1. Big Data 1 - Assignment 1 - Problem 1

Exercise 1: Commonn subsequence of two strings (3 Points)

Write a function that takes two strings, S1 and S2 at its input and returns the longest common subsequence(LCS) of S1 and S2.
Example:
S1 = "ABAZDC"

S2 = "BACBAD"

LCS = "ABAD"

1.1 Approach for the problem:

I did not want to refer to the internet or take any outside help to solve the problem from a logic perspective. Spent a couple of hours to internalize the problem and solve for small strings on paper. It is easy to solve using visual inspection for smaller strings, but moment strings become larger than 10 characters each it gets harder. Chose Python to get hands-on with coding. I had already decided to do "clean-up" of the input strings to trim leading/ trailing/ middle spaces.

There were two approaches I could come up with:

1) Brute force with recursion: I had settled on an approach requiring parsing the strings one character at a time. Therefore I wanted to find an intersection set of the characters present in the two strings; then reduce the input strings to two working strings that retain only the intersection characters. It would reduce the time for loop traversal. But I faced a hurdle as the number of FOR loops would change depending on the length of the input strings and therefore could not be hardcoded. I spent a lot of time trying to create a recursive function call version to solve this problem.
2) Comparing relative positions of matching characters, after implementing intersections logic from approach-1. First I tried to identify the numerical positions of the characters in say string1 and map them to the matching character in string2. Then somehow parse these numerical positions to identify the longest sub-sequence in one shot. Implement a "cost function" to penalize picking up a match as you traverse from left to right on the input strings. I was unable to complete the idea fully. Also, I realized that there may not be a unique sequence of the maximum length so it would be necessary to consider all the sequences and that got hard with the numerical position storage approach. So decided to simply parse the strings using nested FOR loops logic.

1.2 Algorithm:

Algorithm to find one or more longest common sub-sequence of characters for two input strings.

1. Input the two strings from user. Let us call them ***InputString1*** and ***InputString2***.
2. Create a copy of the input strings. Let us refer to these as ***WorkString1*** and ***WorkString2***.
3. Perform basic cleanup on the working strings by removing any spaces at the start, end and in middle of the strings; and convert to lowercase.
4. Find all characters in the working strings that are common to both working strings without bothering about the relative order of the characters. Let us refer to this as the ***IntersectionSetCharacters***.
5. For each working string, only retain those characters that are present in the IntersectionSetCharacters identified in step 4 by discarding the remaining characters from the working strings. However, ensure to preserve any character repetition and the relative order positions from left to right in the working strings.
6. If required, swap the working strings such that: length of WorkingString1 <= length of WorkingString2.
7. Set a flag as False to indicate the longest common sub-sequence is not found as yet. Let us refer to this as the ***LongestSequenceFoundFlag***. Set the FinalOutputArray as empty. Let us refer to this as the ***FinalLongestCommonSubSequenceArray field***.
8. Initialize the length of sub-sequences to extract = length of WorkingString1. Let us refer to this as the ***WantedLength field***. If required, this value may be updated later by reducing its value in a unitary way.
9. Inspect WorkingString1 and WorkingString2 individually, to identify all sub-sequences that are exactly as long as the current value specified in the WantedLength field. Let us call these ***WorkingString1SubSequences field*** and ***WorkingString2SubSequences field***.
10. Cycle through each element of the set of sub-sequences from WorkingString1 (i.e. WorkingString1SubSequences) and check if it is present in set of sub-sequences from WorkingString2 (i.e. WorkingString2SubSequences).
11. If there are any elements common to both WorkingString1SubSequences and WorkingString2SubSequences, then extract those sub-sequences into the FinalLongestCommonSubSequenceArray field.
12. If the FinalLongestCommonSubSequenceArray is empty, then reduce the WantedLength by 1 and repeat from step 9 onwards. Otherwise proceed to next step.
13. But if the FinalLongestCommonSubSequence is not empty then set the LongestSequenceFoundFlag as True***.***
14. As the answer, display all the sub sequence values stored in the FinalLongestCommonSubSequence field as the one (or more) outputs that are the longest common sub-sequence(s) found for the input strings. Also show a count of the number of such sub-sequences and the value of the length of the longest common sub-sequence.

1.3 Github location of the code:

https://github.com/rbewoor/BigDataAssignment1Submission.git

2. Big Data 1 - Assignment 1 - Problem 2

Exercise 2: Pattern Matching (7 Points)

Write a function that takes two string S1 and S2 at its input and returns a boolean denoting whether S1 matches S2

S2 is a sequence of any number of the following:
a-z - which stands for itself
.(dot) - which matches 1 occurrence of any character
*(star)- which matches 0 or more occurrences of the previous single character

Examples:
S1 = "aba" , S2 = "*ab" => false
S1 = "aa" , S2 = "a*" => true
S1 = "ab" , S2 = ".*" => true
S1 = "ab", S2 = "." => false
S1 = "aab" , S2 = "c*a*b" => true
S1 = "aaa" , S2 = "a*" => true

2.1 Approach for the problem:

Again I have not referenced any external resources to build the logic. The idea is to consider the pattern string in elemental pieces of either a DOT-STAR (.*) , a DOT followed by another DOT/character (.. or .@) , a STAR starting position , a character followed by another character/DOT (@@ or @.) , a character followed by a STAR (@*) . These elementary units are compared against necessary sections from the string to be matched. If the pattern string is exhausted and there are still elementary units present in the string to be matched the output is always FALSE.
But before getting into these elementary units matching, if certain basic trivial conditions are not met then there is no complex comparison at all. E.g. the number of characters that the pattern string could define are too short for the input string then we do not need to consider smaller lengths of the pattern string as we move rightward; or that the input string itself contains non-alphabetic and not STAR/DOT characters then the final output is always false; or if we ever find a DOT-STAR presence in the pattern string it is not necessary to compare the strings actually as the output is always TRUE. Both these situations mean there cannot be any match for the strings being considered.

2.2 Algorithm:

Algorithm to indicate if one string matches a pattern specified by another string.

1. Get user to input two strings. Lets us refer to String1 as the string be matched. String2 as the pattern string. Rules to accept the input are as follows:
    a. String 1 – the input string
        i. Cannot be blank(s) or Null string.
        ii. Upper or lower case alphabets allowed with spaces in between allowed
    b. String 2 – the pattern string

        i. Must consist only of alphabets(lower or upper case), DOT or STAR operator with spaces in between allowed
    c. User will be forced to input correct form of strings to proceed further
2. Copy String1 into WorkingString1 and String2 into WorkingString2
3. Performed following steps to reduce complexity of the pattern string as follows:
    a. Repeating STAR operators collapsed to single STAR. E.g. “abc****nm” becomes “abc*nm”
    b. Repeating DOT-STAR pattern to single DOT-STAR. E.g. “abc.*.*nm” becomes “abc.*nm”
4. Performed BASIC LEVEL TRIVIAL condition checks first:
    a. Disregarding DOT and STAR operators present in WorkingString1, if it is still non-alphabetic, the simply declare FINAL output as FALSE. Exit the program.
    b. If a DOT followed by a STAR operator (designated as DOT-STAR) exists anywhere in the WorkingString2, then as long as the WorkingString1 is not the null string (which it cannot be due to above checks), simply declare the FINAL output as TRUE. Exit the program.
5. Setup a FinalOutputFlag as False.
6. Set a pointer P1 at the start of the WorkingString2.
7. Set up a LoopWorkingString2 = all characters from P1 position till the end of WorkingString2.
8. If we are now already at the end of WorkingString2 and we have still not found a situation of a FINAL output being TRUE, then declare the FINAL output as FALSE and exit the program.
9. Proceed to evaluating Low level Trivial conditions:
    a. Only if LoopWorkingString2 starts with STAR, then Increase P1 by one position and repeat from Step 7. Otherwise proceed further.
    b. Only if the LoopWorkingString2 has no STAR operator at all in it, then number of alphabets and DOTs present in LoopWorkingString2 MUST BE AT LEAST = characters in S1. Compare the WorkingString1 and LoopWorkingString2 based on this logic.
    If you find the length of pattern string insufficient to match any possible value in WorkingString1, then simply declare FINAL output as FALSE and exit the program.
    Otherwise proceed further.
10. Set pointers to the start of WorkingString1 and LoopWorkingString2 strings. Let us refer to these pointers as WStr1Index and LoopWS2Index.
11. Set a flag to indicate complex processing is done for this loop. Set is as FALSE. Let us refer to this as FlagProcessingDoneComplex = False
12. Now evaluate more COMPLEX conditions where actual matching of WorkingString1 and LoopWorkingString2 is going to be required.
    Start evaluating LoopWorkingString2 from left to right using the LoopWS2Index and see which of the scenarios below is present. Carry out the steps mentioned for that condition. After the comparison and increasing of the pointer values, if it is ever found that we have reached the end of LoopWorkingString2 AND we have NOT reached the end of WorkingString1, then simply declare the FINAL output as False and exit the program. However, if WorkingString1 has also ended with the last comparison (or set of comparisons for condition of @* situation) and necessary matching is found, then simply declare the FINAL output as True and exit the

program.
Note that @ represents any lowercase alphabet a to z. This is not the same as DOT operator and is only used as a shortform for readability.
Note that there is no reason to evaluate a DOT-STAR presence as this was already detected as part of Basic Level Trivial condition:

a. Condition 1: @
   There is an @ followed either by another @ or a DOT operator, but not the STAR operator.
   If the characters currently being pointed to by our WStr1Index and LoopWS2Index are different, then simply increase the point P1 by 1 character and repeat from Step 7.
   Otherwise increase both pointers WStr1Index and LoopWS2Index to point to the next characters in their strings.
b. Condition 2: @*
   There is a @ followed by STAR operator.
   If the characters currently being pointed to by our WStr1Index and LoopWS2Index are different, then simply increase the point P1 by 1 character and repeat from Step 7.
   Otherwise, we have found one matching character. Now keep using WStr1Index pointer to move rightward down WorkingString1 until the character is NOT the same as the one being pointed to by the LoopWS2Index on LoopWorkingString2.
c. Condition 3:
   There is a DOT followed by another DOT or an @, not a STAR operator.
   Let there be any character at WStr1Index pointer position. Just increase both WStr1Index and LoopWS2Index.
d. Condition 5:
   There is a @ at the end of LoopWorkingString2.
   Compare the two characters being pointed to in WorkingString1 by the WStr1Index pointer position and LoopWorkingString2 by the LoopWS2Index pointer position. If the characters match then just check to see if WorkingString1 is also finished and simply increase the point P1 by 1 character and repeat from Step 7.
e. Condition 6:
   There is a DOT at the end of LoopWorkingString2.
   followed by another DOT or an @, not a STAR operator.
   Let there be any character at WStr1Index pointer position. Just increase both WStr1Index and LoopWS2Index.

2.3 Github location of the code:

https://github.com/rbewoor/BigDataAssignment1Submission.git