Cairo University
Faculty of Engineering
Credit Hour System

SBEN429 Biomedical Data Analytics
Assignment 4

Name: **Sandra Adel Aziz Gebraiel**

ID: **1180059**

Date: **22/12/2021**

Assignment 4 Dynamic Programming
Longest Common Subsequence

1) Explanation of Code:

A) About the Problem:

Longest Common Subsequence is a modified version of the Edit Distance problem. In the Edit Distance, we find the distance between 2 sequences which is the minimum number letter substitutions (mismatches), insertions and deletions done in one sequence to change it into the other among all possible alignments, so in the end we pull off a global alignment algorithm with the cost of matches = 0 and cost of mismatches, insertions and deletions = 1 (counting the number of changes to be do and after completing the alignment table / cost matrix (referring to the DP solution: optimized), the value of the last cell is taken which represents the total cost of aligning/changing the 1st sequence wholly with/to the 2nd sequence wholly.

As for the Longest Common subsequence, we also globally align the 2 sequences but do not consider the mismatch case, and the goal is find the maximum score of an optimal alignment where there is the most number of matches (longest common subsequence) among all possible alignments, if replace the principle of cost with scores, so matches have a score of 1 and indels a score of 0.

B) Naive Solution using Recursion:

This problem can be solved naively recursively by going top to bottom, where the function receives the 2 sequences as parameters, along with pointers pointing to the index of the last letters of each sequence, and we solve the alignment problem column by column. Since we do not consider mismatches, until we reach the base case of finishing either sequence (pointers turning -1), we check if we have a match in the last column, if we do, then the last column is solved, we add a score of 1 to the score of the alignment between the first i-1 letters of 1st sequence and the second j-1 letters of the 2nd sequence, handling it column b column, which is solved
by the recursive call.

If we do not have a match, we find the maximum score of the 2 possible options of indels (gap in the second sequence: deletion) or (gap in the first sequence: insertion), and take it as the score of the alignment at each recursion step, so we have to go through both options until the end, solving each column with the same methodology.
COMPLEXITY: O(2^n)

which occurs in the worst case when all the letters of both sequences mismatch. So, we have n letters, and for each letter we have 2 options, gap in the second sequence: deletion or gap in the first sequence: insertion, for each we find the score and take the maximum. A complementary lag in the naive solution leading it to exponential time is the overlapping substructure of the problem, where the columns which are checked, the sub-problems to be solved, can be solved more than once in different parts of the tree, leading to many unnecessary calculations

C) Optimized Solution using Dynamic Programming:

Since the problem is of overlapping substructure, we avoid the solving problem which are already solved by saving the solution of each sub-problem in a data structure to be recalled after it is solved only once, which is the alignment matrix/table. This matrix is of size (length of 1st sequence +1 x length of 2nd sequence +1), as there are a zeroth column and zeroth row taken into consideration when solving the matrix. Each cell in the matrix is a subproblem on its own for which we find the optimal solution, so that the optimal solution of each cell leads to the optimal solution of the whole problem, representing the optimal substructure of the problem.

Each cell represents the maximum score of the optimal alignment, where there is the longest common subsequence (greatest number of matches), between the first i letters of the 1st sequence and the first j letters of the second sequence.
The score is calculated based on the maximum of three routes to the cell:

1) Diagonally ONLY if there is match, since we do not consider mismatches, and we add one to the score of the alignment in this case. If this is case, it is always of greater score than the other 2 routes, since the score is incremented by 1 in this route. So, we do not need to calculate the score of the other 2.
2) Horizontally, if there isn't a match, which means there is gap in the sequence on the rows at this cell, and we do not put a score in this case.
3) Vertically, also if there isn't a match which means there is gap in the sequence on the columns at this cell, and we also do not put a score in this case.

The highest score is placed in the cell, solution of this subproblem, as we go row by row, in each, column by column, reaching the last cell starting from the first column and first row. The cells of the zeroth column and row are filled by zeros as initial values for solving the subproblems, as in the first row, we are aligning the 1st sequence with nothing of the 2nd sequence, and in the first column, we are aligning the 2nd sequence with nothing of the first.

Finally, the solution would be the value of the last cell which represents the score of the optimal alignment between the first i letters of 1st sequence, which is here its length, and the first j letters of the 2nd sequence, which is also its length here.

COMPLEXITY: O(n x m)

where n represents length of the first sequence and m the length of the second, which is the size of the part of alignment matrix we loop on, disregarding the zeroth column and row.

2) Pseudocode:

A) Optimized Solution: Dynamic Programming

LongestCommonSubsequence(a, b):

Initialize AllignmentMatrix with zeros

For i = 1 → length of sequence a:
    For j = 1 → length of sequence b:

        If ith letter of a = jth letter of b:

            # Score of the previous diagonal cell
            Match ← AllignmentMatrix[i-1, j-1]
            AllignmentMatrix[i, j] ← Match + 1

        Otherwise:

            # Score of previous horizontal cell
            Insertion ← AllignmentMatrix[i, j-1]

            # Score of previous vertical cell
            Deletion ← AllignmentMatrix[i-1, j]

            AllignmentMatrix[i, j] ← Maximum of (Insertion, Deletion)

Return AllignmentMatrix[length of sequence a, length of sequence b]

B) Naïve Solution: Recursion

LongestCommonSubsequenceNaive(a, b, i, j):

If end of either sequences is reached:
    Return 0

If ith letter of a = jth letter of b:
    Return 1 + LongestCommonSubsequenceNaive(a, b, i-1, j-1)

Otherwise:

    Return Maximum of (LongestCommonSubsequenceNaive(a, b, i-1, j) ,
                        LongestCommonSubsequenceNaive(a, b, i, j-1))

3) Naïve Algorithm and Stress Testing Code:

```
# Naive Solution: Recursive
def lcs2_naive(a, b, i, j):

    # Base case: when we reach the end of either one of both sequences
    if (i == -1 or j == -1):
        return 0

    # We add one to the score of the rest of the alignment if the column
    # we are checking is a match, and we move on to recursively solve the
    # rest of the problem
    if ( a[i] == b[j] ):
        return 1 + lcs2_naive(a, b, i-1, j-1)

    # In case of mismatch, we turn to the maximum of each of the insertion
    # and deletion cases. So, we go through both recursively
    else:
        return max( lcs2_naive(a, b, i-1, j),  lcs2_naive(a, b, i, j-1) )


if __name__ == '__main__':

    '''
    input = sys.stdin.read()
    data = list(map(int, input.split()))

    n = data[0]
    data = data[1:]
    a = data[:n]

    data = data[n:]
    m = data[0]
    data = data[1:]
    b = data[:m]

    print(lcs2(a, b))
    #print(lcs2([10**9]*100, [-10***9]*100))
    '''

    while ( 1 ):
            a = []
            n = r.randint(1, 15)
            print('n: ', n)
            for i in range(n):
                a.append(r.randint(1, 20))
            print('a: ', a)
```

```
        b = []
        m = r.randint(1, 15)
        print('m: ', m)
        for i in range(m):
            b.append(r.randint(1, 20))
        print('b: ', b)

        res1 = lcs2_naive(a, b, len(a)-1, len(b)-1)
        res2 = lcs2(a, b)
        print(res2)

        if res1 != res2:
            print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
            break
        else:
            print('OK')
```

4) Resources:

Longest Common Subsequence (opengenus.org)

TopCoder Feature Articles

Longest Common Subsequence (tutorialspoint.com)

MOOC: Algorithmic Toolbox Slides - Week 5