Longest Common Subsequences

Dr.Muhammad Abdullah

Subsequences

Definition: A subsequence is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example:
Consider the sequence `ACCGGTC`. Some possible subsequences are:

- `ACCG` (by picking the first four letters)
- `ACTC` (by picking letters 1, 2, 6, and 7)
- `CCC` (by picking letters 2, 3, and 7)

Formal Definition of a Subsequence

Given a sequence $X = x_1, x_2, \ldots, x_m$, another sequence $Z = z_1, z_2, \ldots, z_k$ is a subsequence of X if there exists a strictly increasing sequence $i_1, i_2, \ldots, i_k$ of indices of X such that:

$$\text{for all } j = 1, 2, \ldots, k, \quad x_{i_j} = z_j$$

Longest Common Subsequence Problem

Problem Statement:
Given two sequences X and Y, find the longest possible sequence that is a subsequence of both X and Y.

Example:

- $X = \text{ABCBDAB}$
- $Y = \text{BDCABA}$

A common subsequence of X and Y is `BCA`. However, it is not the longest common subsequence.

Example of Longest Common Subsequences

Continued Example:

For $X = \text{ABCBDAB}$ and $Y = \text{BDCABA}$:

- `BCBA` and `BDAB` are both longest common subsequences.
- There are no common sequences of length 5 or greater.

Optimal Substructure

Dynamic Programming Approach:
To solve the longest common subsequence (LCS) problem using dynamic programming, we need to use solutions to subproblems to construct the optimal solution.

Key Idea:
There are two possible cases:

1. The last elements of X and Y are equal. In this case, both elements are part of the LCS. We remove these elements and find the LCS of the smaller sequences.
2. The last elements of X and Y are different. In this case, either the last element of X or the last element of Y cannot be part of the LCS. We find the LCS of X and a smaller version of Y, or the LCS of Y and a smaller version of X.

Formal Definition of Optimal Substructure

Given:

- Sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$
- $Z = z_1, z_2, \ldots, z_k$ is a longest common subsequence of X and Y

Let X_i refer to the first i elements of X, and Y_j refer to the first j elements of Y.

The following cases arise:

1. If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is a longest common subsequence of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$, implying Z is a longest common subsequence of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$, implying Z is a longest common subsequence of X and Y_{n-1}.

Combining Solutions

Using the optimal substructure property, we can solve the LCS problem by:

- Finding the longest common subsequence of smaller subproblems
- Combining these solutions to construct the optimal solution for the original problem

A Recursive Solution

Approach:
We use a 2D matrix c to store the solutions to subproblems. Each value $c[i, j]$ represents the length of the longest common subsequence between the first i elements of X and the first j elements of Y.

Goal:
Compute all values $c[i, j]$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.
The final answer (length of the LCS of X and Y) will be stored in $c[m, n]$.

Recursive Formulation

The length of the longest common subsequence $c[i, j]$ is computed as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

Explanation:

- If $i = 0$ or $j = 0$, one sequence is empty, so the LCS length is 0.
- If $x_i = y_j$, we include x_i (or y_j) in the LCS and add 1 to the result of the subproblem $c[i-1, j-1]$.
- If $x_i \neq y_j$, we take the maximum of the LCS values excluding the current element from either X or Y.

Example of Recursion

Example:
Consider sequences $X = \text{ABCBDAB}$ and $Y = \text{BDCABA}$.

- To find $c[7, 6]$, we check if $x_7 = y_6$. Since they are not equal, we compute:
$$c[7, 6] = \max(c[6, 6], c[7, 5])$$
- We continue this process recursively until we reach base cases where $i = 0$ or $j = 0$.

Dynamic Programming Algorithm

Using the Recurrence Relation:
The dynamic programming approach uses the previously defined recurrence relation to fill a 2D table.

Key Idea:

- Populate the table c in a specific order, since some elements depend on others that must already be computed.
- The final solution (length of the LCS) will be in $c[m, n]$, where m and n are the lengths of the sequences X and Y.

LCS Dynamic Programming Algorithm (Pseudocode)

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Running Time of LCS Algorithm

The time complexity of the Longest Common Subsequence (LCS) algorithm using dynamic programming is:

$$O(mn)$$

where:

- m is the length of the first string
- n is the length of the second string

Each entry in the $m \times n$ table is computed in constant time $O(1)$, and there are $m \times n$ entries to compute. Thus, the overall time complexity is $O(mn)$.

Complete Example of LCS Calculation

Example Sequences:

- X = ABCBDAB
- Y = BDCABA

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

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Reconstructing the LCS

Approach:

- Start from $c[m, n]$ and trace back to $c[0, 0]$ to reconstruct the LCS.
- Follow the rules:
  - If $x_i = y_j$, include x_i (or y_j) in the LCS and move diagonally up-left.
  - If $c[i, j] = c[i-1, j]$, move up.
  - If $c[i, j] = c[i, j-1]$, move left.

Reconstruction Algorithm (Code)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```