PRACTICE ASSIGNMENT 3

1.Anagram

Given two strings s1 and s2 consisting of lowercase characters. The task is to check whether two given strings are an anagram of each other or not. An anagram of a string is another string that contains the same characters, only the order of characters can be different. For example, act and tac are an anagram of each other. Strings s1 and s2 can only contain lowercase alphabets.

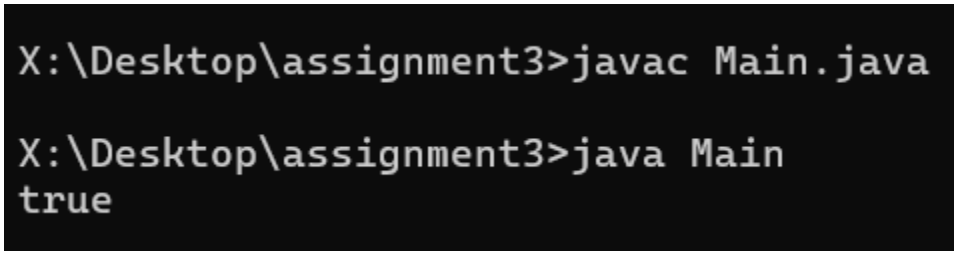
Input: s1 = "geeks", s2 = "kseeg"
Output: true

CODE:

```
class Solution {
    public static boolean areAnagrams(String s1, String s2) {
        int[] charCount = new int[26];
        Arrays.fill(charCount, 0);

        if (s1.length() != s2.length()) {
            return false;
        }

        for (int i = 0; i < s1.length(); i++) {
            charCount[s1.charAt(i) - 'a']++;
        }

        for (int i = 0; i < s2.length(); i++) {
            charCount[s2.charAt(i) - 'a']--;
        }

        for (int i = 0; i < 26; i++) {
            if (charCount[i] != 0) {
                return false;
            }
        }

        return true;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        String s1 = "geeks";
        String s2 = "kseeg";

        boolean result = Solution.areAnagrams(s1, s2);
        System.out.println(result);
    }
}
```

OUTPUT:



TIME COMPLEXITY: O(N)

---

2. Find the row with maximum number of 1s

Given a binary 2D array, where each row is sorted. Find the row with the maximum number of 1s.

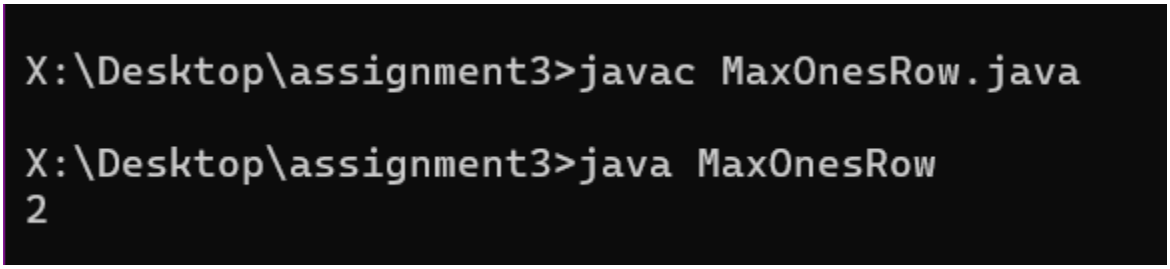
Input
matrix : 0 1 1 1 0 0 1 1 1 1 1 1 0 0 0 0
Output: 2
Explanation: Row = 2 has a maximum number of 1s, that is 4.

CODE:

```
public class MaxOnesRow {

    public static int findRowWithMaxOnes(int[][] matrix) {
        int maxRowIndex = -1;
        int maxCount = 0;
```

```
        for (int i = 0; i < matrix.length; i++) {
            int count = countOnes(matrix[i]);
            if (count > maxCount) {
                maxCount = count;
                maxRowIndex = i;
            }
        }

        return maxRowIndex;
    }
    private static int countOnes(int[] row) {
        int low = 0, high = row.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (row[mid] == 1) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return row.length - low;
    }

    public static void main(String[] args) {
        // Input matrix 1
        int[][] matrix1 = {
            {0, 1, 1, 1},
            {0, 0, 1, 1},
            {1, 1, 1, 1},
            {0, 0, 0, 0}
        };
        System.out.println( findRowWithMaxOnes(matrix1));

    }
}
```

OUTPUT:



TIME COMPLEXITY: O(n log m)

---

3. Longest Consecutive Subsequence

Given an array of integers, find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.

Input: arr[] = {1, 9, 3, 10, 4, 20, 2}
Output: 4
Explanation: The subsequence 1, 3, 4, 2 is the longest subsequence of consecutive elements
Input: arr[] = {36, 41, 56, 35, 44, 33, 34, 92, 43, 32, 42}
Output: 5
Explanation: The subsequence 36, 35, 33, 34, 32 is the longest subsequence of consecutive elements.

CODE:

```
import java.util.ArrayList;
import java.util.Arrays;
class LongestSubsequence {
    static int findLongestSequence(int nums[], int size) {
        Arrays.sort(nums);
        int maxLength = 0, currentLength = 0;
        ArrayList<Integer> uniqueNums = new ArrayList<>();
        uniqueNums.add(nums[0]);
        for (int i = 1; i < size; i++) {
            if (nums[i] != nums[i - 1])
                uniqueNums.add(nums[i]);
        }
        for (int i = 0; i < uniqueNums.size(); i++) {
            if (i > 0 && uniqueNums.get(i) == uniqueNums.get(i - 1) + 1)
```

```
            currentLength++;
        else
            currentLength = 1;

        maxLength = Math.max(maxLength, currentLength);
      }
      return maxLength;
   }

   public static void main(String[] args) {
      int nums[] = { 1, 9, 3, 10, 4, 20, 2 };
      int size = nums.length;

      System.out.println(findLongestSequence(nums, size));
   }
}
```

OUTPUT:

```
X:\Desktop\assignment3>javac LongestSubsequence.java

X:\Desktop\assignment3>java LongestSubsequence
4
```

TIME COMPLEXITY: O(n log n)

---

4. Longest Palindrome Substring

Given a string s, your task is to find the longest palindromic substring within s. A substring is a contiguous sequence of characters within a string, defined as s[i...j] where $0 \leq i \leq j < \text{len}(s)$.

A palindrome is a string that reads the same forward and backward. More formally, s is a palindrome if reverse(s) == s.

Note: If there are multiple palindromes with the same length, return the first occurrence of the longest palindromic substring from left to right.

Input: s = "aaaabbaa"
Output: "aabbaa"
Explanation: The longest palindromic substring is "aabbaa".

CODE:

```
class LongestPalindromeSubstring {
    public static String findLongestPalindrome(String s) {
        int n = s.length();
        if (n < 2) return s;
        int start = 0, maxLength = 1;
        for (int i = 0; i < n; i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
            int len = Math.max(len1, len2);
            if (len > maxLength) {
                maxLength = len;
                start = i - (len - 1) / 2;
            }
        }
        return s.substring(start, start + maxLength);
    }
    private static int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }
    public static void main(String[] args) {
        String s1 = "aaaabbaa";
        System.out.println(findLongestPalindrome(s1));
    }
}
```

OUTPUT:

```
X:\Desktop\assignment3>javac LongestPalindromeSubstring.java

X:\Desktop\assignment3>java LongestPalindromeSubstring
aabbaa
```

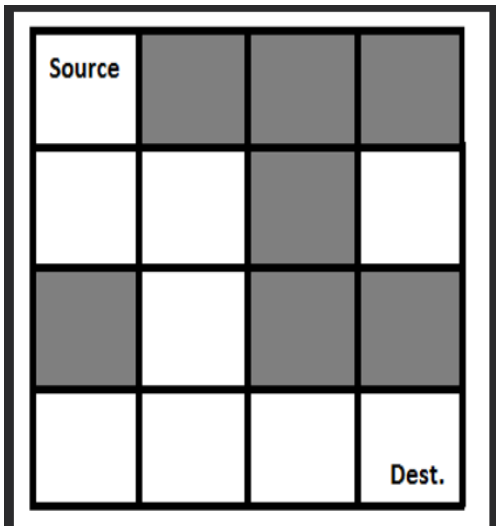
---

5. Rat in a Maze

We have discussed Backtracking and Knight's tour problem in Set 1. Let us discuss Rat in a Maze as another example problem that can be solved using Backtracking.

Consider a rat placed at (0, 0) in a square matrix of order N * N. It has to reach the destination at (N – 1, N – 1). Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are 'U'(up), 'D'(down), 'L' (left), 'R' (right). Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it. Return the list of paths in lexicographically increasing order. Note: In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell.

INPUT:



OUTPUT: DRDDRR

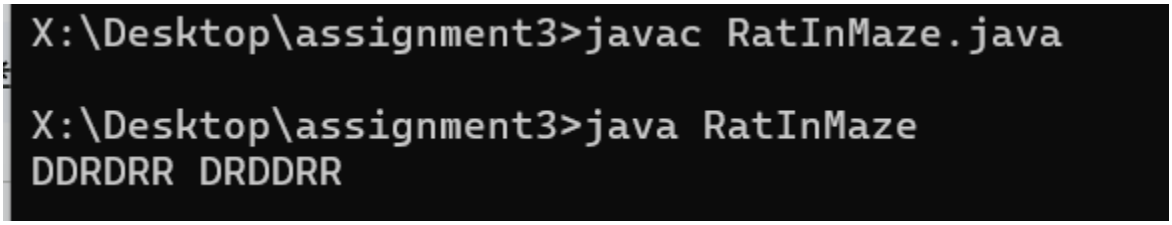
CODE:

```
import java.util.ArrayList;
public class RatInMaze {
   static String moves = "DLRU";
   static int[] rowDelta = { 1, 0, 0, -1 };
   static int[] colDelta = { 0, -1, 1, 0 };
   static boolean isSafe(int x, int y, int size, int[][] grid) {
      return x >= 0 && y >= 0 && x < size && y < size && grid[x][y] == 1;
   }
```

```
    static void explorePaths(int x, int y, int[][] grid, int size, ArrayList<String> paths,
StringBuilder path) {
        if (x == size - 1 && y == size - 1) {
            paths.add(path.toString());
            return;
        }
        grid[x][y] = 0;
        for (int i = 0; i < 4; i++) {
            int nextX = x + rowDelta[i];
            int nextY = y + colDelta[i];
            if (isSafe(nextX, nextY, size, grid)) {
                path.append(moves.charAt(i));
                explorePaths(nextX, nextY, grid, size, paths, path);
                path.deleteCharAt(path.length() - 1);
            }
        }
        grid[x][y] = 1;
    }
    public static void main(String[] args) {
        int[][] grid = {
            { 1, 0, 0, 0 },
            { 1, 1, 0, 1 },
            { 1, 1, 0, 0 },
            { 0, 1, 1, 1 }
        };

        int size = grid.length;
        ArrayList<String> paths = new ArrayList<>();
        StringBuilder path = new StringBuilder();

        if (grid[0][0] == 1 && grid[size - 1][size - 1] == 1) {
            explorePaths(0, 0, grid, size, paths, path);
        }

        if (paths.isEmpty()) {
            System.out.println(-1);
        } else {
            for (String p : paths) {
                System.out.print(p + " ");
            }
```

```
        }
        System.out.println();
    }
}
```

OUTPUT:



TIME COMPLEXITY: O(4N^2)
