

Repeating Decimals to Fractions

Published by Matt in Java ▾

math numbers strings

Performing division on a fraction often results in an infinitely repeating decimal.

```
1/3=.3333333... 1/7=.142857142857...
```

Create a function that takes a decimal in string form with the repeating part in parentheses and returns the equivalent fraction in string form and in lowest terms.

Examples

```
fractions("0.(6)") → "2/3"  
fractions("1.(1)") → "10/9"  
fractions("3.(142857)") → "22/7"  
fractions("0.19(2367)") → "5343/27775"  
fractions("0.1097(3)") → "823/7500"
```

Notes

N/A

I have one query.... when I started the code I realised that I was limited to a 19 digit precision. I wonder if I now decided to change the whole code to BigInteger... would it allow me to explore repetends..... since in the original challenge, we can see that the focus was on this aspect.... I am not after any coding suggestions... but perhaps I can investigate more with number theory with a wider precision... What are your thoughts... And also the impact on performing divisions on BigInteger when reducing fractions...

Thought for 1m 9s

Short answer:

Yes – moving to **BigInteger/BigDecimal** lets you get out of the 19-digit cage and really play with repetends – but the way you explore them should shift to number-theory style (remainders, modular arithmetic), not “just do more double divisions”. And using BigInteger for reduction is mathematically perfect, with the main “impact” being performance, not correctness.

Let me break it down.

1. Why you hit the 19-digit wall

In your current code:

- You're using long for numerator/denominator $\Rightarrow \max \approx 9 \times 10^{18}$ (19 digits).
- You're also tied to double / LIMITPRECISION = 19 for decimal strings.

So even if the *true* repetend is very long (e.g. 1/97 has a 96-digit cycle), you're effectively seeing it chopped to 18–19 digits and then working with that truncated view.

That's fine for an **exploratory prototype**, but it means:

- you can't safely explore very large denominators;
- and you're mixing "actual math" with "what fits in 19 digits".

2. What changes if you move everything to BigInteger?

a) Fraction representation

If you store:

- numerator : BigInteger
- denominator : BigInteger

then:

- No fixed 19-digit limit anymore.
- You can reduce *any* rational a/b exactly using gcd in BigInteger land.
- You can represent denominators like 10^{100} or primes with 50+ digits.

So purely as a **fraction engine**, you're no longer constrained.

b) But decimals themselves are still infinite when repeating

Even with BigInteger, a decimal like:

- $1/7 = 0.142857142857142857\dots$

still has **infinitely many digits**. BigInteger doesn't magically let you "store infinite strings"; it just lets you keep doing exact integer arithmetic as long as you like.

So to "explore repetends", the trick is:

Stop thinking "more double precision" and start thinking "**remainders and modular arithmetic**".

That's exactly where number theory comes in.

3. Number-theory way to explore repetends (perfect for BigInteger)

Key facts:

Let a/b be a reduced fraction in base 10, with $\gcd(b, 10) = 1$ (i.e., b has no factors 2 or 5).

1. Its decimal expansion is **purely repeating**.
2. The length of the repetend is the **multiplicative order of 10 modulo b**:
the smallest $k \geq 1$ such that
 $10^k \equiv 1 \pmod{b}$.

This definition is 100% integer-based – no floating point at all.

With BigInteger you can:

- hold very large b ;
- compute $10^k \bmod b$ exactly;
- find the smallest k where the remainder cycles back to 1.

So yes: **BigInteger is excellent for exploring repetends**, because all the interesting work is done in modular arithmetic, which is entirely integer-based.

You never need to actually *print* all k digits of the repetend to “know” its length.

4. Repeating-decimal \rightarrow fraction formula (fits BigInteger perfectly)

For the original challenge you showed (e.g. "0.19(2367)" \rightarrow "5343/27775"), the usual number-theory formula is:

Let the decimal be:

I . N (R)

I = integer part

N = non-repeating fractional part (length n)

R = repeating block (length r)

Then the fraction is:

$$(\text{integer(INR)} - \text{integer(IN)}) / (10^n (10^r - 1))$$

Everything in that formula is just:

- converting strings of digits to big integers,

- exponentiating 10,
- subtracting, and
- reducing with gcd.

So if you switch your “engine” to BigInteger:

- you can handle **arbitrarily long** repeating blocks in parentheses;
- you can produce an **exact fraction** in lowest terms;
- you never rely on a 19-digit truncated double.

That’s *exactly* in line with the original challenge’s spirit.

5. Impact of using BigInteger for fraction reduction & division

You asked specifically:

And also the impact on performing divisions on BigInteger when reducing fractions...

There are **two different “division” contexts** to keep clear:

(1) Reducing fractions (what you currently do with divisors / GCF)

For reduction, the math is:

- Find $g = \text{gcd}(\text{numerator}, \text{denominator})$
- Set $n' = \text{numerator} / g$, $d' = \text{denominator} / g$

With BigInteger:

- gcd is implemented via fast Euclidean algorithms – very efficient even for large numbers.
- The divisions `numerator.divide(g)` and `denominator.divide(g)` are **exact integer divisions**, no rounding.

So:

-  Conceptually: BigInteger is **perfect** for this.
-  Practically: It’s slower than long, but still very acceptable up to thousands of digits.

In fact, this is what serious math libraries do internally—**all rational simplification is BigInteger-based**.

(2) Computing decimal expansions (long division)

If you actually want digits of the decimal expansion using BigInteger, the usual pattern is:

- Track remainder r , start with $r = \text{numerator} \% \text{denominator}$.
- At each step:
 $r = r * 10$, $\text{digit} = r / \text{denominator}$, $r = r \% \text{denominator}$.
- Digits string grows as you go.

To detect when it repeats:

- Store each remainder in a map (remainder \rightarrow position).
- When a remainder repeats, the digits from that earlier position form the repetend.

Again, all integer operations; BigInteger can do this exactly.

Impact:

- Memory/time grow with the length of the repetend and the size of the denominator.
- For “toy exploration” (denominators up to, say, a few thousand digits) it’s fine;
- for truly massive numbers (millions of digits) it becomes impractical, but that’s a theoretical, not a language limit.

6. So, should you switch to BigInteger if you’re curious about repetends?

From what I’ve seen of your style and interests: **yes**, it’s a very natural evolution.

What switching to BigInteger gives you:

- No more arbitrary 19-digit ceiling on numerators/denominators.
- You can study:
 - length of repetends for large denominators,
 - distribution of period lengths,
 - primes where 10 is a primitive root (full period $p-1$), etc.
- You can implement the “string with parentheses” challenge in a mathematically pure way:
 - no floating point / double at all,
 - everything via digit strings \rightarrow BigInteger.

What it *doesn't* give you:

- Infinite decimals “for free” – you still have to decide how many digits you want to generate or use the number-theory view (order modulo b) rather than pure digit expansion.

7. How this compares to what you've already done

Your current code:

- Mixes long/double with protection heuristics (truncate to 9 digits, LIMITPRECISION, regex on 9 recurring digits).
- Is great as an **exploratory engine under tight resource limits** (online IDE).

A BigInteger-based engine:

- Would be more like a **mathematical laboratory** for:
 - exact rational representation,
 - repetend lengths,
 - terminating vs repeating classification with no compromise,
 - analysing very large denominators.

And because you're *not* asking for code, just conceptually:

- BigInteger is **the right mathematical tool** if your next step is to “really investigate repetends” instead of “approximate them within 19 digits”.

If you'd like, I can next:

- outline (conceptually, not as code) how you'd structure a “Repetend Lab” around:
 - (a) the parentheses notation (0.19(2367) style), and
 - (b) the $1/n \rightarrow$ repetend length exploration using multiplicative order,

so you have a mental model of your *next* project without touching implementation.

what areas of the number theory can i explore moving into BigInteger/BigDecimal

Thought for 38s

You're asking exactly the right question here 😊

Switching to **BigInteger / BigDecimal** doesn't just "make numbers bigger" – it changes *what kind of maths you can realistically play with*. You stop being limited by 19-digit doubles and start doing "proper" number theory experiments.

Let me group some areas that fit **very naturally** with what you've already built.

1. Repeating Decimals & Multiplicative Order (your natural next step)

You're already halfway into this topic.

Core idea

For a reduced fraction $\frac{a}{b}$ in base 10 with $\gcd(b, 10) = 1$:

- The decimal expansion is **purely repeating**.
- The **length of the repeating block** (repetend) is the smallest k such that:

$$10^k \equiv 1 \pmod{b}$$

That "smallest k " is called the **multiplicative order** of 10 modulo b .

What BigInteger lets you do

With BigInteger you can:

- Explore **huge denominators** (b with 30, 50, 100+ digits).
- Compute $10^k \pmod{b}$ exactly, and find multiplicative order.
- Study:
 - How period length grows with b
 - For which primes p you get full length $p-1$ (these are **full reptend primes**)
 - How digit patterns look in long repetends (pseudo-random vs structured)

This is basically turning your current "debug decimal/fraction tool" into a **repetend laboratory**.

Project ideas:

- For all primes up to some bound, compute period length of $1/p$ and log:
 - p , period length, period length $\div (p-1)$, etc.
- Look for primes where period = $p-1$ (full reptend primes).
- Compare different bases (base 2, base 3, base 10) – same denominator, different repetend behaviour.

2. Decimal String with Parentheses → Exact Fraction (exact version of your challenge)

The original challenge you showed:

- "0.6(6)" → 2/3
- "3.(142857)" → 22/7
- etc.

There's a beautiful **pure integer** formula behind this:

If your decimal is:

I.N(R)

I = integer part

N = non-repeating fractional digits (length n)

R = repeating block (length r)

Then the fraction is:

```
int(INR)-int(IN)10n(10r-1)\frac{\text{int}(INR) - \text{int}(IN)}{10^n(10^r - 1)}10n(10r-1)int(INR)-int(IN)
```

Everything there is **just integers** – perfect for BigInteger:

- parse IN, INR as BigInteger
- compute 10^n , $10^r - 1$ as BigInteger
- subtract, divide by gcd for lowest terms

This lets you:

- Handle **arbitrarily long** repeating blocks ((2367...long...))
- Avoid doubles and LIMITPRECISION entirely
- Create a *clean*, mathematically exact version of the “parentheses input” problem.

You could explore:

- For random repeating strings (R), how “complicated” is the resulting fraction?
- Compare different constructing patterns: single digit rep, two-digit blocks, primes, palindromes, etc.

3. Modular Arithmetic & Remainder Cycles (your “state machine” mindset)

The repetend process is actually a **remainder state machine**:

- Start with remainder $r_0 = 1$ ($r_0 = 1$ for $1/b$).
- Next remainder: $r_{n+1} = 10 \cdot r_n \bmod b$
- The digits you see are $r_n * 10 / b$ truncated.

This is just a **cycle in a finite state graph** of size b .

With BigInteger you can:

- Explore remainder graphs for big b .
- Track cycle lengths, pre-period lengths, etc.
- Compare structure between different denominators.

This connects nicely to the way you think in other projects: “coverage of states / transitions” on an array.

4. Continued Fractions & Best Rational Approximations

Now bring in **BigDecimal** for high precision approximations.

Continued fractions let you:

- Convert a (possibly irrational) real number (π , e , $\sqrt{2}$, etc.) into a sequence of **best rational approximations** (convergents).
- Each convergent is a fraction p/q with p, q naturally represented as BigInteger.

Things you can explore:

- Approximate **π** or **e** to, say, 100 or 200 digits (BigDecimal), then:
 - Compute convergents via continued fraction expansion.
 - See famous approximations appear ($22/7$, $355/113$).
- Study how good each convergent is:

$|x - pq| \leq |x - \frac{p}{q}| \leq \frac{1}{q^2}$

and how the error relates to q^2 .

This is a brilliant playground combining:

- BigDecimal (for high-precision real values), and

- BigInteger (for convergent numerators/denominators).

5. Diophantine Equations (e.g. Pell's Equation)

A classic area where BigInteger shines:

Pell's equation:

$$x^2 - Dy^2 = 1$$

for integer D that isn't a perfect square.

- Solutions (x, y) can grow **enormously** – way beyond 64-bit.
- A standard method uses continued fractions for \sqrt{D} .

BigInteger lets you:

- Iteratively compute solutions where x and y have dozens or hundreds of digits.
- Log and explore patterns of growth.

You'd be doing:

- Square, multiply, subtract with BigInteger
- Use your rational/continued fraction intuition to trace how these solutions are generated.

6. Rational Reconstruction & Error Tolerant Matching

You already do a kind of “is this decimal really that fraction?” check.

With BigInteger/BigDecimal you can turn that into a more number-theoretic game:

- Given a decimal truncated to N digits, find **all fractions a/b** (with b below some bound) whose decimal expansion begins with that prefix.
- That's like:
 - choose bound on b (search space),
 - for each candidate fraction a/b, compare its decimal prefix,
 - or use more clever rational reconstruction techniques.

You'd learn about:

- Bounding denominators
- Error intervals (the decimal prefix defines an interval of possible numbers)

- How many rational numbers fit into that interval with “small” denominators.

7. Prime-Related Experiments (linked to repetends)

Primes and repetends come together naturally:

- If p is prime and $\gcd(p, 10) = 1$, the period of $1/p$ divides $p-1$.
- When the period equals $p-1$, p is a **full reptend prime** in base 10.

With BigInteger you can:

- Use probabilistic primality tests (isProbablePrime) to handle large p .
- For chosen primes p , compute the period of $1/p$ (via multiplicative order).
- Collect statistics:
 - proportion of primes with full period,
 - long vs short repetends.

This marries your interest in:

- prime lists
- repetend lengths
- factorial-scale exploration (but in a controlled, number-theory way).

8. BigDecimal Rounding & Error Analysis

Finally, BigDecimal itself is a good tool for **numerical experiments**:

- Different rounding modes (HALF_UP, HALF_EVEN, etc.).
- How rounding interacts with rational numbers vs irrationals.
- Simulating “financial style” cumulative rounding error over many operations.

This is more numeric analysis than pure number theory, but it’s very close to:

- Your interest in precision
- “Session safe” decisions
- Rounding vs exactness for real-world quantities.

Putting it together

If I had to pick **3 next projects** that fit you and your current code best:

1. Repetend Lab (multiplicative order)

- BigInteger denominators, compute repetend lengths of $1/b$ in base 10.
- Identify full reptend primes, log samples, analyse structure.

2. Parentheses Decimal Challenge – exact edition

- Implement the number-theory formula for $I.N(R) \rightarrow$ fraction using BigInteger only.
- No doubles, no LIMITPRECISION – pure rational arithmetic.

3. Continued Fractions Explorer with BigDecimal

- Take constants like π , e , $\sqrt{2}$ (via BigDecimal), generate convergents using BigInteger, and analyse approximation quality.

Each of those uses the same “Amit-style”:

- heavy introspection,
- debug logging,
- stepwise reduction / exploration,

but now powered by **truly unbounded integer precision** instead of being squeezed into 19 digits.

If you decide which of these directions you feel most curious about, I can talk through that one in more depth (conceptually – data structures, flow, and what to measure/print), still without any actual code.

first briefly explain in which context they would be useful in real life / industry.. Or will it be for educational purposes only

Thought for 38s

Good question – a lot of this *sounds* like “pure math”, so it’s natural to ask: *is this just educational, or does industry actually care?*

I’ll go through each area we talked about and mark it as:

- **(P)** = has clear practical / industry uses
- **(M)** = mixed: some real uses, but often appears via libraries / theory
- **(E)** = mostly educational / research, but still great for your mental toolkit

1. Repeating decimals & multiplicative order (repetend length) — (M)

Where it shows up in practice:

- **Cryptography & modular arithmetic:**

Multiplicative order (the “ k where $10^k \equiv 1 \pmod{n}$ ”) is the *same type* of concept used when working with groups in modular arithmetic – which underlies RSA, Diffie-Hellman, and many public-key schemes. In industry you don’t usually compute repetends of $1/n$ directly, but you absolutely *do* care about multiplicative orders and exponent cycles.

- **Random-looking sequences / PRNG building blocks:**

Long repetends in $1/n$ produce digit sequences that “look random-ish”. Similar ideas (period length of a recurrence) are used when designing linear congruential generators and other pseudo-random processes.

Reality check:

You’re unlikely to write “repetend length of $1/97$ ” in a banking app, but understanding orders and cycles is genuinely useful in cryptography, coding theory, and algorithm design. So: **mixed – deep theory behind real systems.**

2. Parentheses decimal → exact fraction (e.g. "0.19(2367)") — (P)

Directly useful places:

- **Finance / accounting software** (BigDecimal style):

Systems often want to store exact rational values instead of floats to avoid rounding bugs. Being able to say “this repeating decimal is exactly p/q ” lets you store the rational and only round at display time.

- **Scientific & engineering software:**

Rational representations avoid cumulative floating-point error in formulas that are naturally fractional (ratios, scaling factors, probabilities).

- **Computer algebra systems:**

Internally, tools like Wolfram Research or The MathWorks need exactly this conversion when a user types something like $0.1(6)$ and then asks to simplify or combine with other expressions.

Verdict:

This is **very practical** – the code you’d write here could be part of a real-world numeric library.

3. Modular arithmetic & remainder cycles — (P)

You're already half-there mentally with your “state machines” and loops.

Real-life contexts:

- **Cryptography:**

Almost everything in classic public-key crypto is modular arithmetic with big integers: exponentiation, inverses, residue classes, orders of elements, etc.

- **Hashing, checksums, error detection:**

Many checksum algorithms use modular arithmetic to detect corruption in packets or files.

- **Scheduling / cyclic systems:**

“Every 7 days”, “every 13 ticks”, clock arithmetic, round-robin scheduling – all of that is congruences and cycles.

Verdict:

Very **practical** and widely used under the hood.

4. Continued fractions & best rational approximations — (M)

Practical links:

- **Digital signal processing / sample rate conversion:**

Ratios like $44.1\text{kHz} \rightarrow 48\text{kHz}$ are converted into reduced fractions (e.g. 147/160) for efficient resampling. Continued fractions explain why those approximations are good and help find others.

- **Control systems & physics simulations:**

Time steps and frequency ratios are often chosen to be rational approximations of “nice” irrational values so that errors stay controlled and cycles line up.

- **Graphics / media timing:**

Frame rates, aspect ratios, and animation timings are frequently rational (e.g. 24000/1001 fps).

Also very educational:

Continued fractions are a fantastic way to see how rationals approximate irrationals like π , e , $\sqrt{2}$ – great for intuition, and occasionally directly used in engineering contexts.

Verdict:

Mixed – in some fields it’s directly useful; otherwise it’s the theory behind how “clever approximations” are found.

5. Pell's equation & Diophantine equations — (E/M)

Mostly:

- **Theoretical / educational:**

Classical number theory topic; used in olympiad problems, research, and as a gateway to algebraic number theory.

Indirect practical relevance:

- Algorithms and ideas from this area appear in:
 - advanced crypto research,
 - algebraic number packages inside CAS,
 - some coding theory.

But if you're writing business software, you won't get a Jira ticket saying "please solve Pell's equation". 😊

Verdict:

Primarily **educational / research**; great for deepening your toolkit, less so for direct industry work.

6. Rational reconstruction & "exact from approximate" — (M/P)

You're already doing a softer version of this when you compare decimal and fraction.

Real-world-ish uses:

- **Computer algebra systems:**

When they do numeric computations (with rounding) and then try to recover the exact rational that would produce that number.

- **Numerical → symbolic transitions:**

E.g. a physics simulation gives 0.3333333333 and you want to infer it's exactly 1/3, not a random float.

- **Data pipelines with approximated values:**

When importing CSVs with decimals that were originally rational, you might want to reconstruct the "intended" fraction (up to some denominator bound).

Verdict:

Mixed, but can be genuinely practical in tools that sit between symbolic math and numeric computation.

7. Prime-related experiments, full reptend primes, etc. — (E/M)

Practically:

- **Crypto** absolutely needs primes and modular exponentiation – but not specifically “full reptend primes in base 10”.
- The same maths (orders, primitive roots) is used when designing groups with good properties.

So:

- Exploring “which primes give full-length repetends in base 10” is mostly **interesting number theory**, not a usual product requirement.
- But the way you *compute* and reason about it (orders, cycles, residues) *is* close to the thinking used in crypto and finite field design.

Verdict:

Mostly **educational / exploratory**, but conceptually close to real cryptographic mathematics.

8. BigDecimal rounding & error analysis — (P)

This one is *very* real-world:

- **Banking & accounting:**
Legal requirements on how to round (HALF_UP, HALF_EVEN, etc.), how many decimal places to keep, and how to avoid bias in long-running calculations.
- **Trading / risk systems:**
Small rounding errors can become material over huge volumes.
- **Scientific & engineering simulations:**
Rounding strategy can affect stability and reproducibility.

Here, BigDecimal is *exactly* what industry Java code uses, and analysing rounding effects is a practical skill.

Verdict:

Strongly **practical**.

Overall: is this “just educational”?

Definitely not. Rough grouping:

- **Strongly practical (P):**

- Parentheses decimal → exact fraction (rational representation)
- Modular arithmetic / remainder cycles
- BigDecimal rounding & error analysis
- **Mixed practical + theoretical (M):**
 - Repeating decimals & multiplicative order
 - Continued fractions & approximations
 - Rational reconstruction
 - Prime / repetend experiments (conceptually close to crypto)
- **Mostly educational (E):**
 - Pell's equation and more exotic Diophantine stuff

Given how you think (engines, logging, state exploration), you're actually sitting right in the sweet spot between **education, tooling, and real-world math software**. A BigInteger/BigDecimal project of yours could easily be framed as:

- a mini “rational arithmetic engine” for finance / numeric tools, with
- extra hooks and logs for **teaching** and **exploration** on top.