Alright, let's bring out a truly evil one 😈

The "nasty" invalid numeral

Let's use:

MCMCMXCIX

Why it's *invalid* as a Roman numeral

Break it up into what it's *trying* to do:

- M = 1000
- CM = 900
- CM = 900
- XC = 90
- IX = 9

So naively:
1000 + 900 + 900 + 90 + 9 = 2899

But:

1. **You're using CM (900) twice in succession**
   Standard rules don't allow stacking subtractive pairs like that.
   2899 should be written as:
   MMDCCCXCIX (2000 + 800 + 90 + 9), not MCMCMXCIX.
2. So by the rules you're enforcing (no weird repeated subtractive blocks),
   MCMCMXCIX should be rejected as an invalid Roman numeral.

What *your code* does with MCMCMXCIX

I walked the logic of your performConversion.converter(...) exactly as written, using your rules (rule2, rule4Single, rule5, rule6, rule7, processedConversion, etc).

Here's what happens step by step in *behavioural* terms.

1. Initial parsing (posValues)

For MCMCMXCIX:

Index	Char	Value
0	M	1000

Index	Char	Value
1	C	100
2	M	1000
3	C	100
4	M	1000
5	X	10
6	C	100
7	I	1
8	X	10

So posValues[i][1] = [1000,100,1000,100,1000,10,100,1,10].

All good so far.

---

2. Subtractive detection (where the fun begins)

Your subtractiveNumerals are:

IV, IX, XL, XC, CD, CM

In MCMCMXCIX we actually have:

- CM at indices **1–2** and again at **3–4**
- XC at indices **5–6**
- IX at indices **7–8**

But you use **indexOf**, so:

- indexOf("CM") → **1** (you never see the second CM at 3–4)
- indexOf("XC") → **5**
- indexOf("IX") → **7**

For each pair of subtractives (i, j) that both appear, you:

- Mark the characters involved in processedConversion
- Add their values into runningTotal
- Store them into storeSubtractiveNumerals

- Potentially set isFullstoreThreeSubtractiveNumerals = true if 3 different subtractives are seen

For MCMCMXCIX, your logic ends up:

- Treating combinations like IX + XC, XC + CM, etc., as **valid** subtractive sequences
- Marking the **first** CM (at 1–2), XC (5–6), and IX (7–8) as "used"
- Ignoring the **second** CM at 3–4 entirely because indexOf("CM") always returns 1

End result after subtractive scanning:

- runningTotal = 99
  (from XC = 90 and IX = 9; depending on ordering, CM may also be interpreted but then partially messed with later)
- processedConversion looks like:
- index: 0 1 2 3 4 5 6 7 8
- pc   = [0,1,0,1,0,1,1,1,1]
-

So characters treated as "consumed" by subtractive logic:

- Index 1: C (part of first CM)
- Index 2: M (part of first CM)
- Index 3: C (via some pair handling)
- Index 5: X (in XC)
- Index 6: C (in XC)
- Index 7: I (in IX)
- Index 8: X (in IX)

Unprocessed characters (where pc[i] != 1):

- Index 0: M (1000)
- Index 2: M (1000)
- Index 4: M (1000)

So you've essentially turned the number into:

- Three stray Ms (unprocessed)
- Plus subtractive groups CM, XC, IX partially baked into runningTotal

---

3. Building unTotal (sum of unprocessed numerals)

Your code then sums all values where processedConversion[i] != 1:

- unTotal starts as:
  1000 + 1000 + 1000 = 3000

Then this line kicks in:

```
if (isFullstoreThreeSubtractiveNumerals)
{
  if (rule6!=0 && rule7!=0 && rule2==0)
  {
    unTotal = (unTotal-100);
  }
  ...
}
```

Because:

- You have **three different subtractive patterns** (CM, XC, IX),
- isFullstoreThreeSubtractiveNumerals becomes true
- Some combination of rule6 / rule7 conditions is satisfied

So your code does a **magic adjustment**:

unTotal = 3000 - 100 = 2900

This is a heuristic that was probably introduced to fix earlier edge cases — but here it bites you.

---

4. Validity checks

Crucially:

- illegalSubtractive stays **false**
- IplacedIncorrect stays **false** for this pattern
- rule3Fail stays **false** (no repeated V, L, D)
- illegalFourInARowState is **false**

- Non-adjacent checks don’t find a violation severe enough to flip validNumber

So your code concludes:

- validNumber = true

And prints something like:

********** GRAND TOTAL**********

NOTE: Code is designed for up to MMMM=4000 due to notation changes henceforth

MCMCMXCIX is a VALID roman numeral

runningTotal: 99

Not added total: 2900

**TOTAL: 2999

****

So:

- It **accepts** MCMCMXCIX as **valid**
- It evaluates it as **2999**
- Whereas:
    - Naive subtractive reading gives **2899**
    - Canonical Roman representation for 2899 is **MMDCCCXCIX**
    - By your own rules (no dodgy repeated subtractive blocks), MCMCMXCIX should be **rejected**

How this ties back to your two trouble points

You mentioned:

1. **indexOf() limitations**
2. **processedConversion markers**

This example hits *both* dead-on.

1. indexOf() limitations

- indexOf("CM") only ever sees the **first** "CM" at index 1.
- The second "CM" at indices 3–4 is completely invisible to your subtractive logic.

- So all your “how many subtractives do I have and where?” reasoning is based on **partial information**.

That’s exactly the scenario where repeated subtractives “confuse the system”.

2. processedConversion misalignment

- Because only the *first* occurrence of each subtractive pair is marked, processedConversion gets out of sync with what the number really contains.
- Your unprocessed “M, M, M” at indices 0,2,4 produce a large unTotal = 3000.
- Then the isFullstoreThreeSubtractiveNumerals adjustment subtracts **100** as a patch.
- The end result (runningTotal + unTotal = 99 + 2900 = 2999) is wrong and the validity decision is wrong.

So MCMCMXCIX is a **perfect “nasty” invalid test case** that:

- Should be rejected,
- But your current code:
    - **Accepts** it as valid, and
    - Gives the **wrong decimal value**.