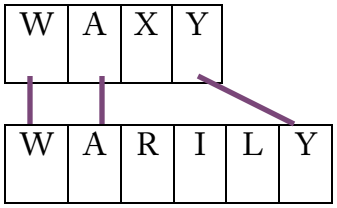
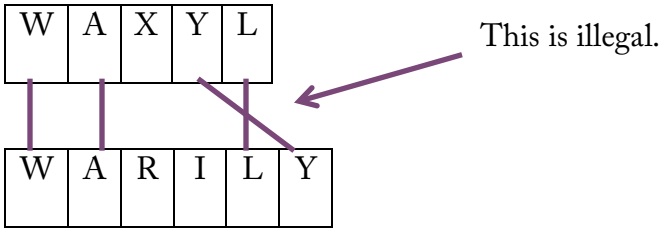
Kyle Wetherald
CS 1501
Dr. John Aronis
April 22, 2013

Longest Common Subsequence Problem

The classic computer science problem of the longest common subsequence (LCS) of two or more strings is useful for many applications including file comparisons and bioinformatics. A subsequence can be defined as a sequence that results from removing elements from another sequence without changing the order of the elements left over. For example, a subsequence of the string "AIDED" could be "ADD" but not "DEAD". A subsequence is said to be a common subsequence of two strings if it is a subsequence of both strings. For example, "WAY" is a common subsequence of the two strings "WAXY" and "WARILY". Visually, we can represent the common subsequence using lines to connect the matching characters:



These lines must never cross, as that would change the order of the characters for at least one of the strings, so it would no longer be a subsequence of one of the strings. For example, if an 'L' were added to the end of "WAXY", both 'L' and 'Y' could not both be a part of the subsequence even though they are common to both strings:



The LCS problem is to find one of the longest common subsequences of two (or more) strings.

One application of the LCS problem within the study of biological data (bioinformatics) is in comparing genes. There are four nucleotide bases that make up DNA: Guanine, Adenine, Thymine, and Cytosine arranged in different orders and complimenting each other. These nucleotides can be represented digitally with the characters G, A, T, and C respectively. According to the U.S. Department of Energy's Human Genome Project, the average gene has around 3000 bases.[1] For a longest common subsequence algorithm to be useful, it needs to be able to process thousands of characters in a reasonable amount of time. There are several possible ways to solve the LCS problem. In this essay I will show why dynamic programming is a preferable method of solving the LCS problem, making it useful for processing thousands of characters.

One possible way of solving the LCS problem is by using a brute-force algorithm. With the brute-force algorithm, to find a longest common subsequence of two strings, one would simply find every possible subsequence of the first string, x, and then check to see if it is a subsequence of the second string, y, then check to see if it is longer than any other substrings previously found. Suppose the length of x is m and the length of y is n. For each subsequence of x, each character x_1 through x_m is either in the subsequence or it is not. Therefore the number of subsequences of x is 2^m. Checking to see if the

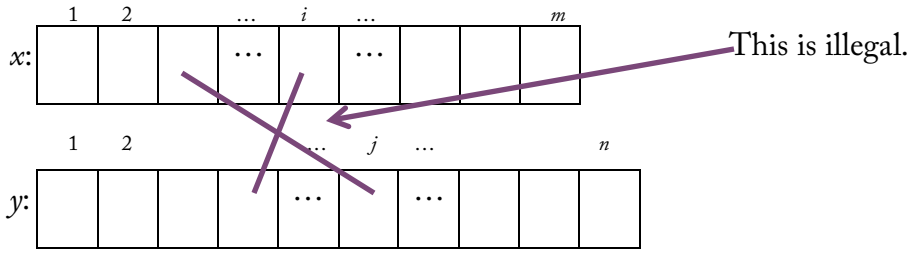
[1] From the Genome to the Proteome, *Human Genome Project Science,* [online] April 2013, http://www.ornl.gov/sci/techresources/Human_Genome/project/info.shtml

subsequence is in *y* would have a linear runtime of $O(n)$. Therefore, the runtime of the brute-force algorithm would be $O(n2^m)$. This is an exponential runtime, and would be unusable for thousands of characters.

A logical next step might be to find the solution recursively. Recursive solutions involve breaking the problem down into smaller problems until a base case is reached. In this case, the recursive function would break the problem down by comparing the last character in each of the two strings and recursing with strings that are smaller by one character until an empty string is reached. The function looks like this:

$$LCS(x_i, y_j) = \begin{cases} \varepsilon & \text{if } i = 0 \text{ or } j = 0 \\ LCS(x_{i-1}, y_{j-1}) \,||\, x_i & \text{if } x_i = y_j \\ \text{longest}\{LCS(x_i, y_{j-1}), LCS(x_{i-1}, y_j)\} & \text{if } x_i \neq y_j \end{cases}$$

This is based on the fact that if *x* and *y* end with the same character, then a longest common subsequence must contain it. If it did not contain it, we could get a longer common subsequence by adding it. If they do not end with the same character, then a longest common subsequence either does not end with x_i or it does not end with y_j. This becomes apparent when we once again visualize using lines connecting matching characters, as the lines would cross if both characters were included in the subsequence:



We must compute both scenarios and see which one results in a longer subsequence. This algorithm is fairly straightforward to implement:

```
LCS(x,y,i,j)
     IF i < 0 or j < 0
          RETURN empty string
     IF x[i]=y[j]
          RETURN LCS(x,y,i-1,j-1)+x[i]
     ELSE
          RETURN MAX{LCS(x,y,i,j-1),LCS(x,y,i-1,j)}
```

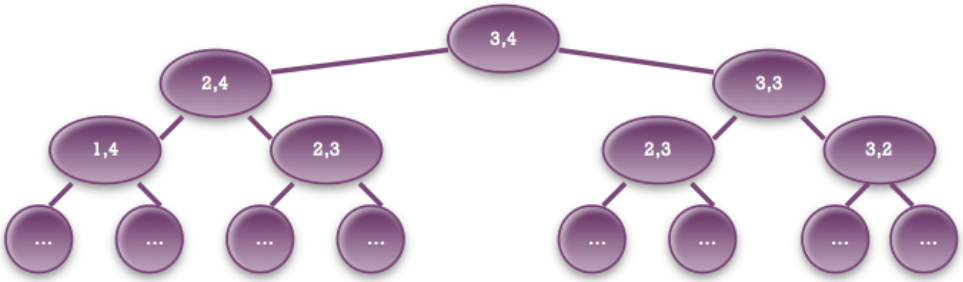
The function would initially be called with:

```
LCS(x,y,m-1,n-1)
```

To determine the runtime for this algorithm, we must consider the worst case scenario, when $x_i \neq y_j$. The recurrence relation would be:

```
T(i,j) = T(i-1,j)+T(i,j-1)+c
```

The runtime becomes apparent if we look at a tree of recursive calls. Take, for instance, a call where m=3 and n=4



It is clear that with every recursive function call in the worst case, two new function calls are made, and the number of function calls is nearly doubled with each level of the tree. The height of the tree would be $m+n$ because in the worst case, each function call decrements either n or m, but not both. $m+n$ function calls are needed to get both m and n to zero. Therefore, the runtime is $O(2^{m+n})$, which is still exponential, making the algorithm unusable for thousands of characters.

However, the tree diagram also shows that some of the work is being done redundantly. For instance, in the example where *m*=3 and *n*=4, we see that the call for *i*=2 and *j*=3 is done twice. This redundancy makes the LCS problem a great candidate for dynamic programming. By adding a memo, we can significantly reduce the runtime of the algorithm.

As we look toward a better algorithm, it is helpful to note that instead of strictly finding the LCS and returning it, a function can first simply find the *length* of the LCS:

$$LCSlen(x_i, y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCSlen(x_{i-1}, y_{j-1}) + 1 & \text{if } x_i = y_j \\ \max\{LCSlen(x_i, y_{j-1}), LCSlen(x_{i-1}, y_j)\} & \text{if } x_i \neq y_j \end{cases}$$

then reconstruct the LCS itself from the results. Also, we can store each of the computed lengths in order to avoid repeated computations. This is the basis of the dynamic programming solution to the longest common subsequence problem. To implement this algorithm, we first set up a matrix, *c,* of size *m*+1 by *n*+1. We know the size of any common subsequence of any string and an empty string is 0, so the 0th column and 0th row are both initialized to zero:

	0	1	2	…	*m*
0	0	0	0	…	0
1	0				
2	0				
3	0				
…	…				
n	0				

Next, we compare each of the x_i characters to each of the y_j characters. If $x_i = y_j$ then the length value at $c[i][j]$ is equal to $c[i\text{-}1][j\text{-}1]+1$. Otherwise, the length is equal to the greater of the values either above or to the left of it. If we start at *i*=0 and *j*=0, then the

previous computations will already have been recorded and will not be repeated. For instance, in the example from the beginning of "WAXYL" and "WARILY", the matrix would start out like this:

	0(ε)	1(W)	2(A)	3(X)	4(Y)	5(L)
0(ε)	0	0	0	0	0	0
1(W)	0					
2(A)	0					
3(R)	0					
4(I)	0					
5(L)	0					
6(Y)	0					

Then, since 'W' and 'W' are a match, the the length value at *c*[1][1] would be the value at *c*[0][0] (which is 0) +1:

	0(ε)	1(W)	2(A)	3(X)	4(Y)	5(L)
0(ε)	0	0	0	0	0	0
1(W)	0	1				
2(A)	0					
3(R)	0					
4(I)	0					
5(L)	0					
6(Y)	0					

'W' and 'A' are not a match, however, so the value of *c*[2][1] would be the greater of what is above or to the left of that value, at *c*[2][1] or *c*[2][1]:

	0(ε)	1(W)	2(A)	3(X)	4(Y)	5(L)
0(ε)	0	0	0	0	0	0
1(W)	0	1	1			
2(A)	0					
3(R)	0					
4(I)	0					
5(L)	0					
6(Y)	0					

The completed matrix looks like this after all of the comparisons have been made and the lengths recorded:

	0(ε)	1(W)	2(A)	3(X)	4(Y)	5(L)
0(ε)	0	0	0	0	0	0
1(W)	0	1	1	1	1	1
2(A)	0	1	2	2	2	2
3(R)	0	1	2	2	2	2
4(I)	0	1	2	2	2	2
5(L)	0	1	2	2	2	3
6(Y)	0	1	2	2	3	3

The implemented algorithm is as follows:

```
LCS(x,y)
   INIT c[0][0] through c[0][m] to 0
   INIT c[1][0] through c[n][0] to 0
   FOR i 0 through m
      FOR j 0 through n
         IF x[i]=y[j]
           SET C[i+1][j+1] equal to c[i][j]+1
         ELSE
          SET C[i+1][j+1] equal to MAX{C[i][j+1],
C[i+1][j]}
```

Next, to reconstruct the LCS, we start at the bottom right of the matrix, where $i=m+1$ and $j=n+1$. If the length changed from one character to the next, we know that there was a match. So, we check above and to the left of the value in the matrix at $c[i-1][j]$. If it is the same value, we decrement i but do not append any characters onto the LCS. Then we check above the value at $c[i][j-1]$. If it too contains the same value, we decrement j but do not append any characters to the LCS. If, however, either of the values to the left or above $c[i][j]$ are less than $c[i][j]$, then we append x_i onto the end of

the LCS and decrement both *i* and *j*. We repeat this process until both *i* and *j* have reached 0. The rest of the implemented algorithm is as follows:

```
SET i equal to length(x)
SET j equal to length(y)
INIT result to empty_string
WHILE i>0 and j>0 DO
   IF c[i][j]=c[i-1][j]
      SET i to i-1
   ELSE IF c[i][j]=c[i][j-1]
      SET j to j-1
   ELSE
      SET i to i-1
      SET j to j-1
      SET result to result+x[i]
ENDWHILE
RETURN result
```

The runtime of the dynamic programming algorithm is significantly better than the previous algorithms. Finding the length of the LCS requires a loop of *n* character comparisons within a loop through *m* characters. It is easy to see then, that the dynamic programming solution has a runtime of $O(m*n)$, which is quadratic. We have finally found a solution that will work for thousands of characters within a reasonable amount of time! Additionally, because we are not recursively calling new functions, the space needed for the algorithm is $O(m*n)$ as well.

Implementing the algorithm in Java was very straightforward and closely followed the pseudo code above. The main difference is that because strings are not immutable in Java, I could not use the `x[i]=y[j]` notations like I would with a character array. Instead I used the `x.charAt(i)` function of the String class.

The longest common subsequence problem is a great example of using dynamic programming. As the runtime analyses of a brute-force solution, a recursive solution, and a dynamic solution show, dynamic programming was the only method tested that was efficient enough with both time and memory to be useful with strings with thousands of characters. In situations like this, dynamic programming is a desirable method of problem solving.