Lecture Notes for CSCI 311: Algorithms

Set 10-Dynamic Programming Example: Longest Common Subsequence

Professor Talmage

February 25, 2026

---

1 Setup

For our final direct example of dynamic programming, we will look at the problem of finding the *Longest Common Subsequence* of two given strings. This is a fundamental problem in text and data processing, since it gives a similarity score for strings, which we can then use to find which of a dataset of strings is most similar to a given target string. This is useful for many things, such as spell checking, where we find a dictionary string most similar to a typed string and either replace or suggest replacing the typed string with the valid one, and inexact search, where we can find almost-matches, which can lead to successfully finding things using an imperfect search string. LCS also has many applications in computational biology, for example DNA comparison, which often needs to find similarity between DNA sequences which are incomplete or may be from different organisms but similarity can help us understand what a new sequence encodes.

We will state the problem formally, then work through our dynamic programming outline to efficiently find the length of the longest common subsequence of two strings.

Problem Statement

Input: Two strings X and Y.

Output: (Length of) the longest common subsequence of X and Y.

- A *subsequence* of a sequence (including a string) is any sequence of characters which appear in the input sequence in that order, not necessarily contiguously.
- Example: Start with "Bucknell". Some subsequence include "Buck", "ckn", "cell", "Bell", "Bull", "ucel", etc.
- A string of length n has 2^n subsequences, so a brute-force approach of trying all possible subsequences of X and Y would be too expensive.
- A *substring* would be similarly defined, but require contiguity. A string of length n has only $\Theta(n^2)$ substrings.

Exercise: Find the longest common subsequences of the following pairs of words:

- "springtime", "pioneer"
- "horseback", "snowflake"
- "bucknell", "bunkbed"
- "bucknell", "buncklel"

2 Dynamic Programming

1. **Characterize Optimal Solution Structure**

   Let $X = \langle x_1, \ldots, x_m \rangle, Y = \langle y_1, \ldots, y_n \rangle$ be two strings and $Z = \langle z_1, \ldots, z_k \rangle$ be an $LCS(X, Y)$. Then

   (a) $x_m = y_n$ implies that $z_k = x_m$ and $\langle z_1, \ldots, z_{k-1} \rangle$ is an LCS of $\langle x_1, \ldots, x_{m-1} \rangle$ and $\langle y_1, \ldots, y_{n-1} \rangle$.
   (b) If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an $LCS(X_{m-1}, Y)$.
   (c) If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an $LCS(X, Y_{n-1})$.

   Using the notation $X_{m-1} = \langle x_1, \ldots, x_{m-1} \rangle$, etc.

   The idea here is that if the last characters of X and Y are the same, then that character is definitely in some LCS and the rest of the LCS is a smaller LCS problem, or we could add it to get a longer common subsequence, and if the last characters are not the same then one must not be in the LCS, so we can discard it and look at a smaller LCS problem. Note that in the case of matching last character, there may be a longest common subsequence where that character is taken from the end of one string and matched against a different occurrence in the other string (e.g. "abcd" vs. "edited", two different matches for "d"), but that leads to an equally optimal solution, and the problem does not care which of many equally optimal solutions we return. Note also that we could compare the first characters instead of the last characters and the logic is the same, though with different indexing. We leave that version of the solution as an exercise.

   We now know that the problem has optimal substructure, since an optimal solution can be broken down into optimal solutions of subproblems.

   If we consider these cases algorithmically, we can combine the last two:

   (a) $x_m = y_n$: one subproblem: $LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) + x_m$
   (b) $x_m \neq y_n$: two subproblems: $LCS(X, Y) = max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1}))$

   All subproblems are of the form $LCS(X_i, Y_j) = LCS(\langle x_1, \ldots, x_i \rangle, \langle y_1, \ldots, y_j \rangle)$. We can thus store subproblem solutions in a two-dimension table. Define $c[i, j] = |LCS(X_i, Y_j)|$, yielding an $m \times n$ table.

2. **Define Optimal Solution Recurrence**

   Exercise: Define the recurrence for the value of $c[i, j]$. What are the base cases? General cases?

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1, j-1] & x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & x_i \neq y_j \end{cases}$$

This recurrence, together with the $m \times n$ size of the subproblem space tells us that dynamic programming must be more efficient than brute force, which would need to explore at least either all 2^m subsequences of X or all 2^n subsequences of Y, if not both. Since there are not that many distinct subproblems, there must be overlapping subproblems in different branches of the recursion.

Exercise: Draw the subproblem table, indicating goal, base cases, invalid cells, and dependencies. What would be a valid bottom-up computation order?

$i \backslash j$	0	1	...	$n-1$	n
0	0	0	...	0	0
1	0				
$\vdots$	0		$\ddots$		
$m-1$	0				
m	0				(goal)

Goal is cell $[m, n]$, representing all of X and Y. Base cases are $i = 0$ or $j = 0$, with value 0, as we have expended one of the inputs, and the longest common subsequence of an empty sequence is empty. There are no invalid cells. Each cell depends on that immediately above, that immediately left, and that immediately diagonally up and left. Bottom-up order is rows top to bottom, filling each row left to right.

3. **Compute Optimal Value**

Exercise: Give bottom-up code to compute $LCS(X, Y)$.

```
1:  function LCS(X, Y)
2:      create c table, set base cases
3:      for i = 1 to m do
4:          for j = 1 to n do
5:              if x_i == y_j then
6:                  c[i, j] = c[i − 1, j − 1] + 1
7:                  b[i, j] =↖
8:              else if c[i − 1, j] ≥ c[i, j − 1] then
9:                  c[i, j] = c[i − 1, j]
10:                 b[i, j] =↑
11:             else
12:                 c[i, j] = c[i, j − 1]
13:                 b[i, j] =←
14:             end if
15:         end for
16:     end for
17: end function
```

4. **Reconstruct Optimal Solution** The second array $b[i, j]$ in the above pseudocode tracks the addition of characters to the LCS: each ↖ indicates that we added a new character. To reconstruct the LCS, follow the arrows back through the table from $b[m, n]$, adding character x_i to the beginning of your string each time you encounter a diagonal arrow in cell $b[i, j]$. When you reach a base case, you have an LCS of X and Y.

Example: Find $LCS(ABCBDAB, BDCABA)$

$i \backslash j$	0	1 (B)	2 (D)	3(C)	4 (A)	5 (B)	6(A)
0	0	0	0	0	0	0	0
1 (A)	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ←
2 (B)	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
3 (C)	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
4 (B)	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
5 (D)	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
6 (A)	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
7 (B)	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Following the arrows from the bottom-right corner, we find diagonal arrows for A, B, C, B, giving the LCS BCBA.

Notes:

- Swapping X and Y flips the table diagonally, but does not change $|LCS|$, just (potentially) which sequence you find.
- Similarly, prioritizing left over up in case of a tie would not change the value of the solution, just which solution you find.
- If we do not care to reconstruct the actual sequence, we only need two rows of the table at a time, since each is created from only the immediately previous, so could solve the problem in $2 * \min\{m, n\}$ space beyond the input.

Exercise: Build the LCS subproblem table to find $LCS(\text{“}book\text{”}, \text{“}block\text{”})$.

Exercise: Build the LCS subproblem table to find $LCS(\text{“}breakiron\text{”}, \text{“}rebroken\text{”})$.