Genesis Scrum Team

Software Engineer Intern

Coding Challenge

Roman Numeral Conversion

Author:
Nathan Q. Yu

School:
Dartmouth College

December 10, 2016

Introduction

It is frequently necessary to convert data from one representation to another. For example, numbers may be stored internally in a binary or packed decimal representation and must be converted to a character string format for display to a user. Similarly, dates may be stored in a variety of formats, including the number of days since a given base date, and must be converted to standard external form such as "May 25, 1988." In this report, I will describe the process of implementing a subroutine that converts a Roman Numeral to an integer. The subroutine is able to handle all positive numbers from 1 to 3999 using the standard accepted definition for Roman Numerals (all letters in upper case).

Preliminary Thoughts

The problem would be trivial if all the letters representing Roman Numerals were unique, and could be solved with a simple character by character conversion. Since the letters are NOT unique, the real problem lies in parsing the correct Roman Numeral symbols from the input string. For example, "XIV" must be interpreted as "X" + "IV", and not "X" + "I" + "V".

After looking over the Roman Numeral correspondences, I determined that there were only three characters of interest: "C", "X", and "I". These are the only values that are used to create a two-character Roman Numeral subtractively by being prepended to another character of same order of magnitude. For example, "IV" = "V" - "I", "CD" = "D" - "C", and "CM" = "M" - "C". With the three characters of interest, an additional check must be made on the subsequent character in the input to determine whether the roman numeral is a two-character symbol, or just the one-character symbol. We can handle all other characters ("M", "D", "L", "V") directly, converting the single character to its decimal representation.

It also came to mind that checking the subsequent character may result in a NullPointerException or an ArrayIndexOutOfBoundsException when converting the last character in the input. Thus the program must make sure to prevent this error in its implementation.

With these thoughts in mind, I began to brainstorm different ways to perform the conversion. In the following sections, I detail the two solutions that I decided to code.

Initial Algorithm

My first solution performs the conversion be parsing each Roman Numeral symbol of the input with an exhaustive block of switch/if-else conditions. This algorithm simply creates a variable *roman*, loop over the characters of the input, and increments *roman* based the the Roman Numeral parsed from the input.

Pseudocode

```
R2I(roman)
 1  decimal = 0
 2  let array be the char array of roman
 3  for i = 0 to array.length
 4       switch array[i]
 5            M: decimal+ = 1000
 6            D: decimal+ = 500
 7            L: decimal+ = 50
 8            V: decimal+ = 5
 9            C: switch array[i + 1]
10                 M: decimal+ = 900
11                 D: decimal+ = 400
12                 Else: decimal+ = 100
13            X: switch array[i + 1]
14                 C: decimal+ = 90
15                 L: decimal+ = 40
16                 Else: decimal+ = 10
17            I: switch array[i + 1]
18                 X: decimal+ = 9
19                 V: decimal+ = 4
20                 Else: decimal+ = 10
21       if two character symbol parsed
22            increment i
23  return decimal
```

Analysis

This algorithm takes $O(1)$ memory to store the *decimal* variable, and $O(n)$ time to loop though each character of the input. In the worst case, the algorithm will need to check the subsequent character for each iteration, but $O(2n)$ simplifies to $O(n)$. This algorithm is guaranteed to return the correct conversion given a properly formatted Roman Numeral because the exact correspondence is used during each step. In the actual code, the NullPointer is handled properly. While this algorithm is correct, the code itself is cluttered and could simplified and made more scalable.

Optimized Algorithm

When dealing with any sort of one-to-one mapping, some sort of Map implementation can be used effectively. After coding the first algorithm, I determined that a HashMap that maps a Roman Numeral to its decimal representation could be used to improve my existing code. Thinking ahead, I decided to use a LinkedHashMap implementation because an ordered keyset would be needed to efficiently convert decimal integers to their Roman Numeral forms.

Pseudocode

R2I($roman$)

```
1  let R2I be a map of Roman Numeral symbols to their decimal values
2  decimal = 0
3  let array be the char array of roman
4  for i = 0 to array.length
5       if R2I containsKey array[i] + array[i + 1]
6            decimal+ = R2I.get(array[i] + array[i + 1])
7            increment i
8       elseif R2I containsKey array[i]
9            decimal+ = R2I.get(array[i])
10 return decimal
```

Analysis

This algorithm takes $O(n)$ memory to store the $R2I$ HashMap, and $O(n)$ time to loop though each character of the input because a HashMap has constant look-up time. This algorithm is guaranteed to return the correct conversion given a properly formatted Roman Numeral because the exact correspondence is used during each step. In the actual code, the NullPointer is handled properly. While this algorithm takes more memory than the previous algorithm with the same runtime, I chose it because this algorithm is significantly more readable, can be easily modified to handle additional Roman Numeral symbols, and also allows for an easy method of converting Roman Numerals to decimal as shown in the code.

*On a very hypothetical aside, if there were to be a very large number of different Roman Numeral symbols, this algorithm would not be maximally efficient because "containsKey" would not have an $O(1)$ runtime in a dense HashMap. We could slightly improve the runtime in this scenario using the property that each subsequent symbol in a properly formatted Roman Numeral will be less than or equal to the current value, and remove all entries from the map that are greater than the current value. We would need to start from a full map for every new conversion.

Testing

The only edge case that I could think of when parsing a properly formatted Roman Numeral would be the case where we have a two-character symbol. This condition was satisfied in both implementations by simply checking for the two-character scenarios exhaustively, and then incrementing the iterator so we do not unintentionally parse a character twice.

With respect to the problem assignment of positive integers from 1 to 3999, I could have written a simple shell script to convert all integers 1-3999 to Roman Numeral and store the results in a text file, and then parse line by line to convert the Roman Numerals back to integer form and store the results in another text file. I did not deem this necessary, so I did not actually implement this. Below are three edge cases that are relevant to the problem, but out of the scope of the problem specifications:

Input greater than Integer.MAX_VALUE can be handled by using the Java BigInteger class instead of the primitive *int* data structure to perform the arithmetic during conversion.

Negative numbers could be handled by parsing the negative sign first by checking if the first character is "-", independently of the the of the input. This would allow us to run the same additive conversion algorithm and return the correct result by simply changing the sign of the returned value ($x* = -1$) under a boolean flag.

Invalid roman numeral format would be more difficult to handle, because not all symbols have unique characters. In this case, we would need to store the previous parsed symbol in a variable, and then use the property that each subsequent symbol in a properly formatted Roman Numeral will be less than or equal to the current value to determine if a symbol violates this property (by comparing the value of the current symbol to the value of the previous symbol).

Conclusion

Overall this was a very fun problem to solve, and made me think about the tradeoffs between efficiency and storage vs. cleanliness of code. with changes that make minimal impact on storage or runtime, I believe that code that is more simple and scalable should be chosen.