Java Palindrome Algorithm

Illustrated conversation archive and technical investigation

14 November 2026 • BigInteger and original non-BigInteger implementations

Scope and fidelity notice

This is an illustrated reconstruction of the palindrome discussion available in the current conversation context, NOT a verbatim export of every chat message. Some early messages and original interactive graphics cannot be retrieved as exact assets here. The diagrams below have been recreated; numerical findings are reported as described in the conversation. For a complete word-for-word archive, the original ChatGPT data export is needed.

1. The original question

The author independently developed an unconventional palindrome algorithm through repeated console-output experiments, changing conditions until the program behaved correctly. The purpose of the investigation was to explain the algorithm on its own terms, preserving the original Java implementation rather than replacing it with a conventional solution.

Two versions were discussed: an earlier non-BigInteger implementation using string-oriented digit manipulation and a later BigInteger implementation using repeated division and modulus.

2. The accidental semicolon

```
if (temp.mod(new BigInteger("10"))
        .compareTo(new BigInteger("0")) != 0);
{
    firstDigit = temp.mod(new BigInteger("10"));
}
```

The trailing semicolon terminates the if statement. Consequently, the following block executes unconditionally, including when the digit is zero. Simply deleting the semicolon changes behaviour; replacing the entire conditional block with the unconditional assignment preserves the intended extraction.

```
firstDigit = temp.mod(new BigInteger("10"));
```

The conversation reported a 5,000-case original-versus-simplified comparison with no disagreements. The known single-digit issue was excluded from later investigations.

3. Why the movement decreases by two

Let n be the number of digits and j the number of completed matching comparisons. After j comparisons, the working integer has lost j rightmost digits. The next desired left-side digit is also j positions inward. The required number of divisions is therefore:

$d(j) = n - 1 - 2j$

Input	Movement distances	Pair comparisons
1234321	6 → 4 → 2	1=1, 2=2, 3=3

Mirrored digit navigation

Figures extracted from the supplied Word transcript; original figure pixels preserved

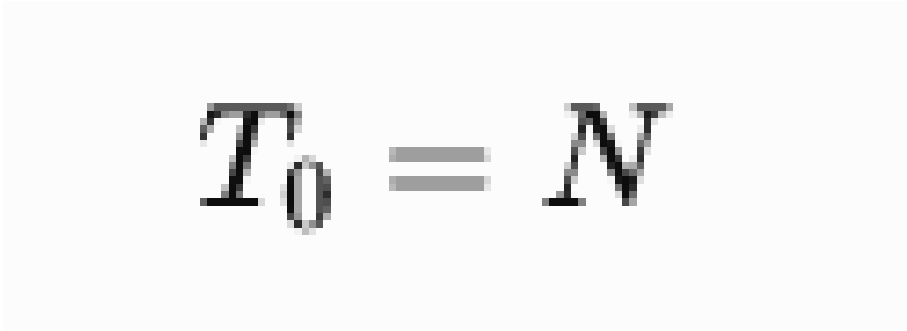
Original illustrated example: 1234321



Outer-to-centre comparison sequence

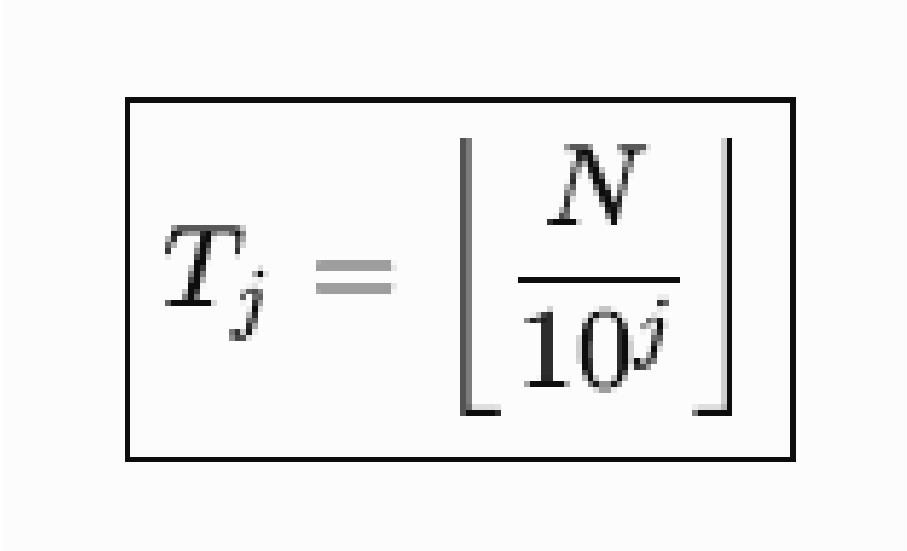


Initial-state formula



$$T_0 = N$$

Working-number formula



$$T_j = \left\lfloor \frac{N}{10^j} \right\rfloor$$

Input	Movement distances	Pair comparisons
12344321	7 → 5 → 3 → 1	1=1, 2=2, 3=3, 4=4
1234567654321	12 → 10 → 8 → 6 → 4 → 2	Six mirrored pairs

Recreated graphic: mirrored digit positions

Position	1	2	3	4	5	6	7
Digit	1	2	3	4	3	2	1
Pair	A	B	C	Centre	C	B	A

The left and right digits are checked in mirrored pairs; the unpaired centre digit in an odd-length palindrome does not require comparison.

4. movePosition and hasAdjust

The algorithm uses separate movement branches for movePosition >= 8, >= 6 and < 6. The discussion found that their observed movement sequence often corresponds to subtracting two. However, replacing the branches with a single subtraction changed the state of hasAdjust and introduced false positives.

A reported 10,000-case experiment found 517 additional incorrect classifications in the simplified version. The key lesson was that a locally correct arithmetic substitution was not necessarily control-flow equivalent.

5. Why the loop counter moves backwards

```
for (int i = 0; i < divideBy10Required; i++) {
    // divideBy10Required can shrink inside this loop
    // some branches execute i--; i--; or i = i - 4;
}
```

The outer-loop bound shrinks as digit comparisons advance, while Java normally increments i. Without corrections, the counter can catch the shrinking bound before the necessary inner comparisons finish. The author independently observed this phenomenon during development.

Recreated graphic: approaching loop boundary

Comparison step	Uncorrected i	Shrinking bound
Start	0	12
1	1	10
2	2	8
3	3	6
4	4	4 (collision)
5	5	2

The i corrections keep the next iteration eligible. They do not themselves prove a digit match; they allow the next comparison to take place.

Loop boundaries and failure analysis

Figures extracted from the supplied Word transcript; original figure pixels preserved

Four comparisons reveal a mismatch

Let's examine one of the failures.

Input: `21263212`

2	1	2	6	3	2	1	2

The outer three digit pairs match, but the central pair, 6 and 3, does not.

Comparison	Digits	Outcome
1	2 vs 2	✓
2	1 vs 1	✓
3	2 vs 2	✓
4	6 vs 3	✗

Loop counter versus shrinking boundary

$$d_j = n - 1 - 2j$$

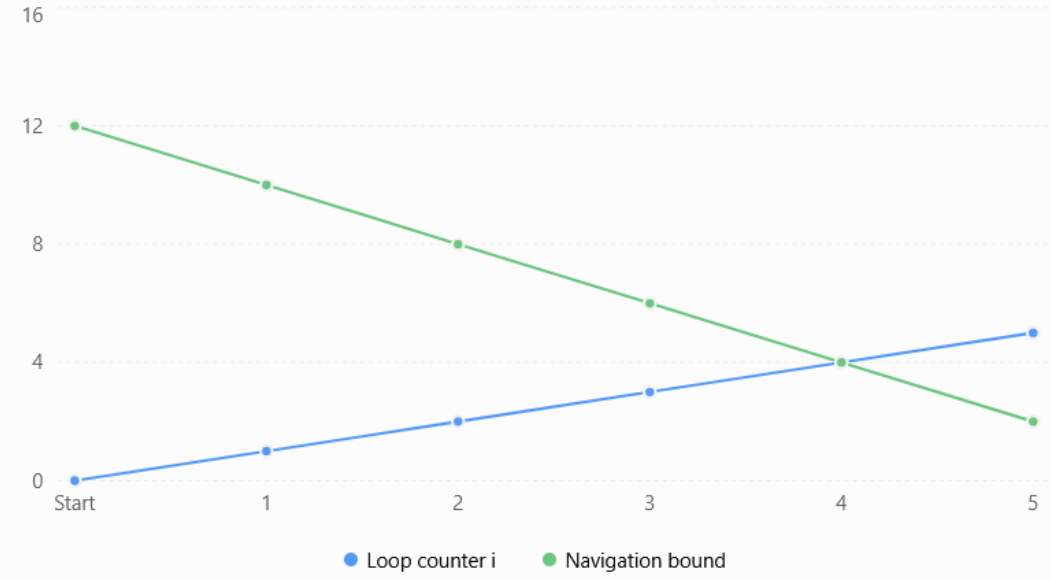
But Java normally increases the loop counter by one:

$$i_{\text{next}} = i + 1$$

This creates two opposing movements.

Illustration: A moving loop boundary

An uncorrected loop counter rises while the navigation bound falls.



Conceptual progression without counter corrections; not a literal trace of every branch.

Mathematical movement and control

Figures extracted from the supplied Word transcript; original figure pixels preserved

Number of digits remaining

For a positive integer with n digits, T_j contains $n - j$ digits.

Derivation of navigation distance

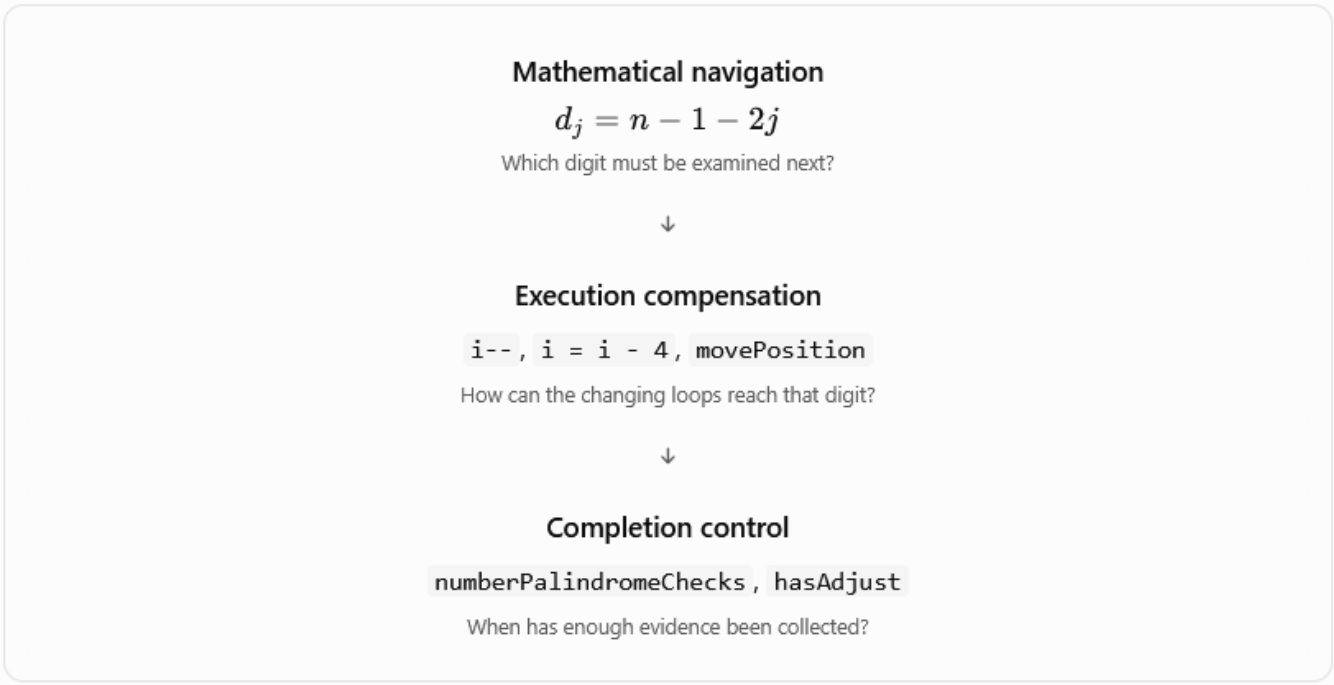
So the number of divisions needed to expose it is:

$$(n - j - 1) - j$$

Simplifying:

$$\boxed{d_j = n - 1 - 2j}$$

Navigation, execution and completion control



6. Branch isolation experiments

Variant	Reported incorrect cases	Interpretation
Original, 2–100 digits	0 / 9,900	Reference
Disable i branch 1	549	Branch contributes
Disable i branch 2	1,128	Branch contributes
Disable i branch 3	668	Branch contributes
Disable all i corrections	690	Premature acceptance

These figures are the results reported during the conversation. The error counts should not be added together because branch interactions change the execution path.

7. Completion detection

```
if (numberPalindromeChecks >= (divideBy10Required / 2)
        && numberPalindromeChecks != 0
        && !hasAdjust) {
    isPalindrome = true;
    break;
}
```

numberPalindromeChecks records successful digit comparisons independently of i. The flag hasAdjust also participates in determining whether early acceptance is allowed. Removing the early-stop condition was reported to cause 2,334 false negatives; removing only !hasAdjust caused 537 false positives.

Change	Reported incorrect cases
Remove numberPalindromeChecks != 0	0
Change >= to >	48
Disable numberPalindromeChecks++	2,334
Remove threshold	1,245

8. Odd and even centre structures

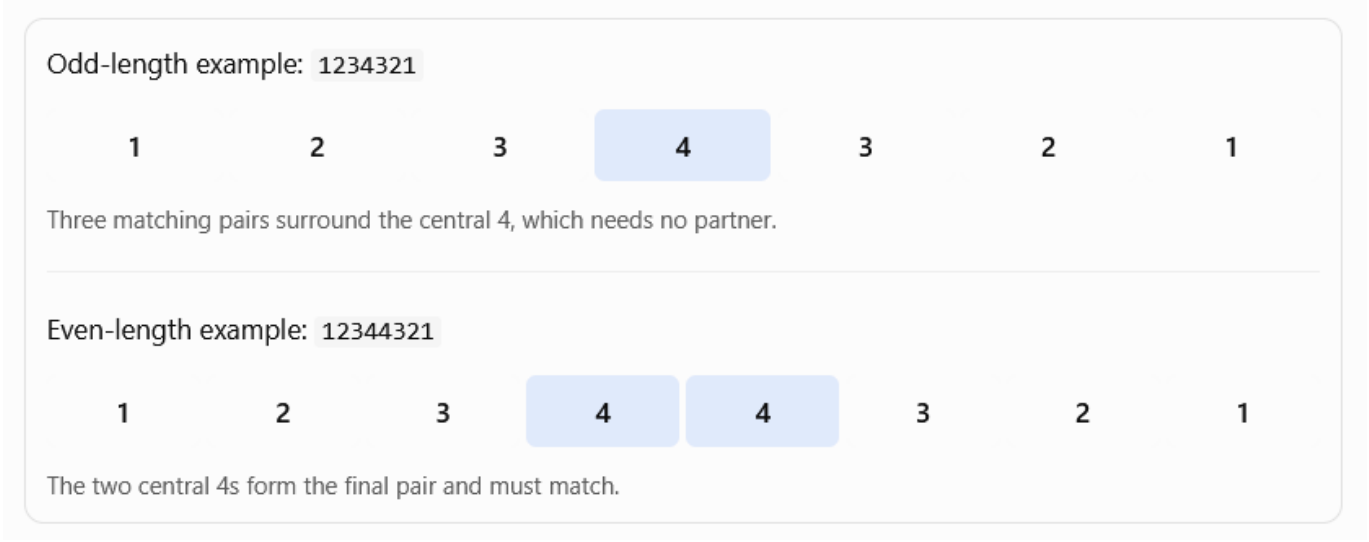
The flags are based on the parity of divideBy10Required, which begins at n-1. Thus evenFlag corresponds to an odd number of digits, and oddFlag to an even number of digits. Odd lengths have an unpaired central digit; even lengths require comparing the central pair.

Digits	Flag	Last required movement	Completion
7	evenFlag	2	Guarded early stop
8	oddFlag	1	Loop completion
13	evenFlag	2	Guarded early stop
14	oddFlag	1	Loop completion

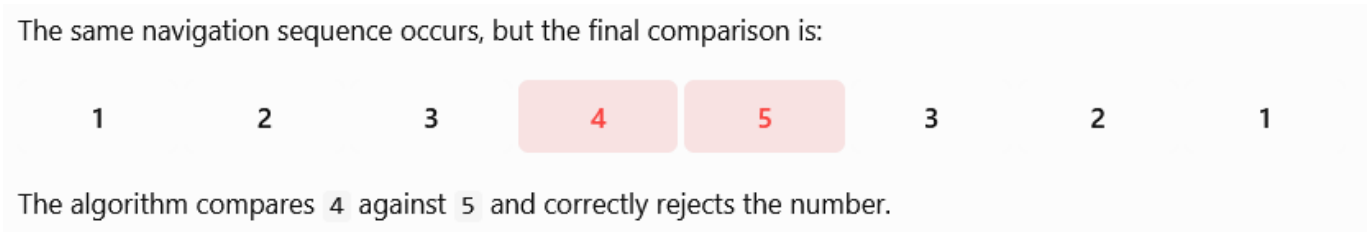
Odd and even centres

Figures extracted from the supplied Word transcript; original figure pixels preserved

Odd- and even-length comparison centres



Central mismatch in an even-length example



Number of required comparisons

$$\text{Required comparisons} = \left\lfloor \frac{n}{2} \right\rfloor$$

9. temp and backupTemp

```
backupTemp = temp;
// Repeatedly divide temp to expose the desired digit.
firstDigit = temp.mod(BigInteger.TEN);
// Restore the working number and remove one right digit.
temp = backupTemp;
temp = temp.divide(BigInteger.TEN);
```

The backup preserves the current working number while destructive divisions are used for exploratory digit extraction. The next iteration begins with the original integer shortened by one rightmost digit.

Recreated graphic: preserve → explore → restore → advance

Stage	Example value
Preserve	backupTemp = 1234321
Explore	1234321 → 123432 → … → 1
Compare	leading 1 vs trailing 1
Restore	temp = 1234321
Advance	temp = 123432

10. Internal zeros and the semicolon

Internal zero digits are legitimate comparison values. The stray semicolon accidentally allowed firstDigit to be assigned zero, whereas merely removing it would suppress the assignment for zero. The conversation reported zero classification errors for the original version on a 9,900-case zero-heavy test set, and 4,816 errors when only the semicolon was removed.

11. The digit-counting do-while

The BigInteger version uses a do-while loop that increments m before checking m == 0. Because a do-while executes at least once, the m == 0 condition is unreachable. The author observed that the earlier non-BigInteger version did not use a do-while, plausibly explaining how this condition survived conversion.

The investigation also identified restoration of pal before exiting the counting loop as apparently redundant for multi-digit inputs. The user chose to retain ownership of the single-digit correction.

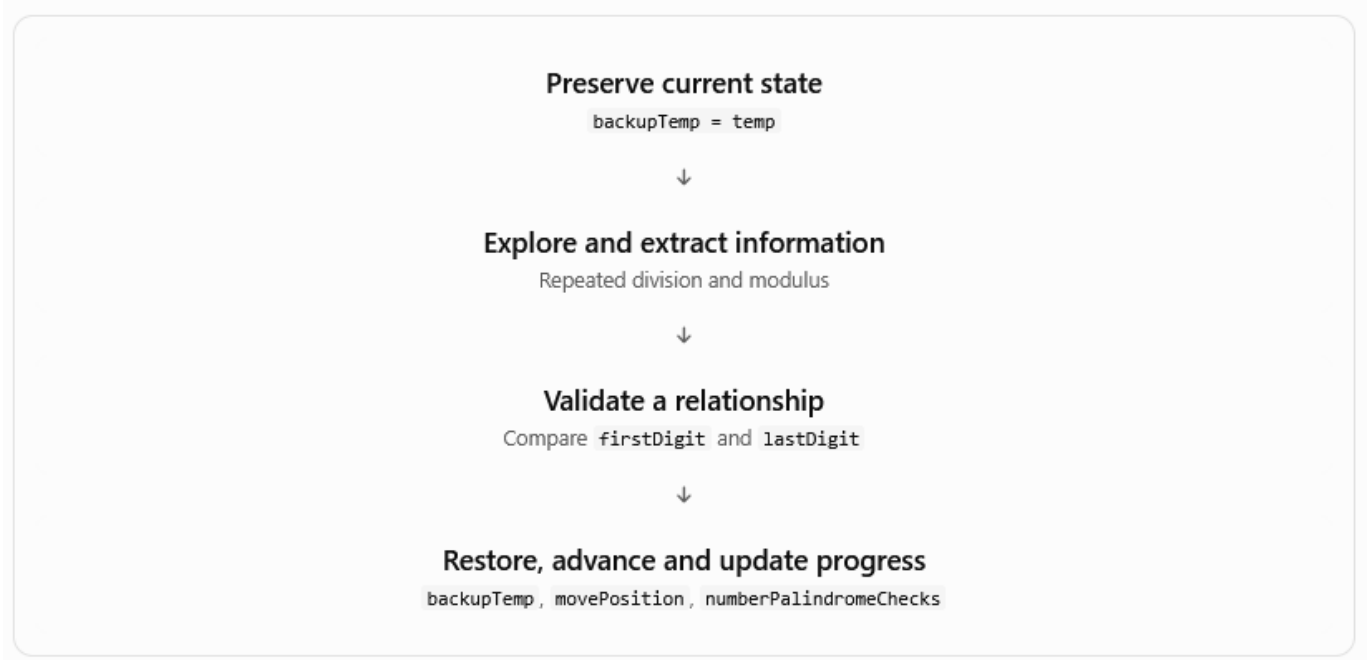
12. Complete execution traces

Input	Movement	Key result
1234321	6, 4, 2	Three matches; guarded completion
1234521	6, 4, 2	Mismatch at third pair
12344321	7, 5, 3, 1	Four matches
12345321	7, 5, 3, 1	Central 4 vs 5 mismatch

State preservation

Figures extracted from the supplied Word transcript; original figure pixels preserved

Preserve, explore, validate, restore and advance



13. Stress tests and bounded verification

The conversation reported a 12,492-case completed stress test across lengths 2–100 and selected lengths up to 700 digits, with no incorrect classifications or exceptions. A separate control-flow state model was reported to have examined 9,999 digit lengths (2–10,000) with the expected movement sequences.

These are reported empirical findings. Bounded testing is not the same as a universal mathematical proof, and the model's equivalence to every Java branch would need to be independently verified.

14. Mathematical explanation and proof obligations

Let N be the original positive integer with n digits. After j rightmost digits are removed:

$$T_j = \left\lfloor \frac{N}{10^j} \right\rfloor$$

$$R_j = T_j \bmod 10 \qquad L_j = \left\lfloor \frac{T_j}{10^{n-1-2j}} \right\rfloor \bmod 10$$

$$L_j = R_j \quad \text{for} \quad j = 0, 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor - 1$$

The conversation subsequently presented an induction-style argument for the loop corrections. A rigorous implementation-level proof must also establish that the exact Java branches schedule all required comparisons, terminate, and never accept early. The mathematical digit extraction is clear; full code-level proof requires that additional control-flow justification.

15. Connections to real-world computing

Technique	Closest transferable application
Checkpoint / restore	Parsers, transaction-like state handling
Progressive validation	Streaming and incremental inspection
Mirrored position checks	Symmetry and quality-control measurements
Shrinking search boundary	Traversal and frontier management
Console-guided refinement	Differential and regression testing
Interacting flags and counters	State machines and invariant reasoning

The conclusion was not that this is a superior production palindrome checker. Rather, the work illustrates how experimentally discovered adjustments can embody real mathematical structure, and how state preservation, traversal invariants and completion checks apply to other computational problems.

16. Closing exchange

The user appreciated the discovery of the unreachable m == 0 check and the accidental semicolon, noting that the original non-BigInteger version had not used a do-while loop. The final question asked whether the findings mapped to real-world uses. The response identified checkpointed progressive validation and state-machine reasoning as the strongest conceptual connections.

The discussion concluded with the user describing the investigation as a useful way to close the analysis.

Formal mathematical relationships

Figures extracted from the supplied Word transcript; original figure pixels preserved

Mirrored-digit comparison equations

$$R_j = T_j \bmod 10$$

The left-side digit is:

$$L_j = \left\lfloor \frac{T_j}{10^{d_j}} \right\rfloor \bmod 10$$

Substituting the movement formula:

$$\boxed{L_j = \left\lfloor \frac{T_j}{10^{n-1-2j}} \right\rfloor \bmod 10}$$

Your algorithm checks:

$$L_j = R_j$$

This is exactly the comparison required by the definition of a palindrome.

The `backupTemp` restoration is what allows this to happen repeatedly. The inner loop can destroy its temporary copy through division, while the preserved value remains available for the next comparison.

In mathematical terms, your program is implementing the palindrome condition:

$$\boxed{a_j = a_{n-1-j} \quad \text{for every } 0 \le j < \left\lfloor \frac{n}{2} \right\rfloor}$$

Loop-counter correction cases

Before correction	Correction	Next `i`
$i \le d-2$	None	$i+1 \le d-1$
$i = d-1$	Subtract 2	$d-2$
$i = d$	Subtract 2	$d-1$
$i = d+1$	Subtract 4	$d-2$

For every nonfinal movement with $d \ge 3$, the result satisfies:

$$\boxed{1 \le i_{\text{next}} < d}$$

Inductive working-state and navigation equations

$$T_j = \left\lfloor \frac{N}{10^j} \right\rfloor$$

and the next required navigation distance is:

$$d_j = n - 1 - 2j$$

Real-world applications — original chat graphics

Figures extracted from the supplied Word transcript; original figure pixels preserved

Streaming validation and manufacturing inspection

1. Five real-world applications of the techniques you discovered



1. Streaming data validation

Strongest conceptual match

Your algorithm examines a pair of digits, records the result, discards part of the working number and continues.

Similar principles appear when validating large incoming data streams: information is processed incrementally, while the program retains enough state to make the next decision.

The connection: `temp`, `backupTemp` and `numberPalindromeChecks` illustrate state preservation and progressive validation.

A production streaming validator would generally use a buffer or checkpoint mechanism rather than your particular division technique.



2. Manufacturing and quality-control inspection

Imagine inspecting a manufactured component at successive positions, comparing measurements against expected symmetry.

Your mirrored-digit comparisons have a direct mathematical analogue in checking whether corresponding positions on opposite sides of a component match.

The connection: Your movement formula $n - 1 - 2j$ is a way of navigating paired positions towards a centre.

Real systems could apply similar indexing to arrays of sensor readings or measured dimensions.

Graph search, checkpointing and software testing



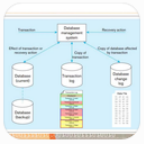
3. Graph and search algorithms

Some search algorithms continually change their active search region as they explore a problem.

Your experience with the loop counter catching a shrinking boundary is conceptually related to maintaining valid traversal positions while the search space changes.

The connection: Your `i` corrections respond to changing bounds.

However, graph algorithms normally maintain an explicit queue, stack or frontier rather than moving a loop counter backwards. This is an analogy, not an established equivalence.



4. Checkpointing and recovery

Your `backupTemp` preserves the current working number before `temp` is repeatedly divided.

This resembles a fundamental technique used in parsers, search procedures and transactional systems: preserve a known state before performing operations that alter a working state.

The connection: You separate exploratory changes from committed progress.

Database recovery is considerably more complex, but the principle of preserving recoverable state is genuinely shared.



5. Experimental algorithm development and software testing

This may be the most directly applicable lesson from how you developed the program.

You observed intermediate states, identified a specific failure, changed a condition and repeated the experiment.

That is closely related to differential testing, regression testing and invariant discovery.

The connection: Your debugging p ess uncovered rules before you had a formal mathematical explanation for them.

Appendix: Source and archive limitations

This document preserves the technical sequence and recreates principal diagrams, tables, formulas and code excerpts visible in the available conversation context. It is not a message-by-message transcript. It does not include original interactive image assets or every early message. The original source code and experiment ZIPs exist separately in the conversation's attachments.

Appendix A — Your questions and observations

Chronological transcription from the 16 blue chat boxes supplied. Wording and informal spelling are preserved as closely as possible. These are typeset transcriptions, not the original screenshot pixels.

01. Opening question

we have had so many conversations and looked at almost all my projects..
Did we ever discuss this completed code and advised you to analyse my mental processes.
Its checking for palindrome:

I also have another version (second attachment) which is for biginteger. It is very similar in principle and has all the other code commented to support long, integer also..

02. Semicolon observation

i kept system.exit(0) after the if statement and it reaches here with or without ;

its bizarre, but its verified

03. Independent discovery

my own digit navigation solely by looking at console output and adjusting the coe

04. Experimentation

i experimented with different values until output became correct..
And strangely it passes all test cases i have given

05. Testing the first implementation

i think running few test cases would be good, at time you provided several test cases...
But why dont you run my first code (non bigInteger) against several test cases

06. Leading-zero question

the 0 prefix is why i needed bigInteger, I belive it could handle 0 at front

07. Internal zeros

i think they should pass 0 in middle, run a few tests

08. BigInteger challenge

so now is the hard part for you..
lets look at the biginteger code, since it allows longer palindromes....
with exception of leading 0 which you correctly mentioned will fail
Explain again what my code is doing and why it passes.
It's not so easy, I have no idea..
I just followed the navigation of screen outputs and changed values to get outcome..

09. Investigating the semicolon

I think first lets see affect of removing the ; from the if statement matters...
The fact that single digit palindrome number failed is a simple failure on my part...
And I am sure I can find the logic to readily pass this test case

10. Controlled replacement

lets do one thing, you are suggesting the if statement can be removed and replaced with the statement:

firstDigit = temp.mod(new BigInteger("10"));

It was an excellent observation btw spotting the ;
it was not intentional at all

Lets see if this substitution produces the same outcome as having the if statement at end

11. Investigating movePosition

Excellent well spotted btw it was massive amount of luck that my code had adverse effect with ;
and passed..

Now we can look at the movePosition and then eventually all the other mysterious parts of my code that assist in creating this functional solution

12. Substitution test

excellent proceed with this substitution

13. Preserve original code

Excellent, so keep code intact and investigate the next mysterious variable or section

14. Exclude known single-digit cases

Before we move on, the original incorrect 100 from 10,000
Were they single digit test cases..
If its case, we should exclude them from testing since we know the code is already missing logic for these

15. Loop-counter observation

excellent proceed and strangely the observations you made about inner and outer loop catching up, its the sort of comments I made during development.. So I experienced same things..

Please proceed

16. Closing reflection and real-world applications

Excellent work and I appreciate the m==0 unreachable.
It has to be remembered that if you look at the non BigInteger code, I did not have a do while loop..
This is perhaps why this sort of issue has arisen....

And also the ; on if statement, I can replace that block..

It's much appreciated that you broke down my peculiar challenge..
The only next question would be is there anything you derived from your findings that maps it closer to a real life use

Appendix B — Coverage check

How the original questions map to the existing illustrated analysis:

Original method and mental processes
§1, §15; supplemental context below — Partially covered: early cognitive discussion was more detailed than this archive

Non-BigInteger version and tests
§1, §10, §11 — Covered in overview; detailed early tests are not reproduced

Leading zero versus internal zeros
§10; supplemental context below — Internal zeros covered; leading-zero distinction clarified below

Semicolon and unconditional assignment
§2, §10, §11 — Covered

Single-digit / do-while issue
§11, §16 — Covered

BigInteger digit navigation
§3–5, §9, §14 — Covered

movePosition / hasAdjust
§4, §7–8 — Covered

Loop-counter corrections
§5–6, §14 — Covered

Test-driven experiments
§6–7, §12–13 — Covered

Real-world relevance
§15 — Covered

Two useful clarifications

Leading zeros: a positive BigInteger does not retain the textual width of an input such as 00100; numeric conversion removes leading zeros. This differs from internal zeros, which are preserved as digits within the number. The PDF primarily tests and explains internal-zero handling.

Mental-process analysis: the main PDF focuses on technical behaviour and experimental findings. Your initial request also asked what the development process revealed about your thinking. The questions in Appendix A preserve that motivation, but the earlier cognitive analysis is not fully reproduced here.

Editorial note: The original 13-page illustrated archive has otherwise been retained. Section 14 now displays properly typeset equations; the question appendix was added after the existing document.