Read: This algorithm is not covered in Kleinberg and Tardos, but it is a simplified version of the Sequence Alignment problem of Section 6.6. This algorithm is presented in Section 15.4 of the Algorithms book by Cormen, Leiserson, Rivest, and Stein.

Dynamic Programming: We begin discussion of an important algorithm design technique, called dynamic programming (or DP for short). The technique is among the most powerful for designing algorithms for optimization problems. (This is true for two reasons. Dynamic programming solutions are based on a few common elements. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

Bottom-Up Construction: Each subproblem has a natural notion of size. Larger subproblems are solved by combining solutions to smaller subproblems.

The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the formulation of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

Optimal substructure: (Sometimes called the principle of optimality.) It states that for the global problem to be solved optimally, each subproblem should be solved optimally. (Not all optimization problems satisfy this. For example, sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

Polynomially many subproblems: An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

Strings: One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and programs like "grep" do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is similar. This arises for example in genetics research and in document retrieval on the web. One common method of measuring the degree of similarity between two strings is to compute their longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = (x_1, x_2, \ldots, x_m)$ and $Z = (z_1, z_2, \ldots, z_k)$, we say that $Z$ is a subsequence of $X$ if
there is a strictly increasing sequence of \( k \) indices \( \langle i_1, i_2, \ldots, i_k \rangle \) (\( 1 \leq i_1 < i_2 < \ldots < i_k \leq n \)) such that \( Z = \langle X_{i_1}, X_{i_2}, \ldots, X_{i_k} \rangle \). For example, let \( X = \langle ABRACADABRA \rangle \) and let \( Z = \langle AADAA \rangle \), then \( Z \) is a subsequence of \( X \).

Given two strings \( X \) and \( Y \), the longest common subsequence of \( X \) and \( Y \) is a longest sequence \( Z \) that is a subsequence of both \( X \) and \( Y \). For example, let \( X = \langle ABRACADABRA \rangle \) and let \( Y = \langle YABBADABBADOO \rangle \). Then the longest common subsequence is \( Z = \langle ABADABA \rangle \). See Fig. 1

\[
\begin{align*}
X & = A B R A C A D A B R A \\
Y & = Y A B B A D A B B A D O O
\end{align*}
\]

\[
\text{LCS} = A B A D A B A
\]

Figure 1: An example of the LCS of two strings \( X \) and \( Y \).

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences \( X = \langle x_1, \ldots, x_m \rangle \) and \( Y = \langle y_1, \ldots, y_n \rangle \) determine a longest common subsequence. Note that it is not always unique. For example the LCS of \( \langle ABC \rangle \) and \( \langle BAC \rangle \) is either \( \langle AC \rangle \) or \( \langle BC \rangle \).

**DP Formulation for LCS:** The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of prefixes will suffice for us. A prefix of a sequence is just an initial string of values, \( X_i = \langle x_1, x_2, \ldots, x_i \rangle \). \( X_0 \) is the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. Let \( c[i, j] \) denote the length of the longest common subsequence of \( X_i \) and \( Y_j \). For example, in the above case we have \( X_5 = \langle ABRAC \rangle \) and \( Y_6 = \langle YABBAD \rangle \). Their longest common subsequence is \( \langle ABA \rangle \). Thus, \( c[5, 6] = 3 \).

Which of the \( c[i, j] \) values do we compute? Since we don’t know which will lead to the final optimum, we compute all of them. Eventually we are interested in \( c[m, n] \) since this will be the LCS of the two entire strings. The idea is to compute \( c[i, j] \) assuming that we already know the values of \( c[i', j'] \), for \( i' \leq i \) and \( j' \leq j \) (but not both equal). Here are the possible cases.

**Basis:** \( c[i, 0] = c[j, 0] = 0 \). If either sequence is empty, then the longest common subsequence is empty.

**Last characters match:** Suppose \( x_i = y_j \). For example: Let \( X_i = \langle ABCA \rangle \) and let \( Y_j = \langle DACA \rangle \). Since both end in \( A \), we claim that the LCS must also end in \( A \). (We will leave the proof as an exercise.) Since the \( A \) is part of the LCS we may find the overall LCS by removing \( A \) from both sequences and taking the LCS of \( X_{i-1} = \langle ABC \rangle \) and \( Y_{j-1} = \langle DAC \rangle \) which is \( \langle AC \rangle \) and then adding \( A \) to the end, giving \( \langle ACA \rangle \) as the answer. (At first you might object: But how did you know that these two \( A \)’s matched with each other. The answer is that we don’t, but it will not make the LCS any smaller if we do.) This is illustrated at the top of Fig. 2.

\[
\text{if } x_i = y_j \text{ then } c[i, j] = c[i - 1, j - 1] + 1
\]
Last characters do not match: Suppose that $x_i \neq y_j$. In this case $x_i$ and $y_j$ cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either $x_i$ is not part of the LCS, or $y_j$ is not part of the LCS (and possibly both are not part of the LCS).

At this point it may be tempting to try to make a “smart” choice. By analyzing the last few characters of $X_i$ and $Y_j$, perhaps we can figure out which character is best to discard. However, this approach is doomed to failure (and you are strongly encouraged to think about this, since it is a common point of confusion.) Instead, our approach is to take advantage of the fact that we have already precomputed smaller subproblems, and use these results to guide us.

In the first case ($x_i$ is not in the LCS) the LCS of $X_i$ and $Y_j$ is the LCS of $X_{i-1}$ and $Y_j$, which is $c[i-1, j]$. In the second case ($y_j$ is not in the LCS) the LCS is the LCS of $X_i$ and $Y_{j-1}$ which is $c[i, j-1]$. We do not know which is the case, so we try both and take the one that gives us the longer LCS. This is illustrated at the bottom half of Fig. 2.

$$\text{if } x_i \neq y_j \text{ then } c[i, j] = \max(c[i-1, j], c[i, j-1])$$

Figure 2: The possible cases in the DP formulation of LCS.

Supplemental 3 Fall 2010
We left undone the business of showing that if both strings end in the same character, then the LCS must also end in this same character. To see this, suppose by contradiction that both characters end in A, and further suppose that the LCS ended in a different character \( B \). Because \( A \) is the last character of both strings, it follows that this particular instance of the character \( A \) cannot be used anywhere else in the LCS. Thus, we can add it to the end of the LCS, creating a longer common subsequence than the LCS, a contradiction.

Combining these observations we have the following formulation:

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

**Implementing the Formulation:** The task now is to simply implement this formulation. We concentrate only on computing the maximum length of the LCS. Later we will see how to extract the actual sequence. We will store some helpful pointers in a parallel array, \( b[0..m, 0..n] \). The code is shown below, and an example is illustrated in Fig. 3

```c
int BuildLCSTable(char x[1..m], char y[1..n]) { // compute LCS table
    int c[0..m, 0..n] // init column 0
    for i = 0 to m
        c[i,0] = 0; b[i,0] = skipX
    for j = 0 to n // init row 0
        c[0,j] = 0; b[0,j] = skipY
    for i = 1 to m // fill rest of table
        for j = 1 to n
            if (x[i] == y[j]) // take X[i] (Y[j]) for LCS
                c[i,j] = c[i-1,j-1]+1; b[i,j] = addXY
            else if (c[i-1,j] >= c[i,j-1]) // X[i] not in LCS
                c[i,j] = c[i-1,j]; b[i,j] = skipX
            else // Y[j] not in LCS
                c[i,j] = c[i,j-1]; b[i,j] = skipY
    return c[m,n] // return length of LCS
}
```

The running time of the algorithm is clearly \( O(mn) \) since there are two nested loops with \( m \) and \( n \) iterations, respectively. The algorithm also uses \( O(mn) \) space.

**Extracting the Actual Sequence:** Extracting the final LCS is done by using the back pointers stored in \( b[0..m, 0..n] \). Intuitively \( b[i,j] = addXY \) means that \( X[i] \) and \( Y[j] \) together form the last character of the LCS. So we take this common character, and continue with entry \( b[i - 1, j - 1] \) to the northwest (\( \searrow \)). If \( b[i,j] = skipX \), then we know that \( X[i] \) is not in the LCS, and so we skip it and go to \( b[i-1,j] \) above us (\( \uparrow \)). Similarly, if \( b[i,j] = skipY \), then we know that \( Y[j] \) is not in the LCS, and so we skip it and go to \( b[i,j-1] \) to the left (\( \leftarrow \)). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

**Memoization:** The algorithm’s iterative structure is different from our DP formulation, which was essentially recursive. You might wonder, “is there a recursive implementation of the algorithm?”
extracting the lcs

```c
getLCS(x[1..m], y[1..n], b[0..m,0..n]) {
    LCSString = empty string
    i = m; j = n // start at lower right
    while(i != 0 && j != 0) // go until upper left
        switch b[i,j]
            case addXY: // add X[i] (=Y[j])
                add x[i] (or equivalently y[j]) to front of LCSstring
                i--; j--; break
            case skipX: i--; break // skip X[i]
            case skipY: j--; break // skip Y[j]
    return LCSstring
}
```

The answer is yes, but it is important for the sake of efficiency that, once an entry of the matrix has been computed by the recursive rule, it must not be computed again. Instead, it’s value is simply looked up and returned.

This process is called memoization. If a matrix entry is not yet computed, we compute it using the recursive approach, and save its value in the matrix. If a matrix entry has been computed, we simply return the value stored in the table. Here is a “memoized” version of the LCS algorithm. It computes just the $c[i,j]$ part, but adding the $b[i,j]$ part is an easy extension.

```c
Build LCS Table

memoizedLCS(x[1..m], y[1..n]) { // compute LCS table
    int c[0..m, 0..n]
    initialize all entries of c to "uncomputed"
    memLCS(m, n)
}

memLCS(i, j) {
    if (c[i,j] != "uncomputed") return c[i,j] // already computed?
    else {
        if (i == 0 || j == 0) c[i,j] = 0
        if (x[i] == y[j])
            c[i, j] = memLCS(i-1, j-1) + 1
        else if (memLCS(i-1, j) >= memLCS(i, j-1))
            c[i, j] = memLCS(i-1, j)
        else
            c[i, j] = memLCS(i, j-1)
        return c[i,j]
    }
}
```

The memoized version has exactly the same asymptotic running time as the original. The reason is that each entry that is computed, is computed exactly once by the recursive rule, which is exactly what the previous algorithm would have done. After that, each entry takes only $O(1)$ additional time to access. has the same $O(mn)$ running time as the original algorithm.