Solutions to Homework 3: Dynamic Programming

Solution 1:

(a) See Fig. 1 (left side).

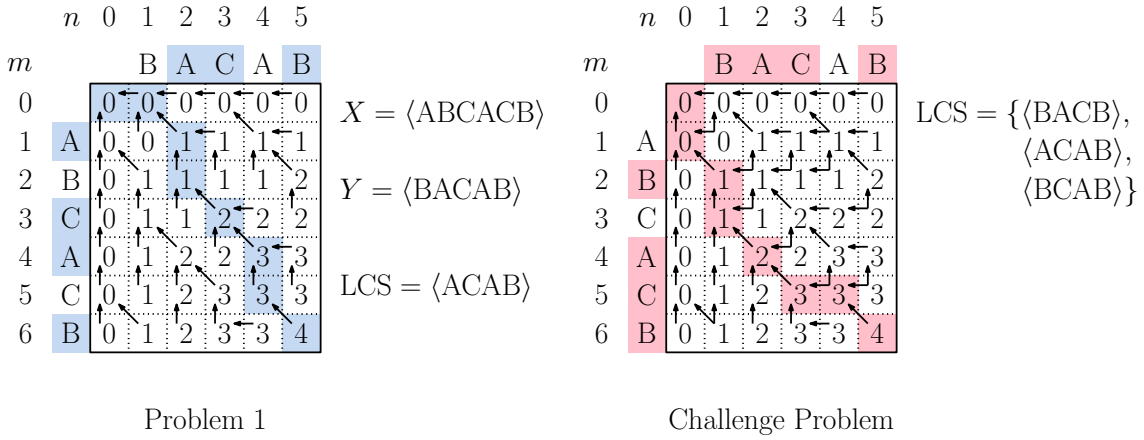

Figure 1: Solution to Problem 1 and the Challenge Problem.

(b) The final LCS depends on the tie-breaking choices made by the algorithm whenever multiple choices lead to the same outcome. If $x_i = y_j$, the algorithm always selects add_{XY} ('$\nwarrow$'). If not the algorithm selects skip_X ('$\uparrow$') if $\text{lcs}[i-1, j] \geq \text{lcs}[i, j-1]$. The algorithm selects the third option skip_Y ('$\rightarrow$') otherwise.

The function get-lcs-sequence traces the resulting path back from the lower-right corner of the table ($\text{lcs}[m, n]$) to the upper-left corner ($\text{lcs}[0, 0]$), and includes all characters where a diagonal move add_{XY} ('$\nwarrow$') was made. The result is $\langle ACAB \rangle$ (see the path of blue squares in Fig. 1 (left)).

Solution 2: As in class, define $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_j = \langle y_1, \ldots, y_j \rangle$.

(a) **Weighted LCS:** For $0 \leq i \leq m$ and $0 \leq j \leq n$, define $\text{wcs}(i, j)$ to be the length of the WCS of X_i and Y_j. For the basis case, if either string is empty then the weight of the WCS is zero, implying that $\text{wcs}(i, 0) = \text{wcs}(0, j) = 0$. Otherwise, let us assume that both i and j are at least 1. There are two cases:

- If $x_i \neq y_j$, then we know that one of these two characters (possibly both) are not in the final WCS. If we exclude x_i the WCS weight is $\text{wcs}(i-1, j)$ and if we exclude y_j the WCS weight is $\text{wcs}(i, j-1)$. We select the option that leads to the larger weight, that is, $\text{wcs}(i, j) = \max(\text{wcs}(i-1, j), \text{wcs}(i, j-1))$.
- If $x_i = y_j$, it follows from the same argument given in class and the fact that weights are positive that this common symbol will be the last symbol of the WCS. Furthermore,

there is no harm in matching x_i with y_j, since any other matching would yield the same cost. By matching these symbols, we receive the weight of $w(x_i) = w(y_j)$ and then recur on the first $i-1$ symbols of X and the first $j-1$ symbols of Y. Thus, we have a total weight of $\text{wcs}(i,j) = w(x_i) + \text{wcs}(i-1, j-1))$.

The final recursive formulation is shown below:

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

The final answer is $\text{wcs}(m,n)$. (Because each entry of the table can be computed in constant time, the total time to implement this is $O(mn)$.)

(b) **LCS with One-Sided Repeats:** For $0 \leq i \leq m$ and $0 \leq j \leq n$, define $\text{lcsr}(i,j)$ to be total number of matched symbols in the LCS with one-sided repeats of X_i and Y_j. As in part (a), for the basis case we have $\text{lcsr}(i,0) = \text{lcsr}(0,j) = 0$. Otherwise, let us assume that both i and j are at least 1. There are two cases:

- If $x_i \neq y_j$, then by exactly the same reasoning as in the standard LCS problem, we have $\text{lcsr}(i,j) = \max(\text{lcsr}(i-1,j), \text{lcsr}(i,j-1))$.
- If $x_i = y_j$, it follows from the same argument given in class that this common symbol will be the last symbol of the LCS. The new wrinkle that comes in the repetition case is that we may want to retain the last symbol x_i to match other symbols within Y. There are two possibilities: (1) match y_j and leave x_i for future reuse or (2) match both y_j and x_i. In the first case we match one symbol and in the second we match two symbols. This yields the following rule: $\text{lcsr}(i,j) = \max(1 + \text{lcsr}(i,j-1), 2 + \text{lcsr}(i-1,j-1))$. (Note that both terms of the max are needed. The second rule alone would fail to count repetitions and the first rule alone would fail to count symbols of X.)

In summary, here is the final recursive formulation:

$$\text{lcsr}(i,j) \;=\; \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max(\text{lcsr}(i-1,j), \text{lcsr}(i,j-1)) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \\ \max(1 + \text{lcsr}(i,j-1),\ 2 + \text{lcsr}(i-1,j-1)) & \text{if } i,j > 0 \text{ and } x_i = y_j. \end{cases}$$

The final answer is $\text{lcsr}(m,n)$. (Because each entry of the table can be computed in constant time, the total time to implement this is $O(mn)$.)

(b') **LCS with One-Sided Repeats:** (Alternate interpretation) Some people misunderstood the problem description, thinking that repetitions must be contiguous substrings of Y. Let us consider a DP formulation for this interpretation.

The basis cases and the case when $x_i \neq y_j$ are the same as before. The case when $x_i = y_j$ differs, however. First, we may elect not to match y_j at all (opting instead for an earlier, longer substring of repeated copies of x_i). This yields the option $\text{lcsr}(i,j-1)$. Otherwise, we initiate a process of matching a suffix of repeated characters Y_j with x_i. It may not always be advantageous to consume the entire run of repeated characters. (For example, if $X = \langle \text{ABA} \rangle$ and $Y = \langle \text{AA} \rangle$, we do not want to match the "AA" run of Y with last "A" of X.)

I struggled to come up with a simple rule for handling this case.[1] In order to allow us to short-circuit, we will try all runs of repeated characters at the end of Y_j, and will select the one that yields the best result. We need some notation to know when a run of repeated characters ends. For $1 \leq j \leq n$, define $q[j]$ to be the smallest index such that all the symbols of $y_{q[j]}, y_{q[j]+1}, \ldots, y_j$ are equal. (If there is no run of repeated characters, then $q[j] = j$.)

Once we start a run of repeats, the run must end at some k, where $q[j] \leq k \leq j$. We match the symbols $\langle y_k, \ldots, y_j \rangle$ of Y with x_i, for a total of $j - k + 2$ matches. After consuming all these symbols, we recurse on X_{i-1} and Y_{k-1}. This yields the following formula for finding the best suffix of repeated symbols to match x_i:

$$\max_{q[j] \leq k \leq j} ((j - k + 2) + \text{lcsr}(i - 1, k - 1)).$$

Putting the two various cases together, we obtain the following DP formulation for this interpretation of the problem:

$$\text{lcsr}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max \left(\begin{array}{c} \text{lcsr}(i, j - 1) \\ \max\limits_{q[j] \leq k \leq j} ((j - k + 2) + \text{lcsr}(i - 1, k - 1)) \end{array} \right) & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(\text{lcsr}(i - 1, j), \text{lcsr}(i, j - 1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Solution 3:

(a) For $0 \leq i \leq n$ and $0 \leq v \leq W$, define $\text{profit}(i, v)$ to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, \ldots, i\}$ for a ship that has a maximum capacity of v tons. To compute this array, we first consider the basis case. If $i = 0$, then there are no items to select from. If $v = 0$, then the ship has no capacity to take any containers. Thus, $\text{profit}(i, v) = 0$ if either $i = 0$ or $v = 0$.

Therefore, let us assume that $i \geq 1$ and $v \geq 1$. Then let us consider the ith container. If we decide not to take this container, we receive no profit and we use up none of the remaining weight capacity. Therefore, the best we can do is to select from the previous $i - 1$ containers but we still have the full v units of weight capacity. Therefore, $\text{profit}(i, v) = \text{profit}(i - 1, v)$. On the other hand, if we decide to take the ith container, then we receive a profit of p_i dollars, but the remaining weight capacity is decreased by w_i, and we can select from the remaining $i - 1$ containers. Thus, $\text{profit}(i, v) = p_i + \text{profit}(i - 1, v - w_i)$. If $w_i > v$, this will yield a negative parameter value. To keep the notation simple and avoid multiple cases, we make the assumption that "*if any parameter is negative, the profit function returns* $-\infty$."

Since we do not know *a priori* which option is better, we try both and take the better of the two options. This yields the following recursive rule:

$$\text{profit}(i, v) = \begin{cases} -\infty & \text{if } v < 0 \\ 0 & \text{if } i = 0 \text{ or } v = 0 \\ \max(\text{profit}(i - 1, v), p_i + \text{profit}(i - 1, v - w_i)) & \text{if } i, v > 0. \end{cases}$$

[1] I think that there may be simpler formulations, but it would take more work to prove that they are optimal.

The final answer is the maximum profit to select from all the n containers for a ship of full capacity W, that is, profit(n, W). (Because each entry of the table takes constant time to compute, this can be implemented in $O(nW)$ time.)

(b) To handle the container limit, we add an additional parameter to the formulation. For $0 \leq i \leq n$, $0 \leq v \leq W$, and $0 \leq c \leq C$, define profit(i, v, c) to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, \ldots, i\}$ for a ship that has a maximum capacity of v tons, and can hold c containers.

The above rule is modified in the following manner. If $c = 0$, then the profit is zero. Otherwise, whenever we choose to take a container, we decrease the number of allowable containers by one. Thus, we have the following rule:

$$\text{profit}(i, v, c) = \begin{cases} -\infty & \text{if } v < 0 \\ 0 & \text{if } i = 0,\ v = 0, \text{ or } c = 0 \\ \max(\text{profit}(i-1, v), & \\ \quad p_i + \text{profit}(i-1, v - w_i, c-1)) & \text{if all indices} > 0. \end{cases}$$

As before, if the v_k index is negative, the value is $-\infty$. The final answer is the maximum profit to select from all the n containers for a ship of full weight capacity W, and full container capacity C, that is, profit(n, W, C). (Because each entry of the table takes constant time to compute, this can be implemented in $O(nWC)$ time.)

(c) Our solution is to replicate the weight and capacity indices, one set for each of the ships. For $0 \leq i \leq n$, $0 \leq v_1, v_2 \leq W$, and $0 \leq c_1, c_2 \leq C$, define profit(i, v_1, c_1, v_2, c_2) to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, \ldots, i\}$ where for $k \in \{1, 2\}$, ship k has a maximum capacity of v_k tons, and can hold c_k containers. With each container we can decide either to leave it, add it to the first ship, or add it to the second. As before, if the v_k index is negative, the value is $-\infty$.

$$\text{profit}(i_1, v_1, c_1, v_2, c_2) = \begin{cases} -\infty & \text{if } \min(v_1, v_2) < 0 \\ 0 & \text{if any parameter is 0} \\ \max(\text{profit}(i-1, v_1, c_1, v_2, c_2), & \\ \quad p_i + \text{profit}(i-1, v_1 - w_i, c_1 - 1, v_2, c_2)) & \\ \quad p_i + \text{profit}(i-1, v_1, c_1, v_2 - w_i, c_2 - 1)) & \text{if all indices} > 0. \end{cases}$$

The final answer is the maximum profit to select from all the n containers with both ships of full weight capacity W, and full container capacity C, that is, profit(n, W, C, W, C). (Because each entry of the table takes constant time to compute, this can be implemented in $O(nW^2C^2)$ time.)

Solution 4: For $1 \leq a \leq H$ and $1 \leq b \leq W$, let profit(a, b) denote the maximum profit achievable by cutting a sheet of height a and width b. To simplify matters, let us define a function $p^*(a, b)$. If there exists i such that $h_i = a$ and $w_i = b$, this function returns the profit p_i. Otherwise, it returns -1.

The basis case is $a = b = 1$, since we cannot split a 1×1 sheet any smaller, we return $p^*(1, 1)$. Otherwise, it is possible to make at least one cut. There are three cases:

- We do not make any further cuts, which yields a profit of $p^*(a, b)$.
- We make a horizontal cut at some y, where $1 \leq y \leq a - 1$. Such a cut results in two pieces, a lower piece of size $y \times b$ and an upper piece of size $(a - y) \times b$. By induction, this yields a profit of $\text{profit}(y, b) + \text{profit}(a - y, b)$.
- We make a vertical cut at some x, such that $1 \leq x \leq b - 1$. Such a cut results in two pieces, a left piece of size $a \times x$ and a right piece of size $a \times (b - x)$. By induction, this yields a profit of $\text{profit}(a, x) + \text{profit}(a, b - x)$.

Since we don't know *a priori* which of these options is the best, we try the all and take the one that produces the highest profit. This yields the following recursive rule.

$$\text{profit}(a, b) \;=\; \max \left(\begin{array}{l} p^*(a, b), \\ \max_{1 \leq y \leq a-1}(\text{profit}(y, b) + \text{profit}(a - y, b)), \\ \max_{1 \leq x \leq b-1}(\text{profit}(a, x) + \text{profit}(a, b - x)) \end{array} \right)$$

Observe that if $a = 1$ or $b = 1$ the associated max-statement degenerates to be a max over zero options. We will assume that in such a case "*the max-statement returns a value of* $-\infty$." Thus, if no cut is possible ($a = b = 1$) then the $p^*(a, b)$ will be chosen. So, this handles the basis case.

Solution to Challenge Problem 1: The modification to obtain all the possible LCS strings involves saving all the possible alternatives, rather than favoring a particular one. For example, if $x_i = y_j$, we know that it is safe to take the common symbol $x_i = y_j$ for the LCS (yielding a length of $1 + \text{lcs}(i - 1, j - 1)$), but if the values $\text{lcs}(i - 1, j)$ and/or $\text{lcs}(i, j - 1)$ yield this same value, then we are justified in taking them as well. Similarly, if $x_i \neq y_j$, and $\text{lcs}(i - 1, j) = \text{lcs}(i, j - 1)$, we can save both as options.

As a result, each helper entry, $H[i, j]$, is a set which contains all the possible values (add_{XY}, skip_X and skip_Y) that lead to the same maximum LCS value for X_i and Y_j.

In order to compute the number of distinct LCS strings, we start at $H[m, n]$ and count the number of paths leading back to $H[0, 0]$. To do this, define $P[i, j]$ to be this number of paths. In order to compute $P[i, j]$ we consult $H[i, j]$. If $\text{skip}_X \in H[i, j]$, then we add $P[i - 1, j]$ to the count, if $\text{skip}_Y \in H[i, j]$, then we add $P[i, j - 1]$ to the count, and if $\text{add}_{XY} \in H[i, j]$, then we add $P[i - 1, j - 1]$ to the count. For the basis, we set $P[0, 0] = 1$ (counting the trivial path to itself). All other entries are initialized to 0 and then incremented as described above. This is shown in the code-fragment below.

Solution to Challenge Problem 2:

(a) We present a memoized version of the metal-cutting problem. Let $\text{Profit}(a, b)$ be a function that returns the maximum profit achievable by cutting a sheet of height a and width b. Whenever we compute a value, we cache it in the array $\text{profit}[a, b]$. The per-piece profit function $p * (a, b)$ described in the solution to Problem 4 can be implemented by storing the ordre entries in a hash-map indexed by the pairs $[a, b]$. Whenever a key $[a, b] = [h_i, w_i]$ is found, we return the profit p_i, and otherwise we return the waste value of -1.

Assuming that each access to the hash-map runs in $O(1)$ time, the algorithm's total running time is $O(HW(H + W))$. The reason is that there are $H \cdot W$ choices for a and b. For each, we either return in constant time, or we invest up to H time in the first for-loop and up to W

```
bottom-up-lcs-all-options() {
    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) {                      // compute lcs
            if (x[i] == y[j]) lcs[i, j] = lcs[i-1, j-1] + 1
            else lcs[i, j] = max(lcs[i-1, j], lcs[i, j-1])

            H[i,j] = empty-set                  // compute helpers
            if (x[i] == y[j]) add ‘addXY’ to H[i, j]
            if (lcs[i, j] == lcs[i-1, j]) add ‘skipX’ to H[i, j]
            if (lcs[i, j] == lcs[i, j-1]) add ‘skipY’ to H[i, j]
        }
    }
    return lcs[m, n]                            // final lcs length
}

get-lcs-count() {                               // count the number of LCSs
    P = new array [0..m, 0..n]
    for (i = 0 to m) {
        for (j = 0 to n) {
            if (i == 0 && j == 0) P[i, j] = 1   // basis case
            else P[i, j] = 0                    // sum the 3 possible options
            if (skipX is in H[i, j]) P[i, j] += P[i-1, j]
            if (skipY is in H[i, j]) P[i, j] += P[i, j-1]
            if (addXY is in H[i, j]) P[i, j] += P[i-1, j-1]
        }
    }
    return P[m, n]                              // return the final count
}
```

Computing the max profit for metal cutting

```
memoized-profit(a, b) {
    if (profit[a, b] is already computed) return profit[a, b]
    else {
        if (a == 1 && b == 1) {                  // basis case 1x1
            prof = p*(1, 1);  help = "No-Cut"
        }
        else {
            prof = p*(a, b);  help = "No-Cut"    // default - no cut

            for (y = 1 to a-1) {                 // try all horizontal cuts
                py = memoized-profit(y, b) + memoized-profit(a-y, b)
                if (py > prof) {
                    prof = py;  help = ("Horizontal", y)
                }
            }
            for (x = 1 to b-1) {                 // try all vertical cuts
                px = memoized-profit(a, x) + memoized-profit(a, b-x)
                if (px > prof) {
                    prof = px;  help = ("Vertical", x)
                }
            }
        }
        profit[a, b] = prof                      // save whichever cut is best
        helper[a, b] = help
        return profit[a, b]                      // return the final profit
    }
}
```

time in the second for-loop, for a total of $(H + W)$ for each table entry computed. In order to create the initial hash-map, we need to store the n orders, which takes (in expectation) $O(n)$ time. Therefore, the total running time is $O(n + HW(H + W))$

(b) The recursive function below outputs the actual sequence of cuts (without any indentation) by unraveling the recursion with the aid of the helper pointers.

Computing the Actual Cuts

```
get-cuts(a, b) {
    if (helper[a, b] == "No-Cut") {             // no further cuts made
        if (p*(a, b) == -1) output "a x b waste for -$1"
        else output "a x b piece for p*(a,b)"
    }
    else {
        if (helper[a, b].direction == "Horizontal") {
            y = helper[a,b].location            // location of the cut
            output "Horizontal cut at y ("
            get-cuts(y, b)                      // output cuts for lower part
            output ") ("
            get-cuts(a-y, b)                    // output cuts for upper part
            output ")"
        } else { /* helper[a, b].direction == "Horizontal" */
            x = helper[a,b].location            // location of the cut
            output "Vertical cut at x ("
            get-cuts(a, x)                      // output cuts for left part
            output ") ("
            get-cuts(a, b-x)                    // output cuts for right part
            output ")"
        }
    }
}
```