Solutions to Practice Problems 6

Solution 1: The DP table $\text{lcs}[0..m, 0..n]$ is shown in Fig. 1. For example, to compute $\text{lcs}[3,3]$, we observe that $x_3 = y_3 =$ 'C', and so we set $\text{lcs}[3,3] = 1 + \text{lcs}[2,2] = 1 + 1 = 2$. On the other hand, to compute $\text{lcs}[5,2]$, we observe that $x_5 =$ 'B' is not equal to $y_2 =$ 'A', and so we set $\text{lcs}[5,2] = \max(\text{lcs}[4,2], \text{lcs}[5,1]$. These are both equal to 1, so the result is 1. (Note that we preferred the "Skip-X" option, due to the manner in which the algorithm from class broke ties, but it would not be incorrect to select "Skip-Y" instead.)

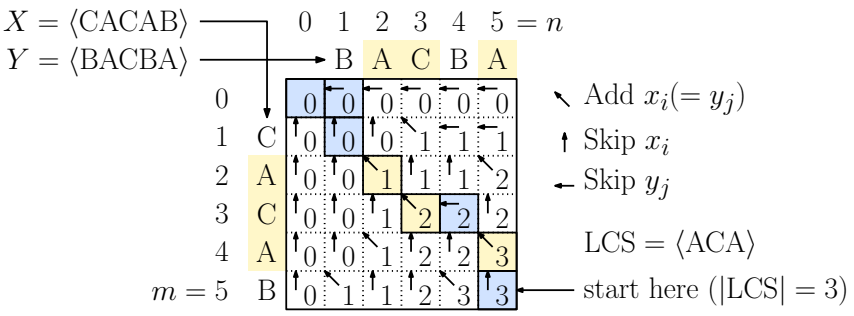

Figure 1: Tracing the LCS algorithm.

The final answer is $\text{lcs}[m,n] = \text{lcs}[5,5] = 3$. To obtain the actual LCS, we trace back the decisions, as illustrated by the arrows. Whenever we see a diagonal arrow (as in $H[4,5]$, we take the common character in each string and decrement both i and j. These are shown with the yellow-shaded entries. Otherwise, if we see a vertical arrow, we go up by decrementing i, and if we see a horizontal arrow, we go left by decrementing j.) These are shown as the blue-shaded entries. This generates the characters of the LCS in reverse order.

Solution 2: As in class, let us assume that $X_i = \langle x_1 \ldots x_i \rangle$ and $Y_j = \langle y_1 \ldots y_j \rangle$. For $0 \le i \le m$ and $0 \le j \le n$, let $\text{lcs}(i,j)$ denote the length of the LCS of X_i and Y_j. The basis case ($i = 0$ or $j = 0$) are the same (the LCS length is zero), so we will assume that both i and j are positive.

(a) (LCS with wild cards) If $x_i = y_j$, but neither is "?", then we claim that the LCS ends with this common character. (If it ended with a different character, then neither of these characters are being used, and we could make the LCS longer by adding this common character.) Also, if either x_i or y_j (both not both) is "?" then we may assume that the LCS ends with the other character. (If it ended with a different character, then either we wasted the wild card, or we used it to match with a different character. Since wild cards can be matched with any character, we may as well use it to match the last character.) In either case, we add $+1$ to the length of the LCS, and we recurse on the remaining substrings X_{i-1} and Y_{j-1}. Otherwise, if neither character is a wild card and they are not equal, we know that one of them cannot be in the LCS. We try removing each, calculate the cost of the resulting LCS, and take the

better of the two options. This leads to the following recursive DP formulation:

$$\text{lcs}(i,j) \;=\; \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ \text{lcs}(i-1,j-1)+1 & \begin{cases} \text{if } x_i = y_j \neq \text{“?” or} \\ \quad \text{either } x_i \text{ or } y_j \text{ (not both) equal “?”} \end{cases} \\ \max(\text{lcs}(i-1,j),\ \text{lcs}(i,j-1)) & \text{otherwise.} \end{cases}$$

The final answer is $\text{lcs}(m,n)$.

(b) (LCS with swaps) We add an additional rule to the standard LCS formulation. If $i,j \geq 2$, and the last two characters of X_i and Y_j are swaps of each other, that is, $\langle x_{i-1}x_i\rangle = \langle y_j y_{j-1}\rangle$, then we add both characters to the LCS and recurse on X_{i-2} and Y_{j-2}. (While I believe that it can be proved that it is always safe to take such a swapped match whenever it emerges, just to be safe, we will simply consider this among the possible options and take the best of all of them.)

$$\text{lcs}(i,j) \;=\; \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ \text{lcs}(i-2,j-2)+2 & \text{if } i,j \geq 2 \text{ and } \langle x_{i-1}x_i\rangle = \langle y_j y_{j-1}\rangle, \\ \text{lcs}(i-1,j-1)+1 & \text{if } i,j \geq 1 \text{ and } x_i = y_j, \\ \max(\text{lcs}(i-1,j),\ \text{lcs}(i,j-1)) & \text{otherwise.} \end{cases}$$

The final answer is $\text{lcs}(m,n)$.

Solution 3:

(a) A counterexample arises by setting $L=3$, and having $w_1=w_2=w_3=w_4=1$ (see Fig. 2(b)). The greedy algorithm generates a layout with the first three words on the first line (penalty 0) and the last word on the second line (penalty 2), for a maximum penalty of 2. However, putting two words on each line (penalties 1 and 1) yields a lower maximum penalty of 1.

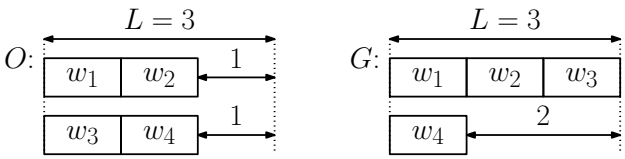

Figure 2: Counterexample to greedy.

You might protest that it is not reasonable to include the last line in the penalty. We can adjust the counterexample to work even in this case. Place a word w_5 after w_4 that is almost as long as L. In either case, this word will need to placed on a line by itself, and w_4 will not be on the last line.

(b) For $0 \leq j \leq n$, let $\text{MP}(j)$ denote the smallest achievable max-penalty for typesetting the first j words. Ultimately, we want to compute $\text{MP}(n)$. For the basis case we have $\text{MP}(0)=0$, since there is nothing to lay out and hence there is no penalty involved. Let us assume we have access to a utility function $\text{len}(i,j)$, which, for $1 \leq i \leq j \leq n$, returns the sum of word lengths $\sum_{k=1}^{j} w_k$. (See the challenge problem for how this can be done.) To compute $\text{MP}(j)$

observe that w_j will be the last word on the last line of the layout, but what is the first word of this line? It will be some word w_i, where $1 \leq i \leq j$ and $\text{len}(i,j) \leq L$. Assuming this, the penalty associated with this line will be $L - \text{len}(i,j)$. Assuming that we lay out the remaining words w_1 through w_{i-1} in the best possible manner, the remaining penalty is $\text{MP}(i-1)$. (Observe that the principle of optimality holds here.) The overall penalty is the maximum of these quantities, that is, $\max(L - \text{len}(i,j), \text{MP}(i-1))$. Among the available options, we select the one that produces the lowest value. Thus, we have

$$\text{MP}(j) \;=\; \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{\substack{1 \leq i \leq j \\ \text{len}(i,j) \leq L}} \max(L - \text{len}(i,j), \text{MP}(i-1)) & \text{otherwise.} \end{cases}$$

This solution involves a 1-parameter function, but requires a loop to determine the best split. Another approach (which yields the same running time, but takes more space) is based on a 2-parameter function. For $1 \leq i \leq j \leq n$, let $\text{MP}'(i,j)$ denote the smallest achievable max-penalty for typesetting the first j words, under the assumption that the last line starts *on or before* w_i. The final answer will be $\text{MP}'(n,n)$. To avoid dealing with cases where the total word length exceeds the line length, let us define

$$\text{penalty}(i,j) \;=\; \begin{cases} L - \text{len}(i,j) & \text{if } \text{len}(i,j) \leq L \\ \infty & \text{otherwise} \end{cases}$$

For the basis case, observe that if $i = 1$, then we are putting all the words on a single line, and the overall penalty is $\text{penalty}(i,j)$. Otherwise, $i \geq 2$. Either the last line starts with w_i, in which case the penalty for this last line is $\text{penalty}(i,j)$. The remaining subproblem is to typeset the first $i-1$ words, which is $\text{MP}'(i-1, i-1)$. The overall max-penalty is the maximum of these two. Otherwise, the last line starts earlier than w_i (or equivalently, on or before w_{i-1}), in which case the overall penalty is given by $\text{MP}'(i-1,j)$. As always, we take the better of these two options.

$$\text{MP}'(i,j) \;=\; \begin{cases} \text{penalty}(i,j) & \text{if } i = 1 \\ \min(\max(\text{penalty}(i,j), \text{MP}'(i-1,i-1)), \text{MP}'(i-1,j)) & \text{otherwise.} \end{cases}$$

Note that once $\text{len}(i,j)$ exceeds L, $\text{MP}'(i,j)$ will be ∞, thus if we were to implement this, we could add this additional check to avoid unnecessary recursive function calls.

(c) We present a memoized implementation in the code block below. (A bottom-up implementation is also quite straightforward). We assume that the array `pred[1..n]` has been precomputed, and we have access to the function $\text{len}(i,j)$. The values are stored in the array $\text{MP}[0..n]$, which is initialized to -1. The initial call is `max-penalty(n)`, which computes the minimum-penalty segmentation of all n words.

Clearly, there are $n+1$ values $\text{MP}[0..n]$ to be computed, and each one involves minimizing over $O(n)$ possibilities. If we can compute $\text{len}(i,j)$ runs in constant time, the overall time is $O(n^2)$.

```
max-penalty(j) {                                    // max-penalty typesetting
    if (MP[j] == -1) {                              // undefined?
        if (j == 0) {                               // basis case
            MP[0] = 0
        } else {
            MP[j] = infinity
            for (i = j downto 1)   {                // try all possible splits
                if (len(i,j) > L) break             // too many words for this line
                thisPenalty = L - len(i,j)          // penalty for last line
                prevPenalty = max-penalty(i-1)      // penalty for prev lines
                MP[j] = min(max(thisPenalty, prevPenalty))
            }
        }
    }
    return MP[j]                                    // return the max penalty
}
```