LONGEST COMMON SUBSEQUENCE PROBLEM

Problem Statement : The Longest Common Subsequence (LCS) problem is as follows. We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.

Solution :
For example, consider : S = ABAZDC and T = BACBAD

In this case, the LCS has length 4 and is the string ABAD. Another way to look at it is we are finding a 1-1 matching between some of the letters in S and some of the letters in T such that none of the edges in the matching cross each other.

For instance, this type of problem comes up all the time in genomics: given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up.

Let's now solve the LCS problem using Dynamic Programming. As subproblems we will look at the LCS of a prefix of S and a prefix of T, running over all pairs of prefixes. For simplicity, let's worry first about finding the length of the LCS and then we can modify the algorithm to produce the actual sequence itself.

So, here is the question: say LCS[i,j] is the length of the LCS of S[1..i] with T[1..j]. How can we solve for LCS[i,j] in terms of the LCS's of the smaller problems?

Case 1: what if S[i] != T[j]? Then, the desired subsequence has to ignore one of S[i] or T[j] so we have:
LCS[i, j] = max(LCS[i − 1, j], LCS[i, j − 1]).

So, we can just do two loops (over values of i and j) , filling in the LCS using these rules. Here's what it looks like pictorially for the example above, with S along the leftmost column and T along the top row

We just fill out this matrix row by row, doing constant amount of work per entry, so this takes O(mn) time overall. The final answer (the length of the LCS of S and T) is in the lower-right corner.

How can we now find the sequence? To find the sequence, we just walk backwards through matrix starting the lower-right corner. If either the cell directly above or directly to the right contains a value equal to the value in the current cell, then move to that cell (if both to, then chose either one). If both such cells have values strictly less than the value in the current cell, then move diagonally up-left (this corresponts to applying Case 2), and output the associated character. This will output the characters in the LCS in reverse order. For instance, running on the matrix above, this outputs DABA.

We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming, that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming".

Basic Idea (version 2): Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like T(n) = 2T(n−1) + n. However, suppose that many of the subproblems you reach as you go down the recursion tree are the same. Then you can hope to get a big savings if you store your computations so that you only compute each different subproblem once. You can store these solutions in an array or hash table. This view of Dynamic Programming is often called

memoizing. For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  return result;
}
```

This algorithm runs in exponential time. In fact, if S and T use completely disjoint sets of characters (so that we never have S[n]==T[m]) then the number of times that LCS(S,1,T,1) is recursively called equals

$$\binom{n+m-2}{m-1}.^{1}$$

In the memoized version, we store results in a matrix so that any given set of arguments to LCS only produces new work (new recursive calls) once. The memoized version begins by initializing arr[i][j] to unknown for all i,j, and then proceeds as follows

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
  return result;
}
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven't already computed the answer in line (*)

In this memoized version, our running time is now just O(mn). One easy way to see this is as follows. First, notice that we reach line (**) at most mn times (at most once for any given value of the parameters). This means we make at most 2mn recursive calls total (at most two calls for each time we reach that line). Any given call of LCS involves only O(1) work (performing some equality checks and taking a max or adding 1), so overall the total running time is O(mn)

Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version pays a penalty in recursion overhead, but can potentially be faster than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand

Worked example :
The longest subsequence common to R = (GAC), and C = (AGCAT) will be found. Because the LCS function uses a "zeroth" element, it is convenient to define zero prefixes that are empty for these sequences: R0 = Ø; and C0 = Ø. All the prefixes are placed in a table with C in the first row (making it a column header) and R in the first column (making it a row header).

LCS Strings

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø					
A	Ø					
C	Ø					

This table is used to store the LCS sequence for each step of the calculation. The second column and second row have been filled in with Ø, because when an empty sequence is compared with a non-empty sequence, the longest common subsequence is always an empty sequence.

LCS(R1, C1) is determined by comparing the first elements in each sequence. G and A are not the same, so this LCS gets (using the "second property") the longest of the two sequences, LCS(R1, C0) and LCS(R0, C1). According to the table, both of these are empty, so LCS(R1, C1) is also empty, as shown in the table below. The arrows indicate that the sequence comes from both the cell above, LCS(R0, C1) and the cell on the left, LCS(R1, C0).

LCS(R1, C2) is determined by comparing G and G. They match, so G is appended to the upper left sequence, LCS(R0, C1), which is (Ø), giving (ØG), which is (G).

For LCS(R1, C3), G and C do not match. The sequence above is empty; the one to the left contains one element, G. Selecting the longest of these, LCS(R1, C3) is (G). The arrow points to the left, since that is the longest of the two sequences.

LCS(R1, C4), likewise, is (G).

LCS(R1, C5), likewise, is (G).

"G" Row Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑ ←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø					
C	Ø					

For LCS(R2, C1), A is compared with A. The two elements match, so A is appended to Ø, giving (A).

For LCS(R2, C2), A and G do not match, so the longest of LCS(R1, C2), which is (G), and LCS(R2, C1), which is (A), is used. In this case, they each contain one element, so this LCS is given two subsequences: (A) and (G).

For LCS(R2, C3), A does not match C. LCS(R2, C2) contains sequences (A) and (G); LCS(R1, C3) is (G), which is already contained in LCS(R2, C2). The result is that LCS(R2, C3) also contains the two subsequences, (A) and (G).

For LCS(R2, C4), A matches A, which is appended to the upper left cell, giving (GA).

For LCS(R2, C5), A does not match T. Comparing the two sequences, (GA) and (G), the longest is (GA), so LCS(R2, C5) is (GA).

"G" & "A" Rows Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	↑←(A) & (G)	↑←(A) & (G)	↖(GA)	←(GA)
C	Ø					

For LCS(R3, C1), C and A do not match, so LCS(R3, C1) gets the longest of the two sequences, (A).

For LCS(R3, C2), C and G do not match. Both LCS(R3, C1) and LCS(R2, C2) have one element. The result is that LCS(R3, C2) contains the two subsequences, (A) and (G).

For LCS(R3, C3), C and C match, so C is appended to LCS(R2, C2), which contains the two subsequences, (A) and (G), giving (AC) and (GC).

For LCS(R3, C4), C and A do not match. Combining LCS(R3, C3), which contains (AC) and (GC), and LCS(R2, C4), which contains (GA), gives a total of three sequences: (AC), (GC), and (GA).

Finally, for LCS(R3, C5), C and T do not match. The result is that LCS(R3, C5) also contains the three sequences, (AC), (GC), and (GA).

Completed LCS Table

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	↑←(A) & (G)	↑←(A) & (G)	↖(GA)	←(GA)
C	Ø	↑(A)	↑←(A) & (G)	↖(AC) & (GC)	↑←(AC) & (GC) & (GA)	↑←(AC) & (GC) & (GA)

The final result is that the last cell contains all the longest subsequences common to (AGCAT) and (GAC); these are (AC), (GC), and (GA). The table also shows the longest common subsequences for every possible pair of prefixes. For example, for (AGC) and (GA), the longest common subsequence are (A) and (G).

My program output :

```
sumanth@IIITA /media/sumanth/Secondary/Subjects/RHM $ python3 LCS.py
Enter the First Sequence
AAATTCGGAGTTGTAGATGCTCAATACTCCAATCGGTTTTTTCGTGCACCACCGCGGGTGGCTGACAAGGGTTTGACATCGAGAAACAAGGCA

Enter the Querying Sequence
GTTCCGGGCTGAAAGTAGCGCCGGGTAAGGTACGCGCCTGGTATGGCAGGACTATG
('AAATTCGGAGTTGTAGATGCTCAATACTCCAATCGGTTTTTTCGTGCACCACCGCGGGTGGCTGACAAGGGTTTGACATCGAGAAACAAGGCA', 'GTTCCGGGCTGAAAGTAGCGCCGGGTAAGGTACGCGCCTGGTATG
GCAGGACTATG')


LCS of given two DNA sequences is TTCGGGTGAAATACCCGGGTAAGGTCGCGGGTATGGCAGGCA
and length of it is : 42
```

Practical Application :

1. Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene's function.
2. In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal growth factor (PDGF) gene.

3. Gene similarities between two genes with known and unknown function alert biologists to some possibilities.
4. Computing a similarity score between two genes tells how likely it is that they have similar functions
5. Dynamic programming is a technique for revealing similarities between genes
6. We find the simillarity score using LCS algorithm which uses Dynamic programming.

References :

1. CS CMU Notes, Design and Analysis of Algorithms – steven felix,steven halim – January 26 2015.
2. Wikipedia :- *https://en.wikipedia.org/wiki/Longest_common_subsequence_problem*
3. Lecture notes of Time's institute : https://time.mk/trajkovski/teaching/eurm/bio/lecture5.pdf