Dynamic Programming 4: Longest Common Subsequence

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

A string s is a **subsequence** of another string t if either $s = t$ or we can convert t to s by deleting characters.

Example: $t =$ ABCDEF

The following are subsequences of t: ABD, ACDF, and ABCDEF.
The following are not: ACB, ACG, and BDFE.

We denote by $|s|$ the **length** of s.

The Longest Common Subsequence Problem

Given two strings x and y, find a common subsequence z of x and y with the maximum length.

We will refer to z as a **longest common subsequence** (LCS) of x and y.

Example: If $x = \texttt{ABCBDAB}$ and $y = \texttt{BDCABA}$, then BCBA is an LCS of x and y, so is BCAB.

If $x = \emptyset$ (empty string) and $y = \texttt{BDCABA}$, their (only) LCS is $\emptyset$.

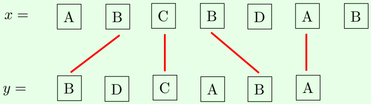
The "Graph" View

A common subsequence z induces a **correspondence graph** between the strings x and y.

- Identify an occurrence of z in x, and an occurrence of z in y.
- For each $i \in [1, |z|]$, draw an edge between
    - the character of x used to match $z[i]$, and
    - the character of y used to match $z[i]$.

If a character of x is connected to a character of y, they are said to **match** each other.

Example: The graph induced by $z = \texttt{BCBA}$:



Note: These edges can be ordered from left to right.

The key to solving the problem is to identify its underlying **recursive structure**.

Specifically, how the original problem is related to subproblems.

The recursive structure will then imply a dyn. programming algorithm.

n = the length of x; m = the length of y

Theorem:

Statement 1: If $x[n] = y[m]$, there **exists** an LCS z of x and y satisfying **both** of the following:

- z induces a correspondence graph where $x[n]$ matches $y[m]$;
- (corollary of the previous bullet) $z[1 : |z| - 1]$ is an LCS of $x[1 : n - 1]$ and $y[1 : m - 1]$.

Statement 2: If $x[n] \neq y[m]$, **any** LCS z of x and y satisfies **at least** one of the following:

- z is an LCS of $x[1 : n - 1]$ and y;
- z is an LCS of x and $y[1 : m - 1]$.

Example:

- Suppose $x = \texttt{BCBDA}$ and $y = \texttt{BDCABA}$.
  The LCS $z = \texttt{BCBA}$ satisfies Statement 1.
- Suppose $x = \texttt{ABCBDAB}$ and $y = \texttt{BDCABA}$.
  The LCS $z = \texttt{BCBA}$ satisfies Statement 2.

Proof of Statement 1:

Take an arbitrary LCS z of x and y. If $x[n]$ matches $y[m]$ in the correspondence graph of z, we are done.

Otherwise, consider the **rightmost** edge e in the correspondence graph. Suppose that the edge matches $x[i]$ with $y[j]$ for some $i \in [1, n]$ and $j \in [1, m]$.

- If $i < n$ and $j < m$, we can add an edge between $x[n]$ and $y[m]$ and thus produce a longer common sequence, giving a contradiction.
- If $i = n$ but $j < m$, replace e with an edge connecting $x[n]$ and $y[m]$.
- If $i < n$ but $j = m$, replace e with an edge connecting $x[n]$ and $y[m]$.

□

Proof of Statement 2:
Take an arbitrary LCS z of x and y and consider the correspondence graph induced by z. Let e be the **rightmost** edge in the graph.

Clearly, e **does not** connect $x[n]$ and $y[m]$ (because they are not identical characters). Thus, either e is not incident on $x[n]$, or e is not incident on $y[m]$. Due to symmetry, we will discuss only the former scenario (e not incident on $x[n]$).

Thus, z also induces a correspondence graph for the input strings $x[1 : n - 1]$ and y. This implies that z must be an LCS of $x[1 : n - 1]$ and y. □

Define $x[1:0] = y[1:0] = \emptyset$ (empty string).

For any $i \in [0, n]$ and $j \in [0, m]$, define

$$opt(i, j) = \text{the LCS length of } x[1:i] \text{ and } y[1:j].$$

Note that $opt(n, m)$ is the LCS length of x and y.

The theorem tells us

$$opt(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ opt(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j] \\ \max\{opt(i, j-1), opt(i-1, j)\} & \text{if } i, j > 0 \text{ and } x[i] \neq y[j] \end{cases}$$

We can compute $opt(n, m)$ in $O(nm)$ time by dynamic programming (last lecture).

Wait! We still need to **generate** an LCS of x and y.

This can be done by using the piggyback technique introduced in the previous lecture. The time complexity remains the same as analyzed earlier. Details are left as a regular exercise.