CPSC 320: DP in 2-D*

The Longest Common Subsequence of two strings `A` and `B` is the longest string whose letters appear in order (but not necessarily consecutively) within both `A` and `B`. For example, the LCS of "computer science" and "mathematics" is the length 5 string `mteic` ("co**m**pu**te**r sc**i**en**c**e" and "**m**a**th**emat**ic**s"). Biologists: If these were DNA base or amino acid sequences, can you imagine how this might be a useful problem?

1. Write down small and trivial instances of the problem.

   SOLUTION: A natural trivial instance is two empty strings. Their LCS is also the empty string. In fact, that generalizes to any instance where one of the strings (`A` or `B`) is empty. In that case, the LCS will also be empty.

   Here are some other small instances:

   - "a" and "a": LCS is "a" of length 1. (We could think of that is one more than what we get recursing on the two empty strings formed when we set aside the letter "a" for inclusion in the LCS.)
   - "a" and "b": LCS is the empty string ("") of length 0.
   - "ab" and "a": Somehow, we need to "strip off" the "b" from the first string to get at the LCS of "a".

2. Now, working backward from the end (i.e., from the last letters, as with the change-making problem where we worked from the total amount of change desired down to zero), let's figure out the first choice we make as we break the problem down into smaller pieces:

   (a) Consider the two strings `compute` and `science`. Describe the relationship of the length of their LCS with the length of the LCS of `comput` and `scienc` (strings `A` and `B` with both of their last letters removed).

   SOLUTION: The length of the LCS of `compute` and `science` is one longer than the length of the LCS of `comput` and `scienc` because we "lose" the final matching letter `e`, which is part of the LCS of the original strings.

   (b) Now consider the two strings `tycoon` and `country`. Describe the relationship of the length of their LCS with the length of the LCS of `tycoon` and `countr` (the same string `A`, and string `B` with its last letter removed).

   SOLUTION: The length of the LCS of `tycoon` and `country` is at least as great as the LCS of `tycoon` and `countr`. Indeed, the LCS of `tycoon` and `countr` is also a common subsequence (if not necessarily the longest one) of `tycoon` and `country`.

3. Given two strings `A` and `B` of length $n > 0$ and $m > 0$, we will denote the length of the LCS of `A` and `B` by `LLCS(A[1..n], B[1..m])`. Describe `LLCS(A[1..n], B[1..m])` as a recurrence relation over smaller instances. Use and generalize your work in the previous problems!

   SOLUTION: If the last letters match, then we are in a situation like with `compute` and `science`: we can move to a subproblem that excludes the final letters of both strings, noting that the LCS of the original strings is one longer than the LCS of the shorter strings.

   Otherwise, either the last character of `A` or the last character of `B` (or both) is not in the LCS. So, we can recurse twice, once with each string truncated, as in the `tycoon` and `countr` instance. That gives us (for $n > 0$ and $m > 0$):

$$\begin{aligned}&\text{LLCS}(A[1..n], B[1..m])\\ &\qquad = \begin{cases} \text{LLCS}(A[1..n-1], B[1..m-1]) + 1, & \text{if } A[n] = B[m],\\ \max\{\text{LLCS}(A[1..n-1], B[1..m]), \text{LLCS}(A[1..n], B[1..m-1])\}, & \text{otherwise .}\end{cases}\end{aligned}$$

   Or equivalently, we can express the recurrence as pseudocode:

```
LLCS(A[1..n], B[1..m]) =

  if A[n] = B[m] then

    return LLCS(A[1..n-1], B[1..m-1]) + 1

  else return the maximum of

    LLCS(A[1..n-1], B[1..m]) and

    LLCS(A[1..n], B[1..m-1])
```

4. Given two strings `A` and `B`, if either has a length of `0`, what is the length of their LCS?

   SOLUTION: This is our trivial case. The length in this case is zero.

5. Convert your recurrence into a memoized solution to the `LLCS` problem.

   SOLUTION: We'll introduce a call that initializes a table and initiates the recursion. The dimensions of the table are $n + 1$ and $m + 1$—the extra "+1" is there so that we can store the base cases. Make sure that your memoized algorithm initializes the table (this is often not necessary with a dynamic programming algorithm).

```
procedure MEMO-LLCS(A[1..n], B[1..m])
    create a 2-dimensional array Soln[0..n][0..m]
    initialize all elements of Soln to −1
    MEMO-HELPER(A[1..n]B[1..m])

procedure MEMO-HELPER(A[1..i], B[1..j])
    ▷ As always with memoization, we wrap the recurrence with a check
    ▷ to see if the Soln array already contains the answer. If not, compute
    ▷ the answer via the recurrence and store it. If so, just return the answer
    if Soln[i][j] is −1 then
        if i == 0 or j == 0 then
```

```
        Soln[i][j] ← 0
    else if A[i] == B[j] then
        Soln[i][j] ← MEMO-HELPER(A[1..i − 1], B[1..j − 1]) + 1
    else
        Soln[i][j] ←
            max{MEMO-HELPER(A[1..i-1],B[1..j]), MEMO-HELPER(A[1..i],B[1..j-1])}
        return Soln[i][j]
```

6. Complete the following table to find the length of the LCS of `tycoon` and `country` using your memoized solution. (The row and column headed with an ϵ, denoting the empty string, are for the trivial cases!)

SOLUTION:

	ϵ	c	co	cou	coun	count	countr	country
ϵ	0	0	0	0	0	0	0	0
t	0	0	0	0	0	1	1	1
ty	**0**	0	0	0	0	1	1	2
tyc	0	**1**	1	1	1	1	1	2
tyco	0	**1**	2	2	2	2	2	2
tycoo	0	1	**2**	**2**	2	2	2	2
tycoon	0	1	2	2	**3**	**3**	**3**	**3**

7. Go back to the table and extract the actual LCS from it. Circle each entry of the table you have to inspect in constructing the LCS. Then, use the space below to write an algorithm that extracts the actual LCS from an `LLCS` table.

SOLUTION: We've colored in blue and bolded (rather than circled) the entries used above. (In fact, we could equivalently have gone some slightly different routes around the "ou" in "country" and the "oo" in "tycoon".) Critically, the table tells us the value of the recurrence at each cell, and the recurrence tells us which cell we need next to reconstruct more of the solution. Our understanding of the recurrence tells us that when it adds one to the length, that's because we've found one letter of the LCS itself. Otherwise, the LCS at the next cell is the same as the LCS at the current one.

```
procedure EXPLAIN-LCS(A[1..n], B[1..m], Soln)
    ▷ Note: Soln]0..n][0..m] is a filled-in LLCS memoization table for A and B

    if n == 0 or m == 0 then ▷ base case
        return ""
    else
        if A[n] == B[m] then
            ▷ the final letters match, so, we add a letter to the LCS
            return EXPLAIN-LCS(A[1..n − 1], B[1..m − 1], Soln) +A[n]
        else
            ▷ which recursive call yielded the max?
            if Soln[n − 1][m] ≥ Soln[n][m − 1] then
                ▷ we don't use the last letter of A in the solution
                return EXPLAIN-LCS(A[1..n − 1], B[1..m], Soln)
            else
                ▷ we don't use the last letter of B in the solution
                return EXPLAIN-LCS(A[1..n], B[1..m − 1], Soln)
```

8. Give an iterative solution that produces the same table as the memoized solution.

   SOLUTION: We need to find an order to traverse the table such that by the time we require the value of any table cell, it has already been calculated. There are **many** working orders, but we'll fill in the table column by column, from top to bottom. We don't need to initialize the table entries in this algorithm, we fill them in directly as we go.

```
procedure DP-LLCS(A[1..n], B[1..m])
    create a 2-dimensional array Soln[0..n][0..m]

    ▷ fill in the base cases
    for i from 0 to n do
        Soln[i][0] ← 0
    for j from 1 to m do
        Soln[0][j] ← 0
    ▷ fill in the recursive cases column-by-column, top-to-bottom
    for i from 1 to n do
        for j from 1 to m do
            if A[i] = B[j] then
                Soln[i][j] ← Soln[i − 1][j − 1] + 1
            else
                Soln[i][j] ← max{ Soln[i − 1][j], Soln[i][j − 1] }
    return Soln[n][m]
```

9. Analyze the efficiency of your memoized (part 5) and dynamic programming (part 8) algorithms in terms of runtime and memory use (not including the space used by the parameters). You may assume the strings are of length n and m, where $n \leq m$ (without loss of generality).

   SOLUTION:

   Both the memoized and dynamic programming solutions fill out each cell of the table exactly once. How long does it take to fill out that cell? Not counting recursive calls (for the memoized solution), filling out a table cell takes constant time: the max over three simple expressions. There are $n * m$ table cells.

   Therefore, both versions take $O(nm)$ time.

   Assuming each table entry takes constant space, both versions also take $O(nm)$ space. (The memoized version uses additional space for the call stack, but the deepest the call stack gets is $O(n+m)$, which is dominated by the table size.)

   The Explain-LCS algorithm stops as soon as it reaches a base case, which takes $O(n+m)$ steps. It never takes a "wrong" turn, and so this is also its runtime. (It uses only constant space beyond the already-stored table, but it does critically rely on that table, as we've currently designed it.)

10. If we only want the **length** of the LCS of `A` and `B` with lengths n and m, where $n \leq m$, explain how we can "get away" with using only $O(n)$ memory in the dynamic programming solution.

    SOLUTION: As in our previous dynamic programming problem, we can store only a constant number of entries along one dimension of the table. In this case, each cell requires the entries above, to the left of, and diagonally above and to the left of itself. Given the traversal order we chose for our DP algorithm above, we can store just one old "column" of the table (one space left of the current column). The recurrence never requires looking further back than that.

    At any time, then, we'll have two columns in memory: the current and previous ones. Each column has $n + 1$ entries, for $O(n)$ memory.

In fact, a solution that uses only $n+2$ entries exists, because the only entries from the previous column we will need after we've computed Soln[i][j] are Soln[$i-1$][j], Soln[$i-1$][$j+1$], ..., Soln[$i-1$][n], and similarly the only entries from the current column we have are Soln[i][0], Soln[i][1], ..., Soln[i][j].