20191017

Longest Common Subsequence

In class I discussed the "diff" utility that is available in UNIX and Linux. Its purpose is to compare two files on a line by line basis and identify the differences. It does this by finding the largest set of lines that occur in the same order in the two files. Lines that do not fit in this "best match" are reported as differences between the files.

This is particularly valuable when comparing two versions of the same file. The diff utility will show exactly which lines were added, deleted or modified between the two versions. This is perhaps one of the most useful version management tools available.

You can read more about diff here: https://www.computerhope.com/unix/udiff.htm ,

here https://www.geeksforgeeks.org/diff-command-linux-examples/ and here:

https://unix.stackexchange.com/questions/81998/understanding-of-diff-output

among many other places.

Now we will explore how to develop a dynamic programming algorithm to solve this problem.

Suppose S is a sequence of characters (from any alphabet). A string T is a subsequence of S if S can be constructed by inserting 0 or more characters into T. Equivalently, T is a subsequence of S if we can get T by deleting some characters of S.

For example, "horse" is a subsequence of "phosphorescent":

p**HO**spho**R**e**S**c**E**nt

or

p**H**osph**OR**e**S**c**E**nt

etc.

Given two strings $P = p_1 p_2 \ldots p_n$ and $Q = q_1 q_2 \ldots q_m$ with length n and m respectively, how can we find the **longest** common (shared) subsequence of P and Q?

As with any dynamic programming development, several questions all get addressed more or less simultaneously:

- What recurrence relation describes the relationship between large problems and their subproblems?
- How can we identify a subproblem by parameters?
- How can we efficiently store the solutions to subproblems?

As is often the case, thinking about how the complete problem reduces to some set of subproblems helps us discover the recurrence relation, which in turn helps us answer all the other questions.

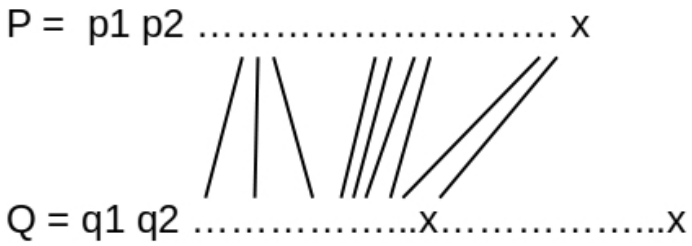
Let's consider the last two characters in the strings: p_n and q_m. Suppose these characters are identical (for example, suppose they are both "x")

$$P = p_1 p_2 \ldots p_{n-1} x$$

$$Q = q_1 q_2 \ldots q_{m-1} x$$

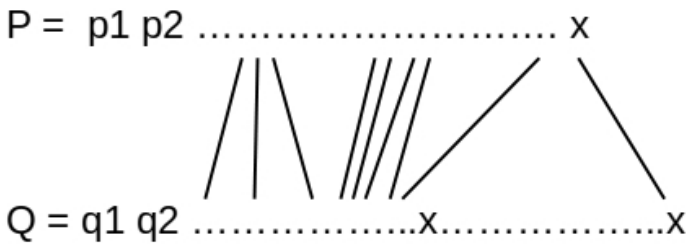
I claim that there is an optimal solution that matches these characters together.

Proof: Suppose there is an optimal solution that does not match either of these characters with anything. Then we could make the solution larger by matching these characters together – but that would be improving an optimal solution, which is a contradiction. Therefore every optimal solution must include at least one of these characters. Without loss of generality, assume O is an optimal solution that matches the x at the end of P with some x in Q. If the x at the end of P is matched in O with the x at the end of Q, then O satisfies the claim. If the x at the end of P is matched with some other x in Q, the situation looks like this:



in which the lines between P and Q indicate character matches. Note that the definition of a common subsequence tells us that none of these lines can cross each other because the matched characters must be in the same order in the two strings.

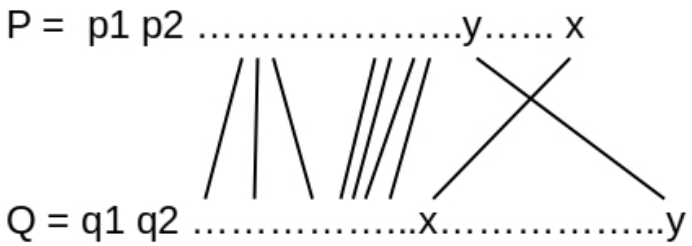
But if O looks like that, then we can clearly create a new matching O′ that looks like this:



O′ has the same number of matches as O, so O′ is optimal, and it satisfies the claim.

So if $p_n == q_n$, we can match the last characters together and reduce the problem to finding the LCS of the rest of the strings.. This looks like a good start on our recurrence relation!

But what if the two final characters don't match? Suppose P ends in x and Q ends in y. Now it is not possible for both of these characters to be in an optimal solution because that would look like this



which is not possible because the matched characters are not in the same order in both strings. So in an optimal solution, we either use one of these (and we don't know which one) or we use neither of them.

Case 1: if the optimal solution doesn't use the x (the last character of P) then we don't lose anything by throwing that character away. The LCS of P and Q is the same as the LCS of P[1..n-1] and Q

Case 2: if the optimal solution doesn't use the y (the last character of Q) then we don't lose anything by throwing that character away. The LCS of P and Q is the same as the LCS of P and Q[1..m-1]

Case 3: if the optimal solution doesn't use either the x or the y, we don't lose anything by throwing both characters away. The LCS of P and Q is the same as the LCS of p[1..n-1] and Q[1..m-1]

How do we know which case applies? We Don't!!! That is the whole idea of dynamic programming. We don't know which reduced problem is the right one so we work out all three of them and use whichever is best.

So to summarize:

I'm going to use LCSL to represent Longest Common Subsequence Length – that is what we will compute – we'll figure out how to construct the LCS later.

To compute LCSL(P,Q):

If $p_n == q_m$:

LCSL(P,Q) = 1 + LCSL(P[1..n-1], Q[1..m-1])

else:

LCSL(P,Q) = max(LCSL(P[1..n-1], Q) ,
LCSL(P, Q[1..m-1]) ,
LCSL(P[1..n-1],Q[1..m-1]))

This successfully expresses the final solution in terms of three subproblems. With a bit of thought we can see that exactly the same logic can be applied to reduce each of the three subproblems to even smaller subproblems, and that basically gives us the general case of our recurrence relation. We end up with this:

To compute LCSL(P[1..i],Q[1..j]):

If $p_i == q_j$:

LCSL(P[1..i],Q[1..j]) = 1 + LCSL(P[1..i-1], Q[1..j-1])

else:

LCSL(P[1..i],Q[1..j]) = max(LCSL(P[1..i-1], Q[1..j]) ,
LCSL(P[1..i], Q[1..j-1]) ,
LCSL(P[1..i-1],Q[1..j-1]))

This is good but it is notationally awkward. We can simplify it by noticing that the substrings we are working with ALWAYS start in position 1 in P and Q. Also, the first substring is always a substring of P and the second is always a substring of Q. So we can redefine LCSL like this:

Let LCSL(i,j) be the length of the longest common subsequence of P[1..i] and Q[1..j]

Now our recurrence looks like this:

To compute LCSL(i, j):

If $p_i == q_j$:

LCSL(i, j) = 1 + LCSL(i-1, j-1)

else:

LCSL(i, j) = max(LCSL(i-1, j) ,
LCSL(i, j-1) ,
LCSL(i-1, j-1))

And voila! We have parameterized our recurrence relation.

We still need to define the base cases. They can be defined in different ways but I will stick with the definition we used in class (corrected here):

LCSL(1,1) = 1	if $p_1 == q_1$	
= 0	otherwise	
LCSL(1,j) = 1	if LCSL(1,j-1) == 1 or $p_1 == q_j$	for all j > 1
= 0	otherwise	
LCSL(i,1) = 1	if LCSL(i-1,1) == 1 or $p_i == q_1$	for all i > 1
= 0	otherwise	

Now that we have the subproblems identified by two parameters (i and j) which run from 1 to n and 1 to m, it is easy to see that we can store the LCSL values in a 2-dimensional array with rows 1 to n and columns 1 to m (or 0 to n-1 and 0 to m-1 if you are under the evil spell of 0-based addressing).

Looking at the recurrence relation and considering the tabular arrangement of the subproblems, we can see that each subproblem's value is determined by the one beside it to the left, the one above it and to the left, and the one directly above it. This means that we can fill in the array row by row, working from left to right.

At long last, let's do an example. If P = "PALINDROME" and Q = "MAILROOM" we can set up the LCSL table with P down the left side and Q across the top:

	M	**A**	**I**	**L**	**R**	**O**	**O**	**M**
p	0	0	0	0	0	0	0	0
A	0							
L	0							
I	0							
N	0							
D	0							
R	0							
O	0							
M	1							
E	1							

At this point the base cases have been filled in.

	M	**A**	**I**	**L**	**R**	**O**	**O**	**M**
p	0	0	0	0	0	0	0	0
A	0	1						
L	0							
I	0							
N	0							
D	0							
R	0							
O	0							
M	1							
E	1							

The next cell gets filled in with 1 because the A's match, and the cell above and to the left contains 0. This 1 propagates along the row because each following cell looks at the one just before it (as well as at the ones above it).

	M	**A**	**I**	**L**	**R**	**O**	**O**	**M**
p	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
L	0							
I	0							
N	0							
D	0							
R	0							
O	0							
M	1							
E	1							

Filling in the next row. Make sure you understand why the 2's are there.

	M	**A**	**I**	**L**	**R**	**O**	**O**	**M**
p	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
L	0	1	1	2	2	2	2	2
I	0							
N	0							
D	0							
R	0							
O	0							
M	1							
E	1							

Completing the table

	M	**A**	**I**	**L**	**R**	**O**	**O**	**M**
p	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
L	0	1	1	2	2	2	2	2
I	0	1	2	2	2	2	2	2
N	0	1	2	2	2	2	2	2
D	0	1	2	2	2	2	2	2
R	0	1	2	2	3	3	3	3
O	0	1	2	2	3	4	4	4
M	1	1	2	2	3	4	4	5
E	1	1	2	2	3	4	4	5

The 5 at the bottom right tells us that the Longest Common Subsequence has length 5 ... but what is it?

If memory space is cheap, we can attach information to each cell of the array indicating the contents of the matching sequence up to that point. In this case, either the subsequence "AIROM" or "ALROM" would be stored in the bottom right corner of the array.

If we don't want to store extra information in the array, we can reconstruct the LCS by working backwards from the bottom right corner – this is the traditional Dynamic Programming approach:

- The "E" doesn't match the "M" so we know we didn't extend the LCS with a match here. We compare the 5 to the values to its left, above and to the left, and above – we see that this 5 is copied from the cell above so we move to that cell.
- The "M" matches the "M" so this character is in the LCS. We move to the cell above and to the left.
- Here the "O" matches the "O" so this character is in the LCS. We move to the cell above and to the left.
- Here the "R" does not match the "O". We determine that this cell's value was copied

from the cell to its left so we move to that cell.

- Here the "R" matches the "R" so this character is in the LCS. We move to the cell above and to the left.
- Here the "D" does not match the "L" ... but all three potential predecessors have a value of 2. In this situation we can choose any of them to step back to – we will end up at an optimal solution no matter which way we go.

I'll let you complete the process of working back to the point where all the characters in an LCS have been identified.

Complexity: Each element of the table is computed by comparing the characters, then looking at no more than three previous elements of the table – all of which have already been computed. Thus each element is computed in constant time. Since the table is n*m in size, this part of the algorithm runs in O(n*m) time where n and m are the lengths of the strings.

The trace-back step also does constant work for each iteration. Each iteration either moves up one row, leftward one column, or both. Thus the maximum number of iterations is O(n+m).

This gives an overall complexity of O(n*m)

In class someone suggested that we can improve the speed of the algorithm by deleting any characters that are not in both strings. This is certainly possible – finding and removing such characters can be done in O(n+m) time. The worst-case complexity of the algorithm would not be changed but the average-case behaviour might be significantly improved. It's a good idea.

Take-away from this study: this is a non-trivial problem that cannot be solved by a divide-and-conquer algorithm (there is no way to split the strings into left and right halves and be sure we haven't cut the LCS into non-matching parts), nor by a greedy algorithm (the first match we find may not be in the LCS). The BFI approach of comparing every subsequence of P to every subsequence of Q to see if they match is clearly infeasible. DP gives us a fast and easily implemented solution.