CHATGPT EXPLANATION 1

but those two //----CHATGPT---- blocks weren't "extra features"; they were **guards that stop silent data corruption** in your current row-based storage model.

If you remove them, your output will usually look "more consistent" for some cases, but it's very easy to re-introduce the exact bug you originally hit: **rows appearing blank, wrong, or inconsistent at the end even though mid-execution prints look correct**.

Here's what each block is preventing, in plain terms.

---

1) The "only initialise when null" block

What it prevents

It prevents this situation:

- You are still working on the **same row** (you haven't done row++ yet)
- Your code loops again (e.g., next startPos, next internal pass)
- You run the initialisation again:
- subsequences[row][0] = new StringJoiner(...);
- ...
- That instantly **wipes** any subsequence data you just stored a moment ago in that same row.

So you can see correct info mid-execution, but by the time you print the table, that row may have been reset.

Why it's relevant even when counter increases

Because counter++ changes which substring of s you scan, **but it does not automatically mean you are storing to a new row**.

Your code can:

- revisit the same row multiple times during one permutation attempt, or
- write partial info, then keep scanning and write more fields for the same row.

So clearing the row each loop destroys previously stored fields.

✅ The null-check means: initialise the containers **once** when the row is first used.

---

2) The “do not overwrite stored length” block

What it prevents

Your length field subsequences[row][3] is used later to compute:

- max subsequence length
- final tie printing

If your logic passes through the same row multiple times, you can accidentally do:

- store length 2 (good match)
- later store length 0 (a later failed scan)
- now the row incorrectly looks like “length 0” at the end

So you end up with the “mentally confusing” situation:

- good match printed during execution
- but end-of-run shows 0 or blank

✅ The guard stops later passes from overwriting the first stored length for that row.

CHATGPT EXPLANATION 2

Yep — this exact mismatch is the **classic symptom** of "same row got reused, and the second pass overwrote the first pass's stored Joiners".

Let's tie it **directly** to what you printed.

What the mid-execution print proves

Mid-execution you printed the *good* data for permutation abb on suffix bcab:

- matched subsequence length = **2**
- presented window = **ab**
- Y indices show a(0) b(1)

So at that moment, your code must have had values like:

- subsequences[row][3] = "2"
- subsequences[row][2] = "ab"
- subsequences[row][4] = "ab"
- subsequences[row][5] = "a index(0) b index(1)"
- and likely store[row][0..1] = 2..3
- and store[row][2] = "abb"

So the data **did exist** in memory.

What the end print proves

At the end, for "Processing row: 3", you printed:

bcab index(0-3) ... permutation: abb (Window size: 4)
Length maximum subsequence: 0
Presented window: *(blank)*
Subsequence: *(blank)*
Subsequence (index on Y): *(blank)*

That means when you reached the final "Processing row: 3" printing code, the stored containers were now effectively:

- subsequences[3][3] is either "0" or empty/null (so it prints as 0)
- subsequences[3][2] is empty/null

- subsequences[3][4] is empty/null
- subsequences[3][5] is empty/null

So the row was **wiped or overwritten** *after* the mid-execution print.

Why "CHECK1 / CHECK2" is the smoking gun

You printed:

- **CHECK1:** ab index(2-3) ... abb (Window size: 2)
- **CHECK2:** bcab index(0-3) ... abb (Window size: 4)

This tells you something important:

Your end printer is comparing two *different stored outputs* for the same row index

That means row 3 is being used to print **more than one "shape" of record**:

- Sometimes it prints the tight window (ab index(2-3))
- Sometimes it prints the scanned suffix (bcab index(0-3))

That can only happen if, at some point, row 3 got partially filled one way and later got partially filled another way.

And that's exactly how your subsequences array gets blanked.

The simple explanation of the bug (using your row 3)

1. **First time row=3 is used for abb/bcab**, you find ab and store:
    - subsequences[3][3] = "2"
    - subsequences[3][2] = "ab"
    - subsequences[3][4] = "ab"
    - etc.
2. Later, the loops run again but **row is still 3** (row didn't advance yet, or you revisit it).
   This time the attempt fails or ends differently and produces **lengthSubsequence = 0** and empty strings.
3. If you do either of these:
    - **re-initialise** the joiners (new StringJoiner) for row 3 again, OR

- **overwrite** subsequences[3][3] with "0",

then row 3 now contains **empty** containers and/or length **0**.

4. So in the final print, row 3 looks blank even though you saw the correct info earlier.

That's why you get:

- mid execution: length 2, window "ab"
- end: length 0, blanks

Exactly which code behaviour causes this

This happens when you store results multiple times per permutation/startPos but the row pointer does not move in sync.

The two ChatGPT guards you were questioning prevent this:

Guard A (initialise only if null)

Stops this from happening:

subsequences[row][2] = new StringJoiner(""); // wipes "ab"

subsequences[row][4] = new StringJoiner(""); // wipes "ab"

subsequences[row][5] = new StringJoiner("\t\t"); // wipes indices

Guard B (do not overwrite length)

Stops this from happening:

subsequences[row][3] = "0"; // overwrites previously stored "2"

One-liner diagnosis for your exact log

When your end report shows:

bcab … abb … Length 0 … blanks

it means: **row 3 got reused after the successful match, and a later "no match / partial match" pass reset subsequences[3][*] back to empty (and length to 0).**

What you are seeing at the end is NOT "the good abb/bcab match got overwritten"

It's that you are printing a **different stored row** for abb — a row that was stored during a pass where **no matches were recorded**, but the **indices were still (wrongly) 0–3**.

So you end up with:

- **Store says:** bcab index(0-3) (because store[c][0]=0, store[c][1]=3)
- **But subsequences are empty:** because that pass recorded no matches

That produces exactly your end output:

bcab index(0-3) … permutation abb … Length 0 … Presented window blank … Subsequence blank

Meanwhile, your **good** mid-execution result for abb (the "ab index(2-3), length 2") is stored in another row (you've seen it as row 4/5/6 in your other output).

The critical "why" in terms of subsequences[row][x]

1) You re-initialise subsequences[row][*] at the start of EVERY counter pass

In your file, right at the start of the loop you do:

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");

int lengthSubsequence=0;

So for any attempt where the code *does not find anything*, the row stays:

- subsequences[row][0].toString() = ""
- subsequences[row][2].toString() = ""
- subsequences[row][3].toString() = "" (and lengthSubsequence remains 0)

That is why your row prints blanks at the end.

2) You still store the row indices EVEN WHEN THERE IS NO MATCH

Later, you store **unconditionally**:

store[row][0]=String.valueOf(firstIndexLocationStringS);

store[row][1]=String.valueOf(lastIndexLocationStringS);

store[row][2]=valuesSet[entry];

row++;

There is no if (lengthSubsequence > 0) gate.

So you can create a row where:

- store[row] says something like 0-3 (full string)
- but subsequences[row][*] is totally empty

That creates “phantom” rows.

3) The reason it becomes bcab index(0-3) specifically

This happens because you **reset** firstIndexLocationStringS to 0 at the end...

firstIndexLocationStringS=0;

...but you **do not reset lastIndexLocationStringS**.

So in a “no-match” attempt:

- firstIndexLocationStringS is 0 (freshly reset)
- lastIndexLocationStringS might still be 3 **left over from a previous successful match** (stale value)

So the stored indices become **0–3**, which prints:

bcab index(0-3) ... abb

Even though the subsequence joiners for that row are empty (because no matches happened in that pass).

✅ That is EXACTLY why your end row shows “bcab index(0-3) ... abb” but has blank subsequence content.

Why your mid-execution print shows correct data but end row looks blank

Because the “good pass” for abb created a row where:

- subsequences[row][2] = "ab"
- subsequences[row][3] = "2"
- store[row][0..1] = 2..3

...but later you also create another row for abb where:

- subsequences[row][*] stays empty (no match)
- store[row][0..1] becomes 0..3 due to **stale lastIndex**

And your end "Processing row" loop prints **every row**, so you see both kinds.

The two "guards" matter for *this exact abb/bcab confusion*

- The **null initialisation guard** prevents re-creating fresh empty joiners in the same row and wiping a good match.
- The **length guard** prevents overwriting a good length (2) with a later 0.

But in your current file, the *main* reason you see the blank bcab index(0-3) ... abb row is actually:

stale indices + unconditional store, not only overwriting.

Minimal, simple fix (no complex conditions)

If you want to stop those phantom rows without redesign:

1. **Reset lastIndexLocationStringS before each attempt**
   Set it to something "invalid" like -1:

```
//----CHATGPT----
// Reset per attempt so we never reuse stale lastIndex from previous attempts.
// Prevents phantom windows like index(0-3) when no match occurred.
lastIndexLocationStringS = -1;
//----END CHATGPT----
```

2. **Only store a row if you matched something**

```
//----CHATGPT----
// Only store a result row if at least one character matched.
// Prevents empty subsequence rows being stored/printed.
if (lengthSubsequence > 0)
{
   store[row][col]=String.valueOf(firstIndexLocationStringS);
```

```
    store[row][col+1]=String.valueOf(lastIndexLocationStringS);
    store[row][col+2]=valuesSet[entry];
    row++;
}
//----END CHATGPT----
```

Those two changes remove the exact “bcab index(0-3) but blanks” rows.

CHATGPT EXPLANATION 3

The “missing / blank at the end but correct mid-execution” issue is **not really about which permutation from valuesSet you’re on**.

It’s about this:

You move across startPos (via counter++), but you sometimes write into the same row again

So when you advance startPos, your code can:

1. **reuse the same row index**, and
2. either **re-initialise** subsequences[row][x] (new StringJoiners), or
3. overwrite subsequences[row][3] (length), or
4. store indices that are stale.

That’s what makes something look right mid-execution, then look blank later.

The simplest way to picture it

For one permutation (say abb) your code does:

- startPos = 0 → finds ab → stores into row = 4
- startPos = 1 → maybe fails → *still row = 4* → re-initialises joiners → row 4 becomes empty
- later end printing reads row 4 → looks blank

So the problem is the **relationship between these three things**:

- counter/startPos advancing
- when row++ happens (or doesn’t)
- when you clear/initialise subsequences[row][x]

If row++ doesn’t happen exactly when you think it does, or if you clear arrays before row advances, you’ll overwrite earlier stored data.