Correctness/Optimization

We first made our own algorithm (naive method) for implementing longest common subsequence functionality in plagiarism checking between two strings. In that algorithm, we supposed first string X be of length m while second string Y of length n. After checking every sentence (subsequence) of X whether it is also present in Y, we returned the whole sentences that were same between the two input strings. But the problem with that approach was- its time complexity was high i.e. There were 2^m combinations of X and the testing combinations whether or not it was present in Y took O(n) time. Ultimately, the naive method could take $O(n2^m)$ time in its execution which was very high.

Because we had to implement the second task in our Project i.e. if a user inputs a folder containing many files for checking plagiarism, our algorithm would take much time for checking. Therefore, we implemented Dynamic Programming in our Plagiarism Checking tool. Of course, it had some space complexity as well, but according to us it was a good compromise.

Complexity

We will find the time and space complexity of Longest Common Subsequence using Dynamic Programming. Before Finding complexities, we should know about the Dynamic Programming Design Recipe that is given below:

1. Characterize the structure of Optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an Optimal solution (typically in bottom up fashion).

4. Construct an Optimal solution from computed information.

1. Characterize the structure of Optimal solution

Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of Z string (subsequence), we will consider the following three cases:

Case I:

When length of X or Y string is 0, then there will no subsequence between them.

it means $Z=Z_0$

Case II:

When last character of both strings matches

For example, when

X=ABCD & Y=AED

In the above case, D which is the last character of both strings must be in our final subsequence. i.e. our subsequence will look like "_____D". Thus we may write as

$Z_k=Z_{k-1}U\ D$

Case III:

When last character of both strings does not match

For example, when

X=ABDF & Y=EBCFD

In the above case, there can be two possibilities. First one is if we eliminate last character from X i.e. F, then both X and Y will have same last letter

X=ABD

Y=EBCFD

So, $Z=LCS(X_{m-1},Y_n)$

The second possibility is if we eliminate last character from Y i.e. D, then compare X and Y.

X=ABDF

Y=EBCF

So, $Z=LCS(X_m,Y_{n-1})$

In both possibilities, the one giving maximum answer (substring) will be our final answer.

2. Recursively define the value of an optimal solution.

We iterate through two dimensional loops of lengths m and n and use the following algorithm to update LCX(X,Y).

- If any of the string X or Y is 0 ($X=X_0$ or $Y=Y_0$) , then LCX(X,Y)= 0.
- if $X_m = Y_n$,i.e. when the characters at ith and jth index matches, $LCX(X,Y)= 1 + LCX(X_{m-1},Y_{n-1})$.
- Otherwise, store the maximum value we get after considering either the character X_m or the character Y_n i.e. $LCX(X,Y)= \max(LCX(X_m,Y_{n-1}), LCX(X_{m-1},Y_n))$.

3. Compute the value of an Optimal solution

Let X=BDCABA & Y=ABCBDAB

	X_0	B	D	C	A	B	A
Y_0		0	0	0	0	0	0
A	0	←0	←0	←0	↖1	←1	↖1
B	0	↖1	←1	←1	←1	↖2	←2
C	0	↑1	←1	↖2	←2	←2	←2
B	0	↖1	←1	↑2	←2	↖3	←3
D	0	↑1	↖2	←2	←2	↑3	←3
A	0	↑1	↑2	←2	↖3	←3	↖4
B	0	↖1	↑2	←2	↑3	↖4	←4

The above table is filled according to the following rules:

1. If any of the string is empty or if both are empty, then we fill 0 (base case).
2. If character matches, then add 1 in the diagonal cell number and fill up.
3. If character does not match, then fill up the cell with maximum value from either above or below cell.

Also for telling the actual subsequence, we need to back track values in the cells i.e. we are starting from 4 (number in the end) and will be highlighting numbers one by one.

	X_0	B	D	C	A	B	A
Y_0		0	0	0	0	0	0
A	0	←0	←0	←0	↖1	←1	↖1
B	0	↖1	←1	←1	←1	↖2	←2
C	0	↑1	←1	↖2	←2	←2	←2
B	0	↖1	←1	↑2	←2	↖3	←3
D	0	↑1	↖2	←2	←2	↑3	←3
A	0	↑1	↑2	←2	↖3	←3	↖4
B	0	↖1	↑2	←2	↑3	↖4	←4

Thus, we may write common string as BDAB, which is the longest common subsequence.

4. Construct an Optimal solution from computed information.

We may finally write pseudocode as:

PRINT-LCS (X, Y)

1. m = X.length
2. n = Y.length
3. Z[m+1][n+1] *//It contains length of LCS of X[0..i-1] and Y[0..j-1]*
4. **FOR** i = 1 to m
5. **FOR** j = 1 to n
6. **IF** i == 0 OR j == 0

```
7.          Z[i][j] = 0
8.        ELSE IF X[i-1] == Y[j-1]
9.            Z[i][j] = Z[i-1][j-1] + 1
10.       ELSE
11.           Z[i][j] = max(Z[i-1][j], Z[i][j-1])
12.  index = Z[m][n]          //For printing out the LCS
13.  lcs[index+1]             //character array for storing same string
14.  lcs[index] = '\0'
15.  i = m
15.  j = n
16.  WHILE i > 0 AND j > 0
17.     IF X[i-1] == Y[j-1]
18.         lcs[index-1] = X[i-1]
19.         i –
20.         j –
21.         index –
22.     ELSE IF Z[i-1][j] > Z[i][j-1]
23.         i –
24.     ELSE
25.         j –
26.  RETURN lcs
```

For telling the time and space complexity, we can clearly see that there are two FOR loops for both the strings (line 4, 5), therefore the time complexity of finding the longest common subsequence using dynamic programming approach is **O(m*n)** where m and n are the lengths of the strings X and Y respectively. Since this implementation involves only m rows and n columns for building LCX (X,Y), thus the space complexity would also be **O(m*n)**.