Practice Problems 6

Problem 1. In this problem, we will trace the LCS algorithm on the input strings $X = \langle CACAB \rangle$ and $Y = \langle BACBA \rangle$.

(a) Show the contents of the lcs and H tables. For consistency, present your answer in the same format as Figure 5(b) from Lecture 9.

For the sake of consistency, follow the same conventions as from class. In particular, whenever there are multiple alternatives of equal value, break ties in favor of (from highest to lowest) "add-xy" ($\nwarrow$), "skip-x" ($\uparrow$), "skip-y" ($\leftarrow$).

(b) Which table entry yields the final value of $|\text{LCS}(X, Y)|$?

(c) Explain step-by-step how the LCS is computed by tracing through the H.

Problem 2. Recall that in the longest common subsequence (LCS) problem the input consists of two strings $X = \langle x_1, \ldots x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ and the objective is to compute the longest string that is a subsequence of both X and Y. Consider the following two variants of the LCS problem. In each case, present an efficient DP algorithm. Express your algorithm as a recursive formula (not pseudocode). Indicate what the initial call is to your recursive function. Justify the correctness of your solution, and derive the running time of the algorithm, if it were to be implemented.

(a) (LCS with wild cards) Each of the strings X and Y may contain a special character "?", which is allowed to match *any single character* of the other string, *except* another wild-card character (see Fig 1(a)).

(b) (LCS with swaps) Any two consecutive characters of either string are allowed to be swapped before matching in the LCS (see Fig 1(b)). Swaps cannot be chained, by which we mean that once two character are swapped, neither character can be swapped again.

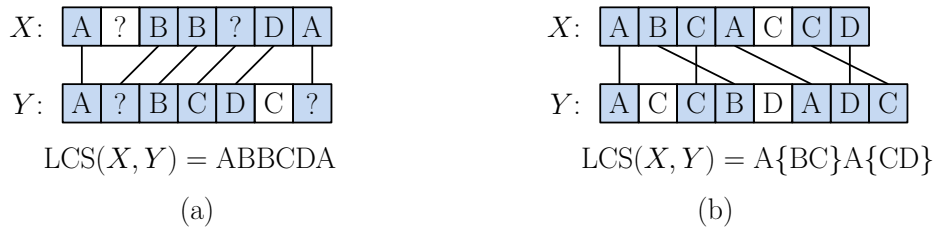

Figure 1: LCS variants.

Hint: Aim for a running time of $O(mn)$.

Problem 3. A common optimization problem for word processors and typesetters is splitting a string of text into lines. The input to this problem consists of a text string, which consists of a sequence of n words $W = \langle w_1, \ldots, w_n \rangle$, where w_i denotes the length of the ith word of the

text. Given a line of length L, our objective is to add line breaks among the words so that the words on each line have a total length of at most L. We call this a *segmentation* (see Fig. 2).

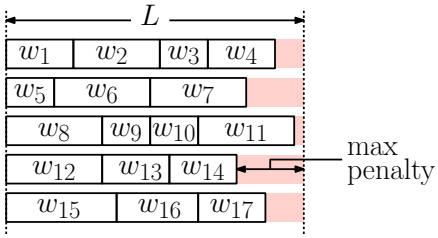

Figure 2: Typesetting words to minimize the maximum penalty.

There are many possible segmentations, and to measure how good one is, we define the *penalty* of each line to be the amount unused space on that line. We define the *maximum penalty* of a segmentation to be the largest penalty over all the lines (see Fig. 2).

(a) There is a simple *greedy algorithm* for segmentation: Fill each line with as many words as possible, so long as the total length does not exceed L. (An example is shown in Fig. 2).

Prove that the greedy segmentation is *not* optimal with respect to max penalty, by presenting a counterexample. (Try to keep your counterexample as short as possible. Show what greedy does on your input, and show what the optimum is.)

(b) Next, let's derive an optimal solution. Present a recursive DP formulation, which given L and the sequence of word lengths $W = \langle w_1, \ldots, w_n \rangle$, determines the segmentation of words to lines (without reordering) that minimizes the maximum penalty (see Fig. 2(a)). As is typical with DP problems, your formulation will compute the maximum penalty, not the actual segmentation. Briefly justify the correctness of your formulation. (We just want the recursive formulation, not pseudocode.)

(c) Present pseudocode for a an implementation of your DP formulation. (You may use either a recursive memoized algorithm or a bottom-up iterative solution.)

What is its running time? It may help to imagine that you have access to a helper function $\text{len}(i, j)$, which 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.