IT251 – DATA STRUCTURES & ALGORITHMS II

ASSIGNMENT 4

Name: **Sachin Prasanna**
Roll No.: **211IT058**

Problem Statements:

Problem 1 : (10 Marks)

- **Levenshtein distance** is a string metric to measure the difference between 2 sequences
- Example – distance between 2 words is the minimum number of single –character edits (insertions, deletions or substitutions)
- Write a program to determine the Levenshtein distance given 2 words as inputs
- Example – inputs: kitten, mitten -> Distance is 1 ; inputs: smitten, sitting – Distance is 3 (1 addition; 1 substitution; 1 addition)
- Mandatory Requirement: Demonstrate solving this problem using Dynamic programming method with the code and comments using all possible aspects of DP taught in class (SRTBOT method)
- The program should be able to run and pass tests of inputs of 2 alphabetic string full fledged statements

containing up to 10 words per statement and up to 15 letters in each word. Do not allow numbers and special characters

- Programming languages : C, C++, Python or Java ONLY
- Assume any other valid requirement, if not mentioned above

Problem 2 : (10 Marks)

- Implement a Linked-list representation of disjoint sets in one of the following languages – C, C++, Python or Java
- When running your program,
    - I should be able to create 2 sets given as inputs
    - I should be able to add objects to the linked list in a predictable manner (ie specify number of objects that need to be added and the details of each object)
    - The linked list formed should be as below
        - SET S1 attributes -> SET id, HEAD, TAIL (and other required attrs for Set Ops)
        - All objects in the set S1 should contain the following attributes
            - Pointer to the HEAD
            - Pointer to the TAIL
        - Set S2 same as above
    - Based on the ranks associated with the Sets, you should be able to run the following operations
    - MAKE-SET – Put all the objects collected with unique ids 1->x in Set 1 & x->y in Set 2
    - UNION – Create a Union of the 2 sets formed. Properly demonstrate S2 is destroyed
    - FIND-SET - You should be able to run FIND-SET on individual Sets S1 and S2 before they are joined and also after they are joined.

- Assume any other valid requirement, if not mentioned above

Answers

1)

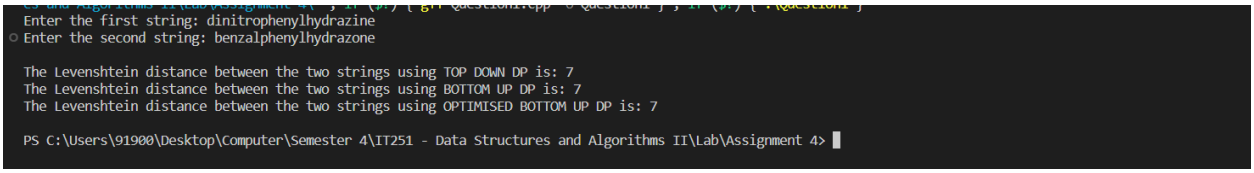
Methods Implemented:

1. Simple Recursion
2. Memoization
3. Tabulation
4. Space optimised Tabulation

The program accepts only alphabets as specified in the question and asks the user to re enter if any other character is inputted.

Outputs (Test Cases inputted):

1)
String str1 = dinitrophenylhydrazine
String str2 = benzalphenylhydrazone

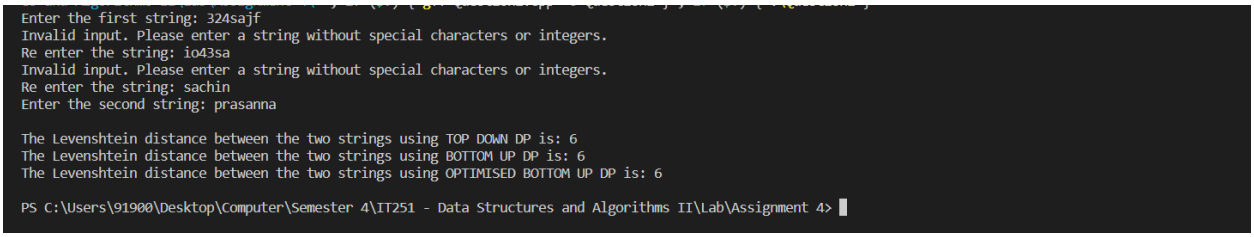


2)
For an Invalid test case, the program asks to re-enter the values

String str1 = 324sajf
//INVALID

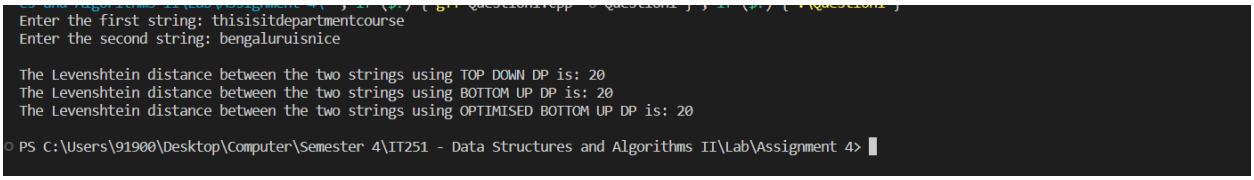
str1 = io43sa
//INVALID

str1 = sachin
str2 = prasanna **//VALID**



3)

String str1 = thisisitdepartmentcourse
String str2 = bengaluruisnice



Analysis of Each Approach:

1. Simple Recursion

The time complexity of this Levenshtein distance algorithm implemented recursively is O(3^n), where n is the length of the longest input string. This is because for each character in one of the strings, there are three possible operations (insertion, deletion, or substitution), and the algorithm recurses over all possible combinations of these operations.

The space complexity is also O(3^n), since in the worst case, the recursive call stack can grow to a depth of n, with three recursive calls being made at each level.

2. Memoization

The time complexity is O(m*n), where m and n are the lengths of the input strings str1 and str2, respectively. This is because the function makes a recursive call for each character in both strings, and each recursive call has three branches.

The space complexity is O(m*n) as well, since a 2D vector (dp) of size m x n is used to store the previously calculated distances.

3. Tabulation

The time complexity is O(mn), where m and n are the lengths of the input strings str1 and str2, respectively. This is because the code fills up an m+1 by n+1 dp vector using nested loops of size O(mn).

The space complexity is also O(mn), since the dp vector requires mn space to store the intermediate results of subproblems.

4. Space Optimised Tabulation

The time complexity is O(m*n), where m and n are the lengths of str1 and str2, respectively, same as before.

The space complexity is O(n), where n is the length of str2. This is because the implementation uses a single vector of length n+1 to store the minimum number of edits required to transform prefixes of str1 into prefixes of str2. It uses this vector to compute the values of the minimum number of edits required for each prefix of str1 and str2 in a bottom-up manner, and only needs to store the values for the current and previous prefixes of str1 as well as the current prefix of str2 in memory.

SRTBOT OF THE MOST OPTIMISED METHOD:

S: The subproblems are finding the minimum number of edits required to transform substrings of str1 and str2 of increasing lengths into each other. Specifically, for i from 1 to m and j from 1 to n, the subproblem is to find the minimum number of edits required to transform the first i characters of str1 into the first j characters of str2.

R: The problem can be broken down into smaller subproblems where we find the minimum number of edits required to transform the first i characters of str1 into the first j characters of str2. This can be recursively computed using the values of previous subproblems.

T: There is no need for a topological sort as there is no dependency between the subproblems.

B: The base case is when one of the strings is empty. In this case, the number of edits required is equal to the length of the non-empty string.

O: The original problem is to find the minimum number of edits required to transform the first m characters of str1 into the first n characters of str2. This can be solved using dynamic programming with an optimised space complexity of O(n).

T: Mentioned before

QUESTION 2) POSTPONED

****