CMSC 451: Lecture 11
Dynamic Programming: Longest Common Subsequence

Thursday, Oct 5, 2017

Reading: This algorithm is not covered in KT or DPV. It is closely related to the Sequence Alignment problem of Section 6.6 of KT and the Edit Distance problem in Section 6.3 of DPV.

Strings: One important area of algorithm design is the study of algorithms for character strings. Finding patterns or similarities within strings is fundamental to various applications, ranging from document analysis to computational biology. One common measure of similarity between two strings is the lengths of their longest common subsequence. Today, we will consider an efficient solution to this problem based on dynamic programming.

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 $\langle i_1, i_2, \ldots, i_k\rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le n)$ such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k}\rangle$. For example, let $X = \langle\text{ABRACADABRA}\rangle$ and let $Z = \langle\text{AADAA}\rangle$, 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 $X = \langle\text{ABRACADABRA}\rangle$ and let $Y = \langle\text{YABBADABBADOO}\rangle$. Then the longest common subsequence is $Z = \langle\text{ABADABA}\rangle$ (see Fig. 1).

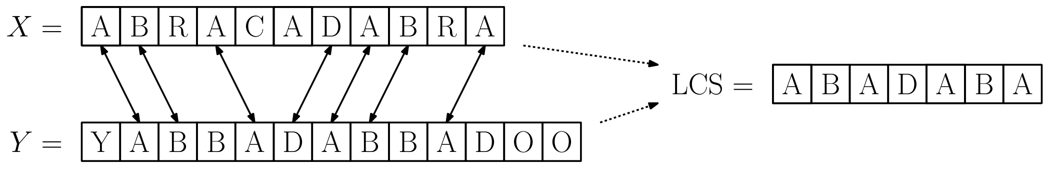

Fig. 1: An example of the LCS of two strings X and Y.

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 the length of their longest common subsequence, and more generally the sequence itself. Note that the subsequence is not necessarily unique. For example the LCS of $\langle\text{ABC}\rangle$ and $\langle\text{BAC}\rangle$ is either $\langle\text{AC}\rangle$ or $\langle\text{BC}\rangle$.

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, $X_i = \langle x_1, \ldots, x_i\rangle$. X_0 is the empty sequence.

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

Let us start by deriving a recursive formulation for computing $\text{lcs}(i, j)$. As we have seen with other DP problems, a naive implementation of this recursive rule will lead to a very inefficient algorithm. Rather than implementing it directly, we will use one of the other techniques (memoization or bottom-up computation) to produce a more efficient algorithm.

Basis: If either sequence is empty, then the longest common subsequence is empty. Therefore, $\text{lcs}(i, 0) = \text{lcs}(j, 0) = 0$.

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', it is easy to see that the LCS *must* also end in 'A'.[1] Also, there is no harm in assuming that the last two characters of both strings will be matched to each other, since matching the last 'A' of one string to an earlier instance of 'A' of the other can only limit our future options.

Since the 'A' is the last character of the LCS, we may find the overall LCS by (1) removing 'A' from both sequences, (2) taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$, and (3) adding 'A' to the end. This yields $\langle ACA \rangle$ as the LCS. Therefore, the length of the final LCS is the length of $\text{lcs}(X_{i-1}, Y_{j-1}) + 1$ (see Fig. 2), which provides us with the following rule:

$$\text{if } (x_i = y_j) \text{ then } \text{lcs}(i, j) = \text{lcs}(i-1, j-1) + 1$$

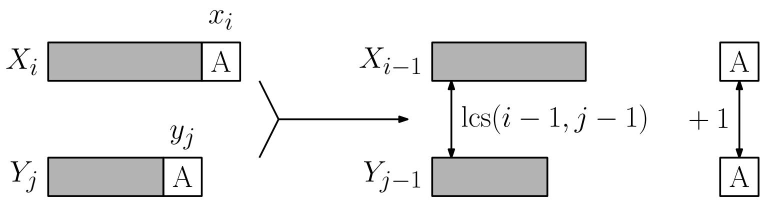

Fig. 2: LCS of two strings whose last characters are equal.

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). Remember the DP selection principle: *When given a set of feasible options to choose from, try them all and take the best.* Let's consider both options, and see which one provides the better result.

Option 1: (x_i is not in the LCS) Since we know that x_i is out, we can infer that the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is given by $\text{lcs}(i-1, j)$.

[1] We will leave the formal proof as an exercise, but intuitively this is proved by contradiction. If the LCS did not end in 'A', then we could make it longer by adding 'A' to its end.

Option 2: (y_j is not in the LCS) Since y_j is out, we can infer that the LCS of X_i and Y_j is the LCS of X_i and Y_{j-1}, which is given by $\text{lcs}(i, j-1)$.

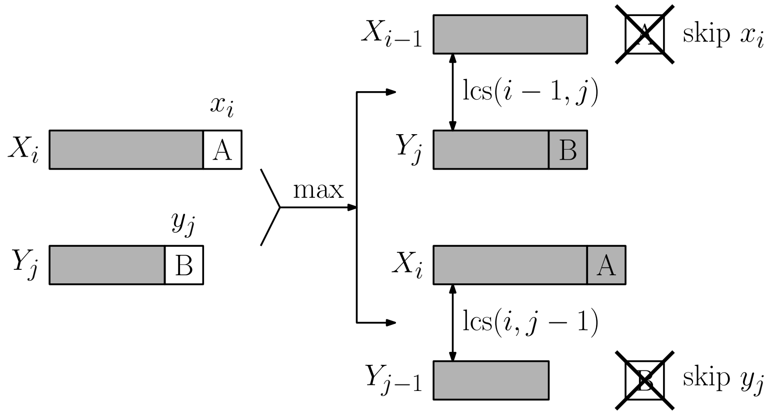

Fig. 3: The possibe cases in the DP formulation of LCS.

We compute both options and take the one that gives us the longer LCS (see Fig. 3).

$$\text{if } (x_i \neq y_j) \;\; \text{then} \;\; \text{lcs}(i,j) = \max(\text{lcs}(i-1,j), \text{lcs}(i,j-1))$$

Combining these observations we have the following recursive formulation:

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

As mentioned earlier, a direct recursive implementation of this rule will be very inefficient. Let's consider two alternative approaches to computing it.

Memoized implementation: The principal source of the inefficiency in a naive implementation of the recursive rule is that it makes repeated calls to $\text{lcs}(i,j)$ for the same values of i and j. To avoid this, it creates a 2-dimensional array $\text{lcs}[0..m, 0..n]$, where $m = |X|$ and $n = |Y|$. The memoized version first checks whether the requested value has already been computed, and if so, it just returns the stored value. Otherwise, it invokes the recursive rule to compute it. See the code block below. The initial call is memoized-lcs(m,n).

The running time of the memoized version is $O(mn)$. To see this, observe that there are $m+1$ possible values for i, and $n+1$ possible values for j. Each time we call memoized-lcs(i,j), if it has already been computed then it returns in $O(1)$ time. Each call to memoized-lcs(i,j) generates a constant number of additional calls. Therefore, the time needed to compute the initial value of any entry is $O(1)$, and all subsequent calls with the same arguments is $O(1)$. Thus, the total running time is equal to the number of entries computed, which is $O((m+1)(n+1)) = O(mn)$.

Bottom-up implementation: The alternative to memoization is to just create the lcs table in a bottom-up manner, working from smaller entries to larger entries. By the recursive rules, in order to compute $\text{lcs}[i,j]$, we need to have already computed $\text{lcs}[i-1,j-1]$, $\text{lcs}[i-1,j]$, and

Memoized Longest Common Subsequence

```
memoized-lcs(i,j) {
    if (lcs[i,j] has not yet been computed) {
        if (i == 0 || j == 0)                   // basis case
            lcs[i,j] = 0
        else if (x[i] == y[j])                  // last characters match
            lcs[i,j] = memoized-lcs(i-1, j-1) + 1
        else                                    // last chars don't match
            lcs[i,j] = max(memoized-lcs(i-1, j), memoized-lcs(i, j-1))
    }
    return lcs[i,j]                             // return stored value
}
```

lcs$[i, j-1]$. Thus, we can compute the entries row-by-row or column-by-column in increasing order. See the code block below and Fig. 4(a). The running time and space used by the algorithm are both clearly $O(mn)$.

Bottom-up Longest Common Subsequence

```
bottom-up-lcs() {
    lcs = new array [0..m, 0..n]
    for (i = 0 to m) lcs[i,0] = 0           // basis cases
    for (j = 0 to n) lcs[0,j] = 0
    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
                lcs[i,j] = lcs[i-1, j-1] + 1
            else
                lcs[i,j] = max(lcs[i-1, j], lcs[i, j-1])
        }
    }
    return lcs[m, n]                         // final lcs length
}
```

Extracting the LCS: The algorithms given so far compute only the length of the LCS, not the actual sequence. The remedy is common to many other DP algorithms. Whenever we make a decision, we save some information to help us recover the decisions that were made. We then work backwards, unraveling these decisions to determine all the decisions that led to the optimal solution. In particular, the algorithm performs three possible actions:

add$_{XY}$: Add $x_i(= y_j)$ to the LCS ('$\nwarrow$' in Fig. 4(b)) and continue with lcs$[i-1, j-1]$

skip$_X$: Do not include x_i to the LCS ('$\uparrow$' in Fig. 4(b)) and continue with lcs$[i-1, j]$

skip$_Y$: Do not include y_j to the LCS ('$\leftarrow$' in Fig. 4(b)) and continue with lcs$[i, j-1]$

An updated version of the bottom-up computation with these added hints is shown in the code block below and Fig. 4(b).

How do we use the hints to reconstruct the answer? We start at the the last entry of the table, which corresponds to lcs(m, n). In general, suppose that we are visiting the entry

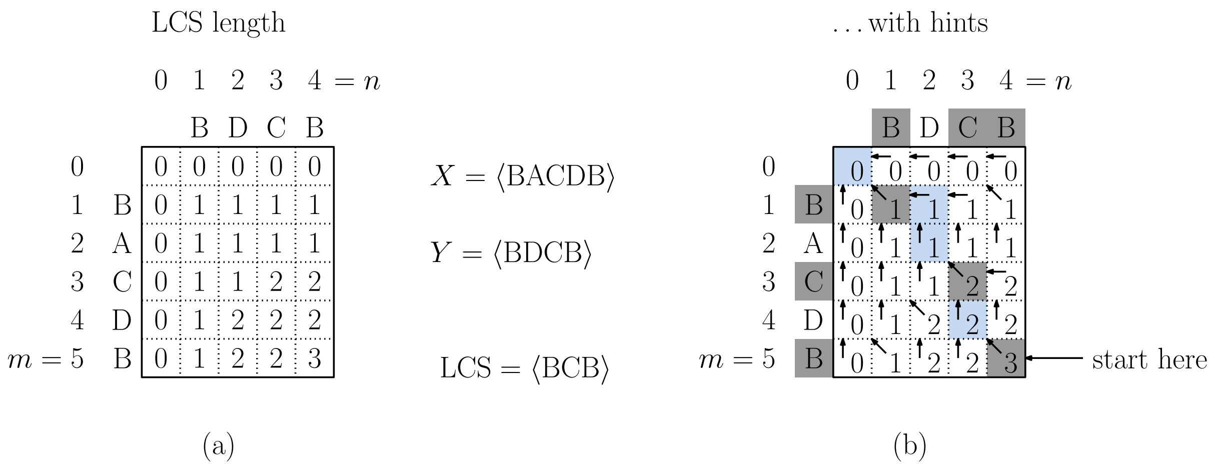

Fig. 4: Contents of the lcs array for the input sequences $X = \langle BACDB \rangle$ and $Y = \langle BCDB \rangle$. The numeric table entries are the values of lcs$[i, j]$ and the arrow entries are used in the extraction of the sequence.

Bottom-up Longest Common Subsequence with Hints

```
bottom-up-lcs-with-hints() {
    lcs = new array [0..m, 0..n]                 // stores lcs lengths
    h = new array [0..m, 0..n]                   // stores hints
    for (i = 0 to m) { lcs[i,0] = 0;  h[i,0] = skipX }
    for (j = 0 to n) { lcs[0,j] = 0;  h[0,j] = skipY }
    for (i = 1 to m) {
        for (j = 1 to n) {
            if (x[i] == y[j])
                { lcs[i,j] = lcs[i-1, j-1] + 1;  h[i,j] = addXY }
            else if (lcs[i-1, j] >= lcs[i, j-1])
                { lcs[i,j] = lcs[i-1, j];  h[i,j] = skipX }
            else
                { lcs[i,j] = lcs[i, j-1];  h[i,j] = skipY }
        }
    }
    return lcs[m, n]                             // final lcs length
}
```

corresponding to $\text{lcs}(m, n)$. If $h[i, j] = \text{add}_{XY}$, we know that $x_i (= y_j)$ is appended to the LCS sequence, and we continue with entry $[i - 1, j - 1]$. If $h[i, j] = \text{skip}_X$ we know that x_i is not in the LCS sequence, and we continue with entry $[i - 1, j]$. If $h[i, j] = \text{skip}_Y$ we know that y_j is not in the LCS sequence, and we continue with entry $[i, j - 1]$. Because the characters of the LCS are generated in reverse order, we *prepend* each one to a sequence, so that when we are done, the sequence is in proper order.

Extracting the LCS using the Hints

```
get-lcs-sequence() {
    LCS = new empty character sequence
    i = m; j = n                                // start at lower right
    while(i != 0 or j != 0)                     // loop until upper left
        switch h[i,j]
            case addXY:                         // add x[i] (= y[j])
                prepend x[i] (or equivalently y[j]) to front of LCS
                i--;  j--;    break
            case skipX: i--;  break             // skip x[i]
            case skipY: j--;  break             // skip y[j]
    return LCS
}
```