

```
//----CHATGPT----
System.out.println("\n*****");
System.out.println("\nNOTE: END REPORT SHOWS WINDOW RESULTS (PREFIX LONGEST SUBSEQUENCE MODE)");
System.out.println("Window = the substring in X from the FIRST matched index to the LAST matched index for the current permutation of Y.");
System.out.println("PREFIX mode = matching starts from the beginning of the permutation of Y (no leading deletions inside Y).");
System.out.println("*****\n");
//----END CHATGPT----
```

PROVIDED CODE

<https://www.amitamlani.com/03022026/3/Main.java>

I realised for clarity, we needed a note in this code similar to attached... So that we know it supports frontal deletions...

Yes, I think adding that note is a very good idea.

Looking at your current algorithm, it is **not a general subsequence matcher**. It supports:

- Removing characters from the **front/middle/end of String Y** when they cannot be matched.
- Matching proceeds from the current first remaining character of Y.
- The reported subsequence is effectively a **prefix-style subsequence of the current permutation after deletions**.

Without a note, somebody reading the end report could wrongly assume:

Subsequence: ba

means the algorithm searched for any arbitrary subsequence inside the permutation.

Your note could be even more explicit:

```
System.out.println("\nNOTE: END REPORT SHOWS WINDOW RESULTS (PREFIX LONGEST
SUBSEQUENCE MODE)");
System.out.println("Window = substring in X from first matched index to
last matched index.");
System.out.println("Matching always starts from the first remaining
character of permutation Y.");
System.out.println("Characters may be removed from Y when no match
exists.");
System.out.println("Therefore the reported subsequence represents the
longest surviving FRONT-ALIGNED");
System.out.println("portion of permutation Y after deletions, not an
arbitrary subsequence of Y.");
```

The key phrase is:

longest surviving FRONT-ALIGNED portion of permutation Y after deletions

because that explains exactly why:

Y = bca
X = a

can end up producing:

Subsequence: a

The algorithm deleted b, deleted c, then matched a.

That behaviour is perfectly valid for your design, but it is not obvious from the output alone. The note removes that ambiguity.

thanks, its well written.. I will add this in...

You're welcome.

I actually think this is one of those cases where a note is more important than additional code changes. When someone revisits the project months later, they will immediately understand the intended behaviour rather than spending hours trying to determine whether:

Y = bca
X = a

producing

Subsequence: a
Length: 1

is a bug or a feature.

The note documents the design philosophy:

- Matching starts at the first remaining character of Y.
- Characters in Y can be discarded when no match exists.
- The reported subsequence is the surviving front-aligned portion after deletions.
- The end report is therefore not a generic longest-common-subsequence implementation.

That's exactly the sort of clarification that future-you will appreciate when looking back at the code. 👍

ok my only concern with the code is that: I have commented out your block of code:

```
ok my only concern with the code is that:

I have commented out your block of code:
//----CHATGPT----

//=====
//=====
// WHY THIS CHANGE WAS MADE (simple explanation)
//
// Your original code worked when every row had a valid
numeric subsequence length.
// But in your design, some rows can legitimately be:
// - null (no data stored for that row slot), or
// - blank (you stored the row but the length field ended up
empty for a failed attempt).
//
// In those cases, this original line can crash the program:
// Integer.valueOf(subsequences[c][3].toString())
//
// because it can throw:
// - NullPointerException (if subsequences[c][3] is null)
// - NumberFormatException (if it's empty or non-numeric)
//
// So I replaced it with a "safe max update":
// 1) Check the length field is not null
// 2) Trim it and ensure it is not empty
// 3) Parse it only when it's valid
//
// This keeps your output logic the same (still finds the true
maximum),
// but prevents rare/edge rows from breaking the whole run.
// The original code is kept below in a /* ... */ block for
reference.

//=====
//=====

/*
// Safe max update: some rows may have null/blank lengths;
skip those.
if (subsequences[c][3] != null)
{
String _lenStr = subsequences[c][3].toString().trim();
if (!_lenStr.isEmpty())
{
int _len = Integer.parseInt(_lenStr);

if (_len > maximumSubsequenceLength)
{
maximumSubsequenceLength = _len;
}
}
}
*/
//----END CHATGPT----
```

I think I would need a workaround as below instead... I used this technique in Longest common subsequence code.. What are your thoughts

```
//using s to guard against StringIndexOutOfBoundsException
if ((Integer.valueOf(subsequences[c]
[3].toString())==maximumSubsequenceLength) && !s.equals(""))
```

Show less ^

Your workaround is good, but it protects against a **different problem**.

This:

```
&& !s.equals("")
```

protects against `StringIndexOutOfBoundsException` later when you do substring/window printing.

But it does **not** protect this line itself:

```
Integer.valueOf(subsequences[c][3].toString())
```

because that can still fail if `subsequences[c][3]` is `null`, blank, or non-numeric.

Best compromise:

```
if (!s.equals("") && subsequences[c][3] != null &&
!subsequences[c][3].toString().trim().equals(""))
{
    if (Integer.valueOf(subsequences[c][3].toString()) ==
maximumSubsequenceLength)
    {
        // print longest subsequence row
    }
}
```

So your instinct is right: keep the `!s.equals("")` guard, but use it **in addition** to the numeric guard, not instead of it.

Your LCS-style guard is valid for substring safety; my guard was for stored-length parsing safety. They solve two separate edge cases.

PROVIDED CODE (Longest subsequence)...

<https://www.amitamlani.com/03022026/2/1/Main.java>
so I think my instinct was probably also wrong for LCS...

Not completely wrong — just **too narrow**.

Your `!s.equals("")` guard only says:

```
String s has content, so substring may be safer.
```

But it does **not** prove this is safe:

```
Integer.valueOf(subsequences[c][3].toString())
```

because `subsequences[c][3]` could still be null/blank/non-numeric.

So in LCS too, the stronger lesson is:

```
// Guard the thing you are about to use
```

For substring:

```
if (!s.equals(""))
```

For stored numeric data:

```
if (subsequences[c][3] != null &&  
    !subsequences[c][3].toString().trim().equals(""))
```

Your instinct was good for avoiding `StringIndexOutOfBoundsException`, but not sufficient for avoiding `NullPointerException` or `NumberFormatException`.

Best readable compromise:

```
if (!s.equals("")  
    && subsequences[c][3] != null  
    && !subsequences[c][3].toString().trim().equals("")  
    && Integer.valueOf(subsequences[c][3].toString()) ==  
    maximumSubsequenceLength)  
{  
    // print result  
}
```

Your comment in the file says the `s.equals("")` guard allowed you to discard the safer numeric block, but that conclusion is the risky part.

Yes, it **could** have included `!s.equals("")`, but I deliberately separated the concerns.

My block was only for:

```
maximumSubsequenceLength
```

So it guards:

```
subsequences[c][3]
```

Your `!s.equals("")` guard belongs where you do:

```
s.substring(...)
```

You *can* combine them like this:

```

if (!s.equals("") && subsequences[c][3] != null)
{
    String lenStr = subsequences[c][3].toString().trim();

    if (!lenStr.equals(""))
    {
        int len = Integer.parseInt(lenStr);

        if (len > maximumSubsequenceLength)
        {
            maximumSubsequenceLength = len;
        }
    }
}

```

But I slightly prefer **not** doing that, because even when s is empty, calculating maximum subsequence length as 0 is still logically safe.

Best structure:

```

// safe length calculation
// no substring protection needed here

```

then:

```

// safe output printing
// substring protection needed here

```

could you give me examples of edge cases that would fail if I chose to keep your code block commented out... For both the LCS and also front truncation code

Yes. These are the kinds of cases that can fail if the safe block stays commented out.

1. Front-truncation code

Your current risky line is:

```
Integer.valueOf(subsequences[c][3].toString())
```

It can fail when a stored row has no clean numeric length.

Examples:

```
String p = "ab";
String s = "";
```

Risk: no real scan happens, but rows/arrays may still be touched. subsequences[c][3] can be blank or not properly filled.

```
String p = "ab";  
String s = "cc";
```

Risk: no characters match. Some rows can represent “attempted but no useful subsequence”.
If length storage is blank/null, parsing crashes.

```
String p = "abcd";  
String s = "ab";
```

Risk: partial scan, incomplete result. If one row is created but length is not cleanly stored,
parsing fails.

Likely errors:

```
NullPointerException  
NumberFormatException
```

Your `!s.equals("")` does not protect against those, because `s` can be non-empty while
`subsequences[c][3]` is still unsafe.

2. LCS code

Same principle. These can expose the issue:

```
String a = "";  
String b = "abc";
```

```
String a = "abc";  
String b = "";
```

```
String a = "abc";  
String b = "xyz";
```

```
String a = "abc";  
String b = null; // if your code ever allows this
```

The important point: `!s.equals("")` only protects string access. It does not prove that the stored
LCS length cell is valid.

So yes, for both codes, I would reinstate the safe numeric block and keep `!s.equals("")` only
around substring/window printing.

So it seems that your block to protect against exceptions and also `!s.equals("")` are both
required

Yes, that's the conclusion I would reach.

They protect **different failure points**:

1. My safe block

Protects:

```
Integer.valueOf(subsequences[c][3].toString())
```

from:

```
NullPointerException  
NumberFormatException
```

Examples:

```
subsequences[c][3] == null
```

or

```
subsequences[c][3].toString().equals("")
```

2. Your `!s.equals("")` guard

Protects:

```
s.substring(...)
```

from:

```
StringIndexOutOfBoundsException
```

or invalid window calculations when:

```
String s = "";
```

My recommendation

Keep both:

```
// Safe maximum subsequence calculation  
if (subsequences[c][3] != null)  
{  
    String lenStr = subsequences[c][3].toString().trim();  
  
    if (!lenStr.equals(""))  
    {  
        int len = Integer.parseInt(lenStr);  
  
        if (len > maximumSubsequenceLength)  
        {  
            maximumSubsequenceLength = len;  
        }  
    }  
}
```

and later:

```
if (!s.equals(""))
    && Integer.valueOf(subsequences[c][3].toString()) == maximumSubsequenceLength)
{
    ...
}
```

In fact, after reviewing the evolution of your code, I would say your `!s.equals("")` guard was a good addition, but it doesn't replace the safe parsing block. The two protections complement each other rather than overlap.

thanks you for assistance, it is actually a very nasty coding challenge in my opinion

You're welcome. And honestly, I agree with you.

This challenge is nasty because it combines **several difficult problems at once**:

1. **Permutation generation**
 - Repeated characters.
 - Duplicate elimination.
 - Large search spaces.
2. **Substring vs subsequence**
 - Very easy to accidentally mix the concepts.
 - Window size and subsequence length are different metrics.
3. **Front truncation / resilience**
 - Once you allow characters to be dropped from String Y, the definition of a "match" becomes much less obvious.
 - Cases like:

Y = "bca"

X = "a"

become philosophical as much as technical:

"Is 'a' a valid subsequence result or not?"

4. **Window bookkeeping**
 - firstIndexLocationStringS
 - lastIndexLocationStringS
 - window size
 - subsequence length

These can all be correct individually while the overall report is wrong.

5. **Reporting layer**
 - Most coding challenges stop after finding the answer.
 - Your code also stores evidence, windows, unmatched characters, subsequence indices, permutations, etc.
 - That's where many of the subtle bugs appeared.
6. **Edge cases**
 - Empty string.
 - Single-character strings.
 - No matches.
 - Partial matches.
 - Repeated letters.
 - Exact-fit windows.
 - End-of-string matches.

A lot of developers could write a solution that passes a few examples. Far fewer would keep extending it until it explains *why* a subsequence was found and prints all the supporting evidence.

The fact that we ended up discussing things like:

```
!s.equals("")
```

versus

```
subsequences[c][3] != null
```

shows the project has moved beyond simply "finding the answer" into making the implementation robust.

So yes, I think "nasty coding challenge" is a fair description. It's the kind of problem where most of the effort ends up in the edge cases and diagnostics rather than the core algorithm itself.