6.6 Longest Common Subsequence

A strand of DNA consists of a string of molecules called bases, which can be one of adenine, cytosine, guanine, and thymine. If we use the first letter to represent each base, we can express a strand of DNA as a string over the 4-element set $\{A, C, G, T\}$. We might have two organisms whose DNA strands are

$$S_1 = ACTGCGTCAGCTGCAGCGTCGA$$
$$S_2 = CAGCTGATCGTAGCTGATCG$$

There are a few ways one may decide how similar two strings like the above are. One way is to find the longest common substring. Another way is to count the number of characters we have to edit from string 1 to get string 2, also known as the edit distance. There is a DP solution to find the edit distance, but we will not focus on this solution today.

What we will attempt to find is the longest common subsequence between the two strings. Don't confuse this with the longest common substring! In the longest common subsequence, the goal is to find a third string whose characters appear in the same order in both S_1 and S_2.

Question 96. Verify that ACTGCGTAGCTGATCG is a valid subsequence of S_1 and S_2.

$$S_1 = ACTGCGTCAGCTGCAGCGTCGA$$
$$ACTGCGTAGCTGATCG$$
$$S_2 = CAGCTGATCGTAGCTGATCG$$

Problem (LCS).

INPUT: Two sequences X with length m and Y with length n.

OUTPUT: The length of the maximum length common subsequence of X and Y.

Question 97. Reason about whether or not this problem has the optimal substructure property.

- Write down a concise statement for the "optimal solution" using the input parameters.
- What are a set of easy top level choices to make, which break the problem into multiple subproblems? Remember, a subproblem must also be a valid instance of the main problem!

① OPT(X, Y): The LCS of ~~strings X with length m~~ and ~~Y with length n~~.
the first m characters of X the first n chars of Y.

② A: If last char of X and last char of Y are the same, use both in the LCS.

X: HEROICALLY
Y: SCHOLARLY

LCS("HEROICALL", "SCHOLARL") → 4 → HOLL (Y)

B: Ignore the last character of X.

X: APPLE$
Y: APPLE

→ LCS("APPLE", "APPLE") = "APPLE" (len 5.)

↖ Ignoring the last character of X still gave the correct LCS, if the last chars are different. ↙

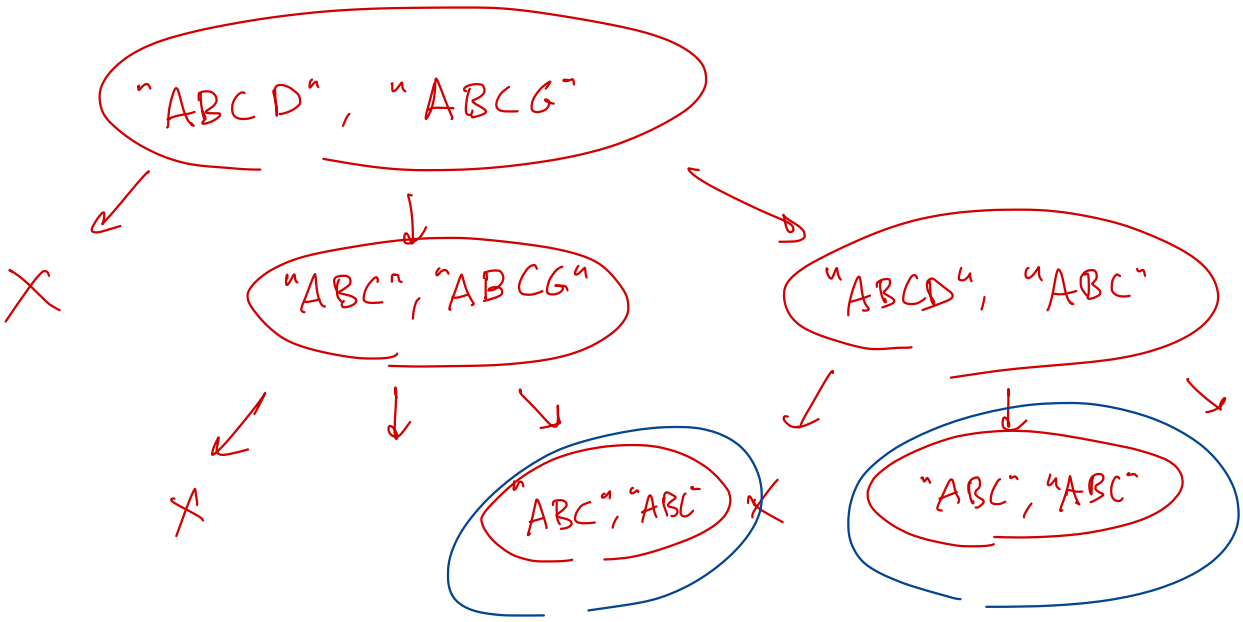
C: Ignore the last character of Y.

X: CART
Y: CATS

Question 98. Using the above analysis, write down a recursive algorithm that solves the problem. This algorithm should not use a memo and will likely be inefficient.

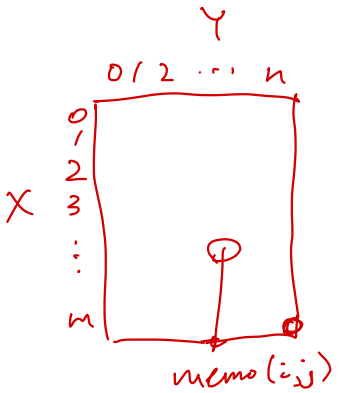
```
def recursive_LCS(X, Y):
    if X == "" or Y == "":
        return                                          } Base case

    if X[-1] == Y[-1]:
        return recursive_LCS(X[:-1], Y[:-1]) + 1        } Recursion.

    else:
        return max(recursive_LCS(X[:-1], Y),
                   recursive_LCS(X, Y[:-1]))
```

Question 99. Given your answer to the previous problem, does the LCS problem also have overlapping subproblems? If yes, write down a scenario where the same subproblem gets called twice.



Question 100. Identify the pieces of the dynamic programming algorithm.

1. **Subproblem domain:** The space of possible subproblems to consider.
2. **Memo table definition:** The items we will be storing in our tables. Alternatively, the name of solutions to subproblems.
3. **Goal:** The location of our solution in the table.
4. **Initial values:** Solutions to trivial subproblems to start building on.
5. **Recurrence:** A compact representation of the recursive function.



(1) The first i chars of X for $i \in [0, m]$
The first j chars of Y for $j \in [0, n]$

(2) memo(i,j) = The length of the LCS between the first i chars of X and the first j chars of Y.

(3) Goal: memo(m,n)

(4) Base cases: LCS w/ any empty string has length 0.
$\Rightarrow$ memo(0,j) = 0 memo(i,0) = 0

(5) Recurrence: The dependencies between memo elements.

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

The following is some python code to compute the LCS length in a bottom up fashion.

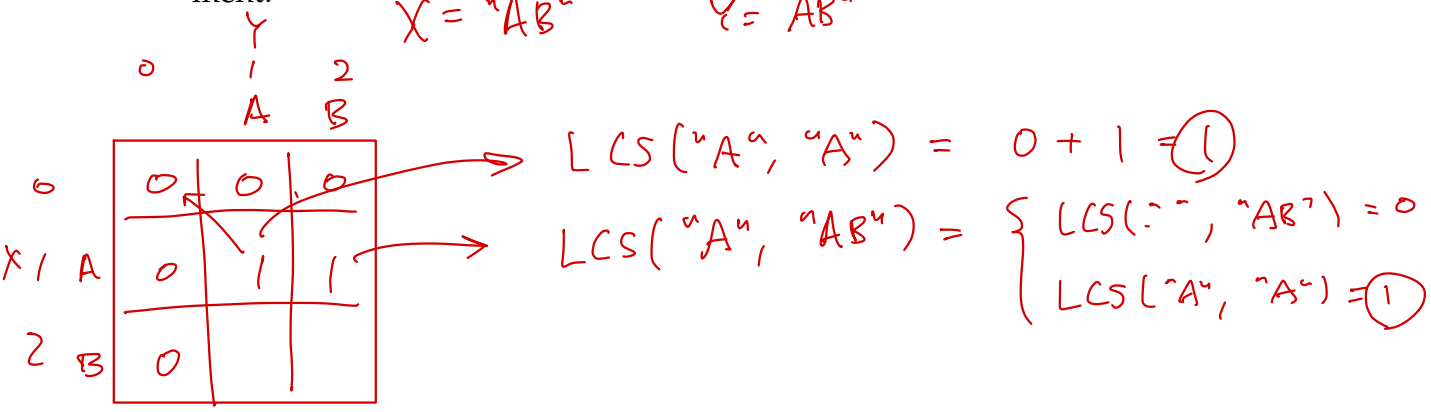
```
1  def LCS(X, Y):
2      m, n = len(X), len(Y)
3      memo =  # Table with [0:m, 0:n]. Stores the LCS of X[1:m], Y[1:n]
4      prev =  # Table with [1:m, 1:n].
5              # Stores the subproblem this solution builds out of.
6      for i in range(1, m + 1):
7          memo[i][0] = 0
8      for j in range(0, n + 1):
9          memo[0][j] = 0
10     for i in range(1, m + 1):
11         for j in range(1, n + 1):
12             if X[i] == Y[j]:
13                 memo[i][j] = memo[i - 1][j - 1] + 1
14                 prev[i][j] = "↖"
15             elif memo[i - 1][j] >= memo[i, j - 1]:
16                 memo[i][j] = memo[i - 1][j] + 1
17                 prev[i][j] = "↑"
18             else:
19                 memo[i][j] = memo[i][j - 1] + 1
20                 prev[i][j] = "←"
21     return memo, prev
```

Base cases.

case 1

else cases.

Question 101. Draw an example that corresponds to each of the branches in the if statement.



The following is an example table that you may have filled after the inputs $X = ABCBDAB$ and $Y = BDCABA$.

i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↑ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	←	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	4

Question 102. Fill out the cells that are empty in the table.

Question 103. Can you use the table above to reconstruct the LCS?

"↑" ignore last char of X

"←" ignore last char of Y

"↖" use char in LCS.

BCBA