PLAN ACTION

Thank you for your assistance. I have a general query..... I am not looking for any advise on the code... But if I was to use my existing code and try to incorporate subsequences, would you think it is worth with my design.. Also what are some of the principles that allow my results to be useful to ascertain the longest subsequence (i.e removing minimum letters from String p) to ensure that it appears in String s

Is it worth adding "subsequence" support to your current design?

Yes — **if your goal is exploration / diagnostics**, your design is a decent fit.

Your current pipeline is basically:

1. generate candidate strings (permutations of p)
2. scan s for "in-order" matches and record windows
3. report smallest / largest windows and where matches happen

That maps nicely to subsequences because subsequence questions are also "try a candidate, scan in-order, record what you can match".

Where it's *not* worth it is if you want something that scales to big p (because permutations explode). For subsequences, the number of candidates can be even bigger than permutations if you brute-force removals.

So: **worth it as a tool for insight**, not as a high-performance library (unless you narrow the search space).

Principles that make your results useful for "longest subsequence" (remove minimum letters from p)

If your new objective is:

Find the **longest subsequence of p** that appears in s
(equivalently: remove the **fewest** letters from p so what remains appears in s)

Then these principles explain why your existing "window + in-order scan" output can help:

1) "In-order matching" already measures feasibility

Your core scan is basically the subsequence test:

- Walk through s
- Consume characters from the candidate in order

That's exactly how you check whether a candidate subsequence exists.

So once you switch from "candidate = permutation(p)" to "candidate = subsequence(p)", your scan engine still applies.

2) Window size gives a *compactness* signal

For a fixed candidate subsequence:

- A **smaller window** means the letters occur closer together in s (stronger match).
- A **larger window** means the letters are more spread out (weaker/looser match).

When searching for "best" subsequences, you can use window size as a tie-breaker:

- Prefer longer subsequence length first
- If equal length, prefer smaller window

3) "How far you got" is a natural score

Right now you treat "fail" as binary. For longest-subsequence, the key idea is:

- Even if you don't match all of p, the number of characters you *did* match in order is useful.

So a strong principle is:

- **Matched prefix length** (or total matched count) can guide which letters are "removable".

This turns your engine into a scoring engine:

- For each startPos, record how many characters of the candidate were matched before running out.

4) Minimal removals = maximal matches

"Remove minimum letters from p" is equivalent to:

- maximize k where some subsequence of length k from p appears in s

So your results become useful if you track:

- best k achieved
- which indices/letters of p formed that match

That directly tells you which letters are "kept" and which are "removed".

The clean conceptual route (no code advice, just the idea)

There are two main ways to interpret "longest subsequence" with your existing style:

A) Longest Common Subsequence (LCS) viewpoint

- "Longest subsequence of p that appears in s" is basically **LCS(p, s)**.
- This is the classic "keep as much as possible from p while respecting order".

Your scan idea aligns with the *meaning* of LCS even if you don't implement the classic DP approach.

B) "Deletion distance" viewpoint

If the LCS length is L, then:

- minimum letters to delete from p so it appears in s is: |p| - L

So if your process can identify L, you immediately get the "minimum deletions" metric.

What would make your approach especially useful (as a design philosophy)

Even without changing the core style, your design is valuable when you want:

- **Explainability**: "Here are the windows and indices where the match was formed."
- **Traceability**: You can log which letters matched and where in s.
- **Multiple solutions**: You can keep not just one best subsequence, but many best ties.
- **Real-world intuition**: shortest windows correspond to more "meaningful" local matches.

If you tell me whether you mean **(1)** "subsequence of p in the original order" or **(2)** "subsequence after permuting p first", I can describe the right principles for each (they lead to different "best" answers).

i would still like to explore permutations... And for each permutation of String p, I would explore subsequences by removing minimum letters in order to get longest possible subsequences... That would be my strategy.. And you are right, my existing code stops execution if a letter in STring p is not found in STring s. I would almost need a level of resilience and continue searching STring s

That strategy is coherent, and it fits your "engine + accountability" style:

- **Outer loop:** permutations of p (candidate orderings)
- **Inner loop:** for each permutation, allow deletions (subsequence) to salvage the best possible match inside s
- **Score:** longest kept length (fewest deletions), then maybe smallest window as tie-breaker

Why this can be worth exploring

Even though it's heavier than classic LCS, it gives you something LCS doesn't give by default:

- You can answer: **"Which ordering of p matches s best?"**
- And: **"How many letters must I delete for that best ordering?"**
- And: **"Where in s does it match (window indices)?"**

That's a useful "diagnostic search engine" view.

What "resilience" means in this context

Right now your matching is basically **all-or-nothing** for a permutation:

- if all letters are found in order → success window stored
- if not → you often end up emptying sb and the permutation is treated as a failure

To support "remove minimum letters", you need your matching step to produce a **best partial match** instead of stopping.

The key principle

For a fixed permutation q of length m, and a given start position in s, you want:

- the **maximum number of characters of q** that can be matched in order while scanning s

That maximum is exactly the length of the **longest subsequence of q found in s** (starting from that start position).

Then:

- **minimum deletions** for that permutation = m - bestMatchedLength

This gives you a meaningful result even when the full permutation cannot be matched.

What to record so results are useful

For each permutation q, store a "best attempt" record, something like:

- matchedLength (how many letters matched in order)

- deletedCount = q.length - matchedLength
- firstIndex and lastIndex in s (window where the matched letters occurred)
- optionally, the **kept subsequence string** (e.g., matched letters)

Then you can compare permutations by:

1. **maximize matchedLength** (equivalently minimize deletions)
2. if tie, **minimize window size**
3. if tie, prefer earliest startPos (stable output)

That keeps your "minimum/maximum window" idea, but now applied to the **best subsequence**, not only full matches.

What changes conceptually in your scan

Instead of deleting from sb until it's empty (full success), you treat sb as "what remains to try" but you never declare total failure just because one letter wasn't found.

Mechanically, resilience looks like:

- continue scanning s even if a particular needed letter never appears
- keep the count of how many you *did* match
- finish the scan and return the best count/window found for that startPos/permutation

You don't need to "exit early" when a letter is missing; you just end with a lower matchedLength.

Two ways to do "subsequence after permutation" (principles, not code)

Option A: Greedy match (simple, fast, very explainable)

For each permutation q and startPos:

- scan through s
- try to match the next needed char in q
- when it matches, advance q pointer, update first/last
- at end, you have matchedLength

This gives a valid "best you can do with this scan rule" and is very transparent for debugging.

Option B: True optimal subsequence inside the window (more work)

Sometimes you might want the absolute best subsequence length for q within s from startPos (or within some region). That's essentially an LCS-style computation between q and s[startPos:].

This is more accurate but heavier. For exploration, greedy is often "good enough" and aligns with your design.

What will blow up (so you can set expectations)

This search space is huge:

- permutations: n! (or n!/repeats!)
- for each permutation, all subsequences is 2^n (you don't want to enumerate)
- but the "resilient scan" approach avoids enumerating subsequences by *computing the best achievable length directly*

So your *practical* version should be:

- **Don't generate subsequences explicitly**
- **Compute the best matched length** during scanning

That still matches your strategy ("remove minimum letters") without exploding.

Minimal behavioral tweak relative to your current mindset

Your code currently treats "not found" as "stop/empty builder".

Your resilient version treats "not found" as:

- "OK, that character remains unmatched; continue scanning to match later characters if possible" (or simply finish and report matched count).

That one mental shift is the heart of resilience.

If you want, I can outline a small set of test cases specifically for this new "permutation + resilient subsequence" behavior (where full match fails, but a strong partial match exists), so you can sanity-check the scoring (matchedLength, deletions, window size).

I need to re-design my code first.... Its a very complicated concept, even this code has taken me lots attempts to get correct Thank you for assistance... I was definitely on right track with my concepts for subsequences... but its tough to keep level head

You're thinking about it the right way. The hard part isn't the "idea" — it's keeping the moving pieces from fighting each other once you add resilience.

If you re-design, a good way to keep a level head (without getting sucked into code details) is to separate the concept into **three clean layers**:

1. **Candidate generator**

- Today: permutations of p
- Later: still permutations, but you might cap how many you generate (or choose a strategy)

2. **Matcher**

- Input: one candidate string (a permutation)
- Output: a "match report" even if it's not perfect:
    - matched length
    - deletions needed (len(candidate) - matched)
    - window indices in s

3. **Scorer / selector**

- Ranks reports:
    - maximize matched length (minimum deletions)
    - then minimize window size
    - then earliest start position (for stability)

That structure lets you change one part without breaking the others — and it keeps your "engine" vibe while staying sane.

When you come back to it, a super steady first milestone is:

- **One permutation only** (don't generate all)
- Get the matcher producing a stable "best partial match" report
- Then re-enable permutations

Anyway, you got your current solution working and you've proven your instincts on subsequences are solid. When you're ready to pick it up again, paste the next draft and we'll keep changes small and controlled like we did with the gate fixes.

so at the moment, I collected minimum and maximum.. I do not think its enough information, since I need to be removing character from STring p in order to compelte subsequence.. And not remove letters from String s. This is exactly where the coding feels mentally confusing

Yeah — the confusion is totally normal here, because "minimum/maximum window" answers a *different* question than "minimum deletions from p".

Right now you record:

- **firstIndex / lastIndex** in s for a *full* match of a permutation
- then you compute min/max **window size in s**

That tells you: *"How tightly can s contain all letters in order?"*

But for subsequences-with-deletions you need a different core measurement:

What you actually need to measure

You want to delete from **p** (not from s) until what remains appears in s.

So the key result is:

Matched length

How many characters from your candidate (permutation of p) were matched in order while scanning s.

Then:

- **deletions from p** = len(p) - matchedLength
- **longest subsequence** = the matched characters (in that order)

Window size becomes optional (a tie-breaker), not the main goal.

Mental model that keeps it clear

Think of it as two pointers:

- pointer i walks through **s**
- pointer j walks through **candidate string** (a permutation of p)

Rules:

- You never delete from s. You just scan it.
- If s[i] == candidate[j], you **keep** that character from p and advance j.
- If they don't match, you only advance i.

At the end:

- j tells you how many characters you successfully kept from p.

This is why it feels confusing in code: you currently "delete from StringBuilder", which feels like removing from p, but you're using it as a *success requirement* (must delete all). For subsequences, you don't require deleting all; you just count how far you got.

What extra information you must store (beyond min/max)

Instead of only storing firstIndex and lastIndex, store a "match report" like:

- matchedLength (most important)
- deletedFromP = p.length - matchedLength
- keptSubsequence (the letters you matched from the permutation)
- firstIndexInS, lastIndexInS (window where those kept letters occurred)
- optional: startPos

Then you can rank permutations by:

1. **max matchedLength** (fewest deletions from p)
2. if tie, **min window size**
3. if tie, earliest startPos

Why min/max window alone can mislead you

Example idea:

- a permutation might match only 3 letters but in a super tiny window
- another permutation might match 7 letters but spread out

Your current min window would prefer the tiny one, but your subsequence goal prefers the 7-letter one.

So: window is about **tightness in s**, deletions are about **how much of p survives**.

A simple way to "upgrade" your thinking without rewriting everything

Keep your current scanning process, but change the success criteria from:

- “sb becomes empty” ✅ full match

to:

- “record best partial progress” ✅ partial match is still valuable

Meaning: even if sb never becomes empty, you still record:

- how many chars were removed (matched)
- which ones they were
- the last matched index

That’s the missing bridge between your current min/max and a subsequence-with-deletions design.

If you want one short guiding sentence to keep your head straight while coding:

“I’m not trying to shrink s. I’m trying to keep as many letters from p as possible while scanning s left-to-right.”

When you’re ready, tell me whether you want:

- “best subsequence for each permutation” (lots of output), or
- “single best permutation overall” (clean summary)

Do i use the maximum result from my current code and perform this operation on it: deletions from p = len(p) - matchedLength longest subsequence = the matched characters (in that order)

Not the **maximum window** result, no.

Your current “maximum” is **maximum window size in s** for a *full match* (or for whatever got stored). That has almost nothing to do with “how many letters from p you can keep”.

What you *can* reuse from your current code is the **same scanning idea**, but you need one extra number that you **don’t currently compute**:

The missing value: matchedLength

- matchedLength = how many characters from the permutation you managed to match in order while scanning s.

Then these formulas are correct:

- **deletions from p** = len(p) - matchedLength
- **longest subsequence** = the matched characters (in that order)

When do those formulas apply?

They apply **per permutation** (and per startPos if you want), based on the **best matchedLength you achieved**.

What about cases where your current code already finds a full match?

If your current code finds a full match for a permutation, then:

- matchedLength = len(p)
- deletions = 0
- longest subsequence is the whole permutation

In those cases, min/max window is still useful as a *tightness* measure, but it's not needed to compute deletions.

What to use instead of "maximum window"

For the subsequence goal, your "best" result should be picked by:

1. **maximum matchedLength** (keep most letters from p)
2. if tie, **minimum deletions** (same thing)
3. if tie, optionally **minimum window size** (tighter match)

So you want "maximum" of **matchedLength**, not maximum of **window size**.

If you tell me what output you want (e.g., "for each permutation print matchedLength + deletions + keptSubsequence"), I can suggest exactly what minimal extra fields to record without changing your overall structure.

ok this makes sense but in order to present the actual subsequence to end user, I need to keep a track of the index (in String p) of where the match occurred. I would then delete indexes of non matches (starting from right hand side to ensure no disruption in offset). I hope this logic is correct

Yes — that logic is correct.

To show the *actual* subsequence (as "letters kept from p"), you need to know **which positions in p were matched**. Then the non-matched positions are exactly what you'd delete (and deleting from right-to-left is the right way to avoid index shifting).

A clean way to think about it:

What you need to track

For a chosen permutation string (call it q), while scanning s in order, maintain:

- matchedIndicesInQ (the indices 0..|q|-1 where you matched)
- optionally matchedIndicesInS (where they were found in s, useful for window)

Then:

- **subsequence string** = q filtered by those matched indices
- **delete list** = all indices of q not in matchedIndicesInQ
- deleting from **right to left** is correct if you're physically deleting characters from a StringBuilder.

Important nuance

Because you're permuting p, there are *two "index spaces"* you could mean:

1. **Index in the permutation string q** (easy and direct)
2. **Index in the original p** (only meaningful if you keep a mapping from q back to original p positions)

If your end-user only needs "the best subsequence for this permutation," then tracking indices in q is enough.

If you want to tell the user "these are the original positions in p that were kept", then you must build the permutation as not just a string, but also as an array of original indices (e.g., permutation of [0,1,2,...]). With repeated letters, this mapping becomes especially important, because there are multiple identical characters.

Why your delete-from-right idea is good

If q = "aab" and you want to delete index 0, doing:

- delete index 0 first works, but if deleting multiple indices, deleting left-to-right causes offsets to shift.
- deleting right-to-left keeps indices stable.

So yes: “collect non-match indices, then delete from right to left” is the safe rule.

Even simpler: you may not need to delete at all

You can present the subsequence without mutation:

- build a StringBuilder kept = new StringBuilder()
- append matched characters as you scan
- output kept.toString()

Deletion is only necessary if you want to literally transform q into the subsequence in-place.

So your logic is solid. The key decision is just whether “index in permutation” is sufficient for your output, or whether you need the mapping back to the original p indices.

ok so I think I have a plan of action... for each permutation, I record the characters matched into StringBuilder array... This would be characters matched. Then surely I would go through each and the longest length are the longest subsequence? Perhaps I have simplified it too much?

That’s a solid plan, and it’s *not* oversimplified — it’s basically the right “first version”.

For each permutation q:

1. Scan s left→right and build a StringBuilder matched by appending whenever you match the next needed char from q.
2. At the end, matched.length() is the **matchedLength** for that permutation.
3. The permutations with the **largest matched.length()** give you the **longest subsequence length** (minimum deletions from p).

That works because you’re not enumerating subsequences — you’re *constructing* one by matching in order.

Two small “gotchas” to be aware of (conceptually, not code advice):

1) Greedy matching gives you *a* longest match for that scan rule

Appending matches as soon as they appear is the normal subsequence check. For “does q exist as a subsequence in s”, it’s perfect.

For “maximum matched length when q cannot fully match”, greedy usually does well, and for a first version it’s the right approach. Just know that if you ever want the provably optimal maximum across all ways of skipping, that becomes an LCS-style problem.

2) You probably want to consider start positions

If you always scan s from index 0, you might miss a longer match that starts later (depending on your scoring goals). Your current code already scans many start positions — you can keep that habit:

- best matched subsequence for permutation q = max over startPos of matched.length()

But you can start with “from 0 only” if you want to reduce complexity for your first prototype.

So yes: **collect matched characters per permutation, pick the maximum length(s)** = longest subsequence (for your permutation-based strategy). That's the cleanest way to keep your head straight while you build it.

ok so its actually a simple concept… but I need to remove logic where it empties stringbuilder when there is no match or insufficient characters in String s I presume

Yes — that's the key shift.

Right now your code treats "can't complete the full permutation" as a **hard failure**, so it does things like:

- **empty sb** to force an exit from the inner loop
- set flags that stop further scanning

For subsequence-with-deletions, you don't want a hard failure. You want:

- "I matched **k** characters so far; keep going and see if I can match more."

So the rule becomes:

Don't empty the builder on "no match"

- If a character isn't found at a certain point, you **don't** delete anything from your target.
- You just keep scanning s to see if later characters match the next needed target letter.

"Insufficient characters left" should end the scan, not wipe progress

When you detect you can't possibly match all remaining letters because you're near the end of s, that should mean:

- "Stop scanning now; the current matchedLength is the best for this startPos."

Not "erase the remaining target and pretend it's a complete success/failure."

Practical mental model

- **sb (target)** stays intact (or you use an index pointer into it)
- **matched builder** grows when you find matches
- at the end, matched.length() is your score

So yes: to add subsequence resilience, you remove/replace the parts that forcibly empty sb on incomplete matches. That's the main conceptual change.

I wonder if that would be enough to prevent empty of stringBuilder and let the code continue to run as it is? I can then just keep a StringBuilder 2D array as discussed above… At each row index I would store the actual substring processed (matched characters) and also the StringBuilder with matched characters… I can then assume that where the difference is minimum, this is where least deletions have occured. And it is the longest subsequence. But I really wanted to keep same code design for both subsequences and also (current minimum/maximum), but it will not be possible due to concept of emptying StringBuilder… I can potentially try to create another method / class to support subsequence

You're thinking about it in the right direction, and yes: **stopping the "empty sb to force failure" behavior is the main thing that lets the code continue running** and produce partial matches.

A few points to keep your logic clean (without getting dragged into lots of code changes):

1) Would "not emptying sb" be enough?

It's close, but the important nuance is: your current loops use **"sb becomes empty"** as the signal to exit inner loops.

So if you simply “don’t empty sb”, you must also ensure you still have a **natural stop condition**, otherwise you risk infinite loops again.

For a subsequence run, the natural stop condition is:

- you reached the end of s (or detected insufficient characters left), so you stop and record whatever you matched.

So: **don’t empty sb**, but do **end the scan** when you hit the end of s (and keep what you matched so far).

2) Your scoring idea is correct

If for each permutation you store:

- matchedString (the characters you successfully matched in order)
  then:
- matchedLength = matchedString.length()
- deletions from p = p.length() - matchedLength

The “least deletions” is equivalent to “largest matchedLength”.

So your statement is correct, just rephrase it precisely:

- **Max matchedLength ⇒ longest subsequence ⇒ minimum deletions from p**

(Using “difference is minimum” = p.length - matchedLength is fine too.)

3) Storing both “substring processed” and “matched characters”

That’s useful:

- the window substring from s helps explain *where* the match came from
- the matched characters shows *what survived from p*

Just be careful with wording:

- the substring of s is not the subsequence; it’s the **window**
- the matched builder is the **subsequence**

4) Keeping one design for both features

You can keep a shared design if you treat “full match” as a special case of subsequence:

- In subsequence mode, you always compute matchedLength
- In current mode (full match), you only store/report when matchedLength == p.length()

That’s the unifying principle.

Why your current “empty sb” conflicts

Because “empty sb” is being used as a *control-flow hack* (“stop now”) rather than as a *real meaning* (“we matched everything”). For subsequences, you need the meaning, not the hack.

So yes: either you refactor that small part, or you keep a separate method.

5) Separate method/class is a very sensible plan

Given you want to preserve your working minimum/maximum implementation, the safest approach is:

- Keep your existing method as-is for full matches.
- Add a new method (or class) like:
    - findBestSubsequencePerPermutation(s, p) or similar that returns “match report” rows.

Then you can still share:

- permutation generation
- printing style
- window index tracking

...but avoid mixing two different stop rules in one nested-loop structure.

If you want a simple “contract” for your new subsequence method, it would output per permutation:

- permutation
- best matchedLength
- deletions = p.length - matchedLength
- matchedSubsequence
- window (firstIndex-lastIndex)

That gives you exactly what you need to rank permutations by “least deletions / longest subsequence”, while keeping your existing min/max window method intact.

yes it makes sense, this is exactly the situation I found myself in the design... Realising the concept of emptying StringBuilder has different role in both codes

Exactly. In your current code, “emptying the StringBuilder” is doing **two jobs at once**:

1. **Meaning job (logical):** “we matched everything”
2. **Control-flow job (mechanical):** “get me out of nested loops / stop this attempt”

For *minimum/maximum window of a full match*, those two jobs line up nicely.

For *subsequence with deletions*, they don’t:

- you can have a good partial match where the builder should **not** become empty
- but you still need a clean way to **stop** scanning and report progress

So your design instinct is right: keep them as two methods (or share a common matcher but with a mode flag) so each one has a clean stop rule.

A simple way to keep your head straight when you implement the subsequence method:

- **Empty builder = success** (full match only)
- **End of s = stop and score** (subsequence mode)

And the score you report is:

- matchedLength (maximize it)

- deletions = |p| - matchedLength (minimize it)

When you draft the subsequence method, even if it's rough, paste it here and we'll keep changes tight and incremental like we did with the off-by-one fixes.

I think it makes sense, I think in order to avoid confusion, it makes sense just to modify my minimum/maximum code for subsequences... In future I can try to think of a multi-class design to support both (since this will be quite complicated).. But what would be benefits (in terms of real life use) if I reported ALL information in a multi-class design: " is a substring containing characters matching permutation: " + valuesSet[entry] (this would suit the current design and also subsequence since no deletions) " is not a substring containing characters matching permutation: " + valuesSet[entry] (terminating when no matched character, current design) " is not a substring containing characters matching permutation: " + valuesSet[entry] (terminating at end of String s, subsequence design, I will also show the matched StringBuilder, the actual substring and number unmatched characters.....) Will it be information overload presenting all this in a single solution or would it to be of benefit? Or is this valuable information?

It *can* be valuable, but only if you present it with a clear "levels" structure. If you dump all three message types for every permutation/startPos, it will become information overload fast.

When it's valuable

A multi-class / multi-output design is genuinely useful in real life when you're using this as a **diagnostic engine** rather than a single-answer algorithm. Examples:

- **Search / fuzzy matching**: "How close did we get?" (subsequence partial match + deletions) is exactly what autocomplete, typo-tolerant search, and approximate keyword match need.
- **Ranking / scoring**: If you're comparing many candidate orders (permutations), you need *why* one scored better (matched length, window, unmatched count).
- **Explainability**: In debugging, data quality checks, or compliance/audit contexts, being able to show "here's the evidence in s" matters.
- **Profiling string patterns**: In DNA/protein-style sequence scanning, log parsing, event chain detection, or "workflow appears in logs in this order", partial matches and where they break is useful.

When it becomes overload

It becomes overload if you show all outputs at once without aggregation. The biggest overload drivers are:

- printing per **startPos** for every permutation
- printing "not a substring..." for every miss (misses are far more common than hits)
- printing full substring windows repeatedly

A good compromise: keep *all* info internally, print only the right level

You can store everything (so it's available), but print in tiers:

Tier 1 (default, end-user friendly)

- Top N results only (e.g., best subsequence per permutation, or best overall)
- Include: permutation, matchedLength, deletions, window indices, matched subsequence

Tier 2 (debug / verbose mode)

- Show "why failed" summary categories:
    - "no character from permutation exists in s"
    - "some characters exist but order prevents full match"

    - “partial match length = k, stopped at end of s”
- Still not every startPos line — just counts + best examples.

Tier 3 (deep audit)

- Full trace per permutation/startPos with the exact messages you listed.

So: **not overload if it’s optional**, **overload if it’s always-on**.

About your three message types

A) "is a substring containing characters matching permutation" (full match)

High value, keep it. It’s your core “success”.

B) "is not a substring..." terminating when no matched character (current design)

This is useful **only as a classification**, not as a line-per-attempt output.
Better as:

- “0 letters matched anywhere” (count) + one example

C) "is not a substring..." terminating at end of s + show matched builder + unmatched count (subsequence design)

This is valuable because it’s basically your “fuzzy match report”.
But again: best shown as:

- “best partial match for this permutation” (one line), not every failed startPos.

Real-life benefit of reporting “not found” vs “partial”

- **Not found at all**: tells you the candidate is irrelevant (wrong alphabet/content) → you can prune it.
- **Partial match**: tells you the candidate is *similar* but incomplete → good for ranking and suggestions.
- **Full match**: tells you it’s a valid ordered occurrence → good for exact detection.

That’s meaningful information — it’s not fluff — but you need **summaries** to keep it usable.

My recommendation (conceptually)

Keep your multi-class design idea, but add a simple “output mode” concept:

- **NORMAL**: print only full matches + final min/max + best subsequence summary
- **VERBOSE**: print best partial per permutation + reasons
- **TRACE**: print everything

That way you retain maximum usefulness without drowning the user.

If you tell me who the “end user” is (you, a developer, or a non-technical user), I can suggest the best default tier and what fields to include in the one-line summaries.