12. Integer to Roman

March 29, 2020 | 18.5K views

Average Rating: 4.63 (16 votes)

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

```
Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, two is written as `II` in Roman numeral, just two one's added together. Twelve is written as, `XII`, which is simply `X` + `II`. The number twenty seven is written as `XXVII`, which is `XX` + `V` + `II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral. Input is guaranteed to be within the range from 1 to 3999.

Example 1:

```
Input: 3
Output: "III"
```

Example 2:

```
Input: 4
Output: "IV"
```

Example 3:

```
Input: 9
Output: "IX"
```

Example 4:

```
Input: 58
Output: "LVIII"
Explanation: L = 50, V = 5, III = 3.
```

Example 5:

```
Input: 1994
Output: "MCMXCIV"
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Solution

Overview

In a lot of countries, Roman Numerals are taught in elementary school-level math. This has made them a somewhat popular "easy" interview question. Unfortunately though, this ignores the fact that *not everybody learned them in school*, and therefore a big advantage has been given to those who did. I suspect it's also difficult for a lot of us who *have* learned them previously to fully appreciate how much easier prior experience makes this question. While this is very unfair, and possibly very frustrating, keep in mind that the best thing you can do is work through this question and the related question Roman to Integer so that you don't get caught out by it in a real interview. In short, if you're here reading this, you've saved yourself from getting caught out by it! Thankfully, questions that rely on this kind of prior knowledge are few and far between.

Have a go at Roman to Integer first

The problem of converting a Roman Numeral to an Integer is simpler. Therefore, we suggest that you have a go at it first if you're finding this question difficult. This will allow you to become more familiar with the concept of Roman Numerals without the "ambiguity" issue that comes up in converting an integer to a Roman Numeral. When converting a Roman Numeral to an integer, there's only one sensible conversion.

Roman Numeral Symbols

Roman Numerals are made with `7` single-letter symbols, each with its own value. Additionally, the subtractive rules (as explained in the problem description) give an additional `6` symbols. This gives us a total of `13` unique symbols (each symbol is made of either 1 letter or 2).

M → 1000
CM → 900
D → 500
CD → 400
C → 100
XC → 90
L → 50
XL → 40
X → 10
IX → 9
V → 5
IV → 4
I → 1

An integer is represented as a Roman Numeral by finding symbols that add to its value.

Handling Ambiguity

One thing that can be a bit confusing if you're not familiar with Roman Numerals is knowing which representation is the "correct" one for a particular integer. For example, consider these possible ways of representing `140`. Which of these is correct?

L + L + XL	= 50 + 50 + 40	= 140
C + X + X + X + X	= 100 + 10 + 10 + 10 + 10	= 140
C + XL	= 100 + 40	= 140
XC + L	= 90 + 50	= 140
XL + XL + XL + X + X	= 40 + 40 + 40 + 10 + 10	= 140
C + X + X + V + V + V + V	= 100 + 10 + 10 + 5 + 5 + 5 + 5	= 140

The system we use to decide is to select the representation with the largest possible symbols, working from left to right. For example, the representations above with the largest symbol at the start are the ones starting with `C`.

C + X + X + X + X	= 100 + 10 + 10 + 10 + 10	= 140
C + XL	= 100 + 40	= 140
C + X + X + V + V + V + V	= 100 + 10 + 10 + 5 + 5 + 5 + 5	= 140

To decide which of these to go with, we look at the next symbol. Two of them have an `X`, which is worth `10`, and one of them has an `XL`, which is worth `40`. Because the `XL` is worth more, we go with that representation. Therefore, the representation for `140` is `CXL`.

This definition of Roman Numerals is, these days, the "most accepted". Interestingly, it still isn't an absolute standard, and throughout history, there have been many variants. If you're interested in math and history, we recommend checking out the Wikipedia article for your own interest.

Approach 1: Greedy

Intuition

Representing a given integer as a Roman Numeral requires finding a sequence of the above 13 symbols, where their corresponding values add up to the integer. This sequence must be in order from largest to smallest, based on symbol value. To remind you, these are the symbol values.

M → 1000
CM → 900
D → 500
CD → 400
C → 100
XC → 90
L → 50
XL → 40
X → 10
IX → 9
V → 5
IV → 4
I → 1

As explained in the overview, the representation should use the largest possible symbols, working from the left. Therefore, it makes sense to use a **Greedy** algorithm. A Greedy algorithm is an algorithm that makes the best possible decision at the current time; in this case taking out the largest possible symbol it can.

So to represent a given integer, we look for the *largest* symbol that fits into it. We subtract that, and then look for the largest symbol that fits into the remainder, and so on until the remainder is 0. Each of the symbols we take out are appended onto the output Roman Numeral string.

For example, suppose we need to make the number `671`.

The largest symbol that fits into `671` is `D` (which is worth `500`). The next symbol up, `CM`, is worth `900` and so is too big to fit. Therefore, we now have the following.

```
Roman Numeral so far: D
Integer remainder: 671 - 500 = 171
```

We now repeat the process with `171`. The largest symbol that fits into it is `C` (worth `100`).

```
Roman Numeral so far: DC
Integer remainder: 171 - 100 = 71
```

Repeating this with `71`, we find the largest symbol that fits in is `L` (worth `50`).

```
Roman Numeral so far: DCL
Integer remainder: 71 - 50 = 21
```

For `21`, the largest symbol that fits in is `X` (worth `10`).

```
Roman Numeral so far: DCLX
Integer remainder: 21 - 10 = 11
```

For `11`, the largest symbol that fits in is again `X`.

```
Roman Numeral so far: DCLXX
Integer remainder: 11 - 10 = 1
```

Finally, the `1` is represented with a `I` and we're done.

```
Roman Numeral so far: DCLXXI
Integer remainder: 1 - 1 = 0
```

In pseudocode, this algorithm is as follows.

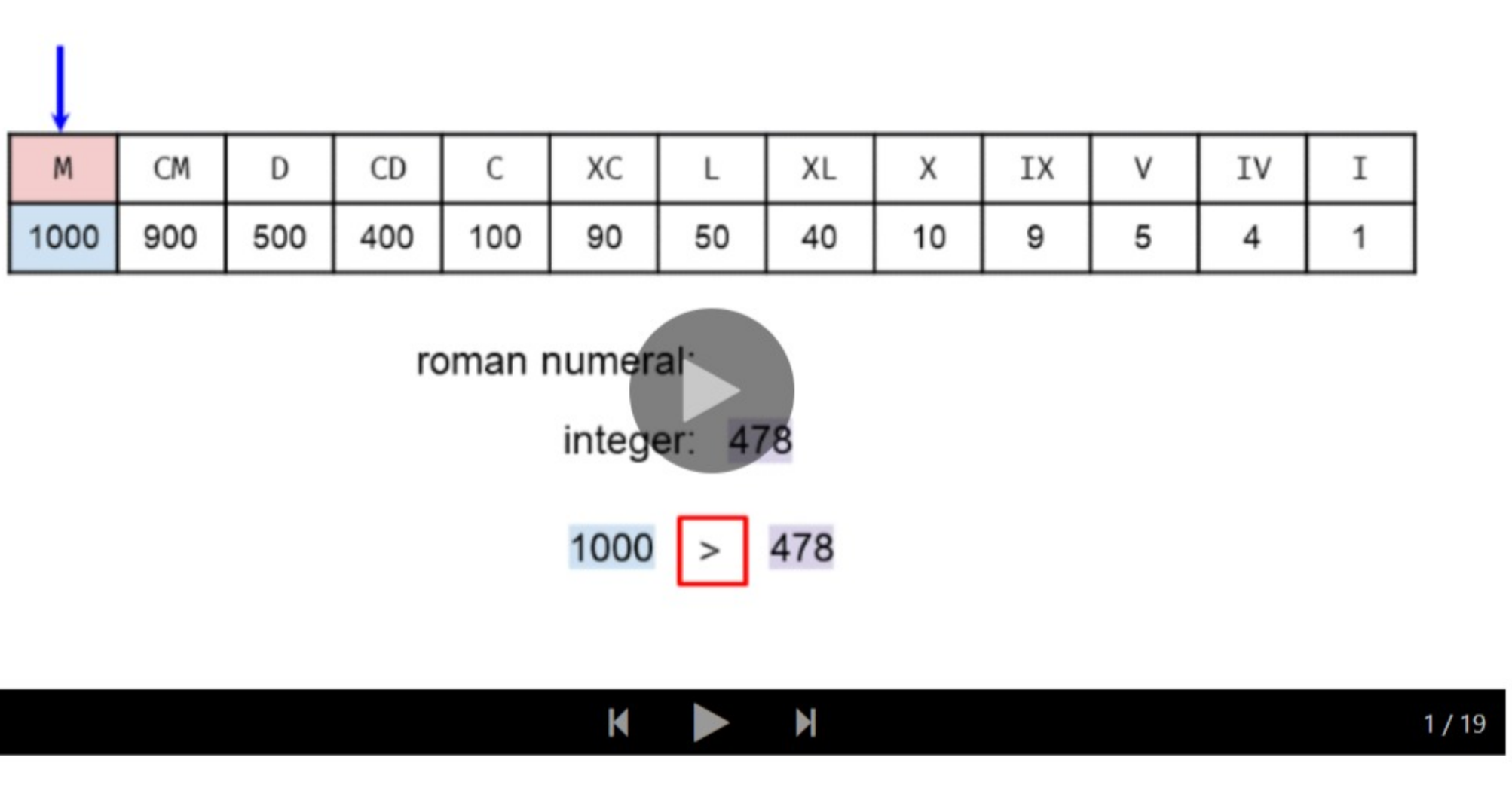
```
define function to_roman(integer):
    roman_numeral = ""
    while integer is non-zero:
        symbol = biggest valued symbol that fits into integer
        roman_numeral = concat roman_numeral and symbol
        integer = integer - value of symbol
    return roman_numeral
```

The cleanest way to implement this in code is to loop over each symbol, from largest to smallest, checking *how many* copies of the current symbol fit into the remaining integer.

```
define function to_roman(integer):
    roman_numeral = ""
    for each symbol from largest to smallest:
        if value of symbol is greater than integer:
            continue
        symbol_count = number of times symbol value fits into integer
        repeat symbol_count times:
            roman_numeral = concat roman_numeral and symbol
        integer = integer - (value of symbol * symbol_count)

    return roman_numeral
```

Here's an animation showing this algorithm run on the number `478`.



Algorithm

Java Python Copy

```python
digits = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"), (90, "XC"),
          (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")]

def intToRoman(self, num: int) -> str:
    roman_digits = []
    # Loop through each symbol.
    for value, symbol in digits:
        # We don't want to continue looping if we're done.
        if num == 0: break
        count, num = divmod(num, value)
        # Append "count" copies of "symbol" to roman_digits.
        roman_digits.append(symbol * count)
    return "".join(roman_digits)
```

Complexity Analysis

- Time complexity : $O(1)$.

  As there is a finite set of roman numerals, there is a hard upper limit on how many times the loop can iterate. This upper limit is `15` times, and it occurs for the number `3888`, which has a representation of `MMMDCCCLXXXVIII`. Therefore, we say the time complexity is constant, i.e. $O(1)$.

- Space complexity : $O(1)$.

  The amount of memory used does not change with the size of the input integer, and is therefore constant.

Approach 2: Hardcode Digits

Intuition

Please don't panic and assume you need to memorize the values in this approach. The first approach should be fine, and in-fact has the added bonus of being more flexible if we were to extend the Roman Numeral symbol set to have symbols over 1000. This second approach is only included for completeness. Do try to understand how we derived this approach, though.

An interesting observation that can be made is that each of the digits in the integer's decimal representation can be treated *independently* when converting the integer into a Roman Numeral. Notice that all of the symbols can be split into groups based on their highest factor out of `1000`, `100`, `10`, and `1`.

thousands →	M
hundreds →	CM, D, CD, C
tens →	XC, L, XL, X
ones →	IX, V, IV, I

While the number is at least `1000`, an `M (1000)` will be appended to the output and `1000` will be subtracted from the integer. The other symbols won't even be considered until the number is below `1000`. Additionally, the `M (1000)` s cannot represent any lower part of the number. Therefore, we can represent the thousands digit of the integer entirely with `M (1000)` s.

Now, assume we have a remainder of between `100` and `999`. The next symbols considered are those in the hundreds row. The highest symbol that could fit in right now is `CM (900)`, and the lowest is `C (100)`. None of the symbols in this range can possibly modify the tens or ones. As long as the remainder is still above `100`, we can still take at least `C (100)` out of it. This means that we'll only be subtracting symbols from the hundreds row for as long as the number is at least `100`.

The same argument applies for the tens, and then the ones.

We can, therefore, work out what the representation for each digit, in each place, is. There are only 34 of them; `0`, `1`, `2`, `3` and `4` for the thousands column, and `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` and `9` for each of the hundreds, tens, and ones. So with a pencil, paper, and some patience, you can hopefully work out the representation for each of these possibilities and hardcode them. Then, converting an integer to a Roman Numeral will require breaking the integer into digits and appending the relevant representation for each digit.

	thousands	hundreds	tens	ones
0	-	-	-	-
1	M	C	X	I
2	MM	CC	XX	II
3	MMM	CCC	XXX	III
4	-	CD	XL	IV
5	-	D	L	V
6	-	DC	LX	VI
7	-	DCC	LXX	VII
8	-	DCCC	LXXX	VIII
9	-	CM	XC	IX

Getting each digit of the number can be done using the modulus and division operators. The division operator removes the digits *below* the place we want, and the modulus operator removes the digits from *above*. This simply leaves the digit we want.

```
thousands_digit = integer / 1000
hundreds_digit = (integer % 1000) / 100
tens_digit = (integer % 100) / 10
ones_digit = integer % 10
```

Then, we can simply look these up in the hardcoded table, and append the results together!

Algorithm

The cleanest way to go about it in code is to have 4 separate arrays; one for each place value. Then, extract the digits, look up their symbols in the relevant array, and append them all together.

Java Python Copy

```python
def intToRoman(self, num: int) -> str:
    thousands = ["", "M", "MM", "MMM"]
    hundreds = ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"]
    tens = ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"]
    ones = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"]
    return thousands[num // 1000] + hundreds[num % 1000 // 100] + tens[num % 100 // 10] + ones[num % 10]
```

Complexity Analysis

- Time complexity : $O(1)$.

  The same number of operations is done, regardless of the size of the input. Therefore, the time complexity is constant.

- Space complexity : $O(1)$.

  While we have Arrays, they are the same size, *regardless of the size of the input*. Therefore, they are constant for the purpose of space-complexity analysis.

The downside of this approach is that it is inflexible if Roman Numerals were to be extended (which is an interesting follow-up question). For example, what if we said the symbol `H` now represents `5000`, and `P` now represents `10000`, allowing us to represent numbers up to `39999`? Approach 1 will be a lot quicker to modify, as you simply need to add these 2 values to the code without doing any calculations. But for Approach 2, you'll need to calculate and hardcode ten new representations. What if we then added symbols to be able to go up to `399,999,999`? Approach 2 becomes more and more difficult to manage, the more symbols we add.