CPSC 320: DP in 2-D*

The Longest Common Subsequence of two strings `A` and `B` is the longest string whose letters appear in order (but not necessarily consecutively) within both `A` and `B`. For example, the LCS of “computer science” and “mathematics” is the length 5 string `mteic` (“$\overset{A}{\text{computer science}}$” and “$\overset{B}{\text{mathematics}}$”). Biologists: If these were DNA base or amino acid sequences, can you imagine how this might be a useful problem?

1. Write down small and trivial instances of the problem.

   Trivial : either A and/or B is empty (ϵ)
   $\Rightarrow$ length of longest common subsequence
   LLCS is zero.

   Small : both A and B have length 1

2. Now, working backward from the end (i.e., from the last letters, as with the change-making problem where we worked from the total amount of change desired down to zero), let’s figure out the first choice we make as we break the problem down into smaller pieces:

   (a) Consider the two strings `compute` and `science`. Describe the relationship of the length of their LCS with the length of the LCS of `comput` and `scienc` (strings `A` and `B` with both of their last letters removed).

   LLCS(compute, science) $\overset{?}{\geq}$ LLCS(comput, scienc)
   = 1 + ''

CPSC 320: DP in 2-D*

The Longest Common Subsequence of two strings `A` and `B` is the longest string whose letters appear in order (but not necessarily consecutively) within both `A` and `B`. For example, the LCS of "computer science" and "mathematics" is the length 5 string `mteic` ("computer science" and "mathematics"). Biologists: If these were DNA base or amino acid sequences, can you imagine how this might be a useful problem?

computer science

mathematics

1. Write down small and trivial instances of the problem.

Trivial : either A and/or B is empty (ϵ)
$\Rightarrow$ length of longest common subsequence
LLCS is zero.

Small : both A and B have length 1

2. Now, working backward from the end (i.e., from the last letters, as with the change-making problem where we worked from the total amount of change desired down to zero), let's figure out the first choice we make as we break the problem down into smaller pieces:

   (a) Consider the two strings `compute` and `science`. Describe the relationship of the length of their LCS with the length of the LCS of `comput` and `scienc` (strings `A` and `B` with both of their last letters removed).

LLCS(compute, science) ?≥? LLCS(comput, scienc)
= 1 + " "

Given two strings `A` and `B` of length $n > 0$ and $m > 0$, we will denote the length of the LCS of `A` and `B` by `LLCS(A[1..n], B[1..m])`.

Clicker Question

Suppose $A[n] = B[m]$. Complete the recurrence for the LLCS in this case:

$\text{LLCS}(A[1 \ldots n],\ B[1 \ldots m]) = \ldots$

A. $\text{LLCS}(A[1 \ldots n-1],\ B[1 \ldots m-1])$
B. $1 + \text{LLCS}(A[1 \ldots n-1],\ B[1 \ldots m-1])$
C. $\text{LLCS}(A[2 \ldots n],\ B[2 \ldots m])$
D. $1 + \text{LLCS}(A[2 \ldots n],\ B[2 \ldots m])$

(b) Now consider the two strings `tycoon` and `country`. Describe the relationship of the length of their LCS with the length of the LCS of `tycoon` and `countr` (the same string `A`, and string `B` with its last letter removed).

LLCS(tycoon, country) ??? LLCS(tycoon, countr)

=

$\geq$

Given two strings `A` and `B` of length $n > 0$ and $m > 0$, we will denote the length of the LCS of `A` and `B` by `LLCS(A[1..n], B[1..m])`.

A[1 2 ---- n]

B[1 2 -- - m]

Clicker Question

Suppose $A[n] \neq B[m]$. Which of the following is **always true** about LLCS($A[1 \ldots n],\ B[1 \ldots m]$)?

A. LLCS($A[1 \ldots n],\ B[1 \ldots m]$) = LLCS($A[1 \ldots n-1],\ B[1 \ldots m-1]$)

B. LLCS($A[1 \ldots n],\ B[1 \ldots m]$) = LLCS($A[1 \ldots n],\ B[1 \ldots m-1]$)

C. LLCS($A[1 \ldots n],\ B[1 \ldots m]$) = LLCS($A[1 \ldots n-1],\ B[1 \ldots m]$)

D. At least one of B or C is true

E. None of the above statements are always true

3. Given two strings A and B of length $n > 0$ and $m > 0$, we will denote the length of the LCS of A and B by `LLCS(A[1..n], B[1..m])`. Describe `LLCS(A[1..n], B[1..m])` as a recurrence relation over smaller instances. Use and generalize your work in the previous problems!

$$LLCS(A[1..n], B[1..m]) = \begin{cases} 1 + LLCS(A[1..n-1], B[1..m-1]), & \text{if } A[n] = B[m] \\ \max\{LLCS(A[1...n], B[1...m-1]), \; LLCS(A[1...n-1], B[1...m])\}, & \text{if } A[n] \neq B[m] \end{cases}$$

A ~~greedy~~ dynamic programming algorithm proceeds by:

- ~~Making a single choice based on a simple, local criterion~~
- Considering a (polynomial) number of possible choices
- Solving one or more subproblem(s) that result from each choice
- Combining the choice and its associated subproblem(s) (e.g., by calculating the total score)
- Selecting the best choice (with maximum or minimum score).

This technique is useful when the possible choices lead to **repeated or overlapping** subproblems

4. Given two strings A and B, if either has a length of 0, what is the length of their LCS?

0

5. Convert your recurrence into a memoized solution to the LLCS problem.

First, recall the form of a memoized algorithm, derived from recurrence:

$$C(n) = \begin{cases} \infty & \text{if } n < 0 \quad \text{// boundary cases} \\ 0 & \text{if } n = 0 \quad \text{// base case} \\ \min\{C(n-25)+1, C(n-10)+1, C(n-1)+1\} & \text{otherwise} \quad \text{// general case} \end{cases}$$

Making Change

```
function Memo-Change(n)
    create a new array Soln[1..n]
    for i from 1 to n do Soln[i] ← −1
    return Memo-Change-Helper(n)

function Memo-Change-Helper(i)
    if i < 0 then
        return infinity
    else if i = 0 then
        return 0
    else
        if Soln[i] == −1 then
            Soln[i] ← the minimum of:
                Memo-Change-Helper(i − 25) + 1,
                Memo-Change-Helper(i − 10) + 1,
                Memo-Change-Helper(i − 1) + 1.
        return Soln[i]
```

General Form

```
function Memo-XX(instance)
    create a new array Soln[...]
    initialize entries in the array, e.g., to Init
    return Memo-XX-Helper(instance)

function Memo-XX-Helper(sub-instance)
    if ... then          // boundary cases if needed

    else if ...          // base cases

    else                 // general case(s)
        if Soln[...] == Init then
            // use recurrence and recursion to
            // store a value in Soln[...]
```

5. Convert your recurrence into a memoized solution to the LLCS problem.

$$\mathrm{LLCS}(A[1..n], B[1..m]) = \begin{cases} 0, & \text{if } n = 0 \text{ or } m = 0 \quad \text{// base case} \\ \mathrm{LLCS}(A[1..n-1], B[1..m-1]) + 1, & \text{if } A[n] = B[m], \\ \max\{\mathrm{LLCS}(A[1..n-1], B[1..m]), \mathrm{LLCS}(A[1..n], B[1..m-1])\}, & \text{otherwise .} \end{cases}$$

Start with the top-level function.

```
function Memo-LLCS(n, m)
  create Soln[0...n][0...m]
  for i from 0 to n
     for j from 0 to m
          Soln[i][j] ← -1
  return MEMO-LLCS-Helper(n, m)
```

```
function MEMO-XX(instance)
   create a new array Soln[...]
   initialize entries in the array e.g., to Init
   return MEMO-XX-HELPER(instance)
```

Then the helper function.

```
function MEMO-LLCS-Helper(i, j)
  if Soln[i, j] == -1 then
    if (i==0 or j==0) then
       Soln[i, j] ← 0.
    else if A[i] == B[j] then
       Soln[i, j] ← 1 +
          MEMO-LLCS-Helper(i-1, j-1)
    else
       Soln[i, j] ← max{
          MEMO-LLCS-Helper(i, j-1),
    endif "      "      " (i-1, j)}
  endif
  return Soln[i, j]
```

```
function MEMO-XX-HELPER(sub-instance)

  if Soln[...] == Init then
    if ...        // base cases
      // store base case in Soln[...]
    else
      // use recurrence and recursion to
      // store a value in Soln[...]
  return Soln[..]
```

6. Complete the following table to find the length of the LCS of `tycoon` and `country` using your memoized solution. (The row and column headed with an ϵ, denoting the empty string, are for the trivial cases!)

tycoon
country

Soln array

	ϵ	c	co	cou	coun	count	countr	country
ϵ	0	0	0	0	0	0	0	0
t	0	0	0	0	0	1	1	1
ty	0	0	0	0	0	1	1	2
tyc	0	1	1	1	1	1	1	2
tyco	0	1	2	2	2	2	2	2
tycoo	0	1	2	2	2	2	2	2
tycoon	0	1	2	2	3	3	3	3

7. Go back to the table and extract the actual LCS from it. Circle each entry of the table you have to inspect in constructing the LCS. Then, use the space below to write an algorithm that extracts the actual LCS from an `LLCS` table.

	ϵ	c	co	cou	coun	count	countr	country
ϵ	0	0	0	0	0	0	0	0
t	0	0	0	0	0	1	1	1
ty	0	0	0	0	0	1	1	2
tyc	0	1	1	1	1	1	1	2
tyco	0	1	2	2	2	2	2	2
tycoo	0	1	2	2	2	2	2	2
tycoon	0	1	2	2	3	3	3	3

base case ↑ o≠c c=c o=o o≠u n=n n≠t n≠r n≠y

c o n

procedure Extract-LCS (A[1...n], B[1...m], Soln[0..n][0..m])

if (n==0) or (m==0)

return ε (empty string)

elseif A[n] == B[m] then

return Extract-LCS (A[1...n-1], B[1.. m-1], Soln ...) + A[n]

elseif Soln[n, m-1] ≥ Soln[n-1, m]

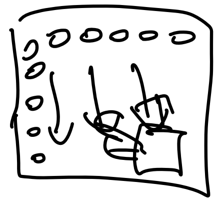
return Extract-LCS (A[1.. n], B[1...m-1], Soln[ ])

else // Soln[n, m-1] < Soln[n-1, m]

return Extract-LCS (A[1.. n-1], B[1.. m], Soln[..])

dynamic programming

8. Give an iterative solution that produces the same table as the memoized solution.



$$\mathrm{LLCS}(A[1..n], B[1..m]) = \begin{cases} 0, \text{ if } n=0 \text{ or } m=0 & \text{// base case} \\ \mathrm{LLCS}(A[1..n-1], B[1..m-1]) + 1, & \text{if } A[n] = B[m], \\ \max\{\mathrm{LLCS}(A[1..n-1], B[1..m]), \mathrm{LLCS}(A[1..n], B[1..m-1])\}, & \text{otherwise .} \end{cases}$$

```
function DP-LLCS(A[1..n], B[1..m])
   create Soln[0..n][0..m]
n     for i = 0 to m
+         Soln[0, i] ← 0
m     for j = 1 to n
          Soln[j, 0] ← 0

n     for i from 1 to n
×         for j from 1 to m
m
                if A[i] == B[j]
                    Soln[i, j] ← Soln[i-1, j-1] + 1
O(1)            else
                    Soln[i, j] ← max{ Soln[i, j-1], Soln[i-1, j] }

   return Soln[n, m]
```

General Form of DP Algorithm

```
function DP-XX(instance)
    create a new array Soln[...]
    fill in base case entries of Soln

    for [all other entries]
    // may need nested for loop
        Soln[entry] ←
              // use recurrence + smaller
              // entries of Soln to
              // compute this value
    return desired Soln entry
```

9. Analyze the efficiency of your memoized (part 5) and dynamic programming (part 8) algorithms in terms of runtime and memory use (not including the space used by the parameters). You may assume the strings are of length n and m, where $n \leq m$ (without loss of generality).

DP is $O(nm)$, also

Memoization is $O(nm)$

10. If we only want the **length** of the LCS of `A` and `B` with lengths n and m, where $n \leq m$, explain how we can "get away" with using only $O(n)$ memory in the dynamic programming solution.

When filling in row i of Soln array, we need row $i-1$, but not earlier rows.
We can delete rows $1 \ldots i-2$ once we're on to row i.

So $O(n)$ memory.