## 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 [Integer to Roman](https://leetcode.com/problems/integer-to-roman/) so that you don't get caught out by it in a real interview.

Can we assume the input is valid?

Yes. Here on Leetcode, you can make that assumption because you *haven't* been told what to do if it *isn't*.

In a real interview, this is a question you should ask the interviewer. Don't ever assume without asking in a real interview that the input has to be valid.

Is there only one valid representation for each number?

This is more relevant to the other question, [Integer to Roman](https://leetcode.com/problems/integer-to-roman/), however we'll still briefly look at it now.

Given that the representation for 3 is III, it could seem natural that the representation for 15 is VVV, because that would be 5 + 5 + 5. However, it's actually XV, which is 10 + 5. How are you even supposed to know which is correct?

The trick is to use the "biggest" symbols you can. Because X is bigger than V, we should use an X first and then make up the remainder with a single V, giving XV.

We'll talk more about this in the [Integer to Roman](https://leetcode.com/problems/integer-to-roman/) article. This question is a lot simpler because there's only one logical way of converting from a *Roman Numeral to an Integer*. This is also why this question is labelled as "easy", whereas the other is labelled as "medium".

A few more examples

If you're not very familiar with Roman Numerals, work through these examples and then have another go at writing your own algorithm before reading the rest of this solution article.

*What is CXVII as an integer?*

Recall that C = 100, X = 10, V = 5, and I = 1. Because the symbols are ordered from most significant to least, we can simply add the symbols, i.e. C + X + V + I + I = 100 + 10 + 5 + 1 + 1 = 117.

*What is DXCI as an integer?*

Recall that D = 500.

Now, notice that this time the symbols are *not* ordered from most significant to least—the X and C are out of numeric order. Because of this, we subtract the value of X (10) from the value of C (100) to get 90.

So, going from left to right, we have D + (C - X) + I = 500 + 90 + 1 = 591.

*What is CMXCIV as an integer?*

Recall that M = 1000.

The symbols barely look sorted at all here—from left-to-right we have 100, 1000, 10, 100, 1, 5. Do not panic though, we just need to look for each occurrence of a smaller symbols *preceding* a bigger symbol. The first, third, and fifth symbols are all smaller than their next symbol. Therefore they are all going to be subtracted from their next.

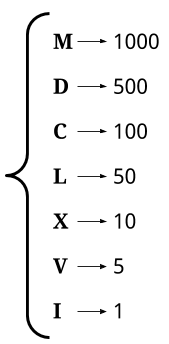
* The first two symbols are CM. This is M - C = 1000 - 100 = 900
* The second two symbols are XC. This is C - X = 100 - 10 = 90.
* The final two symbols are IV. This is V - I = 5 - 1 = 4.

Like we did above, we add these together. (M - C) + (C - X) + (V - I) = 900 + 90 + 4 = 994.

#### Approach 1: Left-to-Right Pass

Intuition

Let's hard-code a mapping with the value of each symbol so that we can easily look them up.



Now, recall that each symbol adds its own value, *except for* when a *smaller valued symbol* is *before* a *larger valued symbol*. In those cases, instead of adding both symbols to the total, we need to *subtract the large from the small*, adding that instead.

Therefore, the simplest algorithm is to use a pointer to scan through the string, at each step deciding whether to add the current symbol and go forward 1 place, or add the difference of the next 2 symbols and go forward 2 places. Here is this algorithm in pseudocode.

total = 0

i = 0

while i < s.length:

if at least 2 symbols remaining AND value of s[i] < value of s[i + 1]:

total = total + (value of s[i + 1]) - (value of s[i])

i = i + 2

else:

total = total + (value of s[i])

i = i + 1

return total

Here is an animation of the above algorithm.

1 / 10

Recall that the *input is always valid*. This means that we don't need to worry about being given inputs such as ICD.

Algorithm

Complexity Analysis

Let

n

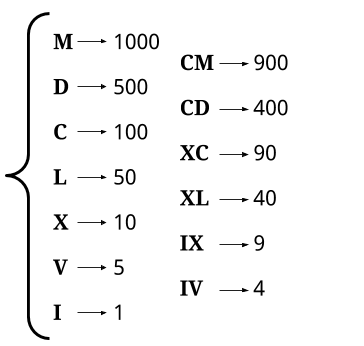
*n* be the length of the input string (the total number of symbols in it).

* Time complexity :
* O(1)
* *O*(1).  
  As there is a finite set of roman numerals, the maximum number possible number can be 3999, which in roman numerals is MMMCMXCIX. As such the time complexity is
* O(1)
* *O*(1).  
  If roman numerals had an arbitrary number of symbols, then the time complexity would be proportional to the length of the input, i.e.
* O(n)
* *O*(*n*). This is assuming that looking up the value of each symbol is
* O(1)
* *O*(1).
* Space complexity :
* O(1)
* *O*(1).  
  Because only a constant number of single-value variables are used, the space complexity is
* O(1)
* *O*(1).

#### Approach 2: Left-to-Right Pass Improved

Intuition

Instead of viewing a Roman Numeral as having 7 unique symbols, we could instead view it as having 13 unique symbols—some of length-1 and some of length-2.



For example, here is the Roman Numeral MMCMLXXXIX broken into its symbols using this definition:

Splitting the numeral into parts

We can then look up the value of each symbol and add them together.

Adding up the sum of the numeral

After making a Map of String -> Integer with the 13 "symbols", we need to work our way down the string in the same way as before (we'll do left-to-right, however right-to-left will work okay too), firstly checking if we're at a length-2 symbol, and if not, then treating it as a length-1 symbol.

total = 0

i = 0

while i < s.length:

if at least 2 characters remaining and s.substing(i, i + 1) is in values:

total = total + (value of s.substring(i, i + 1))

i = i + 2

else:

total = total + (value of s[i])

i = i + 1

return total

Here is an animation showing the algorithm.

1 / 8

Algorithm

Complexity Analysis

* Time complexity :
* O(1)
* *O*(1).  
  Same as Approach 1.
* Space complexity :
* O(1)
* *O*(1).  
  Same as Approach 1.

#### Approach 3: Right-to-Left Pass

Intuition

This approach is a more elegant variant of Approach 1. Just to be clear though, *Approach 1 and Approach 2 are probably sufficient for an interview. This approach is still well worth understanding though.*

In the "subtraction" cases, such as XC, we've been updating our running sum as follows:

sum += value(C) - value(X)

However, notice that this is mathematically equivalent to the following:

sum += value(C)

sum -= value(X)

Utilizing this means that we can process *one* symbol each time we go around the main loop. We still need to determine whether or not our current symbol should be added or subtracted by looking at the neighbour though.

In Approach 1, we had to be careful when inspecting the next symbol to not go over the end of the string. This check wasn't difficult to do, but it increased the code complexity a bit, and it turns out we can avoid it with this approach!

Observe the following:

1. Without looking at the next symbol, we *don't* know whether or not the left-most symbol should be added or subtracted.
2. The right-most symbol is *always* added. It is either by itself, or the additive part of a pair.

So, what we can do is initialise sum to be the value of the right-most (last) symbol. Then, we work *backwards* through the string, starting from the second-to-last-symbol. We check the symbol *after* (i + 1) to determine whether the current symbol should be "added" or "subtracted".

last = s.length - 1

total = value(last)

`

for i from last - 1 down to 0:

if value(s[i]) < value(s[i+1]):

total -= value(s[i])

else:

total += value(s[i])

return sum

Because we're starting at the second-to-last-index, we know that index i + 1 always exists. We no longer need to handle its potential non-existence as a special case, and additionally we're able to (cleanly) use a for loop, as we're always moving along by 1 index at at time, unlike before where it could have been 1 or 2.

Here is an animation of the above approach.

1 / 9

Algorithm

Complexity Analysis

* Time complexity :
* O(1)
* *O*(1).  
  Same as Approach 1.
* Space complexity :
* O(1)
* *O*(1).  
  Same as Approach 1.

[Report Article Issue](https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues)