1. What should happen conceptually

For the ULTIMATE test:

0,0,

50,51,52,53,54,55,

-12,-11,-10,-9,-8,-7,

10,11,12,13,14,15,

100,101,102,103,104,105,

-2,-1,0,1,2,3,

...

Ideal behaviour:

- You discover several runs of length 6:
    - 50..55
    - -12..-7
    - 10..15
    - 100..105
    - -2..3
- currentMaximumConsectiveNumbers = 6
- storeMaxConsecutiveSequences should contain *all* of these rows (or at least not miss -2..3).

But in your actual code, for the -2,-1,0,1,2,3 row, the second phase (analysis over Store[i]) fails to recognise it as "another sequence of length 6 worth storing".

---

2. Where the logic goes wrong (big picture)

The failure happens in the **analysis phase**, when you scan each Store[i] row and try to:

1. Compute the effective length of the streak (currentMaximumConsectiveNumbers), and
2. Decide whether this row is:
    - a new maximum (replace stored sequences), or
    - equal to the current maximum (append as another best), or

- shorter (ignore)

For the -2,-1,0,1,2,3 row, two things combine to break it:

1. Your code **mis-measures the streak length** for that row because of the way posZero, captureIndex, and countIndexLocationsNoZero interact with the zeros.
2. Even if the final numeric value reaches 6 at some point, the **"equality" conditions** for storing another max sequence are extremely strict and aren't satisfied for this row.

So the bug is not in phase 1 (building Store): it *does* contain [-2,-1,0,1,2,3,0,0,...].
It's in phase 2: the logic that tries to infer the length from that row is fragile and tuned to simpler patterns.

---

3. Why this particular streak is so awkward for your heuristics

Look at this row's shape as it appears in Store:

[-2, -1, 0, 1, 2, 3, 0, 0, 0, ...]

0 1 2 3 4 5 6 7 8

For this row:

- The *real* streak is indices 0..5 inclusive → length 6
- There is a **0 in the middle** (index 2) → part of the streak
- There are **padding zeros at the end** (indices 6,7,8,...) → not part of the streak

But your analysis logic doesn't have a clean way to distinguish:

- "0 at index 2 = real data"
- "0 at index 6,7,... = padding"

because:

- There is **no storeLen[i]** telling you "this row stops at index 6", and
- There is **no null marker**; everything unused is also 0.

So instead you tried to approximate "end of real streak" using:

- posZero – a drifting pointer that sometimes tracks last non-zero, sometimes last zero
- captureIndex – last zero index
- countIndexLocationsNoZero – how many non-zeros you saw

- A bunch of special checks in the catch block and after (+1 / -1 patches)

This worked on many simpler test cases, but with this ULTIMATE array:

- There are *many* zeros scattered before and after this row.
- posZero is **not reset per row** – it carries history from previous rows.
- The "negative → 0 → positive" bridge logic was tuned assuming fewer 0 variations.

By the time you reach the -2,-1,0,1,2,3 row, posZero and captureIndex are already influenced by earlier rows and zeros in the dataset. That makes the heuristic that's supposed to say:

"The real streak here is of length 6"

either:

- under-estimate (treat it as 5 because it misses the 0 in the middle), or
- over-estimate and then try to "correct" it with one of the +/-1 patches, ending up with a number that doesn't match the current max of 6 in the moment where you test equality.

Either way, the key effect is:

For that row, your code never ends up in the branch that says
"this sequence's length **equals** currentMaximumConsectiveNumbers; store it as another max".

---

4. The equality storage branch is *very* picky

To store sequences that tie the current max, you rely on this part:

```
if (!hasMoreConsecutive)
{
  if ((countIndexLocationsNoZero==currentMaximumConsectiveNumbers)
    && countIndexLocationsNoZero+1==currentMaximumConsectiveNumbers  &&
Store[i][0]==0)
  {
    // store equal-length sequence
  }
  else
```

```
  {
    if ((j>=posZero && posZero!=0 && zeroFound &&
        countIndexLocationsNoZero>=posZero && Store[i][j-1]!=0 && j>=nums.length)
     || ((countIndexLocationsNoZero==currentMaximumConsectiveNumbers &&
(j==nums.length)
      || countIndexLocationsNoZero+1==currentMaximumConsectiveNumbers  &&
Store[i][0]==0)))
    {
      // store equal-length sequence
    }
  }
}
```

For -2,-1,0,1,2,3:

- Store[i][0] is **not 0** (it's -2), so the first equality case cannot trigger.
- hasMoreConsecutive can easily remain true at the wrong time because you check Store[i][j+1] != 0 in the try block; as long as you see another non-zero ahead, hasMoreConsecutive stops the equal-length branch from running.
- countIndexLocationsNoZero is counting *non-zero* elements (5 here), not the true streak length including that internal 0. You rely on additional +1 patches (bridge logic, "0 in original dataset" bump, etc.) to make it line up with the max – but those patches were tuned around simpler patterns, and in this ultimate test they don't land cleanly on "6" at the right moment.

Result: At no point do you hit a clean condition like:

countIndexLocationsNoZero == currentMaximumConsectiveNumbers
or
(patched length including zero) == currentMaximumConsectiveNumbers

and simultaneously satisfy all the other flags (hasMoreConsecutive == false, j at correct boundary, posZero conditions, etc.).

So the row gets treated as "interesting but not exactly equal to the current max"; it's never stored.

---

5. Why this is exactly the "cliff edge" of your design

This ULTIMATE test is basically the worst-case combination for the heuristics you built:

- Multiple disjoint long streaks (so currentMaximumConsectiveNumbers is updated several times across rows).
- Many zeros scattered everywhere (start, middle, end, padding).
- A "negative → 0 → positive" streak (-2,-1,0,1,2,3) that appears **after** several other sequences and zeros, so your "global" variables (posZero, etc.) are already "contaminated" by earlier rows.

Your code was *very* specifically tuned to earlier tests; for those, the patches around 0, posZero, and captureIndex did line up well enough.

But this ultimate array is exactly the kind of input that says:

"Okay, if you don't have a clean length (storeLen[i]), or a clean marker (null),
and you rely on 0 doing double duty with a web of conditions...
here's the kind of case that slips through the cracks."

---

6. So what did we learn from this failure?

You basically discovered:

1. **It's not enough to "sort of" track the end of the streak** with blended variables like posZero, captureIndex, and countIndexLocationsNoZero.
   You need an explicit, unambiguous definition of:

"This is where the real data stops."

2. **0 cannot reliably serve as both real data and padding** when your logic depends so heavily on detecting 0 to make structural decisions.
3. **Global state (like posZero not reset per row)** makes behaviour depend on the *order* of rows, not just their content – which is why some streaks of the same shape get treated differently depending on where they appear in the full dataset.

And that's exactly why the solution we talked about earlier (with storeLen[i] or Integer[][] + null, or simply reusing the +1 property cleanly) avoids this entire class of bugs:
you stop guessing where the streak ends and instead **know**.

Why your code handles the other tests but not this pattern

From what you've observed:

- Multiple equal-length streaks *without* crossing zero → you handle fine.
- Negative→0→positive streaks *on their own* or with smaller neighbours → also fine.
- Even some mixed cases with zeros *before* or *after* a streak → still fine.

So your (very elaborate) logic *does* successfully cover:

- "simple" equal max streaks,
- "simple" crossings through 0,
- some noise zeros.

The one that breaks it is more specific:

"A late, max-length streak that crosses 0, where:

- **earlier max streaks already set currentMaximumConsectiveNumbers, and**
- **there are trailing zeros in nums after that streak."**

In that scenario:

- Phase 1 still builds a nice row like
  [-2,-1,0,1,2,3,0,0,...]
- But Phase 2 (the posZero / captureIndex / countIndexLocationsNoZero logic):
  - has posZero and friends already influenced by earlier rows,
  - misjudges where the real streak ends in this particular row,
  - or never satisfies the *very strict* "equal max" conditions at the end of the loop.

So that last -2,-1,0,1,2,3 gets treated as *not quite* equal to the existing max=6, and it never gets stored as a joint max.

So, in one line:

Your code doesn't generally "blow up on lots of cases"; it fails on a **narrow, nasty family** of inputs:

Later max-length streaks that cross 0 (negative → 0 → positive) and are followed by zeros, when other max streaks have already been found earlier.

That's exactly why everything "felt fine" until you hit the ULTIMATE test and my example 6 – they're the first ones that really combine all those conditions at once.

🔬 **Test 1 – "Hyper-ultimate multi-max, no bridging"**

Several **disjoint** max sequences of the *same* length, with duplicates and noise, but no numbers that accidentally join them into a longer run.

```
static int[] nums = new int[] {
  50,                    // noise

  // some scattered members of real sequences (out of order)
  -2, 12, 101,

  // Max streak 1: crosses zero: length 8
  -4, -3, -2, -1, 0, 1, 2, 3,

  // Max streak 2: positive block: length 8
  10, 11, 12, 13, 14, 15, 16, 17,

  // Max streak 3: larger positive block: length 8
  100, 101, 102, 103, 104, 105, 106, 107,

  // duplicates inside sequences (shouldn't change max length)
  0, 11, 14, 101, 105,

  // random noise
  200, -100, 999
};
```

Expected:

- Longest consecutive sequence: 8
- Theoretical max-length sequences (any correct solution should see these as max):

1. [-4, -3, -2, -1, 0, 1, 2, 3]
2. [10, 11, 12, 13, 14, 15, 16, 17]
3. [100, 101, 102, 103, 104, 105, 106, 107]

There's **no** -5, 4, 9, 18, 99, or 108, so none of these three can grow to length 9.

🔬 **Test 2 – "Zeros everywhere + multiple max streaks"**

Lots of zeros and repetition, with three different max streaks of the same length:

```
static int[] nums = new int[] {
  0, 0, 0,                  // padding zeros

  // Max streak 1: crosses zero, length 6
  -3, -2, -1, 0, 1, 2,

  100,                      // noise in the middle

  // Max streak 2: positive-only, length 6
  10, 11, 12, 13, 14, 15,

  // Max streak 3: negative-only, length 6
  -10, -9, -8, -7, -6, -5,

  0, 0, 0,                  // more padding zeros

  // Repeat the cross-zero sequence again to test duplicates
  -3, -2, -1, 0, 1, 2
};
```

Expected:

- Longest consecutive sequence: 6

- Theoretical max sequences (length 6):
    1. [-3, -2, -1, 0, 1, 2] (appears twice in different positions)
    2. [10, 11, 12, 13, 14, 15]
    3. [-10, -9, -8, -7, -6, -5]

Zeros at the edges are **not** able to extend any streak beyond 6, and we've avoided neighbours that would join streaks into something longer.

🔬 **Test 3 – "Heavy duplicates & repeated max sequence"**

Here the dataset is messy, with many duplicates of the same max sequence scattered around, plus other shorter sequences:

```
static int[] nums = new int[] {
  5, 5, 5,          // duplicates

  0, 0,             // zeros that are not part of any longer run yet

  3, 4, 1, 2,       // scrambled sub-run pieces

  // bits of the real max run, mixed:
  -1, 1, 2, 3, 4, 5, 0,

  100,              // noise

  20, 21, 22,       // short consecutive run length 3

  // Full max streak explicitly again:
  -1, 0, 1, 2, 3, 4, 5
};
```

Expected:

- Longest consecutive sequence: 7
- Theoretical max sequence of length 7:
    - [-1, 0, 1, 2, 3, 4, 5]

We've *deliberately* not included -2 or 6, so nothing can grow beyond length 7, no matter how you join things.

Your code may:

- detect the [-1..5] run starting from more than one index,
- and may store multiple rows that *represent* that same streak (just with 0-padding after).

That's fine as long as the **max length** is 7 and those rows' leading part is exactly that run.

What those 3 new tests really verified

1. **Test 1**
    - Multiple disjoint max streaks in totally different ranges:
        - negatives, small positives, big positives
    - Duplicates inside sequences
    - Noise in between
    - No "bridge" numbers
      → Confirms your logic can find several separate max runs and still keep the correct max length.
2. **Test 2**
    - Zeros at:
        - the start,
        - between streaks,
        - and again at the end
    - Three different max streaks of equal length:
        - [-3..2], [10..15], [-10..-5]

- One of them duplicated later
  → This really hammers the parts of your code that:
    - distinguish *real* zeros in streaks from padding zeros,
    - handle **multiple** max-length streaks,
    - don't accidentally merge across padding.

3. **Test 3**
   - Heavy duplicates (5,5,5, 0,0)
   - Scrambled subsequences (3,4,1,2)
   - One true max streak [-1..5] that:
     - is partly "hinted" earlier,
     - then appears cleanly in full at the end
       → Confirms that:
     - shorter partial chains don't fool your logic into thinking they're the max,
     - the final, complete chain still wins with the correct length (7).

Across all of that, your code:

- Reports the **correct longest length** each time.
- Stores sequences that match those theoretical max streaks (with trailing zeros, as expected from your Store design).
- No extra bogus longer run appears.