CS161 Final Project, 06/04/14

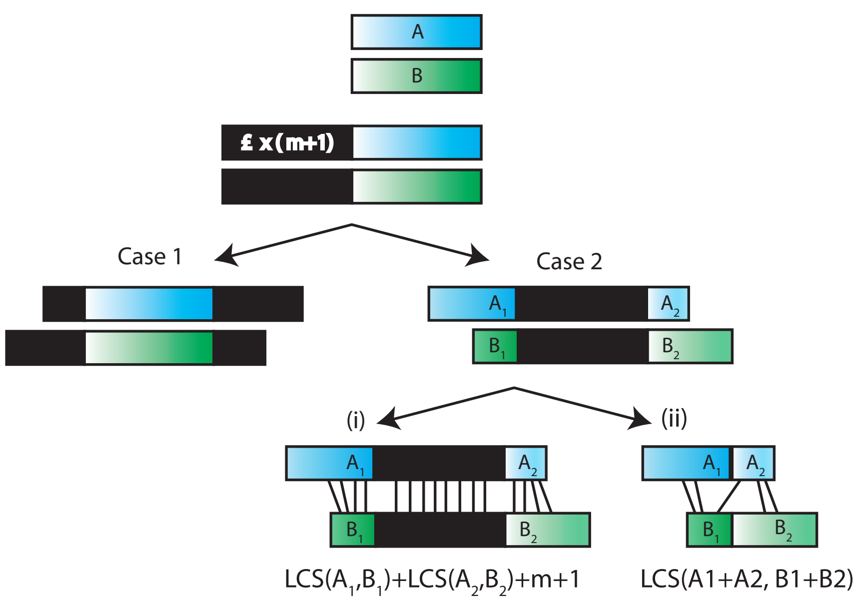
Aaditya Shidham, Clementine Jacoby, Thomas Stevens

(a)

Add $m+1$ foreign characters £ (e.g. any character not present in the alphabet of A and B) to the beginning of both A and B. Since m is the max length of LCS(A, B), when we treat CLCS as a black box, it will keep the positions of A and B fixed in order to pair the foreign character sequences with each other. Prepending the $m+1$ characters ensures that for any cut, matching the foreign characters will always lengthen the LCS more than any prepended strings' ends.

Prove this by considering an arbitrary cut i for the appended A and some cut at j for the appended B. In the first case, both cuts happen in the regions of A and B containing £. These cuts will produce the LCS trivially, since the ordering of the original characters of A and B was preserved after the cuts.

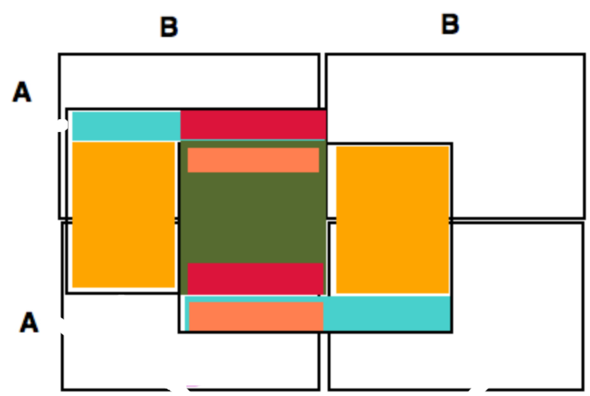
In the second case, either or both cuts happen in the non-£ region of the sequence. Then, there is some sequence of characters, A_1 that precedes the £ sequence and some sequence of characters that comes after the foreign block, A_2. Then define B_1 and B_2 similarly. We know that the LCS of A and B cut in this manner is either, (i), includes the $m+1$ matches created by matching the sequence block of £ or, (ii), does not include them. In case (i), the LCS length is $m+1$+LCS(A_1, B_1)+LCS(A_2, B_2). In case (ii), the LCS length is at most m characters since m is the max length of LCS(cut(A, i),cut(B, j)) where the £ region is ignored. So we note that the £ matching wins out in the creation of the LCS for any arbitrary cut.



Then the length of the LCS produced by this arbitrary cut is upper bounded by LCS$(A, B)+m+1$ (where $i = j = 0$), since the relative ordering of the characters in A and B are preserved in the substrings of A and B, and they ignore the additional matchings between A_1 and B_2 that the $i = j = 0$ solution considers. So the uncut strings will return the largest value for the LCS, making them the $CLCS$.
The original problem inputs to CLCS are of length m and n. The transformed inputs to CLCS are of length $m+(m+1) = (2m+1)$ and $n+(m+1) = (m+n+1)$ respectively. The product of the input sizes $(2m + 1)(m + n + 1) = mn + 2m^2 + \ldots$. $m \leq n \implies m^2 \leq mn$, and the product is still $O(mn)$ as it was originally.
We can find the LCS(A, B) by subtracting $m + 1$ from the value returned by CLCS, to account for the prepended foreign characters £. This added operation removes at most $m + 1$ characters from the array will increase the running time by at most $O(m)$. This implies that if CLCS can be solved in $T(m, n) = o(mn)$ time, then LCS could also be solved in $O(T(m, n))$ time.

(b)

A graphical proof follows. The graph below illustrates that two overlapping regions representing LCS solutions for different cuts are composed of identical subregions. All rectangles of the same color are identical (per the correspondence shown in part (c), they contain the same nodes and edges, and thus the same shortest path solutions).



Consider the graph below this section:

- The sub-problem LCS(cut(A, i),cut(B, j)) is represented by the red rectangle, with the red line representing the shortest path through the region.
- We can transform this path into a path starting at $B = 0$ by cutting off the lower right region and pasting it into the upper left corner.
- This new path represents a common subsequence C =(cut(A, k),cut$(B, 0)$).

- Then C is the common subsequence constructed by cutting the blue rectangle and pasting it in the upper left corner.

Length of C:

- Note that the common subsequence is defined by the number of diagonals taken on the path.
- The number of diagonals is not changed by copying and pasting C, since the regions over which C is defined (in the lower right and upper left corners shown below) contain the same nodes and edges (and therefore the same diagonal arrows).
- Because the number of diagonals is maintained, $|C| = \text{LCS}(\text{cut}(A, i), \text{cut}(B, j))$.

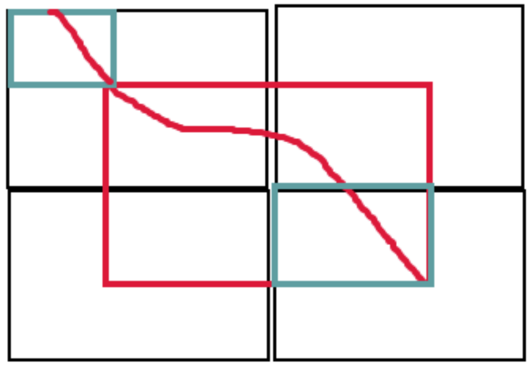
Relationship between LCS(cut(A, i),cut(B, j)) and LCS(cut(A, k),cut$(B, 0)$):

- We know that C is formed by cutting A at some k, and cutting B at 0. Therefore it is some subsequence of the form LCS(cut(A, k),cut$(B, 0)$).
- By definition, LCS(cut(A, k),cut$(B, 0)$) is greater than or equal to any common subsequence of the form LCS(cut(A, k),cut$(B, 0)$).
- $\therefore$ LCS(cut(A, k),cut$(B, 0)$) $>=$ $|C|$ =LCS(cut(A, i),cut(B, j)).

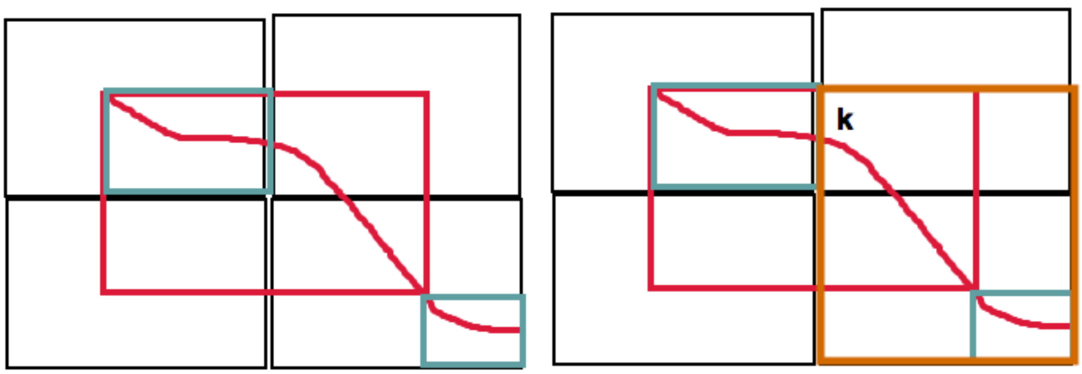
Correctness.

- Therefore, it suffices to find the longest LCS(cut(A, k),cut$(B, 0)$) over all possible choices of k, because for every fixed choice of i and j, $\exists k$: LCS(cut(A, k),cut$(B, 0)$)$>=$ $|C|$ =LCS(cut(A, i),cut(B, j)).
- Given cuts on two strings, we can always identify where to make the single cut in A that will give us the correct result. Using this method, every pair of cuts gives a solution that corresponds to a single cut on string A.
- We can do this in $O(mn)$ time because there are m possible choices of k and each LCS takes $O(mn)$ work to compute. Finding the longest LCS(cut(A, k),cut$(B, 0)$) is simply solving LCS m times, which gives the algorithm a running time of $O(mn)$.

The example above covers one set of cases, but there are other cases where this procedure won't produce a path starting at $B = 0$. Consider the graph below, where the procedure cuts B at some non-zero value:



We need to define our procedure so that it covers these cases where the shortest path crosses a vertical boundary (a boundary between B and B) before a horizontal one (a boundary between A and A). In this case, we "copy down" rather than copying up. That is, we copy the upper left region of the red rectangle into the lower region. We're left with a graph where $B = 0$.



This shows that for any fixed values (i, j), we can reconstruct an equal-length or shorter path starting at $(0, k)$.

(c)

We show the following 3 steps:

1. Given a common subsequence of the strings, show how to create a corresponding path in the graph.
   (a) The DP table forms a two-dimensional array, where each cell holds an entry. We can also look at this table as a directed acyclic graph (DAG), where each entry in the table corresponds to a vertex in the graph. A pair of adjacent entries in the DP table corresponds to an edge between the corresponding vertices in the DAG.
   (b) To see why the corresponding graph is a DAG, note that all edges in the table are directed (either down, to the right, or diagonally in the right-and-down direction).

(c) Since we only have these types of edges, the graph is necessarily acyclic.

(d) This means that, given a common subsequence of two strings described by a set of arrows used to traverse the table, we can create a corresponding path in the graph by including the edges that correspond to the arrows used to traverse the table.

This shows that all of the common subsequences are represented in the graph as a path problem.

2. Given a path from the top left to the bottom right, show how to recover a common subsequence.

   (a) Consider the path followed by the DP solution to the problem LCS(A,B), call it P.

   (b) There is a direct correspondence between the arrows forming P and the edges forming a shortest path tree of the graph G described above.

   (c) G is a DAG, so it has a root, namely (0,0).

   (d) This correspondence means that we can solve LCS by solving a shortest path problem on the DAG G_0.

   (e) This shows that you can get back to an answer to the original problem from an answer to the graph problem.

3. The SHORTEST paths in the graph correspond to the LONGEST common subsequences.

   (a) Note that every edge in the DAG has unit length, and that these edges correspond to adjacencies in the DP solution.

   (b) Having a shortest path in the graph means using the fewest number of edges to traverse it, since all edges have the same length.

   (c) Note that following a diagonal arrow in the table means taking a single path of length 1 rather than 2 (the horizontal and vertical arrows).

   (d) Therefore, taking the maximum number of diagonal arrows to traverse the table will use the fewest number of arrows, which will result in the shortest path through the DAG.

   (e) Since only diagonal moves accumulate $|LCS|$, solving the SHORTEST path problem from the top left to the bottom right produces a LONGEST common subsequence of the strings.

Lemma 1. The shortest path has the most diagonal edges.
Use Theorem 15.1, p392 of CLRS and the LCS-LENGTH implementation of this problem. The longest path from $(0, 0)$ to $(m, n) = m + n$, which follows the table perimeter and uses

0 diagonal edges.
In the case $m = n$, the shortest path is m, using m diagonal edges.
The edges "$\downarrow$" and "$\rightarrow$" in G correspond to the cases when the characters did not match for $[i, j]$, and the DP table is filled to reference $[i-1, j]$ or $[i, j-1]$ respectively.
Of the total $m + n$ characters, these steps only proceed past 1 character from either A or B in the table.
On the other hand, the "$\searrow$" edge corresponds to a match at $[i, j]$, and the DP table is filled diagonally from $[i-1, j-1] + 1$. This step proceeds past 1 character in A and 1 in B.
A path P will examine all $m + n$ characters in A and B, though an "$\searrow$" edge of length 1 examines 2 characters at a time.
The |arrow| notation below denotes the number of edges in the graph G of that type.

$$
\begin{aligned}
m + n &= |\rightarrow| + |\downarrow| + 2|\searrow| \\
m + n &= (|\rightarrow| + |\downarrow| + |\searrow|) + |\searrow| \\
m + n &= |P| + |\searrow| \\
|P| &= m + n - |\searrow|
\end{aligned}
$$

Thus $\forall m, n,\ \min |P| \implies \max|\searrow|$, and the shortest path has the most diagonal edges.

By Lemma 1, the shortest path will return the longest string through the procedure PRINT-LCS below, and therefore will produce the LCS.
Given the shortest path in G, traverse the path starting from (m, n). Each time there is a diagonal entry from some (i, j) to $(i-1, j-1)$ in the path, append $A[i]$ to the beginning of the LCS. Do this until the path hits the origin. This is the procedure PRINT-LCS from CLRS p395.

(d)

Given $x \in G_{U_i}$, p_j must at some point meet p_i, the lower bound of G_{U_i}, before reaching x. Suppose that when p_j reaches a vertex in p_i, it continues along $p_j = p_i$ until reaching column n: Since p_i is the lower bound of G_{U_i}, where x resides, and p_i is a shortest path from the first point of intersection with p_j, this path will always be shorter than a path through x. Stated equivalently, because a path through x must be longer than the path described, a path through x must be longer than the shortest path.

By contradiction, claim there exists a subpath p' through x rather than along the subpath constructed previously, p_j. Then this p' makes p_j shorter–that is, p' is a more optimal subpart than the subpart of p_i, called p'_i, used in the original p_j. But this means that by using p' instead of the subpart of p'_i, we can form a path shorter than p_i using p' instead of p'_i. $|p'| < |p'_i| \implies p_i$ is NOT a shortest path, which is a contradiction.

(e)

We aim to bound the running time of FindShortestPaths by showing (1) the work done in the subproblems at each level, and (2) the total recursion depth. The call FindShortest-Paths$(A, B, p, 0, m)$ will recursively generate shortest paths on the graph G (equivalently the DP table: $\{0, \ldots, 2m\} \times \{0, \ldots, n\}$). Let p_i denote the shortest path computed from starting point $(i, 0), 0 \leq i \leq m$. Prior to the call, compute p_0, the shortest path from $(0, 0) \to (m, n)$. p_m is defined by shifting p_0 down m indices, such that it runs from $(m, 0) \to (2m, n)$.

1. Work at each level of subproblems

We argue geometrically that the first subproblem, namely the table interior bounded above by p_0 and below by p_m, has $O(mn)$ entries. Let:

- $G_{U_0} = \{\{\text{table above } p_0\} \cup p_0\}$
- $G_{L_0} = \{\text{table below } p_0 \text{ until row } m \text{ inclusive}\}$.
- $G_{U_m} = \{\{\text{table above } p_m \text{ until row } m \text{ inclusive}\} \cup p_m\}$
- $G_{L_m} = \{\text{table below } p_m\}$.

The interior region to compute is $G_{L_0} \cup G_{U_m}$.
By the inclusion-exclusion principle:

$$\begin{aligned} |G_{L_0} \cup G_{U_m}| &= |G_{L_0}| + |G_{U_m}| - |G_{L_0} \cap G_{U_m}| \\ &= |G_{L_0}| + |G_{U_m}| - |\text{row } m| && \text{both graphs include row m.} \\ &= |G_{L_0}| + |G_{U_m}| - (n+1) \end{aligned}$$

left endpoint of $p_m = (m, 0)$ and right endpoint of $p_0 = (m, n)$.

$$= |G_{L_0}| + |G_{U_0}| - (n+1)$$

the two U regions are the same size since p_m is a translation of p_0.

$$\begin{aligned} &= |G_{L_0} \cup G_{U_0}| - (n+1) && \text{regions are mutually exclusive} \\ &= (m+1)(n+1) - (n+1) && \text{together these compose the full upper table} \\ &= mn + n \\ &= O(mn) && \forall m > 1, n < mn \end{aligned}$$

All paths generated by lower recursive calls must fall in the inclusive region bounded by the initial p_0 and p_m.

2. Recursion depth

Each subproblem defines a new starting midpoint $(mid, 0)$ for the current shortest-path problem. The path p_{mid} divides each "parent" region into two "child" regions. Thus, the recursion structure has the form of a binary tree, with a pair of child subproblems generated by dividing the parent with a path starting at the parent's midpoint.

Given the initialization step with $i = \{0, m\}$, the set of remaining midpoints $= \{0, \ldots, m\}/\{0, m\} = \{1, \ldots, m-1\}$. Since m is assumed to be a power of 2, $|\{1, \ldots, m-1\}| = m - 1 = 2^{\lg m} - 1$, and the subproblems correspond to a binary tree of height $h = \lg m$.

Consider the first pair of child calls from the initial bounding region:
After generating p_{mid} with $O(mn)$ work as described in (1), the regions is split into 2 sub-regions, both including the overlapping path p_{mid}. The maximum path length of p_0 at initialization can at most traverse the perimeter of the table, yielding $(m+1)+(n+1)$. In the child subproblem, the vertical length is restricted by p_{mid} and becomes $(\frac{m}{2}+1)+(n+1)$. Thus, by the inclusion-exclusion argument used in (1), the sizes of the child subproblems sum to the parent subproblem plus an extra count of the shared path p_{mid}. For a given recursion tree level, h, the total work done by subproblems at that level is

$$O(mn) + \frac{m}{2^h} + n + 2 = cmn + \frac{m}{2^h} + n + 2 \quad \text{for} c > 0$$

Each level of the recursion tree has 2^h nodes, $0 \le h \le \lg m - 1$. The overlapping subproblem occurs for each pair of children and thus once for each parent. Given tree level h, there will be 2^{h-1} overcounting events, since there are 2^{h-1} parents at that level.
The algorithm does O(1) work when the upper and lower bounding path start points are next to one another. This occurs at the lowest level of recursion depth ($h = \lg m - 1$), since the parent starting indices must have had difference $(u - \ell) = 2$. The work at the terminating level is $2^{\lg m - 1} * O(1) = m - 1 = O(m)$.
Adding this to the sum of work over the remaining nodes:

$$\begin{aligned}
T(m,n) &= O(m) + \sum_{h=0}^{\lg m - 2} \left[cmn + 2^{h-1}\left(\frac{m}{2^h} + n + 2\right)\right] \\
&= O(m) + \sum_{h=0}^{\lg m - 2} \left[cmn + \frac{m}{2} + 2^{h-1}(n+2)\right] \\
&= O(m) + (\lg m - 1) * (O(mn) + O(m)) + \sum_{h=0}^{\lg m - 2} 2^{h-1}(n+2) \\
&= O(m) + (\lg m - 1) * (O(mn) + O(m)) + (n+2)\frac{2^{\lg m - 1} - 1}{2 - 1} \\
&= O(m) + (\lg m - 1) * (O(mn) + O(m)) + (n+2)(m-1) \\
&= O(m) + O(mn \lg m) - O(mn) + O(m \lg m) - O(m) + O(mn) + O(m) + O(n) + O(1) \\
&= \boxed{O(mn \lg m)}
\end{aligned}$$

Since $\forall n > 1, mn \lg m > m \lg m$ and $\forall m > 2, mn \lg m > mn$. The linear and constant terms are trivially less than the product terms with these same inequalities.