Lecture 25: Longest Common Subsequence

(April 28, 1998)

Read: Section 16.3 in CLR.

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 functions 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. Genetic codes are stored as long DNA molecules. The DNA strands can be broken down into a long sequences each of which is one of four basic types: C, G, T, A.

But exact matches rarely occur in biology because of small changes in DNA replication. Exact substring search will only find exact matches. For this reason, it is of interest to compute similarities between strings that do not match exactly. The method of string similarities should be insensitive to random insertions and deletions of characters from some originating string. There are a number of measures of similarity in strings. The first is the *edit distance*, that is, the minimum number of single character insertions, deletions, or transpositions necessary to convert one string into another. The other, which we will study today, is that of determining the length of the longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$, 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 which is both a subsequence of X and Y.

For example, let X be as before and let $Y = \langle YABBADABBADOO \rangle$. Then the longest common subsequence is $Z = \langle ABADABA \rangle$.

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 $\langle BC \rangle$.

Dynamic Programming Solution: 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. 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). We begin with some observations.

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

Last characters match: Suppose $x_i = y_j$. 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 explain why later.) 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.)

Thus, if $x_i = y_j$ 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).

In the first case 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 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.

Thus, if $x_i \neq y_j$ then $c[i, j] = \max(c[i - 1, j], c[i, j - 1])$.

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. But this would contradict the definition of the LCS as being longest.

Combining these observations we have the following rule:

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

Implementing the Rule: The task now is to simply implement this rule. As with other DP solutions, we concentrate on computing the maximum length. We will store some helpful pointers in a parallel array, $b[0..m, 0..n]$.

Longest Common Subsequence

```
LCS(char x[1..m], char y[1..n]) {
    int c[0..m, 0..n]
    for i = 0 to m do {
        c[i,0] = 0    b[i,0] = SKIPX                // initialize column 0
    }
    for j = 0 to n do {
        c[0,j] = 0    b[0,j] = SKIPY                // initialize row 0
    }
    for i = 1 to m do {
        for j = 1 to n do {
            if (x[i] == y[j]) {
                c[i,j] = c[i-1,j-1]+1               // take X[i] and Y[j] for LCS
                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];
    }
```

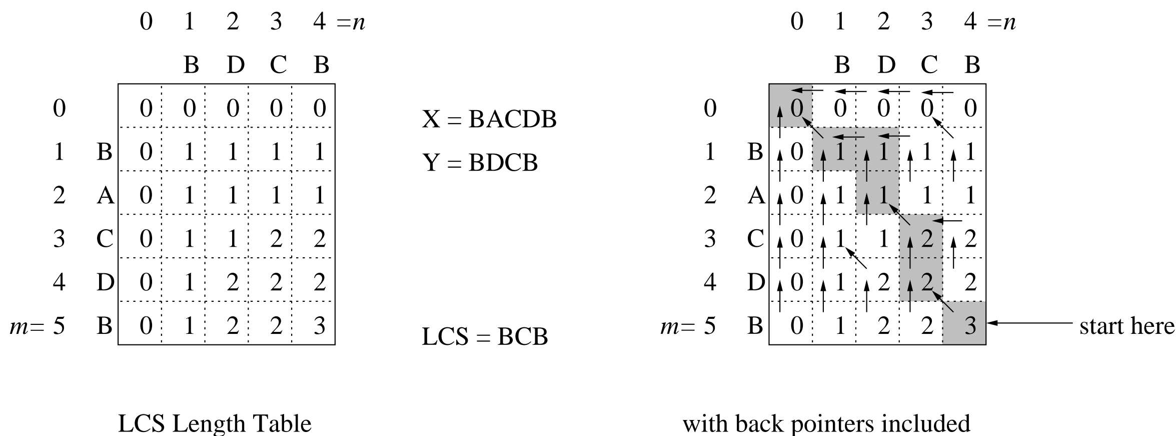

Figure 32: Longest common subsequence example.

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 ($\nwarrow$). 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.

Print Subsequence

```
getLCS(char x[1..m], char y[1..n], int b[0..m,0..n]) {
    LCS = empty string
    i = m
    j = n
    while(i != 0 && j != 0) {
        switch b[i,j] {
        case ADDXY:
            add x[i] (or equivalently y[j]) to front of LCS
            i--;   j--;   break
        case SKIPX:
            i--;  break
        case SKIPY:
            j--;  break
        }
    }
    return LCS
}
```