I'll pick **MCMXCIV** (which should be **1994**) because:

- It has **three subtractive pairs**: CM, XC, IV
- It's a classic "stress test" for subtractive logic

What your code does with MCMXCIV

1. Initial parsing

numeralsToString = "MCMXCIV"

You build:

Index: 0 1 2 3 4 5 6

Char: M C M X C I V

Value:1000 100 1000 10 100 1 5

So:

- inputtedNumerals = ['M','C','M','X','C','I','V']
- posValues = {{?,1000}, {?,100}, {?,1000}, {?,10}, {?,100}, {?,1}, {?,5}}
- processedConversion[] = all zeros initially
- runningTotal = 0, unTotal = 0

---

2. Subtractive detection (IV, IX, XL, XC, CD, CM)

Your subtractiveNumerals array:

{ "IV", "IX", "XL", "XC", "CD", "CM" }

In MCMXCIV you actually have:

- "CM" at index **1**
- "XC" at index **3**
- "IV" at index **5**

Now the fun part:

a) First important pairing: "IV" with "XC"

For i = "IV", j = "XC":

- indexOf("IV") = 5

- indexOf("XC") = 3
- Both are found ⇒ twoSubtractiveNotations = true
- Since 5 > 3, you go into the **“i after j”** branch:
    - You add both to storeSubtractiveNumerals → now it has "IV" and "XC"
    - You set:

 processedConversion[3] = 1 // 'X' of 'XC'

processedConversion[4] = 1 // 'C' of 'XC'

processedConversion[5] = 1 // 'I' of 'IV'

processedConversion[6] = 1 // 'V' of 'IV'

 You add their values to runningTotal:

-
    - runningTotal = 4 (IV) + 90 (XC) = 94
    -

So at this point, XC and IV are fully “claimed”.

b) Triple subtractive recognition

Then you also consider "IV" with "CM" in another pair, and "CM" with "XC" in others.

Through those pairings you fill:

storeSubtractiveNumerals = ["IV", "XC", "CM", ...]

isFullstoreThreeSubtractiveNumerals = true;

But **critically**:

- "CM" never gets its own runningTotal += 900
- Its positions [1,2] never get set in processedConversion by this stage

Because the relevant branch only *checks* ordering (checkComplete) and doesn’t actually mark or sum "CM" in this triple-subtractive situation.

So after subtractive loops:

- runningTotal = 94 ← counts XC + IV
- processedConversion:
- index: 0 1 2 3 4 5 6

- value: 0 0 0 1 1 1 1
-
- isFullstoreThreeSubtractiveNumerals = true

---

3. Main left-to-right pass (rules 2, 5, 6, 7)

You loop m from 1 to 6, but only process entries where processedConversion[m] != 1.

So only m = 1 and m = 2 (C and M) get processed.

m = 3..6 are skipped because they're part of XC and IV.

a) For m = 1 (C after M)

- posValues[1] = 100, posValues[0] = 1000
- Since 100 < 1000 and previous is unprocessed, you treat this as a **Rule 5** additive block:

 rule5 += 100 + 1000 = 1100

 You also detect that C is between two Ms in another check and accumulate for rule7:

- rule7 += (1000 - 100) = 900
-

b) For m = 2 (second M)

- You detect C(100) before M(1000) as a legal subtractive situation (Rule 6):
- processedConversion[1] = 1
- rule6 += (1000 - 100) = 900
-

So now:

- rule5 = 1100
- rule6 = 900
- rule7 = 900
- processedConversion:
- index: 0 1 2 3 4 5 6
- value: 0 1 0 1 1 1 1

-

---

4. unTotal – leftover numerals

You now sum all positions where processedConversion[i] != 1:

- Index 0: M → 1000
- Index 2: M → 1000

So:

unTotal = 1000 + 1000 = 2000

---

5. Triple subtractive adjustment

Because isFullstoreThreeSubtractiveNumerals == true and:

- rule6 != 0 (900)
- rule7 != 0 (900)
- rule2 == 0

You do:

unTotal = unTotal - 100;

Result:

unTotal = 2000 - 100 = 1900

That "−100" is one of your hacks to fix overcounting/overlap in triple-subtractive cases.

---

6. Validity + final total

You run your non-adjacent checks (Rule 7 style stuff) on pairs like:

- (M, M), (C, X), (M, C), (X, I), (C, V)

They all pass (left >= right in each 2-step window), so:

validNumber = true

Then you print the **grand total**:

runningTotal = 94 // XC + IV

unTotal = 1900 // the two Ms plus adjustments

TOTAL = 94 + 1900 = 1994

So your program ultimately outputs:

- "MCMXCIV is a VALID roman numeral"
- "runningTotal: 94"
- "Not added total: 1900"
- "**TOTAL: 1994"

✅ **Correct final decimal.**

---

7. The weird "OLD PROGRAMME LOGIC" totals

After that, you dump a lot of diagnostics:

- rule4: 0
- rule2: 0
- rule4Single: 0
- rule5: 1100
- rule6: 900
- rule7: 900
- threeInRow: 0
- Untotal: 1900

Then in the big if (!illegalSubtractive && ... && validNumber) you hit this branch:

```
if (unTotal!=0 && rule5!=0 && rule6!=0 && !hasProcessLegacyTotal)
{
   System.out.println("**Total6: " + (unTotal + rule6));
   hasProcessLegacyTotal = true;
}
```

That prints:

**Total6: 2800 // 1900 + 900

This doesn't match the *real* final answer; it's just your older experiment logic still echoing weird numbers. But your **actual "GRAND TOTAL"** is based on runningTotal + unTotal (1994), which is correct.

Bottom line for MCMXCIV

- Your code **accepts it as valid** ✅
- Your main total calculation prints **1994** ✅
- Some internal rule totals (rule5, rule6, rule7, Total6: 2800) are inconsistent and confusing, but they're effectively "legacy debug noise" now.

If you'd like, next I can do the *opposite* kind of test: run the same style of reasoning on a **nasty invalid** numeral like MCMCMXCIX and show you whether your code correctly rejects it or accidentally lets it through.