ALL SUBSEQUENCE MODE OF OPERATIONS

Subsequence modes beyond "prefix-only"

1) Prefix-subsequence of Y in X (what your current design behaves like)

- You try to match Y from **Y[0] onward** in order.
- If Y[0] doesn't match anywhere, you report length 0.
- This is: "How much of Y can I match starting from the beginning?"

Useful when:

- Y is a command sequence / protocol header
- you only care if the sequence "starts correctly"

Example:

- Y="ba", X="a" → length 0 (because it doesn't start with b)

---

2) Best subsequence of Y in X (delete any characters from Y, including the front)

- You can drop *any* chars from Y to maximise what remains in X.
- This is the "true" longest subsequence (LCS variant when X fixed and order preserved).

Example:

- Y="ba", X="a" → "a" length 1

This is what you *thought* you were doing, but it conflicts with your prefix gate.

---

3) Best subsequence of Y in X within a window (windowed subsequence)

- Same as #2, but you require the matched characters in X to lie inside a window X[i..j]
- Then you optimise either:
    - longest subsequence, or
    - smallest window for a given subsequence length

This pairs naturally with your "minimum / maximum window" mindset.

---

4) Longest common subsequence (LCS) between X and Y

- Both sides can "skip" characters.
- This is the classic dynamic programming problem.

Example:

- X="bcab", Y="abb" → LCS is "ab" (length 2)

This is the cleanest mathematically, but it's a different engine than your current scanning approach.

---

5) Longest common substring (contiguous, not subsequence)

- Must be continuous block in both strings.
- Much stricter than subsequence.

Often confused with subsequence, but totally different.

---

6) Approximate / edit-distance matching

Instead of "delete-only", allow:

- insertions, deletions, substitutions
  Score results by Levenshtein distance.

Useful when:

- typos, OCR noise, fuzzy matching

---

7) Constrained subsequence (anchored / must-include)

Examples:

- must include the first character of Y (your current mode)
- must include last character
- must include a specific set of "required" characters
- must match at least K characters

This is common in rule engines.

Which modes fit your design best?

Your code is naturally good for:

- **Mode 1 (prefix-subsequence)** and
- **Mode 3 (windowed subsequence with constraints)**

It's not naturally suited to Mode 2/4 without a redesign, because those require the matcher to freely skip inside Y (including leading chars) and keep consistent bookkeeping.