Longest common sub-sequence problem

You are given two strings, S and T, and want to find the longest subsequence that appears in both strings, where the characters of the subsequence do not have to appear consecutively in the original strings.

Examples:
LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.
LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

So a string of length n has 2^n different possible
sub-sequences.

The straightforward recursive algorithm can be written in pseudocode as:

```
function LCS(S, T) is
    if S is empty or T is empty then return empty string

    if first char of S == first char of T then
        return (first char of S) + LCS(S - first char, T - first char)
    otherwise // first chars are different
        return longer of LCS(S - first char, T) and LCS(S, T - first char)
```

In the *first case*, one or both of the strings is empty, so the longest common subsequence is empty.

In the *second case*, both strings begin with the same character, so we declare that character to be part of the longest common subsequence and recursively calculate the longest common subsequence of the remaining characters.

In the *third case*, the strings begin with different characters, so at least one of the first character of S or the first character of T is not part of the longest common subsequence. Therefore, we recursively calculate the longest common subsequences that we would get by dropping the first character of S and by dropping the first character of T, and keep whichever answer is longer.

Overlapping Sub-problems

Consider what happens in the recursive LCS algorithm on inputs such as "ABCDE" and "FGHI". The progress of the algorithm can be summarized as

```
LCS("ABCDE", "FGHI")
= longer of { LCS("BCDE","FGHI"),
LCS("ABCDE","GHI") }
= longer of {
longer of { LCS("CDE","FGHI"),
LCS("BCDE","GHI") },
longer of { LCS("BCDE","GHI"),
LCS("ABCDE","HI") }}
= longer of {
longer of {
longer of { LCS("DE","FGHI"), LCS("CDE","GHI") },
longer of { LCS("CDE","GHI"), LCS("BCDE","HI") }},
longer of {
longer of { LCS("CDE","GHI"), LCS("BCDE","HI") },
longer of { LCS("BCDE","HI"), LCS("ABCDE","I") }}}
= ...
```

Notice how certain subproblems appear over and over again. Already we see three calls to LCS("CDE","GHI") and another three calls to LCS("BCDE","HI"). Eventually we'll end up with a whopping 35 calls to LCS("E","I").

With Memoization

The memo table can be implemented as a hash table. This is a very common representation of memo tables, especially when the inputs are strings.

Instead of passing around entire strings, we can pass around indices into those strings. Then the memo table can be a simple two-dimensional array indexed by integers. The catch is that we need arrays of different sizes whenever we call the main function with different initial strings. So we allocate a new array when we start the main function, and deallocate the array when we finish. We use an index of 0 to indicate that we are at the beginning of a string, and an index one past the last position in the string to indicate that we have reached the end of the string.

The memoization-based algorithm processes the memo table in a top-down, demand-driven fashion. Taking the final step to full-fledged DP involves processing the memo table bottom-up, instead.

DP throws away the recursion and simply focuses on filling in the table, using a few loops to initialize the table from the smallest subproblems to the biggest subproblems.

```
function LCS(S, T) is
    allocate an array A[0..length of S, 0..length of T]
    for i = length of S downto 0 do
        for j = length of T downto 0 do
            if i == length of S or j == length of T then
                A[i,j] = empty string
            else if S[i] == T[j] then
                A[i,j] = S[i] + A[i+1,j]
            else
                A[i,j] = longer of A[i+1,j] and A[i,j+1]

    answer = A[0,0]
    deallocate A
    return answer
```

In the recursive algorithm, it was most natural to obtain subproblems by peeling off characters from the fronts of the strings. But that means that the smaller subproblems are at the backs of the strings, which in turn means that, if you want to process the table from smallest subproblems to biggest subproblems, you end up working backward.

With DP, however, it is probably more natural to work front to back. Fortunately, this is a very easy change to make.

```
function LCS(S, T) is
    allocate an array A[0..length of S, 0..length of T]
    for i = 0 upto length of S do
        for j = 0 upto length of T do
            if i == 0 or j == 0 then
                A[i,j] = empty string
            else if S[i-1] == T[j-1] then
                A[i,j] = A[i-1,j] + S[i-1]
            else
                A[i,j] = longer of A[i-1,j] and A[i,j-1]
    answer = A[length of S,length of T]
    deallocate A
    return answer
```

This answer is fine as is, but aficionados of DP often take the further step of initializing the base cases outside the main loop, as in the following version.

```
function LCS(S, T) is
    allocate an array A[0..length of S, 0..length of T]

    // initialize base cases
    for i = 0 upto length of S do
        A[i,0] = empty string
    for j = 0 upto length of T do
        A[0,j] = empty string

    // main loop
    for i = 1 upto length of S do
        for j = 1 upto length of T do
            if S[i-1] == T[j-1] then
                A[i,j] = A[i-1,j] + S[i-1,j]
            else
                A[i,j] = longer of A[i-1,j] and A[i,j-1]

    answer = A[length of S,length of T]
    deallocate A
    return answer
```

Reference

- topcoder notes url