Problem Set 7: Dynamic Programming

Handed out Friday, October 11. Due at the start of class Monday, October 21.

Problem 1. (10 points) Show the results of the DP algorithm for computing the longest common subsequence (LCS) as presented in class on the following input:

$$X = \langle \text{ABCACB} \rangle \qquad Y = \langle \text{BACAB} \rangle$$

(a) Show the contents of the lcs table (see Lecture 17). Also, show the matrix of helper values to obtain the solution. I would suggest representing them as arrows (see, Slide 8 from Lecture 17), but you may represent them as you like, as long as you explain your convention.

(b) Which sequence do you get if you apply the function get-lcs-sequence as given in class. (**Warning:** There are multiple LCS's of the same length, and the function given in class returns one of these. I will give 50% credit for any LCS, and full credit for the same one generated by the algorithm given in class. See the Extra-Credit Problem for more.)

Problem 2. (20 points) In this problem we consider some variations of the longest common subsequence (LCS) problem. In all instances the input consists of two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ and the output involves the length or weight of a common subsequence (or a variant thereof).

In each of the following instances, present a recursive DP formulation. (You do *not* need to give pseudocode, just the recursive rule.) Justify the correctness of your solution.

Hint: For all DP formulations, don't forget to include (1) the basis case(s), and (2) how to obtain the final answer given your formulation.

(a) **Weighted CS:** The letters are drawn from an alphabet Σ. For each symbol $z \in \Sigma$, let $w(z)$ denote the *weight* of this symbol. The *weighted common subsequence* (WCS) is the common subsequence $Z = \langle z_1, \ldots, z_k \rangle$ that maximizes the total weight $\sum_{j=1}^{k} w(z_j)$. (For example, in Fig. 1(a), the standard LCS is $\langle$ACCBA$\rangle$, which has a weight of 22 but the WCS is $\langle$BBA$\rangle$, which has a weight of 25.)

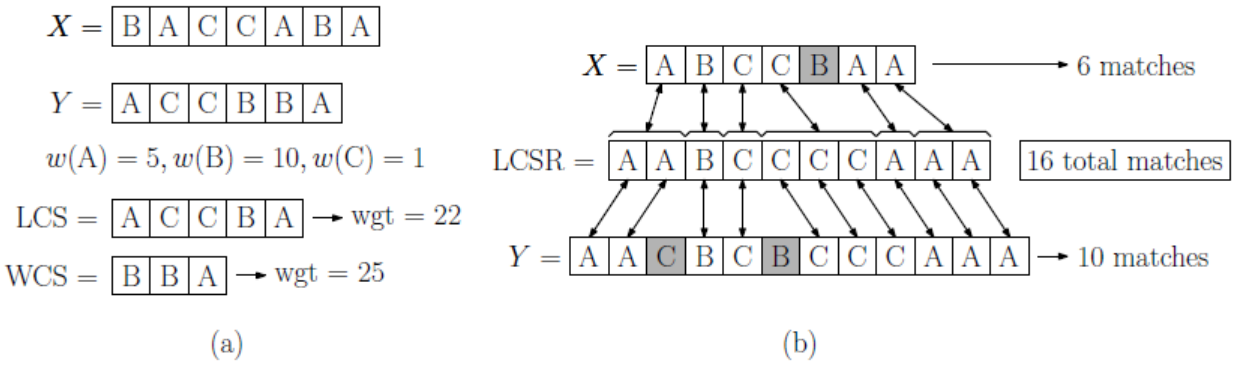

Figure 1: Problem 2.

(b) **LCS with One-Sided Repeats:** Given X and Y, we define a *common subsequence with repeats* to be a sequence $Z = \langle z_1, \ldots, z_k \rangle$ that is a subsequence of Y, and if some repeated contiguous symbols of Z are collapsed to a single symbol (e.g., "CCCC" $\rightarrow$ "C"), the result is a subsequence of X. The objective is to maximize the sum of the number of symbols of X that are matched in the LCS and the number of symbols of Y that are matched in the LCS. (For example, in Fig. 1 we show a possible input-output combination. Six symbols of X are matched and ten symbols of Y are matched, for a total of 16.)

While you do not have to implement your algorithm, for full credit it should be clear that your solution can be implemented in $O(mn)$ time.

Problem 3.(15 points) Let's return to the typesetting problem from Homework 4. Recall that we are given a line of length L and a paragraph consisting of a sequence of words whose lengths are $W = \langle w_1, ..., w_n \rangle$. (We assume that $w_i < L$ for all i.) We are to place words in order along each line subject to the condition that the sum of word lengths on any line does not exceed L. The penalty for each line is defined to be the difference between the sum of word lengths on this line and L. The objective is to place the words to minimize the *maximum penalty* over all the lines (see Fig. 2(a)). In Homework 4 we showed that a greedy strategy

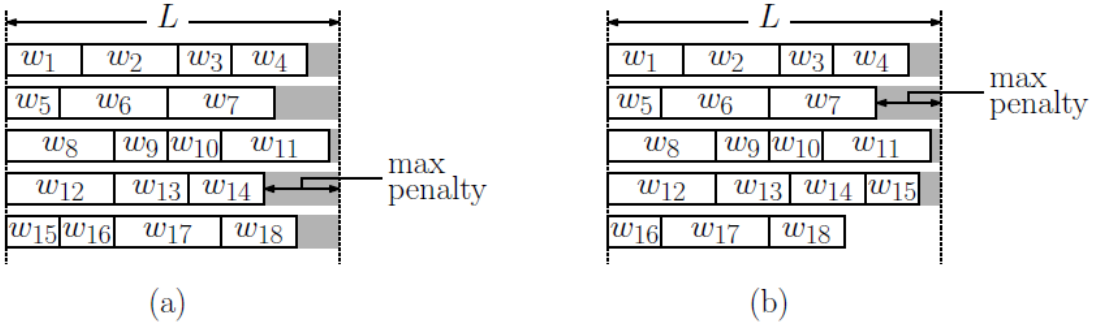

Figure 2: Problem 3.

is not optimal. In this problem we will show that this problem can be solved optimally by dynamic programming.

(a) Derive a (recursive) dynamic programming rule, which given L and the word sequence W, determines the layout that minimizes the maximum penalty (see Fig. 2(a)). Actually, I don't care about the layout, just the final value of the maximum penalty, and I don't need a full algorithm, just the recursive DP formulation.
Briefly justify the correctness of your algorithm and derive its running time (if it were implemented). It may help to imagine that you have access to a function $W(i, j)$ that returns the sum of word lengths from w_i up to w_j (assuming that $1 \leq i \leq j \leq n$) that runs in constant time.

(b) In practice, when laying out a paragraph we do not care whether the last line is "ragged". Modify your solution from part (a) to compute the layout that minimizes the maximum layout excluding the last line. (For example, by this metric the layout shown in Fig. 2(b) has a lower cost than the layout from Fig. 2(a).)
As in part (a), briefly justify the correctness of your algorithm and derive its running time.

Problem 4. (10 points) Programming (upload code to Moodle)

Consider the shortest common supersequence (SCS) problem. The input to the SCS problem is two strings $A = \langle A_1 ... A_m \rangle$ and $B = \langle B_1 ... B_n \rangle$, and the output is the length of the shortest string that contains both A and B as subsequences.

Example: $A = \langle ABCBABA \rangle$, $B = \langle BCAABAB \rangle$, then the $SCS = \langle ABCAABABA \rangle$, so the len$(SCS) = 9$.

Given that the recurrence relationship is as follows, implement a dynamic programming algorithm to solve the SCS problem. Let $SCS(i, j)$ be the length of the shortest common supersequence of $A[1..i]$ and $B[1..j]$. Your algorithm **must** run in $O(mn)$ time for full credit. Your program should prompt the user to enter two strings and output the length of the shortest common supersequence, not the actual SCS.

$$SCS(i,j) = min \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ SCS(i-1, j-1) + 1 & i, j > 0 \text{ and } x_i = y_j \\ SCS(i, j-1) + 1 & i, j > 0 \text{ and } x_i \neq y_j \\ SCS(i-1, j) + 1 & i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Extra Credit: Add backtracking hints to your code and use them to output the SCS as well as the length of the SCS.

Challenge Problem. (Extra Credit) Returning to Problem 1, observe that in this example there are many LCS's of equal length, such as $\langle BCAB \rangle$,$\langle ACAB \rangle$, and $\langle BACB \rangle$. How would you modify the LCS algorithm (both filling out the LCS table (lcs$[i, j]$) and the helper table ($h[i, j]$) so that it would be possible to obtain *all* the LCS strings?

How would you modify the algorithms to output the *number* of distinct LCS sequences?