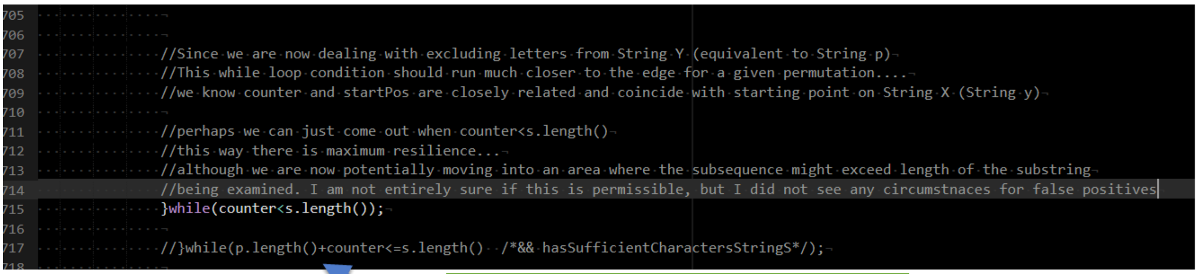
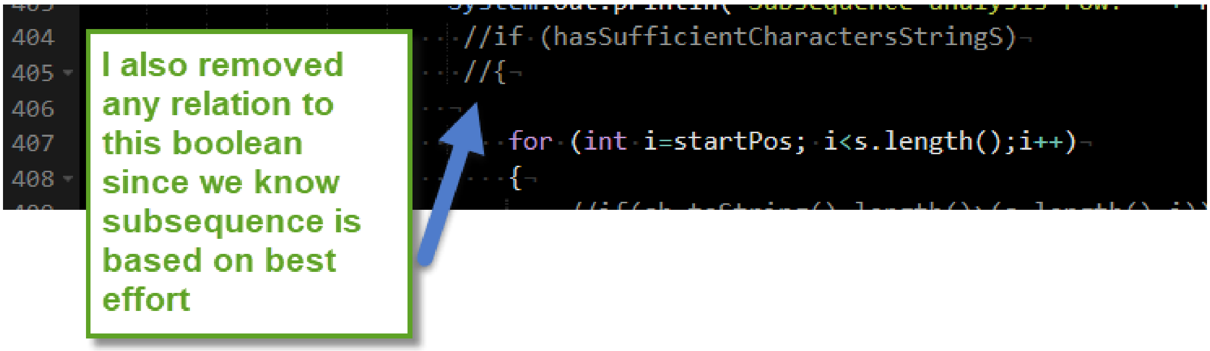
I have continued coding until it has reached an error free state. I did notice that since I was heavily confused on termination point for the do while.



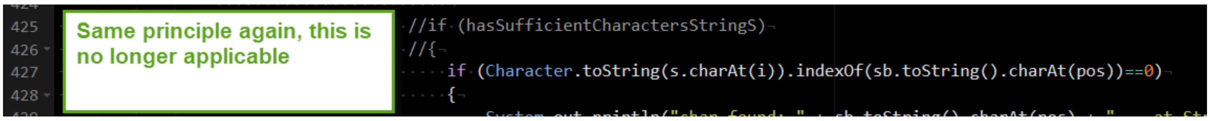
```
                //Since we are now dealing with excluding letters from String Y (equivalent to String p)
                //This while loop condition should run much closer to the edge for a given permutation....
                //we know counter and startPos are closely related and coincide with starting point on String X (String y)

                //perhaps we can just come out when counter<s.length()
                //this way there is maximum resilience...
                //although we are now potentially moving into an area where the subsequence might exceed length of the substring
                //being examined. I am not entirely sure if this is permissible, but I did not see any circumstnaces for false positives
                }while(counter<s.length());

                //}while(p.length()+counter<=s.length()  /*&& hasSufficientCharactersStringS*/);
```

Strictly speaking we know that in can never reach the last character in String p (equivalent to String y) with condition active..
This while loop would cut out lots executions where String x is shorter than String y



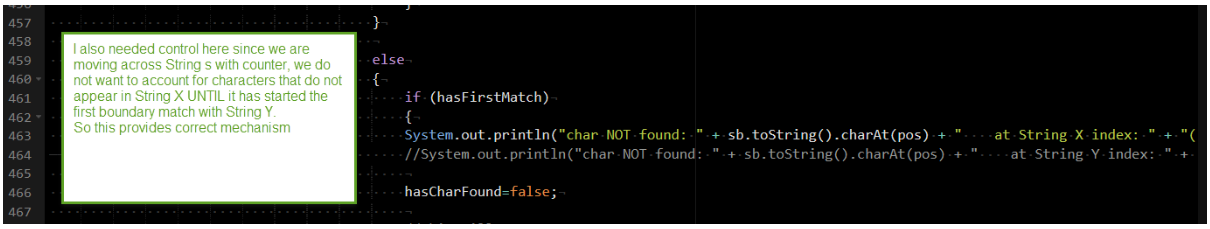
I also removed any relation to this boolean since we know subsequence is based on best effort

```
  //if (hasSufficientCharactersStringS)
  //{

    for (int i=startPos; i<s.length();i++)
    {
```



Same principle again, this is no longer applicable

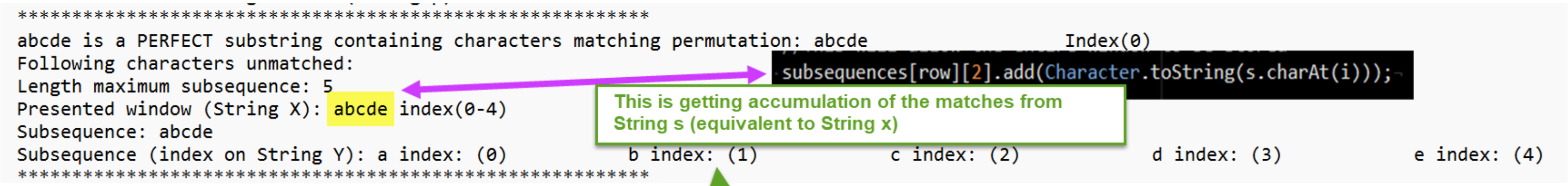
```
//if (hasSufficientCharactersStringS)
//{
    if (Character.toString(s.charAt(i)).indexOf(sb.toString().charAt(pos))==0)
    {
```



I also needed control here since we are moving across String s with counter, we do not want to account for characters that do not appear in String X UNTIL it has started the first boundary match with String Y.
So this provides correct mechanism

```
        }

        else
        {
            if (hasFirstMatch)
            {
            System.out.println("char NOT found: " + sb.toString().charAt(pos) + "    at String X index: " + "(
            //System.out.println("char NOT found: " + sb.toString().charAt(pos) + "    at String Y index: " +

            hasCharFound=false;
```

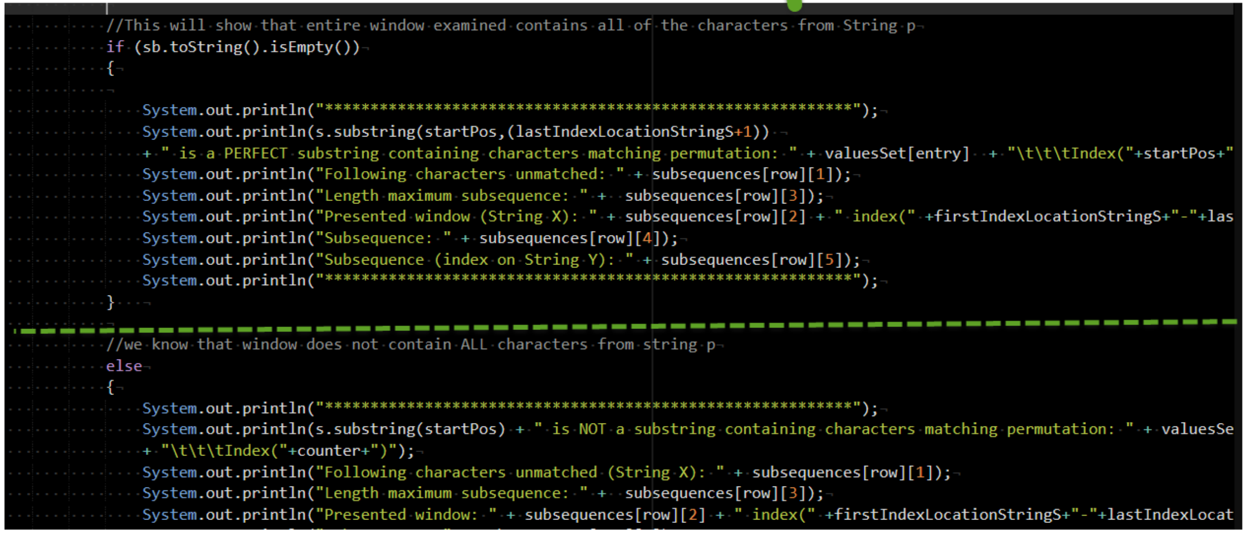
I adapted a similar style of code to previous exercise, however I had to remove all of the booleans from inside the if condition... We can see as below, it shows a perfect substring for the permutation....



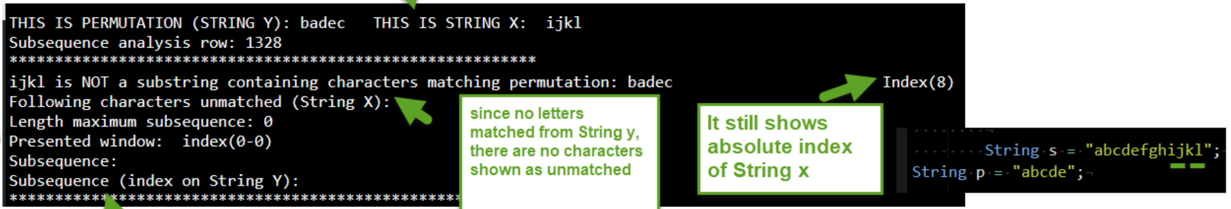
```
*************************************************************
abcde is a PERFECT substring containing characters matching permutation: abcde                  Index(0)
Following characters unmatched:
Length maximum subsequence: 5
Presented window (String X): abcde index(0-4)
Subsequence: abcde
Subsequence (index on String Y): a index: (0)           b index: (1)            c index: (2)            d index: (3)            e index: (4)
*************************************************************
```

```
subsequences[row][2].add(Character.toString(s.charAt(i)));
```

This is getting accumulation of the matches from String s (equivalent to String x)



```
        //This will show that entire window examined contains all of the characters from String p
        if (sb.toString().isEmpty())
        {

            System.out.println("*************************************************************");
            System.out.println(s.substring(startPos,(lastIndexLocationStringS+1))
            + " is a PERFECT substring containing characters matching permutation: " + valuesSet[entry]  + "\t\t\tIndex("+startPos+"
            System.out.println("Following characters unmatched: " + subsequences[row][1]);
            System.out.println("Length maximum subsequence: " +  subsequences[row][3]);
            System.out.println("Presented window (String X): " + subsequences[row][2] + " index(" +firstIndexLocationStringS+"-"+las
            System.out.println("Subsequence: " + subsequences[row][4]);
            System.out.println("Subsequence (index on String Y): " + subsequences[row][5]);
            System.out.println("*************************************************************");
        }

        //we know that window does not contain ALL characters from string p
        else
        {
            System.out.println("*************************************************************");
            System.out.println(s.substring(startPos) + " is NOT a substring containing characters matching permutation: " + valuesSe
            + "\t\t\tIndex("+counter+")");
            System.out.println("Following characters unmatched (String X): " + subsequences[row][1]);
            System.out.println("Length maximum subsequence: " +  subsequences[row][3]);
            System.out.println("Presented window: " + subsequences[row][2] + " index(" +firstIndexLocationStringS+"-"+lastIndexLocat
```



```
THIS IS PERMUTATION (STRING Y): badec   THIS IS STRING X:  ijkl
Subsequence analysis row: 1328
*************************************************************
ijkl is NOT a substring containing characters matching permutation: badec                  Index(8)
Following characters unmatched (String X):
Length maximum subsequence: 0
Presented window:  index(0-0)
Subsequence:
Subsequence (index on String Y):
*************************************************************
```

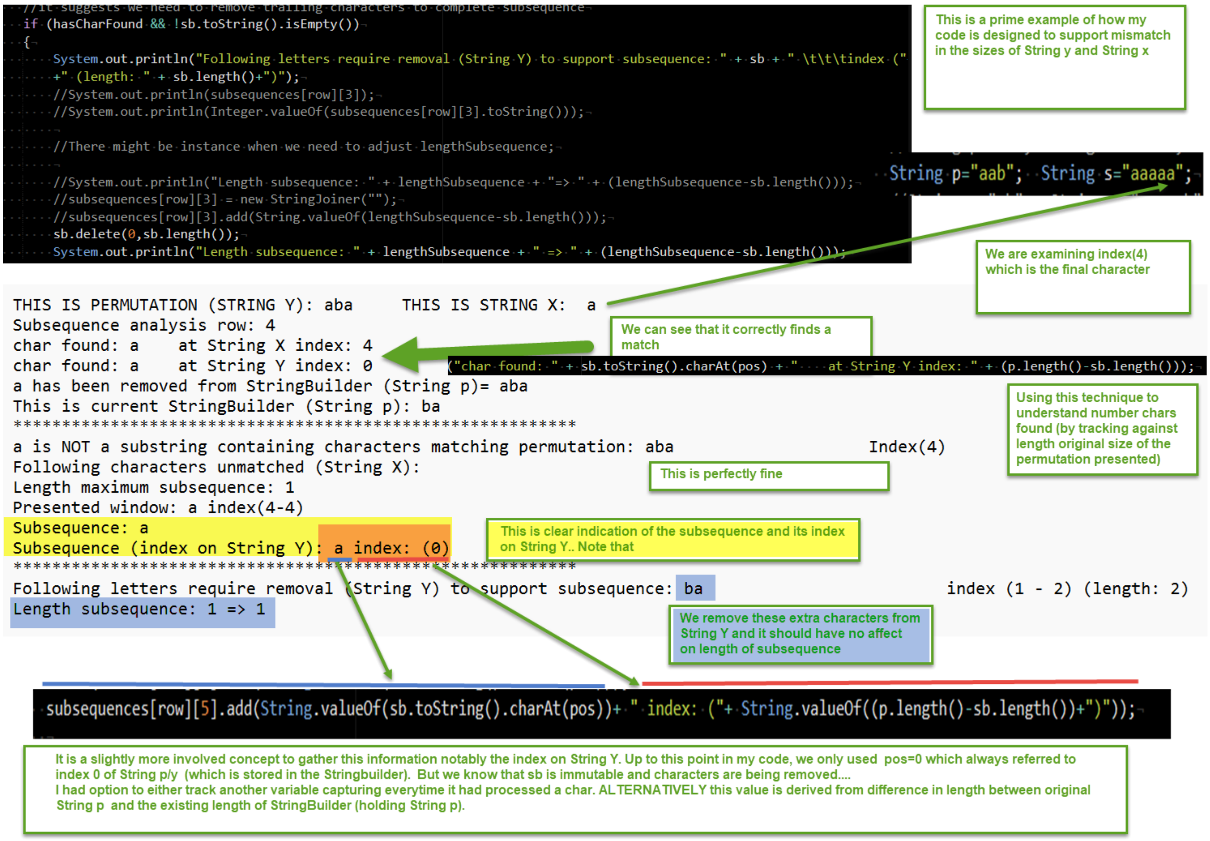
```
        String s = "abcdefghijkl";
String p = "abcde";
```

The window has no length since there is no start of end index found in String s

since no letters matched from String y, there are no characters shown as unmatched

It still shows absolute index of String x

As expected no subsequence



```
//It suggests we need to remove trailing characters to complete subsequence
if (hasCharFound && !sb.toString().isEmpty())
{
    System.out.println("Following letters require removal (String Y) to support subsequence: " + sb + " \t\t\tindex ("
    +" (length: " + sb.length()+")");
    //System.out.println(subsequences[row][3]);
    //System.out.println(Integer.valueOf(subsequences[row][3].toString()));

    //There might be instance when we need to adjust lengthSubsequence;

    //System.out.println("Length subsequence: " + lengthSubsequence + "=> " + (lengthSubsequence-sb.length()));
    //subsequences[row][3] = new StringJoiner("");
    //subsequences[row][3].add(String.valueOf(lengthSubsequence-sb.length()));
    sb.delete(0,sb.length());
    System.out.println("Length subsequence: " + lengthSubsequence + " => " + (lengthSubsequence-sb.length()));
```

This is a prime example of how my code is designed to support mismatch in the sizes of String y and String x

```
String p="aab";  String s="aaaaa";
```

We are examining index(4) which is the final character

```
THIS IS PERMUTATION (STRING Y): aba     THIS IS STRING X:  a
Subsequence analysis row: 4
char found: a    at String X index: 4
char found: a    at String Y index: 0
a has been removed from StringBuilder (String p)= aba
This is current StringBuilder (String p): ba
**********************************************************
a is NOT a substring containing characters matching permutation: aba                Index(4)
Following characters unmatched (String X):
Length maximum subsequence: 1
Presented window: a index(4-4)
Subsequence: a
Subsequence (index on String Y): a index: (0)
**********************************************************
Following letters require removal (String Y) to support subsequence: ba            index (1 - 2) (length: 2)
Length subsequence: 1 => 1
```

We can see that it correctly finds a match

```
("char found: " + sb.toString().charAt(pos) + "    at String Y index: " + (p.length()-sb.length()));
```

Using this technique to understand number chars found (by tracking against length original size of the permutation presented)

This is perfectly fine

This is clear indication of the subsequence and its index on String Y.. Note that

We remove these extra characters from String Y and it should have no affect on length of subsequence

```
subsequences[row][5].add(String.valueOf(sb.toString().charAt(pos))+ " index: ("+ String.valueOf((p.length()-sb.length())+")"));
```

It is a slightly more involved concept to gather this information notably the index on String Y. Up to this point in my code, we only used pos=0 which always referred to index 0 of String p/y (which is stored in the Stringbuilder). But we know that sb is immutable and characters are being removed....
I had option to either track another variable capturing everytime it had processed a char. ALTERNATIVELY this value is derived from difference in length between original String p and the existing length of StringBuilder (holding String p).

## CHATGPT CONVERSATION

in subsequences, can string p be shorter than string s

Yes — and that's the normal case.

**Subsequence basics**

A string p is a **subsequence** of s if you can delete **some characters from s** (without reordering the remaining ones) and get p.

So for p to be a subsequence of s, you must have:

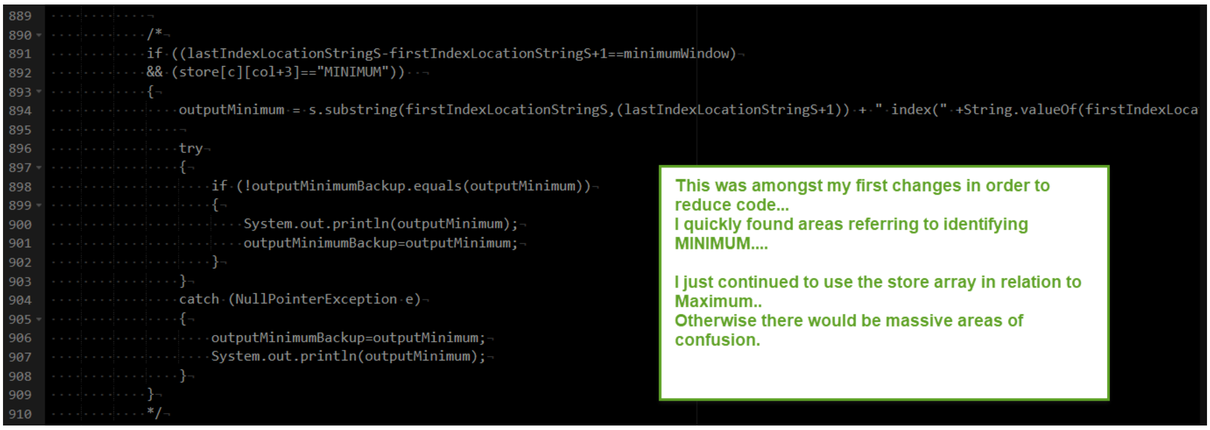
- len(p) ≤ len(s)

If len(p) > len(s), then p **cannot** be a subsequence of s.

**Examples**

- p="ace" is a subsequence of s="abcde" (take a,c,e).
- p="abc" is **not** a subsequence of s="acb" (order doesn't work).
- p="abcd" cannot be a subsequence of s="abc" (too long).

If you meant your *other* setup (delete from **p** to make it appear in **s**), then even if p is longer than s, you can still delete from p until the remaining subsequence fits inside s.



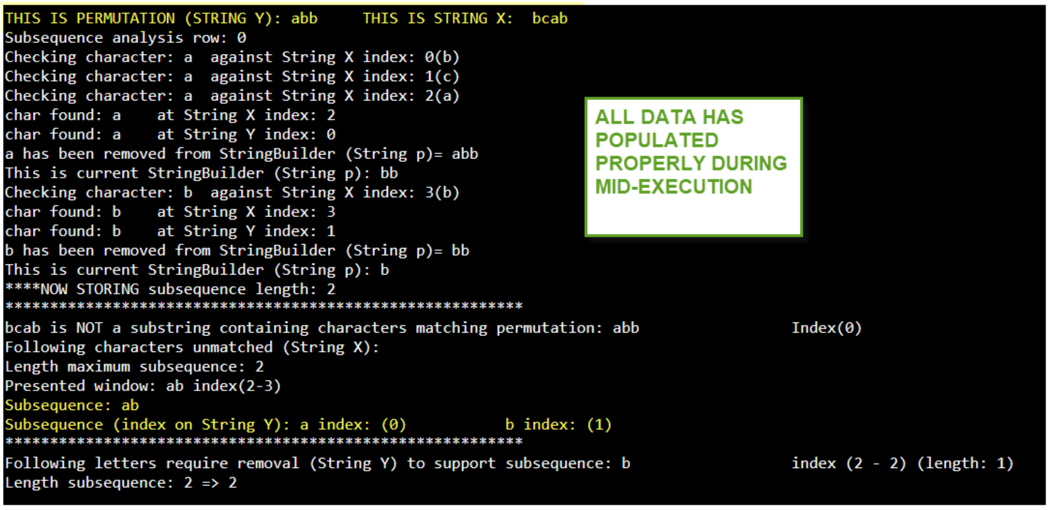
```
            
            /*
            if ((lastIndexLocationStringS-firstIndexLocationStringS+1==minimumWindow)
            && (store[c][col+3]=="MINIMUM"))  
            {
                outputMinimum = s.substring(firstIndexLocationStringS,(lastIndexLocationStringS+1)) + " index(" +String.valueOf(firstIndexLoca
                
                try
                {
                    if (!outputMinimumBackup.equals(outputMinimum))
                    {
                        System.out.println(outputMinimum);
                        outputMinimumBackup=outputMinimum;
                    }
                }
                catch (NullPointerException e)
                {
                    outputMinimumBackup=outputMinimum;
                    System.out.println(outputMinimum);
                }
            }
            */
```

This was amongst my first changes in order to reduce code...
I quickly found areas referring to identifying MINIMUM....

I just continued to use the store array in relation to Maximum..
Otherwise there would be massive areas of confusion.

With all this extreme hard effort, I was at a point where the single test case was failing...
I particularly found this a scenario which would be extremely difficult to isolate.
It is reminiscent of level of issues I encountered on several challenges in which there was a mismatch between mid-execution and summary at the end.
Here is the exact test case:

## TEST CASE: Mismatch in output mid-execution and summary



```
THIS IS PERMUTATION (STRING Y): abb     THIS IS STRING X:  bcab
Subsequence analysis row: 0
Checking character: a  against String X index: 0(b)
Checking character: a  against String X index: 1(c)
Checking character: a  against String X index: 2(a)
char found: a    at String X index: 2
char found: a    at String Y index: 0
a has been removed from StringBuilder (String p)= abb
This is current StringBuilder (String p): bb
Checking character: b  against String X index: 3(b)
char found: b    at String X index: 3
char found: b    at String Y index: 1
b has been removed from StringBuilder (String p)= bb
This is current StringBuilder (String p): b
****NOW STORING subsequence length: 2
**********************************************************
bcab is NOT a substring containing characters matching permutation: abb              Index(0)
Following characters unmatched (String X):
Length maximum subsequence: 2
Presented window: ab index(2-3)
Subsequence: ab
Subsequence (index on String Y): a index: (0)           b index: (1)
**********************************************************
Following letters require removal (String Y) to support subsequence: b              index (2 - 2) (length: 1)
Length subsequence: 2 => 2
```

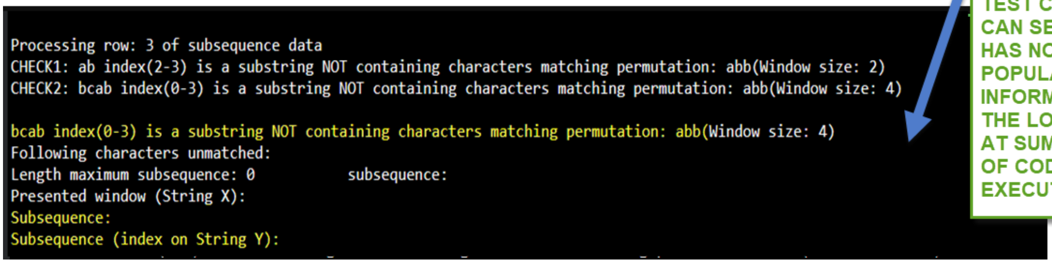
ALL DATA HAS POPULATED PROPERLY DURING MID-EXECUTION

2

But if we examine the data during the code execution, we can see that there are no issues. This is almost close to an impossible issue to solve at human capacity since it is so difficult to calculate where the branching would differ with other test cases

1

ONLY FOR THIS TEST CASE, WE CAN SEE THAT IT HAS NOT POPULATED INFORMATION IN THE LOWER PART AT SUMMARY PART OF CODE EXECUTION

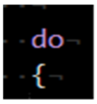


```
Processing row: 3 of subsequence data
CHECK1: ab index(2-3) is a substring NOT containing characters matching permutation: abb(Window size: 2)
CHECK2: bcab index(0-3) is a substring NOT containing characters matching permutation: abb(Window size: 4)

bcab index(0-3) is a substring NOT containing characters matching permutation: abb(Window size: 4)
Following characters unmatched:
Length maximum subsequence: 0          subsequence:
Presented window (String X):
Subsequence:
Subsequence (index on String Y):
```

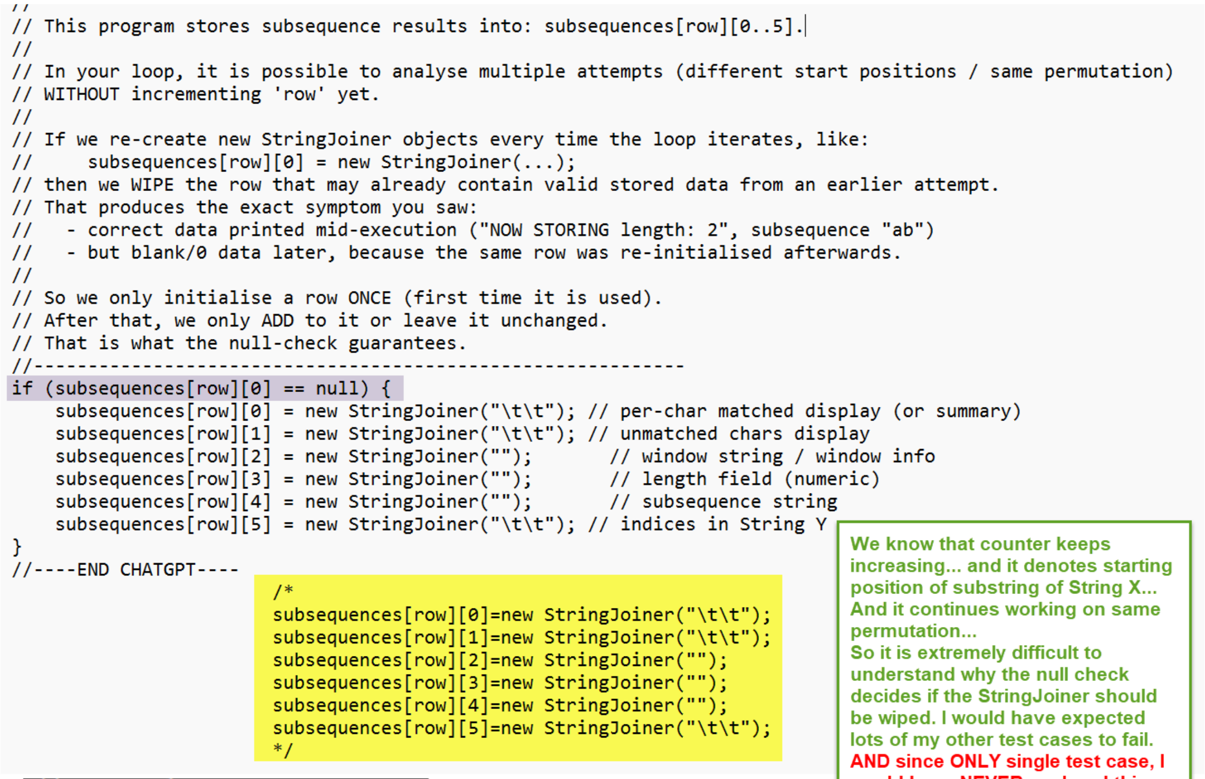
I have let ChatGPT examine my code for the first time.
Once again, one or two were human errors which I might have noticed if I tidied my code.
But there were several changes....
Note, some modifications with the usual ------------chatGPT-------------
------------end ChatGPT---------
are the modifications from the inclusion of String p in String s

I will go through each of the inputs by ChatGPT to demonstrate my understanding. As per aforementioned, it is logic which does not resonate with me.

## 1) CHATGPT CHANGE **(I consider this as highly technical, it refers to my immediate issue above)**

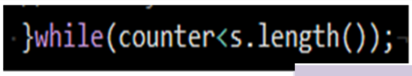


```
do
{
```



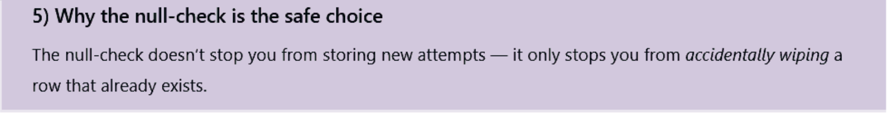
```
//
// This program stores subsequence results into: subsequences[row][0..5].
//
// In your loop, it is possible to analyse multiple attempts (different start positions / same permutation)
// WITHOUT incrementing 'row' yet.
//
// If we re-create new StringJoiner objects every time the loop iterates, like:
//     subsequences[row][0] = new StringJoiner(...);
// then we WIPE the row that may already contain valid stored data from an earlier attempt.
// That produces the exact symptom you saw:
//   - correct data printed mid-execution ("NOW STORING length: 2", subsequence "ab")
//   - but blank/0 data later, because the same row was re-initialised afterwards.
//
// So we only initialise a row ONCE (first time it is used).
// After that, we only ADD to it or leave it unchanged.
// That is what the null-check guarantees.
//-------------------------------------------------------------
if (subsequences[row][0] == null) {
    subsequences[row][0] = new StringJoiner("\t\t"); // per-char matched display (or summary)
    subsequences[row][1] = new StringJoiner("\t\t"); // unmatched chars display
    subsequences[row][2] = new StringJoiner("");       // window string / window info
    subsequences[row][3] = new StringJoiner("");       // length field (numeric)
    subsequences[row][4] = new StringJoiner("");       // subsequence string
    subsequences[row][5] = new StringJoiner("\t\t"); // indices in String Y
}
//----END CHATGPT----
```

```
/*
subsequences[row][0]=new StringJoiner("\t\t");
subsequences[row][1]=new StringJoiner("\t\t");
subsequences[row][2]=new StringJoiner("");
subsequences[row][3]=new StringJoiner("");
subsequences[row][4]=new StringJoiner("");
subsequences[row][5]=new StringJoiner("\t\t");
*/
```



```
}while(counter<s.length());
```

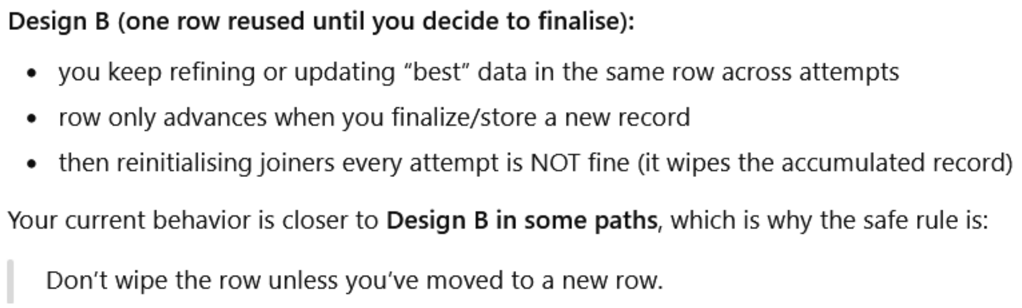
**We know that counter keeps increasing... and it denotes starting position of substring of String X... And it continues working on same permutation... So it is extremely difficult to understand why the null check decides if the StringJoiner should be wiped. I would have expected lots of my other test cases to fail. AND since ONLY single test case, I would have NEVER explored this a underlying issue.**



5) Why the null-check is the safe choice

The null-check doesn't stop you from storing new attempts — it only stops you from *accidentally wiping* a row that already exists.

## CHATGPT



**Design B (one row reused until you decide to finalise):**

- you keep refining or updating "best" data in the same row across attempts
- row only advances when you finalize/store a new record
- then reinitialising joiners every attempt is NOT fine (it wipes the accumulated record)

Your current behavior is closer to **Design B in some paths**, which is why the safe rule is:

> Don't wipe the row unless you've moved to a new row.

So it's totally possible for the program to do 10 attempts (counter 0..9) while still writing to subsequences[row] where row stayed the same.

**2) How can the "same row" be used until the loop exits?**

Because many parts of your code are built like this pattern:

- set up variables
- run an attempt
- maybe store a result
- restore/reset some variables
- increment counter
- try again

# 2) CHATGPT CHANGE (**I consider this as highly technical, it refers to my immediate issue above**)

```
//----CHATGPT----
// Do not overwrite an existing stored length for this row.
if (subsequences[row][3] == null || subsequences[row][3].toString().isEmpty())
{
    subsequences[row][3]=new StringJoiner("\n");
    subsequences[row][3].add(String.valueOf(lengthSubsequence));
}
//----END CHATGPT----
```

I added that guard for one specific reason: **your code was sometimes writing multiple times into the same row,** and a later "worse" attempt (or an empty/0 attempt) could overwrite a earlier "good" stored length for that row.

So this block was meant to enforce your Option (1) policy:

**Once a row has a stored subsequence length, don't let later loop iterations wipe it.**

**The situation it was protecting against**

Inside a single permutation you do multiple attempts (different counter / startPos). In your design, it's possible to:

1. Find a good subsequence early (e.g., length 2) and store it in subsequences[row]
2. Continue looping without incrementing row
3. Later attempt stores length 0/blank (or just re-initialises) and overwrites the good length

That produces the exact symptom you saw earlier:

- correct data printed mid-execution
- blank/0 data at the end

The guard stops the **second store** from replacing the first.

**When this guard is not appropriate**

If your current design is:

- "store every attempt as a new row" (so row++ happens for each attempt)

then **you don't need this guard** at all, and it can actually hide useful updates.

Also, if you actually want **the best length** for a row (not just the first length), then the guard should be different:

- update only if lengthSubsequence > existingLength

**The best version depends on your row meaning**

- **Row = one attempt** → remove this guard
- **Row = best attempt for a permutation** → change guard to "overwrite only if better"
- **Row = first stored result and never change** → current guard matches that

# 3)CHATGPT CHANGE (I consider this as highly avoidable)

You’ve basically already found the root cause in your own logs:

- Your debug trace shows you successfully matched **a** (index 0 in String Y) and **b** (index 1 in String Y) for permutation **abb** inside **bcab**.
- So the **maximum subsequence length should be 2** and the subsequence should be **ab**.

The reason your *final “subsequence” section* ends up showing **length 0 / empty subsequence** is because in your uploaded file **Main(192).java**, the variable:

```
int lengthSubsequence=0;
```

is **never updated anywhere** before you store it into:

```
subsequences[row][3].add(String.valueOf(lengthSubsequence));
```

So it stores **0**, and your final summary prints 0 / blank even though the match logic worked.

**Minimal fix (1 line of logic + comments)**

Right before this line:

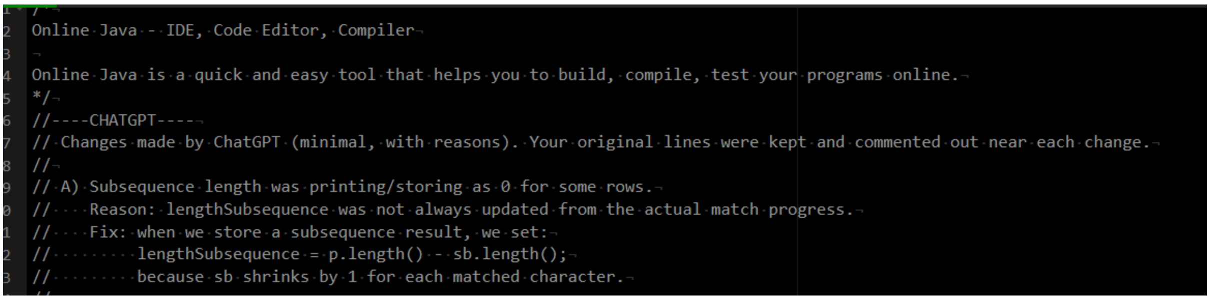
```
System.out.println("****NOW STORING subsequence length: " +  lengthSubsequence);
```

set:

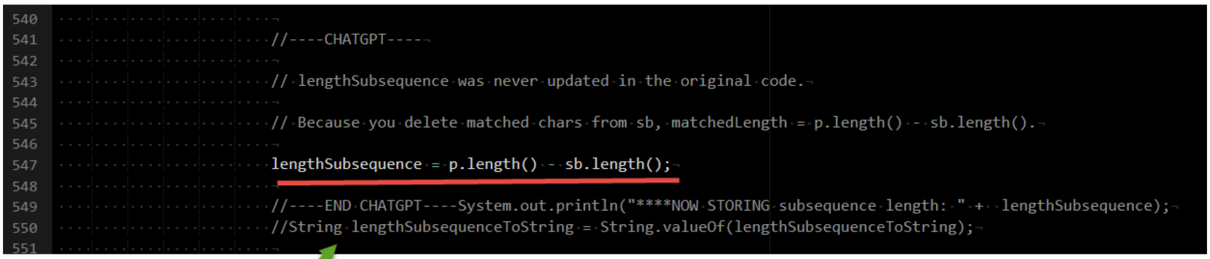
```
lengthSubsequence = p.length() - sb.length();
```

Why this works:

- You delete one char from sb every time you find a match.
- So p.length() - sb.length() = number of matched characters = matchedLength.



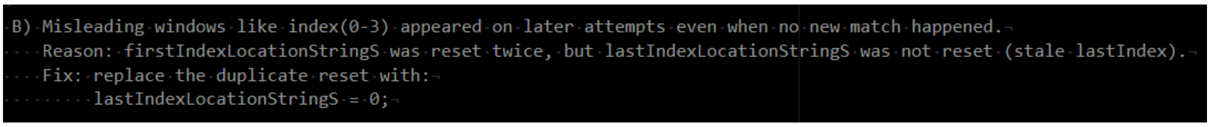
```
Online Java - IDE, Code Editor, Compiler

Online Java is a quick and easy tool that helps you to build, compile, test your programs online.
*/
//----CHATGPT----
// Changes made by ChatGPT (minimal, with reasons). Your original lines were kept and commented out near each change.
//
// A) Subsequence length was printing/storing as 0 for some rows.
//    Reason: lengthSubsequence was not always updated from the actual match progress.
//    Fix: when we store a subsequence result, we set:
//         lengthSubsequence = p.length() - sb.length();
//         because sb shrinks by 1 for each matched character.
```



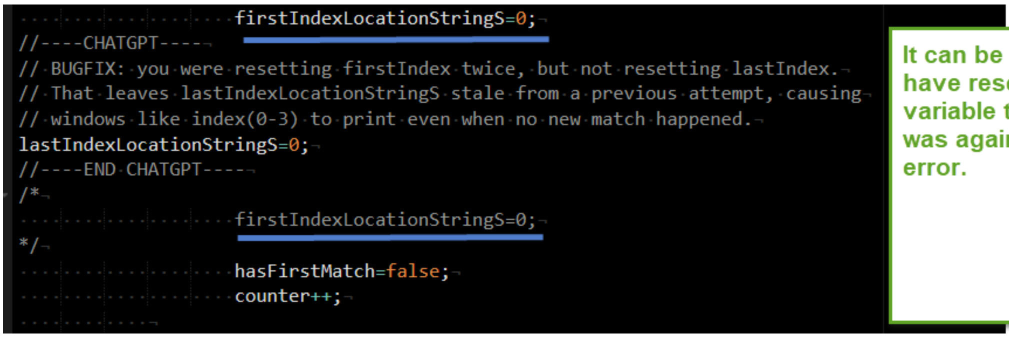
```
                        //----CHATGPT----

                        // lengthSubsequence was never updated in the original code.

                        // Because you delete matched chars from sb, matchedLength = p.length() - sb.length().

                        lengthSubsequence = p.length() - sb.length();

                        //----END CHATGPT----System.out.println("****NOW STORING subsequence length: " + lengthSubsequence);
                        //String lengthSubsequenceToString = String.valueOf(lengthSubsequenceToString);
```

I was previously looking at technique to capture the length subsequence... But I was converting it to String. And my intention was to convert it back to Integer. This is due to syntax issues that occured.. And unfortunately amongst all this chopping and changing, I never re-instated the calculation which chatGPT completed!

## 4)CHATGPT CHANGE (I ALSO CONSIDER THIS AS HIGHLY AVOIDABLE)



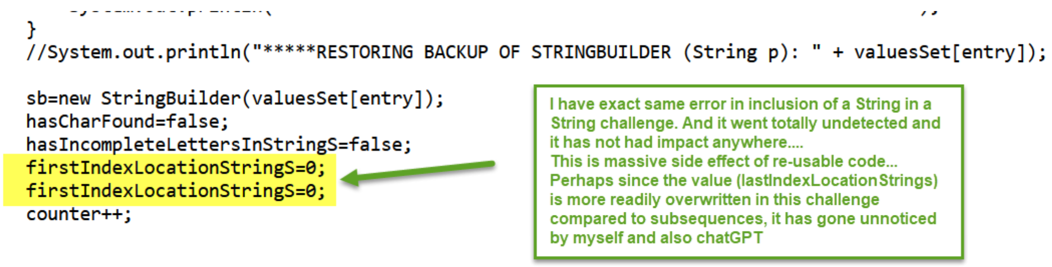
```
B) Misleading windows like index(0-3) appeared on later attempts even when no new match happened.
   Reason: firstIndexLocationStringS was reset twice, but lastIndexLocationStringS was not reset (stale lastIndex).
   Fix: replace the duplicate reset with:
        lastIndexLocationStringS = 0;
```



```
                    firstIndexLocationStringS=0;
//----CHATGPT----
// BUGFIX: you were resetting firstIndex twice, but not resetting lastIndex.
// That leaves lastIndexLocationStringS stale from a previous attempt, causing
// windows like index(0-3) to print even when no new match happened.
lastIndexLocationStringS=0;
//----END CHATGPT----
/*
                    firstIndexLocationStringS=0;
*/
                    hasFirstMatch=false;
                    counter++;
```

**It can be seen that I have reset the same variable twice... This was again a human error.**

<u>In my previous challenge, same error transpired.</u>



```
}
//System.out.println("*****RESTORING BACKUP OF STRINGBUILDER (String p): " + valuesSet[entry]);

sb=new StringBuilder(valuesSet[entry]);
hasCharFound=false;
hasIncompleteLettersInStringS=false;
firstIndexLocationStringS=0;
firstIndexLocationStringS=0;
counter++;
```

I have exact same error in inclusion of a String in a String challenge. And it went totally undetected and it has not had impact anywhere....
This is massive side effect of re-usable code...
Perhaps since the value (lastIndexLocationStrings) is more readily overwritten in this challenge compared to subsequences, it has gone unnoticed by myself and also chatGPT

## CHATGPT

You've actually exposed the real bug in your own output: **this isn't a subsequence-matching problem at all** — it's a **stale index reset bug** that only shows up on some rows.

**What's happening**

Row **4** for permutation **abb** is correct:

- subsequence length 2
- subsequence ab
- window ab index(2-3)

Row **7** later prints:

- bcab index(0-3) ... permutation: abb
- but subsequence fields are blank / length 0

That means:

- the subsequence fields for that attempt truly ended up empty (no match for that attempt), **BUT**
- the printed index(0-3) window is **wrong** and comes from **stale lastIndexLocationStringS** from an earlier attempt.

**The exact bug in your code**

You reset this twice:

firstIndexLocationStringS=0;

firstIndexLocationStringS=0;

So you **never reset** lastIndexLocationStringS.

That causes exactly what you're seeing: later attempts can print windows like index(0-3) even though they didn't find any new match, because lastIndexLocationStringS is still sitting at 3 from a previous match.

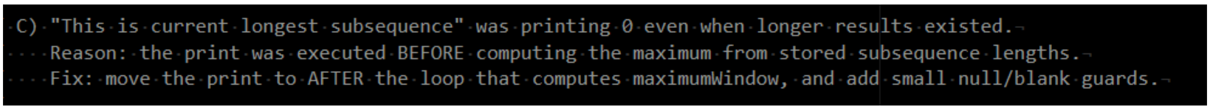
✅ **Fix:** replace the second line with:

lastIndexLocationStringS=0;

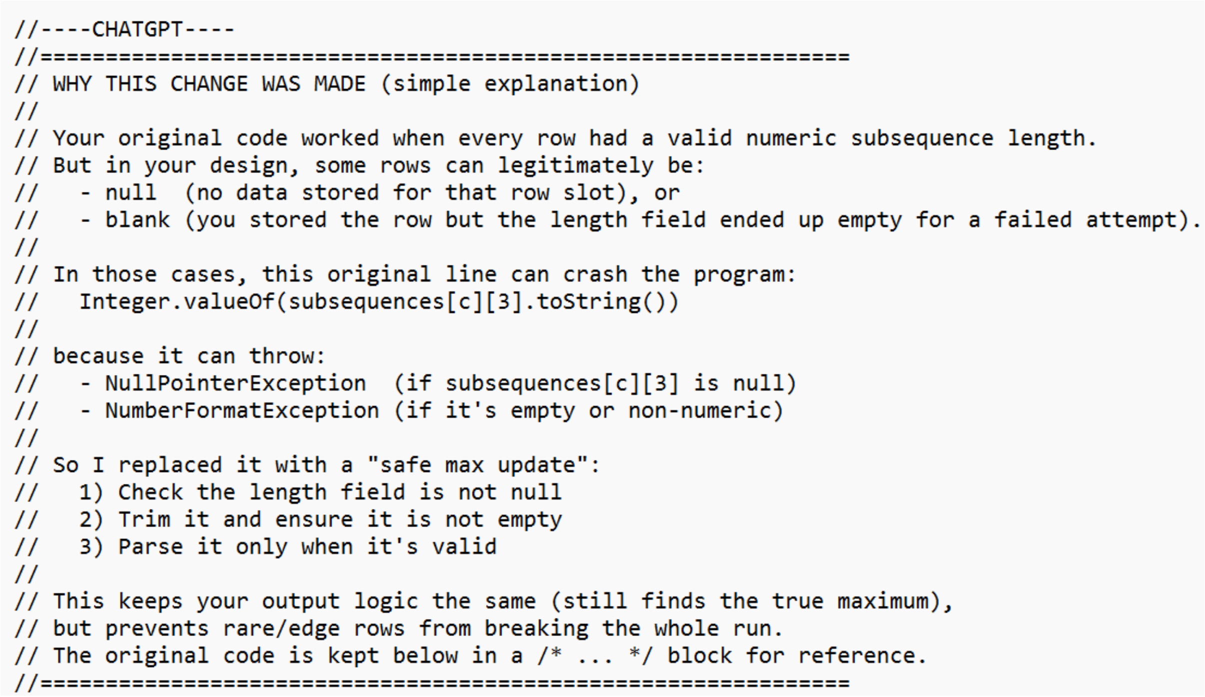
And wrap it in your markers so it's obvious.

---

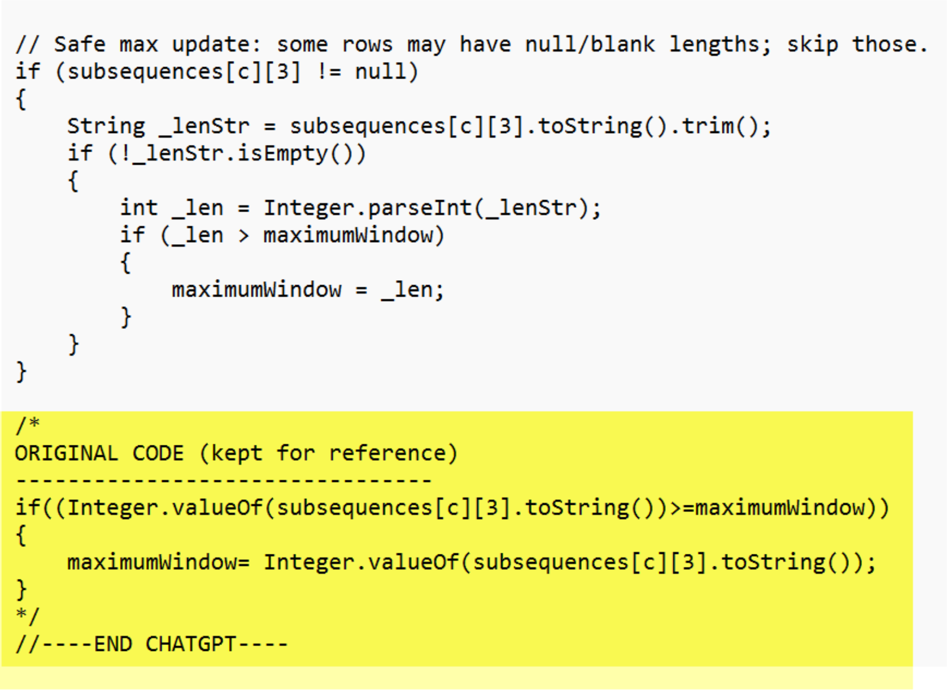
# 5)CHATGPT CHANGE (I CONSIDER THIS AS TECHNICAL CHANGE)



```
C) "This is current longest subsequence" was printing 0 even when longer results existed.
    Reason: the print was executed BEFORE computing the maximum from stored subsequence lengths.
    Fix: move the print to AFTER the loop that computes maximumWindow, and add small null/blank guards.
```



```
//----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 > maximumWindow)
        {
            maximumWindow = _len;
        }
    }
}

/*
ORIGINAL CODE (kept for reference)
--------------------------------
if((Integer.valueOf(subsequences[c][3].toString())>=maximumWindow))
{
    maximumWindow= Integer.valueOf(subsequences[c][3].toString());
}
*/
//----END CHATGPT----
```

**I did not actually experiemce any issues with my code... But I think chatGPT was safeguarding against something... I still think its code is very complex and in future it is perhaps keeping my existing code and using the exception handling should I encounter any test cases failing...**

So my immediate response was to ask ChatGPT on test cases in which my existing code (yellow) would cause exceptions…

An option for me is to remove its code and re-instate mine:

## CHATGPT CONVERSATION

could you provide test cases that would give exception (if I kept my code active instead of yours)

Yes — your original max-update line can throw **NullPointerException** or **NumberFormatException** depending on what ends up in subsequences[c][3].

Your original code:

```
if((Integer.valueOf(subsequences[c][3].toString())>=maximumWindow))
{
   maximumWindow= Integer.valueOf(subsequences[c][3].toString());
}
```

**When it breaks**

- **NullPointerException** if subsequences[c][3] == null
- **NumberFormatException** if subsequences[c][3].toString() is "", " ", "\n", etc.

Because your program stores rows for "not containing permutation" attempts and some of those rows can end up with length not populated (depending on branch/early exit), these situations happen on certain inputs.

---

**Test case set that *commonly* creates null/blank length rows**

**1) No character of p exists in s (lots of "0 match" attempts)**

```
String p = "abc";
String s = "zzzzzz";
```

Why it can break:

- Many attempts never match anything.
- Depending on your branch logic, you may still create a row but never add a length → subsequences[row][3] stays null or empty.

**2) s is shorter than p (your subsequence logic stores "failed" rows)**

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

Why it can break:

- You terminate attempts early due to insufficient characters.
- Some branches print debug but don't store a numeric length string.

**3) p has repeats, s has only one occurrence (partial matches + early termination)**

```
String p = "aaab";

String s = "a";
```

Why it can break:

- You match at most 1 char.
- Some permutations/attempts can produce blank length if the “store length” line is not executed on that path.

**4) Empty-ish s (fast failure paths)**

```
String p = "ab";

String s = "";
```

Why it can break:

- Your code likely exits earlier in some paths; any partially initialised row can leave length null/blank.

*(If your code blocks empty s earlier, use s=" " or s="x".)*

---

**A more “realistic” reproducer based on your earlier style**

This one tends to generate a mix of:

- good subsequences
- and rows where some fields were never written (depending on startPos)

```
String p = "abb";

String s = "bcab";   // you’ve already seen mixed outcomes here
```

If any row is created but subsequences[c][3] isn’t populated in one branch, your original max-update line will fail.

---

**How to *force* the exception quickly (if you want proof)**

If you temporarily add **one debug line** right before your original max-update loop:

```
System.out.println("DEBUG len raw = [" + subsequences[c][3] + "]");
```

and you see:

- DEBUG len raw = [null] → original code will throw **NullPointerException**
- DEBUG len raw = [] or [ ] → original code will throw **NumberFormatException**

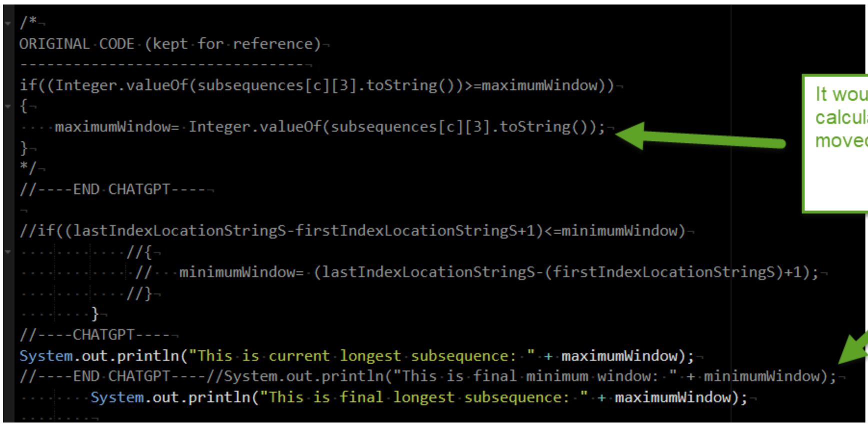
---

# 6)CHATGPT CHANGE (**I CONSIDER THIS AS SLIGHT USER EXPERIENCE CHANGE AND ALSO STRAIGHT FORWARD HUMAN ERROR**) – this is line regarding maximum……

```
//----CHATGPT----
// Moved this print to AFTER we compute maximumWindow from stored subsequence lengths.
// Previously it printed 0 because maximumWindow was still at its initial value here.
/*
        System.out.println("This is current longest subsequence: " + maximumWindow);
*/
//----END CHATGPT----
```

It had moved the system.out.println lower down after it had computed the maximumWindow via scrolling through the rows. This is most likely an error due to extreme fatigue and working in early morning hours...
Also, we can see having variable maximumWindow is highly misleading for subsequence challenge

It has been moved correctly here:



```
/*
ORIGINAL CODE (kept for reference)
--------------------------------
if((Integer.valueOf(subsequences[c][3].toString())>=maximumWindow))
{
    maximumWindow= Integer.valueOf(subsequences[c][3].toString());
}
*/
//----END CHATGPT----

//if((lastIndexLocationStringS-firstIndexLocationStringS+1)<=minimumWindow)
            //{
            //   minimumWindow= (lastIndexLocationStringS-(firstIndexLocationStringS)+1);
            //}
        }
//----CHATGPT----
System.out.println("This is current longest subsequence: " + maximumWindow);
//----END CHATGPT----//System.out.println("This is final minimum window: " + minimumWindow);
        System.out.println("This is final longest subsequence: " + maximumWindow);
```

It would have completed the calculation here, so it can be moved after it as shown below

You're right — in your current file you print the "current longest subsequence" **before** you actually calculate it, so it naturally shows **0**, even though later rows have lengths 1/2/3.

### What's wrong in the code

In `Main192_subsequence_fix_option1_v2.java`, you have:

```java
System.out.println("This is current longest subsequence: " + maximumWindow);

for (int c=0; c<row; c++)
{
    if ((Integer.valueOf(subsequences[c][3].toString()) >= maximumWindow))
    {
        maximumWindow = Integer.valueOf(subsequences[c][3].toString());
    }
}
```

So the print happens while `maximumWindow` is still its initial value (0).
Only after that do you compute it.

### Minimal fix I applied

1. **Commented out** the early print.
2. **Moved the print to after** the loop that computes the maximum.