Longest Common Subsequence Problem

The problem is to find the longest common subsequence in two given strings. A subsequence of a string is simply some subset of the letters in the whole string in the order they appear in the string. In order to denote a subsequence, you could simply denote each array index of the string you wanted to include in the subsequence. For example, given the string "GOODMORNING", the subsequence that corresponds to array indexes 1, 3, 5, and 6 is "ODOR."

Here is the basic idea behind solving the problem:

If the last characters of both strings s1 and s2 match, then the LCS will be one plus the LCS of both of the strings with their last characters removed.

If the last characters of both strings do NOT match, then the LCS will be one of two options:

1) The LCS of x and y without its last character.
2) The LCS of y and x without its last character.

Thus, in this case we will simply take the maximum of these two values. Also, we could just as easily have compared the *first* two characters of x and y and used a similar technique.

Let's examine the code for both the recursive solution to LCS and the dynamic programming solution:

```
// Arup Guha
// 3/2/05

// The method below solves the longest common subsequence
// problem recursively.
import java.io.*;

public class LCS {

  // Precondition: Both x and y are non-empty strings.
  //                        0 < len1 <= x.length() , 0 < len2 <= y.length
  public static int lcsrec(String x, String y) {

    // If one of the strings has one character, search for that
    // character in the other string and return the appropriate
    // answer.
    if (x.length() == 1)
        return find(x.charAt(0), y);
    if (y.length() == 1)
        return find(y.charAt(0), x);

    // Solve the problem recursively.

    // Corresponding last characters match.
    if (x.charAt(len1-1) == y.charAt(len2-1))
        return 1+lcsrec(x.substring(0, x.length()-1),
                         y.substring(0,y.length()-1));

    // Corresponding characters do not match.
    else
        return max(lcsrec(x, y.substring(0, y.length()-1)),
                    lcsrec(x.substring(0,x.length()-1), y));

  }
```

Now, our goal will be to take this recursive solution and build a dynamic programming solution. The key here is to notice that the heart of each recursive call is the pair of indexes, telling us which prefix string we are considering. In some sense, we can build the answer to "longer" LCS questions based on the answers to smaller LCS questions. This can be seen trace through the recursion at the very last few steps.

If we make the recursive call on the strings RACECAR and CREAM, once we have the answers to the recursive calls for inputs RACECAR and CREA and the inputs RACECA and CREAM, we can use those two answers and immediately take the maximum of the two to solve our problem!

Thus, think of *storing* the answers to these recursive calls in a table, such as this:

	R	**A**	**C**	**E**	**C**	**A**	**R**
C							
R							
E							
A			**XXX**				
M							

In this chart for example, the slot with the XXX will store an integer that represents the longest common subsequence of CREA and RAC. (In this case 2.)

Now, let's think about building this table. First we will initialize the first row and column:

	R	A	C	E	C	A	R
C	**0**	**0**	**1**	**1**	**1**	**1**	**1**
R	**1**						
E	**1**						
A	**1**						
M	**1**						

Basically, we search for the first letter in the other string, when we get there, we put a 1, and all other values subsequent to that on the row or column are also one. This corresponds to the base case in the recursive code.

Now, we simply fill out the chart according to the recursive rule:

1) Check to see if the "last" characters match. If so, delete this and take the LCS of what's left and add 1 to it.

2) If not, then we try to possibilities, and take the maximum of those two possibilities. (These possibilities are simply taking the LCS of the whole first word and the second work minus the last letter, and vice versa.)

Here is the chart:

	R	A	C	E	C	A	R
C	**0**	**0**	**1**	**1**	**1**	**1**	**1**
R	**1**	**1**	**1**	**1**	**1**	**1**	**2**
E	**1**	**1**	**1**	**2**	**2**	**2**	**2**
A	**1**	**2**	**2**	**2**	**2**	**3**	**3**
M	**1**	**2**	**2**	**2**	**2**	**3**	**3**

Now, let's use this to develop the dynamic programming code.

A little trick when coding is to add an extra row and column to the beginning to indicate an "empty character" so that you don't have to initialize the first row and column of the DP table:

		R	**A**	**C**	**E**	**C**	**A**	**R**
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
C	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**
R	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**2**
E	**0**	**1**	**1**	**1**	**2**	**2**	**2**	**2**
A	**0**	**1**	**2**	**2**	**2**	**2**	**3**	**3**
M	**0**	**1**	**2**	**2**	**2**	**2**	**3**	**3**

The 0s are naturally filled in (in Java) when creating your array, so you can start your loops for index 1 into your DP array instead of index 0. This allows you to safely index into i-1 and j-1 without worrying about going out of bounds. One caveat to using this method is that now, the letter stored in index 0 of each string really corresponds to index 1 in the DP array.