3. In this problem you are to apply the dynamic programming algorithm given in the worked example at the end of this problem set to (i) compute the length of the longest subsequence and (ii) find an actual longest common subsequence for the pair of strings $X = \text{ACGTAA}$ and $Y = \text{CGTCAG}$. Specifically, you are to compute the following.

   a. The $7 \times 7$ array $L$ where $L[i, j]$ records the length of the longest common subsequence for $X[1 : i]$ and $Y[1 : j]$, for $0 \leq i, j \leq 7$. (Note that $L[0, j] = 0$ for it denotes the length of the longest subsequence of the empty string $X[1 : 0]$ and $Y[1 : j]$; similarly, $L[i, 0] = 0$.) Use the recursive formulation to compute the table entries row by row, starting with row 1.

   b. The $6 \times 6$ array $C$ where, for $1 \leq i, j \leq 6$, $C(i, j)$ has three possible entries: M, U or B (short for Match, Up and Back, respectively). M indicates that $L[i, j]$ is obtained by adding 1 to $L[i - 1, j - 1]$ because $X[i]$ and $Y[j]$ match, U indicates that $L[i, j] = L[i - 1, j]$, and B indicates that $L[i, j] = L[i, j - 1]$ (if both U and B are possible choose B).

   c. Which locations of array $C$ are visited and in what order so as to form a longest common substring? Show the construction of this longest common substring named $\text{LS}$ as the relevant locations of $C$ are visited. (Use the procedure in part e of the sample problem solution.)

4. Let $X[1 : m]$, $Y[1 : n]$ and $Z[1 : p]$ be three strings of lengths $m$, $n$, and $p$, respectively. The task is to find a string $S$ which is a longest common subsequence for $X$, $Y$ and $Z$. Note that the following approach does not work: compute $S_1$, a longest common subsequence for $X$ and $Y$, and then compute $S_2$, a longest common subsequence for $X_1$ and $Z$. The following example illustrates this: $X = \text{addbeec}$, $Y = \text{affbdde}$, $Z = \text{acbbfc}$; $S_1 = \text{addc}$ is the longest common subsequence for $X$ and $Y$, $ac$ is the longest common subsequence for $S_1$ and $Z$, but the longest common subsequence for $X$, $Y$ and $Z$ is $\text{abc}$.

   Give a dynamic programming algorithm to compute $S$. Your algorithm should be described in the following stages.

   a. Write a recursive formula for the longest common subsequence.

   b. Write a recursive procedure to compute the formula given in part a, in pseudo-code.

   c. Augment the program with table lookup for efficiency.

   d. Record the best choices in a second table so as to be able to quickly find an actual optimal solution and not just its value.

   e. Give a second procedure that outputs the optimal solution once an dynamic programming
procedure has been run.
f. Give the Driver procedure.
g. Analyze the runtime of your algorithm. (This is basically the number of table entries
times the time to compute a single table entry exclusive of recursive calls.)

5. Consider the following bead solitaire game. It is played with a sequence of \( n \) cups 
\( C_1, C_2, \ldots, C_n \) arranged in a row. Each cup holds zero or more blue beads and zero or more
red beads. The player can see the number of beads of each color in each of the cups. For
our convenience this data is stored in arrays \( R[1:n] \) and \( B[1:n] \), where \( R[i] \) records the
number of red beads in cup \( i \) and \( B[i] \) the number of blue beads.

On each move the (solitary) player empties one or more cups in sequence from the right
end, keeping all the beads from the emptied cups if collectively they contain more blue than
red bids, and otherwise discarding the beads. The player’s goal is to keep as many beads as
possible. The game continues until all the cups have been emptied.

Your task is to give a dynamic programming algorithm to provide the optimal strategy
for the player. Again proceed using steps a–g.

Precomputing the sums \( SR[i] = R[i] + R[i+1] + \ldots + R[n] \) and \( SB[i] = B[i] + B[i+1] +
\ldots + B[n] \) will facilitate the description of the algorithm.

6. Consider the following road-building decision making. The Department of Transport has
\( M \) to spend on road repairs. There are roads \( R_1, R_2, \ldots, R_n \) it could repair. Repairing \( R_i \)
will cost \( \$c_i \), and will save \( t_i \) hours of travel time per year. The goal is to maximize the
travel time saved, given the constraint that no more than \( \$M \) can be spent. \( M, c_1, \ldots, c_n \)
and \( t_1, \ldots, t_n \) are all integers.

Give a recursive formulation for the function \( \text{Time}(L, k) \) which returns the maximum time
saved when spending up to \( \$L \) on repairing (some of) the roads \( R_1, R_2, \ldots, R_k \). Explain what
is the runtime of the dynamic programming implementation of \( T(M, n) \), augmented to list
the set of roads to repair, justifying your answer.

7. Consider the longest common subsequence problem, but with an additional parameter
\( k \). The input comprises \( n \) character strings \( X \) and \( Y \) as before, and the task is to re-
port the longest common subsequence in the event that it has length at least \( n - k + 1 \).
However, your algorithm must run in time \( O(nk) \). You may find it helpful to compute
\( \text{BestSubSeq}(X, i, Y, j, h) \) (BSS for short), which reports the length of the longest common
subsequence of \( X[1:i] \) and \( Y[1:j] \) assuming it has length at least \( \max\{i, j\} - h + 1 \) and
outputs \(-1\) otherwise.
Sample problem
Let $X[1 : m]$ and $Y[1 : n]$ be two strings of lengths $m$ and $n$, respectively. The task is to find a string $S$ which is a longest common subsequence for $X$ and $Y$.

Give a dynamic programming algorithm to compute $S$. Your algorithm should be described in the stages a–g as in problem 4.

Solution

a. $LSS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LSS(i - 1, j - 1) + 1 & \text{if } i, j \geq 1 \text{ and } X[i] = Y[j] \\ \max\{LSS(i - 1, j), LSS(i, j - 1)\} & \text{if } i, j \geq 1 \end{cases}$

b. Hencerforth, we assume that $X$ and $Y$ have been declared as global variables.

function $LSS(i, j)$;
    if $i = 0$ or $j = 0$ then return(0)
    else if $X[i] = Y[j]$ then return($1 + LSS(i - 1, j - 1)$)
    else return($\max\{LSS(i - 1, j), LSS(i, j - 1)\}$)
end

c. Driver;
    global variable $L[0 : m, 0 : n]$, initialized to have all entries -1;
    $LSS(m, n)$
end

function $LSS(i, j)$;
    if $L[i, j] = -1$ then do
        if $i = 0$ or $j = 0$ then $L[i, j] \leftarrow 0$
        else if $X[i] = Y[j]$ then $L[i, j] \leftarrow 1 + LSS(i - 1, j - 1)$;
        else do
            $l_1 \leftarrow LSS(i - 1, j)$; $l_2 \leftarrow LSS(i, j - 1)$;
            if $l_1 < l_2$
                then $L[i, j] \leftarrow l_2$
            else $L[i, j] \leftarrow l_1$
        end; (* else do *)
    end; (* then do *)
    return($L[i, j]$)
end

d. We use M, U, B as the possible entries in $C$ to indicate whether there was a match (M), or index $i$ was reduced by 1 (U), or index $j$ was reduced by 1 (B).
Driver;

**global variable** $L[0 : m, 0 : n]$, initialized to have all entries -1;
**global variable** $C[1 : m, 1 : n]$, uninitialized;

$LSS(m, n)$

end

function $LSS(i, j)$;

if $L[i, j] = -1$ then do
    if $i = 0$ or $j = 0$ then $L[i, j] \leftarrow 0$
    else if $X[i] = Y[j]$ then do
        $L[i, j] \leftarrow 1 + LSS(i - 1, j - 1)$;
        $C[i, j] \leftarrow M$
    end (* then do *)
else do
    $l_1 \leftarrow LSS(i - 1, j)$; $l_2 \leftarrow LSS(i, j - 1)$;
    if $l_1 < l_2$ then do
        $L[i, j] \leftarrow LSS(i, j - 1)$;
        $C[i, j] \leftarrow B$
    end (* then do *)
else do
    $L[i, j] \leftarrow LSS(i - 1, j)$;
    $C[i, j] \leftarrow U$
end (* else do *)
end (* else do *)

return ($L[i, j]$)
end

e. We assume that $l = LSS(i, j)$. Then the following procedure builds a length $l$ longest common subsequence of $X$ and $Y$, from left to right, storing it in the first $l$ locations of array $A[1 : n]$.

$BuildS(i, j, l)$

if $l > 0$ then do
    if $C[i, j] = M$ then do
        $BuildS(i - 1, j - 1, l - 1)$; $ALS[l] \leftarrow X[i]$
    end (* then do *)
else if $C[i, j] = U$ then $BuildS(i - 1, j, l)$
    else $BuildS(i, j - 1, l)$
end (* then do *)
f. Driver;
   global variable $L[0 : m, 0 : n]$, initialized to have all entries -1;
   global variable $C[1 : m, 1 : n]$, uninitialized;
   $l ← \text{LSS}(m, n)$;
   global variable $\text{ALS}[1 : l]$, uninitialized;
   $\text{BuildS}(m, n, l)$
end

g. There are $(m + 1)(n + 1)$ table entries in $L$ and $mn$ entries in $C$ and each entry takes time $O(1)$ to compute aside from recursive calls. Consequently, the whole algorithm runs in $O(mn)$ time. Note that the time to run the BuildS procedure is linear in the length of the longest common subsequence which is $O(m + n)$.

Short solution for parts b-g.
We compute the solution recursively using a table $L$ to store the solutions as they are found. Before making a recursive call we first check if the solution is already in $L$ and if so simply report the stored solution. We also use a second table $C$ to record the choice made in computing each entry of $L$ for $i, j \geq 1$, namely which assignment is used when computing the value $L[i, j]$. The computation of each entry in $L$ and $C$ takes $\Theta(1)$ time aside the cost of recursive calls, and as there are $O(mn)$ entries filled in all, the total cost for running the recursive procedure on $X$ and $Y$ is $O(mn)$.

To compute an actual longest subsequence we use a choice recovery procedure. It recursively follows the choices stored in $C$ starting at $C[m, n]$. This will take time $O(m + n)$, as each recursive call decrements the length of at least one of the strings being considered.