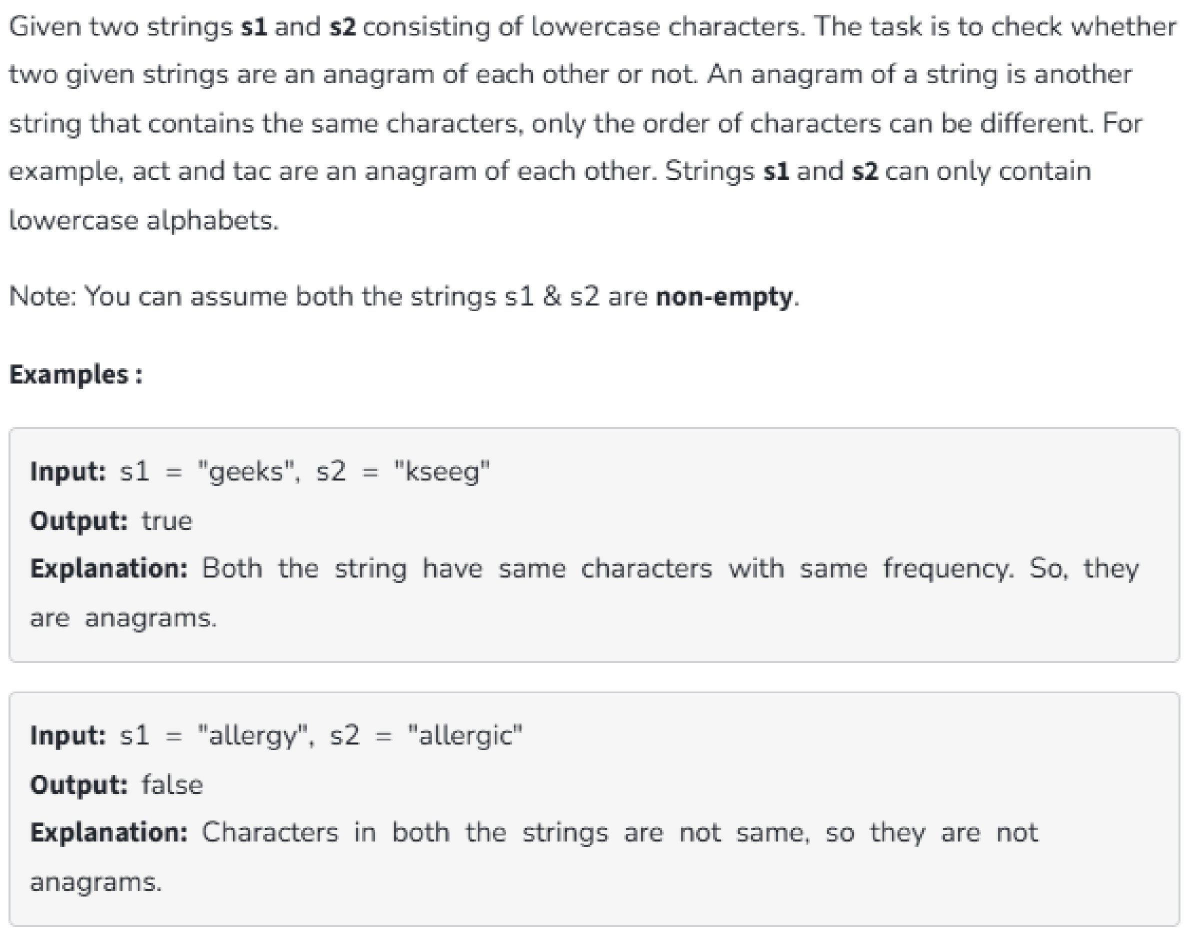
Coding Practice-3

1. Anagram program



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.

Note: You can assume both the strings s1 & s2 are **non-empty**.

Examples :

Input: s1 = "geeks", s2 = "kseeg"
Output: true
Explanation: Both the string have same characters with same frequency. So, they are anagrams.

Input: s1 = "allergy", s2 = "allergic"
Output: false
Explanation: Characters in both the strings are not same, so they are not anagrams.

Code:

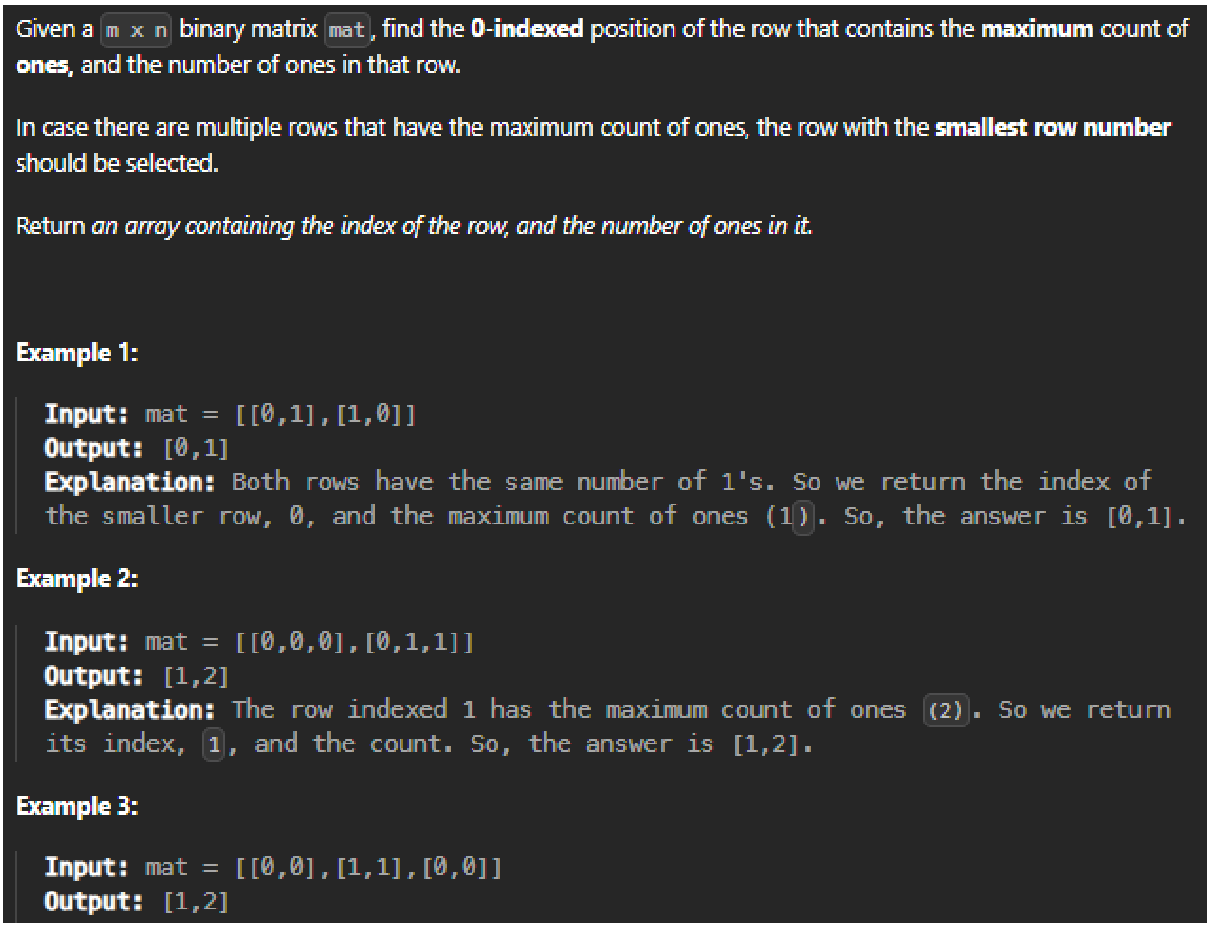
```
import java.util.HashMap;
import java.util.Scanner;
public class Anagram {
 public static boolean isAnagram(String s1, String s2) {
 if (s1.length() != s2.length()) {
 return false;
 }
 HashMap<Character, Integer> charCountMap = new HashMap<>();
 for (char c : s1.toCharArray()) {
 charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
 }
 for (char c : s2.toCharArray()) {
 if (!charCountMap.containsKey(c) || charCountMap.get(c) == 0) {
 return false;
 }
 charCountMap.put(c, charCountMap.get(c) - 1);
 }
 return true;
 }
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("Enter first string: ");
 String s1 = scanner.nextLine();

 System.out.print("Enter second string: ");
 String s2 = scanner.nextLine();

 System.out.println(isAnagram(s1, s2));

 scanner.close();
 }
}
```

Output:

```
Enter first string: Swetha
Enter second string: Priyanka
False
```

Time Complexity: O(n)

2. Row with max 1s'


Given a `m x n` binary matrix `mat`, find the **0-indexed** position of the row that contains the **maximum** count of **ones,** and the number of ones in that row.

In case there are multiple rows that have the maximum count of ones, the row with the **smallest row number** should be selected.

Return *an array containing the index of the row, and the number of ones in it.*

Example 1:

```
Input: mat = [[0,1],[1,0]]
Output: [0,1]
Explanation: Both rows have the same number of 1's. So we return the index of
the smaller row, 0, and the maximum count of ones (1). So, the answer is [0,1].
```

Example 2:

```
Input: mat = [[0,0,0],[0,1,1]]
Output: [1,2]
Explanation: The row indexed 1 has the maximum count of ones (2). So we return
its index, 1, and the count. So, the answer is [1,2].
```

Example 3:

```
Input: mat = [[0,0],[1,1],[0,0]]
Output: [1,2]
```

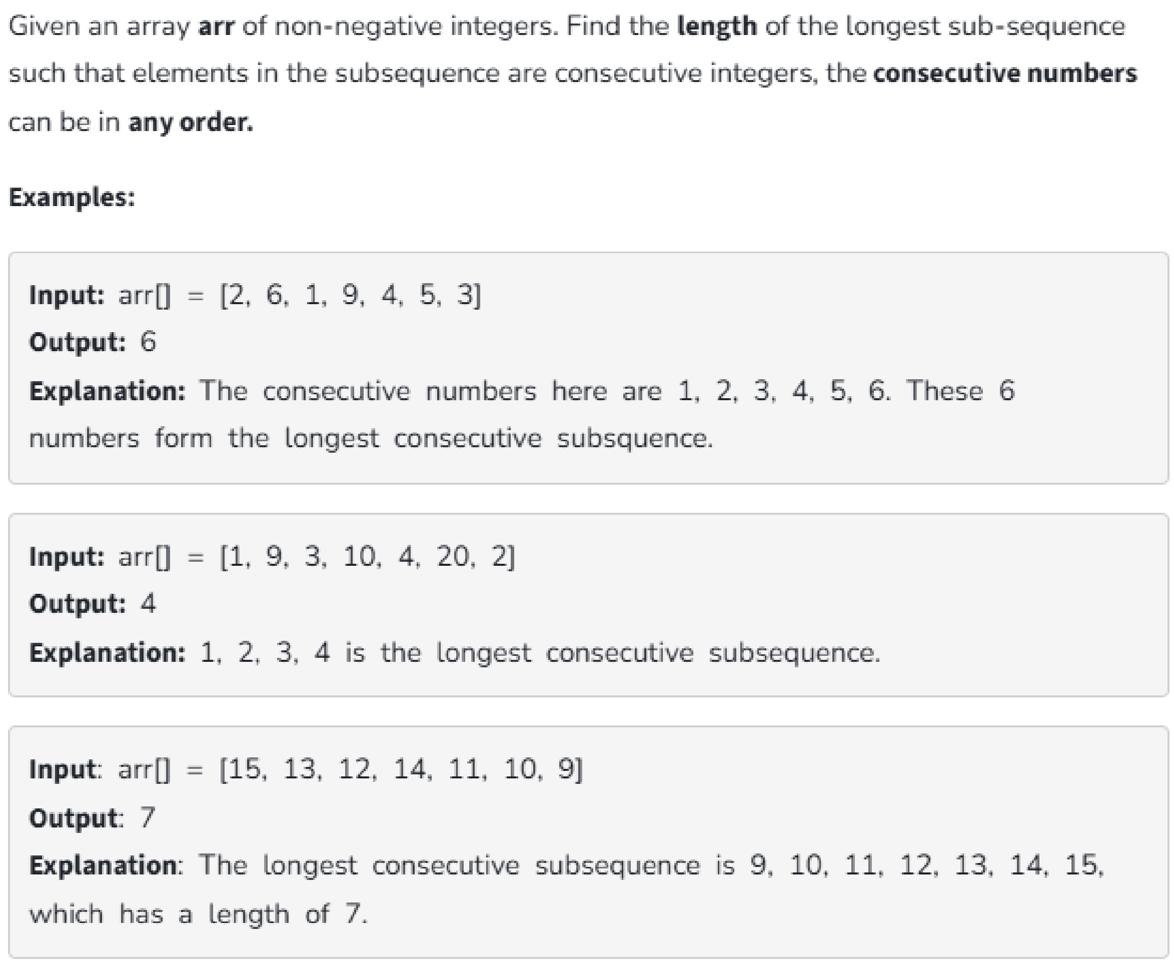
Code:

```
class MaxRow {
    public int[] rowAndMaximumOnes(int[][] m) {
        int x=0;
        int p=0;
        for(int i=0;i<m.length;i++){
            int c=m[i][0];
            for(int j=1;j<m[0].length;j++){
                m[i][j]+=c;
                c=m[i][j];
            }
            if(c>x){
                x=c;
                p=i;
            }
        }
        return new int[]{p,x};

    }
    public static void main(String[] args) {
        MaxRow maxRow = new MaxRow();

        int[][] matrix1 = {
            {1, 0, 1, 1},
            {1, 1, 1, 0},
            {0, 0, 1, 1},
            {1, 1, 0, 0}
        };
        int[] result1 = maxRow.rowAndMaximumOnes(matrix1);
        System.out.println("Row with maximum ones: " + result1[0] + ", Number of ones: "
+ result1[1]);



    }
}
```

Output: Row with maximum ones: 0, Number of ones: 3

Time Complexity: O(n+m)

3. Longest consecutive subsequence



Given an array **arr** of non-negative 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.**

Examples:

Input: arr[] = [2, 6, 1, 9, 4, 5, 3]
Output: 6
Explanation: The consecutive numbers here are 1, 2, 3, 4, 5, 6. These 6 numbers form the longest consecutive subsquence.

Input: arr[] = [1, 9, 3, 10, 4, 20, 2]
Output: 4
Explanation: 1, 2, 3, 4 is the longest consecutive subsequence.

Input: arr[] = [15, 13, 12, 14, 11, 10, 9]
Output: 7
Explanation: The longest consecutive subsequence is 9, 10, 11, 12, 13, 14, 15, which has a length of 7.

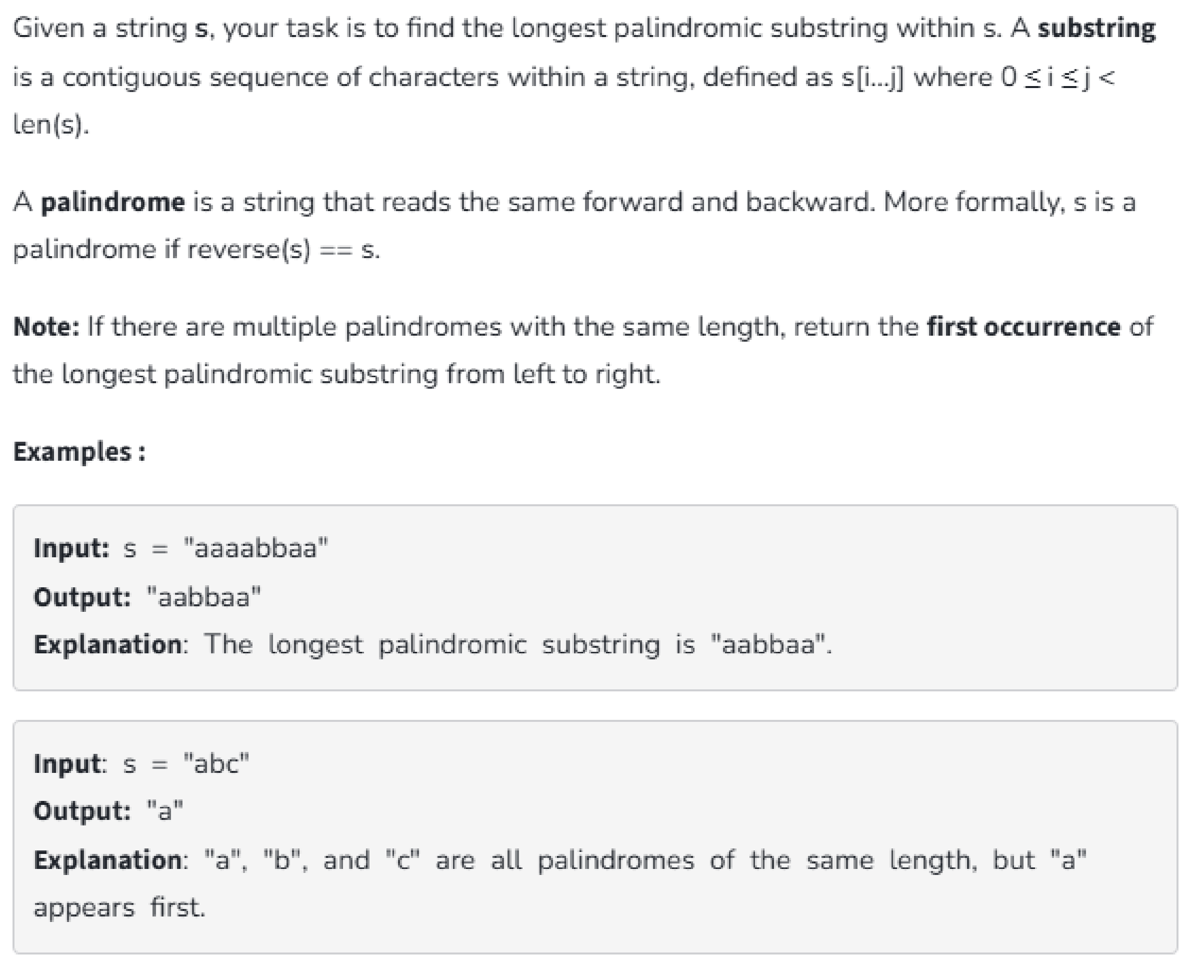
Code:

```
import java.util.HashSet;
import java.util.Arrays;

class LongCon {
    public int findLongestConseqSubseq(int[] arr) {
        if (arr.length == 0) {
            return 0;
        }

        // Sort the array
        Arrays.sort(arr);

        int longestStreak = 1;
        int currentStreak = 1;


        for (int i = 1; i < arr.length; i++) {
            if (arr[i] != arr[i - 1]) {  // Ignore duplicates
                if (arr[i] == arr[i - 1] + 1) {
                    currentStreak++;
                } else {
                    longestStreak = Math.max(longestStreak, currentStreak);
                    currentStreak = 1;
                }
            }
        }


        return Math.max(longestStreak, currentStreak);
    }
    public static void main(String[] args){
        LongCon ob= new LongCon();


        int[] testCase1 = {2, 6, 1, 9, 4, 5, 3};
        System.out.println( ob.findLongestConseqSubseq(testCase1));


    }
}
```

Output:6

Time Complexity: O(nlogn)

4. Longest palindrome in a string



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 ≤ i ≤ j < 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.

Examples :

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

Input: s = "abc"
Output: "a"
Explanation: "a", "b", and "c" are all palindromes of the same length, but "a" appears first.

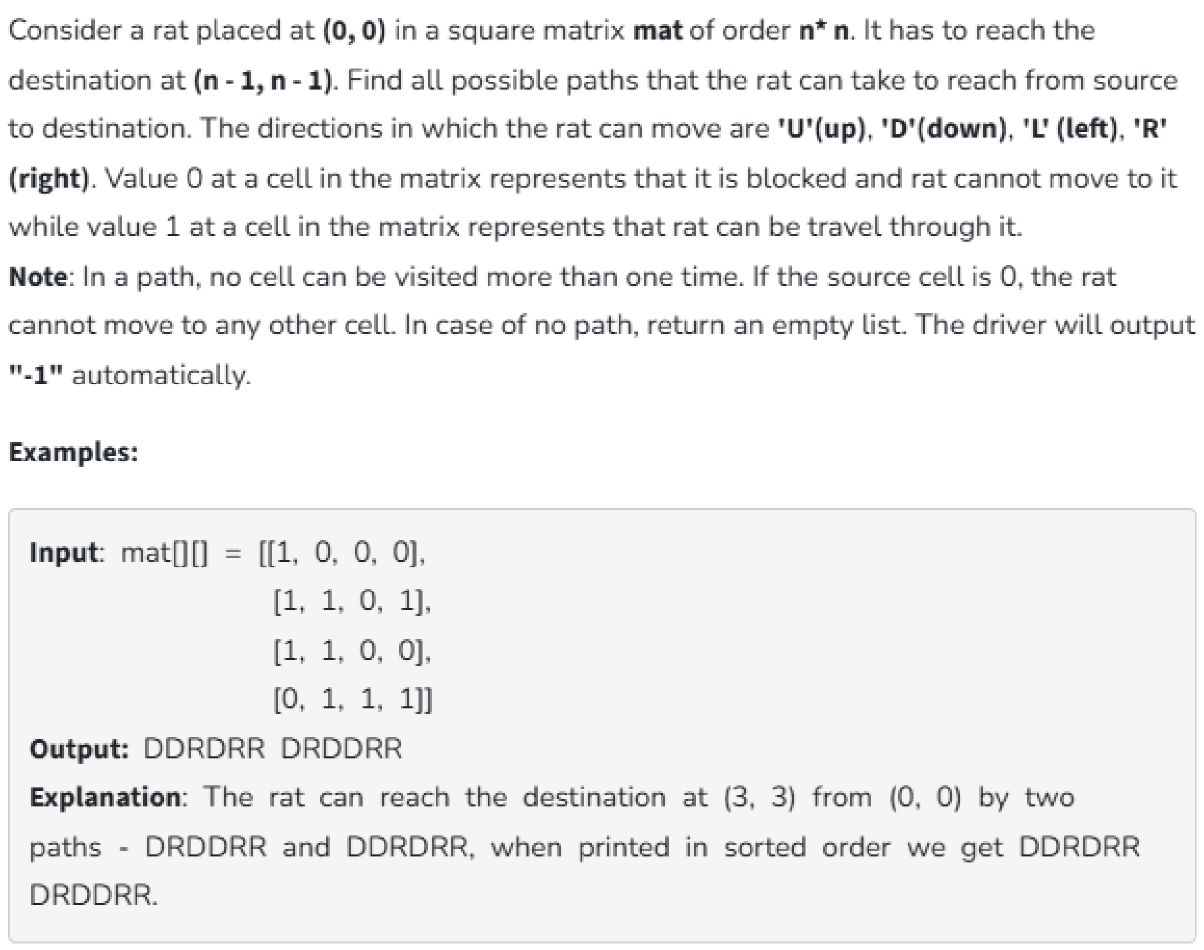
Code:

```
public class LongPal {

    static boolean checkPal(String s, int low, int high) {
        while (low < high) {
            if (s.charAt(low) != s.charAt(high))
                return false;
            low++;
            high--;
        }
        return true;
    }

    static String longestPalSubstr(String s) {
        int n = s.length();
        int maxLen = 1, start = 0;

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                if (checkPal(s, i, j) && (j - i + 1) > maxLen) {
                    start = i;
                    maxLen = j - i + 1;
                }
            }
        }

        return s.substring(start, start + maxLen);
    }

    public static void main(String[] args) {
        String s = "whdhaiueaknfsdnaijdjcjk";
        System.out.println(longestPalSubstr(s));
    }
}
```

Output:hdh

Time Complexity:O(n^3)

5. Rat in a maze problem



Consider a rat placed at **(0, 0)** in a square matrix **mat** 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.
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. In case of no path, return an empty list. The driver will output **"-1"** automatically.

Examples:

```
Input: mat[][] = [[1, 0, 0, 0],
                  [1, 1, 0, 1],
                  [1, 1, 0, 0],
                  [0, 1, 1, 1]]
Output: DDRDRR DRDDRR
Explanation: The rat can reach the destination at (3, 3) from (0, 0) by two paths - DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR DRDDRR.
```

Code:

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

public class Rat{

   static String direction = "DLRU";
   static int[] dr = { 1, 0, 0, -1 };
   static int[] dc = { 0, -1, 1, 0 };

   static boolean isValid(int row, int col, int n, int[][] maze) {
       return row >= 0 && col >= 0 && row < n && col < n && maze[row][col] == 1;
   }
```

```
    static void findPath(int row, int col, int[][] maze, int n, ArrayList<String> ans,
StringBuilder currentPath) {
        if (row == n - 1 && col == n - 1) {
            ans.add(currentPath.toString());
            return;
        }
        maze[row][col] = 0;
        for (int i = 0; i < 4; i++) {
            int nextrow = row + dr[i];
            int nextcol = col + dc[i];
            if (isValid(nextrow, nextcol, n, maze)) {
                currentPath.append(direction.charAt(i));
                findPath(nextrow, nextcol, maze, n, ans, currentPath);
                currentPath.deleteCharAt(currentPath.length() - 1);
            }
        }
        maze[row][col] = 1;
    }

    public static void main(String[] args) {
        int[][] maze = {
            { 1, 0, 0, 0 },
            { 1, 1, 0, 1 },
            { 1, 1, 0, 0 },
            { 0, 1, 1, 1 }
        };

        int n = maze.length;
        ArrayList<String> result = new ArrayList<>();
        StringBuilder currentPath = new StringBuilder();
```

```
if (maze[0][0] != 0 && maze[n - 1][n - 1] != 0) {
    findPath(0, 0, maze, n, result, currentPath);
}

if (result.size() == 0)
    System.out.println(-1);
else
    for (String path : result)
        System.out.print(path + " ");
System.out.println();

int[][] testMaze = {
    { 1, 0, 0, 0 },
    { 1, 1, 0, 1 },
    { 1, 1, 0, 0 },
    { 0, 1, 1, 1 }
};

int testN = testMaze.length;
ArrayList<String> testResult = new ArrayList<>();
StringBuilder testCurrentPath = new StringBuilder();

if (testMaze[0][0] != 0 && testMaze[testN - 1][testN - 1] != 0) {
    findPath(0, 0, testMaze, testN, testResult, testCurrentPath);
}

if (testResult.size() == 0)
    System.out.println(-1);
else
    for (String path : testResult)
```

```
                System.out.print(path + " ");
        System.out.println();
    }
}
```

Output:

```
 DDRDRR DRDDRR
DDRDRR DRDDRR
```

Time Complexity: O(4^(n^2))