Proof-by-condition classification

Below are the **strongest "exact certainty" items first** (true logical contradictions / structural impossibility), then "effectively unreachable due to compare/typo", then "logically possible but state-rare".

A) Guaranteed impossible under current control-flow / local logic

B00077

- Condition: if (k == nums.length - 1)
- **Why impossible:** it sits inside a loop that only iterates while k <= nums.length-2 (so k==nums.length-1 can't occur).

B00180

- Condition: if (String.valueOf(nums[k+1]).equals(potentialfurtherAscendingBeyondThisEnd))
- **Why impossible:** immediately before this check, the code does:
- potentialfurtherAscendingBeyondThisEnd="";

so RHS is always "", and String.valueOf(nums[k+1]) is never empty.

B00181

- This is the else of:
- if ((Math.abs(nums[k] - (nums[k-1] - difference)) <epsilon) || (Math.abs(nums[k] - (nums[k-1] + difference)) <epsilon))
- **Why effectively impossible in practice:** in the merged runs, the if above is **always true whenever that outer block is reached** (you can see B00177 and B00178 have identical hit totals in the file, so the else never happens).
  That's not just "rare", it means the outer path that leads here constructs a state where the condition is invariantly true.

B00185 / B00186 / B00187

- These are in parseTickerTokens(...)
- **Why impossible:** you call it as:
- parseTickerTokens(normalizeTickerForComparison(...))

and normalizeTickerForComparison(...) strips or normalizes the exact characters those branches look for:

  - B00185 checks whitespace / * / - → removed/normalized away

- B00186 checks “not a letter” → never present after normalization
- B00187 checks raw repeated runs like "SSS" → your ticker is already compressed via appendSCompressed(...)

So those branches are *logically blocked by preprocessing*.

B) Never-hit blocks that are dead due to dependency on another dead block

These are “exactly sure” **if** the upstream block is truly dead (and we *do* have that proof for some of them).

B00043 depends on **B00181**

- B00043 requires standaloneTemp to be non-empty.
- standaloneTemp is only assigned here:
- standaloneTemp = start;

which lives inside **B00181** (never entered).

- Therefore **B00043 cannot happen unless B00181 happens**, and B00181 is dead ⇒ B00043 dead.

B00070 depends on B00069 (nested inside it)

B00095 depends on B00094 (nested inside it)

B00124 depends on B00123 (nested inside it)

B00174 depends on B00173 (nested inside it)

B00130 / B00131 / B00132 / B00133 depend on B00129 (all inside the B00129 branch)

C) “Dead because of String compare bug / typo” (effectively unreachable as written)

These are not “mathematically impossible”, but **they are “practically unreachable” because the code uses reference comparisons** (== / !=) for emptiness checks.

Blocks where this matters directly

B00082 / B00135 / B00154

- They use if (temp != "") (reference compare)
- Your temp is built via concatenation when non-empty (backupStart+"->"+end), and is reset to "" literal in several places.

- This makes “empty/non-empty” behavior inconsistent and strongly biases these blocks toward never firing.
- Correct check should be !temp.isEmpty() or !temp.equals("").

B00066 / B00067 / B00074 / B00108 / B00148

- They contain conditions like potentialfurtherAscendingBeyondThisStart=="" or != ""
- But those variables are often assigned via String.valueOf(...) or concatenation, so reference equality with "" is unreliable.

B00090

- Condition is:
- if (potentialfurtherAscendingBeyondThisEnd!="" && potentialfurtherAscendingBeyondThisEnd!="")
- That’s almost certainly a typo (it checks **End twice**, and doesn’t check Start).
- Even if values exist, this block can be blocked by the typo + reference comparison.

D) The remaining blocks are “logically possible but state-rare”

These are not provably impossible from a single local contradiction, but they require a very specific combination of:

- counter value at the right time,
- stored range strings being set/not set,
- hasWrittenRepeatNumber state,
- exact “difference step” adjacency and sometimes plateau adjacency.

Remaining “RARE/state” IDs (from your merged list):
B00029, B00040, B00050, B00054, B00062, B00066, B00067, B00068, B00069, B00074, B00076, B00079, B00084, B00088, B00090, B00094, B00096, B00100, B00103, B00106, B00108, B00115, B00119, B00123, B00127, B00129, B00138, B00146, B00148, B00151, B00162, B00167, B00169, B00171, B00173, B00192
(You can inspect each exact guard in never_entered_contexts.txt.)

What is *usually safe* to remove when hit=0

These are high-confidence “byproduct unused code” candidates:

✅ **Duplicate debug output blocks** (System.out.println(...) branches)
✅ **Alternative formatting branches** that only change display, not logic
✅ **Redundant else branches** where the condition is logically always true (you can prove it from code)
✅ **Old experimental paths** guarded behind flags you never enable

---

What is *not safe* to remove just because hit=0

These are common traps:

⚠️ **Bounds/guard checks** (may only trigger on bad input but protect you)
⚠️ **Error-handling paths** (NaN/Infinity/empty input)
⚠️ **Rare state transitions** in complex sequence logic (like your plateau/junction stuff)
⚠️ **Code that resets state**, even if the branch didn’t execute in tests

For these, “quarantine” + real-world regression testing is safer than deletion.