CPSC 320 Sample Solution: DP in 2-D

March 18, 2018

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 `eleanor` and `naomi` is the length 2 string `no` (or equivalently the length 2 string `ao`).

(Biologists: If these were DNA base or amino acid sequences, can you imagine how this might be a useful problem?)

1. Write at least three trivial or small instances and their solutions.

   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 of "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 of the empty string ("")
   - "ab" and "a": Somehow, we need to "strip off" the "b" from the first string to get at the LCS of "a".

2. 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. Now consider the two strings `stable` and `marriage`. Describe the relationship of the length of their LCS with the length of the LCS of `stabl` and `marriag` (strings `A` and string `B` with their last letters removed).

   SOLUTION: The length of the LCS of `stable` and `marriage` is one longer than the length of the LCS of `stabl` and `marriag` because we "lose" the final matching letter `e`, which is part of the LCS of the original strings.

4. Given two strings `A` and `B` of length $n > 0$ and $m > 0$, break the problem of finding the length of the LCS `LLCS(A[1..n], B[1..m])` down into a recurrence over smaller problems. **USE** and generalize your work in the previous problems!

   SOLUTION: If the last letters match, then we are in a situation like with `stable` and `marriage`: 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:

```
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])
```

5. 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.

6. The previous two problems give a recurrence to solve `LLCS`. Does this recurrence repeatedly solve suproblems many times? (That is, might we want to use memoization or dynamic programming on it?) Sketch enough of the recursion tree to justify your answer.

   SOLUTION: Absolutely! If the last character matches, we're actually in good shape. (We have only one recursive call; so, that particular portion of the tree cannot be the one that introduces multiple paths to the same subproblem, although it can participate in a path created from a "split" higher up.) However, consider the second and third calls. Each can lead (via another recursive call in the "opposite direction") to the same subproblem where both strings are one letter shorter. We will solve certain subproblems exponentially often if we naively implement this algorithm.

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

   SOLUTION: We'll introduce a trampoline call that initializes a table and initiates the recursion:

```
LLCS(A, n, B, m):
  let Table[0..n][0..m] be a 2-dimensional array
  initialize all elements of Table to null
  Helper(Table, A, B, n, m)

Helper(T, A, B, n, m):
  // As always with memoization, we wrap the recurrence with a check
  // to see if the table already contains the answer. If not, compute
  // the answer via the recurrence and store it. If so, just return.
  if T[n][m] is null:
    if n = 0 or m = 0:
      T[n][m] = 0
    else:
      if A[n] = B[m]:
        Table[n][m] = Helper(T,A,B,n-1,m-1) + 1
      else:
        Table[n][m] = max(Helper(T,A,B,n-1,m),
                          Helper(T,A,B,n,m-1))
  reurn T[n][m]
```

8. 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 a blank are for the trivial cases!)

   SOLUTION: Inline below...

	__	c	o	u	n	t	r	y
__	0	0	0	0	0	0	0	0
t	0	0	0	0	0	1	1	1
y	*0*	0	0	0	0	1	1	2
c	0	*1*	1	1	1	1	1	2
o	0	*1*	2	2	2	2	2	2
o	0	1	*2*	*2*	2	2	2	2
n	0	1	2	2	*3*	*3*	*3*	*3*

9. 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 italicized 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.

   Our algorithm is inline below...

```
// Note: len(A) = n, len(B) = m, and Table is a filled-in
//       (n+1)x(m+1) LLCS memoization table for A and B
ExplainLCS(A, B, Table):
  if len(A) = 0 or len(B) = 0:  // recurrence's base case
    return ""                   // so the LCS is the empty string

  else:
    value = Table[n][m]         // the value computed by the recurrence

    if A[n] = B[m]:
      // The final letters matched. So, we actually need to add a letter to the LCS.
      return ExplainLCS(A[1..n-1], B[1..m-1], Table) + A[n]
    else:
      // which recursive call yielded the max?
      // although this doesn't look QUITE as similar to the memoized code as the rest
      // of this function, consider that all we're doing is writing out what the max
      // function actually does explicitly!
      if Table[n-1][m] >= Table[n][m-1]:
        // We decided not to use the last letter of A in the solution.
        return ExplainLCS(A[1..n-1], B, Table)
      else:
        // We decided not to use the last letter of B in the solution.
        return ExplainLCS(A, B[1..m-1], Table)
```

10. Give a dynamic programming 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.

```
LLCS(A, n, B, m):
  let Table[0..n][0..m] be a 2-dimensional array
  initialize all elements of Table to null

  // Fill in the base cases
  for i = 0 to n: Table[i][0] = 0
  for i = 0 to m: Table[0][i] = 0

  // Fill in the recursive cases column-by-column, top-to-bottom
  for i = 1 to n:
    for j = 1 to m:
      if A[i] = B[j]:
        Table[i][j] = Table[i-1][j-1] + 1
      else:
        Table[i][j] = max(Table[i-1][j],
                          Table[i][j-1])
  return Table[n][m]
```

11. Analyse the efficiency of your memoized, DP, and "explain" algorithms in terms of runtime and (additional, beyond the parameters) memory use. You may assume the strings are of length n and m, where $n \leq m$ (without loss of generality).

    SOLUTION: Both the memoized and DP 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" algorithm stops as soon as it reaches a base case, which takes at most $O(n+m)$ 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.)

12. 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.