        head2.next.next.next = new ListNode(4);

        System.out.println(isPalindrome(head2));

    }

    public static boolean isPalindrome(ListNode head) {

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

            return true;

        }

        ListNode slow = head, fast = head;

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

            slow = slow.next;

            fast = fast.next.next;

        }

        ListNode secondHalf = reverseList(slow);

        ListNode secondHalfCopy = secondHalf;

        ListNode firstHalf = head;

        while (secondHalf != null) {

            if (firstHalf.val != secondHalf.val) {

                return false;

            }

            firstHalf = firstHalf.next;

            secondHalf = secondHalf.next;

        }

        reverseList(secondHalfCopy);

        return true;

    }

    private static ListNode reverseList(ListNode head) {

        ListNode prev = null, curr = head;

        while (curr != null) {

            ListNode next = curr.next;

            curr.next = prev;

            prev = curr;

            curr = next;

        }

        return prev;

    }

}

**Time Complexity**

**O( n )**

**5.** **Rat in a maze**

import java.util.\*;

class Ratinamaze {

    static int[] xa = { 0, 1, -1, 0 };

    static int[] ya = { 1, 0, 0, -1 };

    static String[] str = { "R", "D", "U", "L" };

    public ArrayList<String> findPath(int[][] mat) {

        ArrayList<String> result = new ArrayList<>();

        int m = mat.length;

        int n = mat[0].length;

        int[][] visited = new int[m][n];

        if (mat[0][0] == 1)

            findPaths(0, 0, m, n, mat, result, visited, "");

        return result;

    }

    public void findPaths(int x, int y, int m, int n, int[][] grid, List<String> results,

            int[][] visited, String path) {

        if (x == m - 1 && y == n - 1) {

            results.add(path);

            return;

        }

        visited[x][y] = 1;

        for (int i = 0; i < 4; i++) {

            int newX = x + xa[i];

            int newY = y + ya[i];

            if (newX >= 0 && newX < m && newY >= 0 && newY < n &&

                    grid[newX][newY] == 1 && visited[newX][newY] == 0) {

                findPaths(newX, newY, m, n, grid, results, visited, path + str[i]);

            }

        }

        visited[x][y] = 0;

    }

}

**Time Complexity**

**O(m n)**