DSA Questions

Mogan babu C

Rows with Maximum Ones

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]

Explanation: The row indexed 1 has the maximum count of ones (2). So the answer is [1,2].

Program :

```
import java.util.Scanner;

class MaxOnes {
   public int[] rowAndMaximumOnes(int[][] mat) {
      int index = 0;
      int maxOnes = 0;

      for (int i = 0; i < mat.length; i++) {
         int onesCount = 0;

         for (int j = 0; j < mat[0].length; j++) {
            onesCount += mat[i][j];
         }

         if (onesCount > maxOnes) {
            maxOnes = onesCount;
            index = i;
         }
      }

      return new int[] {index, maxOnes};
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Enter the number of rows: ");
      int rows = scanner.nextInt();

      System.out.print("Enter the number of columns: ");
```

```
        int cols = scanner.nextInt();

        int[][] mat = new int[rows][cols];

        System.out.println("Enter the matrix elements (0 or 1): ");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                mat[i][j] = scanner.nextInt();
            }
        }

        scanner.close();

        MaxOnes solution = new MaxOnes();
        int[] result = solution.rowAndMaximumOnes(mat);
        System.out.println("Row with maximum ones: " + result[0]);
        System.out.println("Maximum count of ones: " + result[1]);
    }
}
```

Output :

Enter the number of rows: 3
Enter the number of columns: 2
Enter the matrix elements (0 or 1):
1 1 0
0 0 1
Row with maximum ones: 0
Maximum count of ones: 2

The time complexity is O(m+n) and space complexity O(n).

2) Valid Anagram :

Given two strings s and t, return true if t is an

anagram of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

1 <= s.length, t.length <= 5 * 104
s and t consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

Program :

```
import java.util.HashMap;
class Solution {
  public boolean isAnagram(String s, String t) {
     HashMap<Character,Integer> map = new HashMap<>();
     HashMap<Character,Integer> T = new HashMap<>();

     if(s.length()!=t.length()) return false;
```

```
        for(int i=0; i<s.length(); i++){
            map.put(s.charAt(i), map.getOrDefault(s.charAt(i),0)+1);
        }
        for(char ch : t.toCharArray()){
            if(!map.containsKey(ch)){
                return false;
            }
            map.put(ch,map.get(ch)-1);
        }
        for(int i : map.values()){
            if(i>0){
                return false;
            }
        }
        return true;

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

        String s1 = "anagram";
        String t1 = "nagaram";
        System.out.println("Is anagram (anagram, nagaram): " +
solution.isAnagram(s1, t1)); // true

        String s2 = "rat";
        String t2 = "car";
        System.out.println("Is anagram (rat, car): " + solution.isAnagram(s2, t2)); //
false

        String s3 = "aabbcc";
        String t3 = "ccbbaa";
```

```
        System.out.println("Is anagram (aabbcc, ccbbaa): " +
solution.isAnagram(s3, t3)); // true

        String s4 = "hello";
        String t4 = "bello";
        System.out.println("Is anagram (hello, bello): " + solution.isAnagram(s4,
t4)); // false
    }
}
```

Output :

Is anagram (anagram, nagaram): true
Is anagram (rat, car): false
Is anagram (aabbcc,ccbbaa): true
Is anagram (hello,bello): false

The time complexity is O(n) and space complexity is O(1)

3) Longest Consecutive Sequence

Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in O(n) time.

Example 1:

Input: nums = [100,4,200,1,3,2]

Output: 4

Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:

Input: nums = [0,3,7,2,5,8,4,6,0,1]

Output: 9

Constraints:

0 <= nums.length <= 105
-109 <= nums[i] <= 109

Program :

```
import java.util.HashSet;

class LongConsecutive {
   public int longestConsecutive(int[] nums) {
      if (nums.length == 0) {
         return 0;
      }
```

```
        HashSet<Integer> set = new HashSet<>();
        for (int num : nums) {
            set.add(num);
        }

        int longest = 0;

        for (int num : set) {
            if (!set.contains(num - 1)) {
                int currentNum = num;
                int length = 1;

                while (set.contains(currentNum + 1)) {
                    currentNum++;
                    length++;
                }

                longest = Math.max(longest, length);
            }
        }

        return longest;
    }

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

        int[] nums1 = {100, 4, 200, 1, 3, 2};
        System.out.println("Longest consecutive sequence length: " +
solution.longestConsecutive(nums1));  // Output: 4

        int[] nums2 = {0, 9, 1, 8, 7, 6, 5};
        System.out.println("Longest consecutive sequence length: " +
solution.longestConsecutive(nums2));  // Output: 5

        int[] nums3 = {1, 2, 0, 1};
```

```
        System.out.println("Longest consecutive sequence length: " +
solution.longestConsecutive(nums3));  // Output: 3
    }
}
```

Output :

Longest consecutive sequence length: 4
Longest consecutive sequence length: 5
Longest consecutive sequence length: 3

The time complexity is O(n) and the space complexity is O(1)

4)Longest Palindrome :

Given a string s which consists of lowercase or uppercase letters, return the length of the **longest palindrome** that can be built with those letters.

Letters are **case sensitive**, for example, "Aa" is not considered a palindrome.

Example 1:

Input: s = "abcccdd"

Output: 7

Explanation: One longest palindrome that can be built is "dccaccd", whose length is 7.

Example 2:

Input: s = "a"

Output: 1

Explanation: The longest palindrome that can be built is "a", whose length is 1.

Constraints:

1 <= s.length <= 2000
s consists of lowercase **and/or** uppercase English letters only.

Program :

```
import java.util.HashMap;

class Palindrome {
   public int longestPalindrome(String s) {
      int m = 0;
      boolean hasOdd = false;
      HashMap<Character, Integer> map = new HashMap<>();
```

```
        for (char l : s.toCharArray()) {
            map.put(l, map.getOrDefault(l, 0) + 1);
        }

        for (int i : map.values()) {
            if (i % 2 == 0) {
                m += i;
            } else {
                m += i - 1;
                hasOdd = true;
            }
        }

        if (hasOdd) {
            m += 1;
        }

        return m;
    }

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

        String s = "bananas";
        int result = Palindrome.longestPalindrome(s);

        System.out.println("The length of the longest palindrome that can be built
is: " + result);
    }
}
```

Output :

The length of the longest palindrome that can be built is: 5

The time complexity is O(n) and the space complexity is O(n).

5) Rat in a Maze Problem - I

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.

Input: mat[][] = [[1, 0],
[1, 0]]
Output: -1
Explanation: No path exists and destination cell is blocked.

Expected Time Complexity: O(3n^2)

Expected Auxiliary Space: O(l * x)

Here l = length of the path, x = number of paths.

Constraints:

$2 \leq n \leq 5$

$0 \leq mat[i][j] \leq 1$

Program :

```
import java.util.ArrayList;
import java.util.List;
public class MazePaths {
    private static final String directions = "DLRU";
    private static final int[] rowMove = {1, 0, 0, -1};
    private static final int[] colMove = {0, -1, 1, 0};
    private static boolean isValid(int row, int col, int n, int[][] maze) {
        return row >= 0 && row < n && col >= 0 && col < n && maze[row][col]
== 1;
    }
    private static void findPath(int row, int col, int[][] maze, int n, List<String>
paths, StringBuilder path) {
        if (row == n - 1 && col == n - 1) {
            paths.add(path.toString());
            return;
        }
        maze[row][col] = 0;
        for (int i = 0; i < 4; i++) {
            int nextRow = row + rowMove[i];
```

```
            int nextCol = col + colMove[i];
            if (isValid(nextRow, nextCol, n, maze)) {
                path.append(directions.charAt(i));
                findPath(nextRow, nextCol, maze, n, paths, path);
                path.deleteCharAt(path.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;
        List<String> result = new ArrayList<>();
        if (maze[0][0] != 0 && maze[n - 1][n - 1] != 0) {
            findPath(0, 0, maze, n, result, new StringBuilder());
        }
        if (result.isEmpty()) {
            System.out.println(-1);
```

```
        } else {
            System.out.println(String.join(" ", result));
        }
    }
}
```

Output :

DDRDRR DRDDRR

Time complexity is O(m*n)

Space Complexity : O(m*n)