Putting Numbers Behind the Numerals

Milton Rosenbaum
McGill
milton.rosenbaum@mail.mcgill.ca

1 Introduction

Though the idea that Roman numerals can be suited for arithmetic has become more mainstream Detlefsen et al. (1976), Bruderer (2019), even its most ardent defenders have shied away from division algorithms. Indeed, when presented with $DCVII$ and $LXXI$ and asked to divide, most people would not know where to begin. Macbeth (2018)'s recent attempt was even used as part of an argument *against* Roman numeral's suitability for numerical reasoning.

Since Roman numerals impose the structure of the natural numbers onto the symbols I, V, X, L, C, D, and M, we should be able to rely on that structure while reasoning. Furthermore, formal systems are not merely tools to express reasoning but come to constitute the same number sense they were developed to represent Schlimm (2018). Since formal systems shape human reasoning, this leads to a question: How do we compare systems? Which ones work best for abstract thought?

Roman numerals are an interesting case study. As stated above, the common intuition seems to be that Roman numerals are an example of a system that enables counting or addition but limits other arithmetical reasoning. Roman numerals already have reasonable algorithms for addition, subtraction, and multiplication Schlimm and Neth (2008). It would be a relevant result, then, if a system existed that enabled reasoning in three of the four basic arithmetic operations but was unsuited for reasoning about division.

Given the poor state of division algorithms, it seems reasonable to conclude that Roman numerals are unsuited for division. However, I show that the standard long division algorithm has a direct translation in Roman numerals. I further demonstrate that this algorithm is a clear improvement from the previous state-of-the-art algorithm presented in Macbeth 2018.

The presentation of an algorithm does not demonstrate that abstract reasoning is possible within Roman numerals, so I must make this argument somewhat obliquely. The algorithm I present is based on a large amount of abstract numerical reasoning and is more efficient than MacBeth's algorithm, which was based on a more mechanical interpretation of the system. As such, even if I cannot directly prove that Roman numerals admit genuine reasoning, their use as a computational tool seems to benefit from reasoning, indicating that it can be present within the system.

Number Systems Arabic Numerals form a place-value system, where ordered sequences of individual digits represent numbers. Each digit represents a quantity of some power of the base, though the specific power of the base depends on its distance from the least significant digit. *Arabic* numerals refer to our specific choice of ten as a base and the common set of ten symbols for our digits. As such, any string of Arabic digits $d_3d_2d_1d_0$ implicitly encodes the sum $10^3d_3 + 10^2d_2 + 10^1d_1 + 10^0d_0$.

Roman numerals, on the other hand, are an additive system. Each symbol encodes the same number, no matter its position. Any additive system with a value for one would be able to encode for every single natural number and so would be informationally equivalent to Arabic numerals. In this case, *Roman* numerals are defined as the additive system that uses the familiar I, V, X, L, C, D, M representing $1, 5, 10, 50, 100, 500$ and 1000 respectively. When applicable, numbers larger than MMM will be represented using a *vinculum*, where the same symbols written with a bar over the top encode for that many Ms. So fifty thousand is written $\overline{L}$.

Symbolic and Mathematical Reasoning Formal systems can be read in two different, yet

equally valid ways. First, we could imagine that each numeral refers directly to some numerical object. The numerals 4 or $IIII$, for example, would refer to the natural number between three and five. However, formal systems can also be understood symbolically, as just a set of symbols and the rules that govern valid transitions from one symbol to another.

In addition to Roman numeral's numerical definition, they can be defined purely as a set of symbols and their transition rules. Along with the standard set of symbols, we introduce another set of grouping and ungrouping rules, $IIIII \leftrightarrow V$, $VV \leftrightarrow X$, and so on. This new formalization is equivalent to their numerical presentation above. To define addition, we would shuffle the two numerals' symbols together and apply grouping rules until the numeral is as short as possible.

In Arabic numerals, addition could be understood as the repeated application of rules of the form: "If a 6 is written above a 7, write a 3 below the line and a 1 in the next column."

These presentations seem to strip the underlying numbers away from the symbols. Indeed, since any algorithm in any formal system has such a symbolic reading, it is impossible to determine if a user of the system is performing reasoning or blind symbolic manipulation. However, it seems to be the case that *some* symbolic systems encourage reasoning, while others encourage mechanical manipulation.

Comparing Arabic to Roman numerals, Arabic numeral addition is difficult to memorize when considered as a large set of rules. It depends on the relative positions of symbols, and each rule has several exceptions. However, once some measure of fluency in the notation itself has been achieved, it becomes difficult, for instance, to imagine making column transposition errors. For Roman numerals, however, symbolic manipulation is straightforward and often easier than reasoning about the abstract sums first, and then trying to encode sums as numerals.

Although this claim may be intuitive, in the comparison of algorithms that follows, I demonstrate that rigorous reasoning is indeed possible with Roman numerals, as one would expect with any formal system. I present a comparison between the division algorithm presented in MacBeth 2018 and an algorithm of my design. I claim that my algorithm is an example of reasoning within the system of Roman numerals. Not only is the presented algorithm based on a similar example of reasoning found in Arabic numerals, but it is substantially more efficient than the algorithm presented by MacBeth. This demonstrates that Roman numerals are not condemned to mere manipulation, but admit reasoning.

This claim, however, does not rebuke the rest of the philosophy presented in her paper. Indeed, her claim that Roman numerals only represent collections of things, and are therefore unsuited to reasoning about numbers, follows from her other definitions. Since each symbol of a Roman numeral may be considered independently from its other symbols, it represents each number as a collection of smaller parts, which in turn represent collections of smaller parts. This, she argues, is different from representing numbers as individual abstract objects, as is done by positional systems, whose individual digits only represent a definite quantity when placed in a numeral. The grounds of each of these claims are argued from a fully formed system of thought which I do not contradict. Instead, my argument serves to demonstrate that additive systems admit genuine mathematical reasoning by demonstrating that such reasoning improves their ability to perform calculations.

2 Models

To measure and compare the symbolic complexity of both algorithms, I developed a grid-based agent that attempts to model all the thoughts and actions a person would perform in order to implement an algorithm.

2.1 Grid Agents

Each agent has access to an underlying grid where every square can contain at most one symbol. The grid serves as an abstraction of a blank sheet of paper and makes logging, information retrieval, and writing explicit. To interact with the grid, the agent has an "eye" which can be pointed to one cell of the grid at a time. The agent can move its eye freely but must use some of its active memory to determine where. After moving its eye, the agent can get the value of a letter at that point. Additionally, the agent is given a "pen" which moves independently of the eye. The pen can write at most one symbol at a time. Each higher-order ability that the agent has — such as reading whatever numeral is positioned at a certain point on the grid —

is accomplished by the composition of these primitive operations.

Every time an agent interacts with the grid, either to retrieve or write a symbol, it is logged along with the distance from the previous action of that type. Taken separately, these only capture a small portion of my abstract understanding of symbolic complexity. However, taken together, these measurements provide a comprehensive account of the motor and eye movements that would be required to implement an algorithm on paper. Though a human implementing the algorithm would find places to save time and symbol writing, they should never be required to take any action that the grid-agent has not. As such, the performance of the agent should provide an upper bound on the total motor operations that a human would perform to implement the same algorithm.

In addition to being able to read and write symbols, agents can recall ungrouping, ordering, grouping, and multiplication facts. Every time such a fact is recalled, it is logged by the agent. With the combination of writes, recalls, and eye movements, the entire behavior of the agent is fully reproducible.

2.2 Long Division Agent

This agent implements an algorithm of my design, based on standard Arabic long division with minor modifications to work with Roman numerals. Since it uses addition, multiplication, and simplification as key sub-algorithms, they are presented as well.

To begin a division problem, the agent is presented with a grid that already contains the divisor and dividend. For instance, to divide $MCCXI$ by $XVIII$, the agent would receive a grid initialized to:

```
MCCXI
  XVIII
```

The first step of Arabic numeral long division is to find the largest single-digit multiple of the divisor that is still smaller than the first few digits of the dividend. That multiple is then written directly below those digits. This can be broken down into two smaller steps. First, writing the product near the most significant digits of the dividend is equivalent to repeatedly multiplying the divisor by ten. This step is trivial in Arabic numerals but must be computed in Roman numerals. As such, before it can begin calculating the quotient, the Roman numeral agent "precomputes" all of the single-symbol multiples of the divisor, so that we can refer back to the values while computing the quotient.

Multiplication The agent is given symbolic knowledge of multiplication facts for each pair of symbols. This is analogous to the memorized times table that is standard for users of Arabic numerals. All such facts are presented in the table below:

	I	V	X	L	C	D	M
I	I	V	X	L	C	D	M
V		XXV	L	CCL	D	MMD	$\overline{V}$
X			C	D	M	$\overline{V}$	$\overline{X}$
L				MMD	$\overline{V}$	$\overline{XXV}$	$\overline{L}$
C					$\overline{X}$	$\overline{L}$	$\overline{C}$
D						$\overline{CCL}$	$\overline{D}$
M							$\overline{M}$

Recall that the agent is still in the precomputation step. It is trying to find the smallest single-letter multiple of the divisor that is still below the dividend. When performing the required multiplications, the agent writes the result of the multiplication next to the symbol it is multiplying by.

The agent begins this step by multiplying $XVIII$ and V. First, the agent checked the table for the combination of X and V, saw that this produced an L, and wrote that symbol to the grid. Next, it looked up the product of V and V, which returned XXV. The agent wrote that to the grid beside the L. Likewise, the agent performed similar three lookups for V and I.

```
MCCXI
I XVIII

  XVIII
V LXXVVVV
```

Simplification Given a Roman numeral, the agent writes the number in simplest form two lines below the original on the grid. In order to do this, the agent begins from the smallest symbol in the numeral it is considering. It then steps to the left, counting each occurrence of that symbol. If the symbol can be grouped into a larger symbol, the agent will write the larger symbol onto the line below. When the agent reaches a new symbol, the count is set to however many groupings it per-

formed for that symbol. Often numerals are in simplest form or only require a few groupings, so the number is most often initialized to zero.

After stepping through the number once and generating the new groups, the agent will step through the number a second time, this time beginning at the most significant symbol. If it sees a group, it will write that group down and step forward by the group size that the letter generates.

After the simplification, the grid now reads:

```
MCCXI
I XVIII

  XVIII
  LXXVVVV
     X X
V LXXXX
```

In the example above, the agent started by considering V, it recalled the grouping fact that $VV \leftrightarrow X$. In the next step, it observes the second V and writes X directly below it. It does the same for the next two V's. Once it has stepped through the entire numeral, it begins from L. Seeing no group below it, it will rewrite the L two lines below. When it arrives at the first V, the agent will see that there is an X written below it, and write an X in the cell directly below. Since X ungroups into VV, the agent will move its eye by two characters, while moving its pencil by only one. It will then see the next V and the X below, and so write the final X in the result.

After each simplification, the agent checks if the resulting numeral exceeds the dividend. Since both numbers are in simplest form, the agent just needs to step through both numerals letter by letter, stopping if the two digits are ever different and checking which of the two digits are larger.

Once a product exceeds the dividend, the agent moves on to the next phase of the algorithm. In this case, that will be when the grid looks like:

```
MCCXI
I XVIII

  XVIII
  LXXVVVV
     X X
V LXXXX
```

```
  XVIII
  CLXXX

X CLXXX

  XVIII
  DCCLLLL
     C C
L DCCCC

  XVIII
  MDCCC

C MDCCC
```

Addition Given a pair of two numerals, the agent shuffles the two sets of symbols together. It begins at the first pair of either symbol, and recalls an ordering fact. Whichever symbol of the two is larger is written to the grid, and the agent shifts its eye to the next symbol in whichever numeral that symbol came from. This process repeats until both numerals are exhausted.

At this phase of the algorithm, the Roman numeral agent finished the precomputation step, and will now move on to calculating the quotient through repeated addition. In Arabic long division, the analogous part of the algorithm uses repeated subtraction rather than addition. The algorithm also assumes that the user not only has procedural knowledge of all multiplication facts but also the ability to consider all of these facts and determine which produces the largest product without exceeding some number. In Roman numerals, each letter encodes a portion of a sum and there are far fewer composition rules to track, so we can instead perform repeated addition directly, without needing to recall such higher-order facts.

The agent begins the repeated summation step by creating two columns, quotient and accumulator. In the accumulator column, the agent will calculate an ongoing sum, using each product it just computed as many times as possible without exceeding the dividend. In the quotient column, the agent will keep track of the single letters that were used to create each product.

In this case, since $DCCCC$ is the largest valid product from the previous phase, it will be placed into the accumulator column, and L into the quotient column. Since two Ls group to form C, and we already know that $XVIII \times C$ exceeds the quotient, the agent moves on to Xs. The agent

will repeatedly add X to the quotient column, and $CLXXX$ to the accumulator column, until it either adds the fourth X or the accumulator exceeds the dividend.

In this case, after adding X to the quotient column twice, the agent will have exceeded the dividend, and the grid will read:

```
MCCXI
I XVIII

...

XVIII
  CLXXX

X CLXXX

...

DCCCCCLXXX              LX
MD
MLXXX                   LX
MCLLXXXXXX              LXX
  C L
MCCLX                   LXX

MLXXX                   LX
```

The dots are not part of the grid, and stand in for lines that have already been seen, and which are not relevant to this portion of the algorithm.

This process continues until the agent exhausts all of the letters. Whatever the quotient column is at this point is the answer, and subtracting the final value in the accumulator from the dividend yields the remainder. In this case, the last few lines of the grid that the agent produced are:

```
MCLXXXVIII              LXVI

MCLXXXVIII              LXVI

MCLXXXXVVIIIIII         LXVII
  CL   X V
MCCVI                   LXVII
MCCXVVIIII              LXVIII
    X
MCCXXIIII               LXVIII

MCCVI                   LXVII
```

2.3 MacBeth Division Agent

The MacBeth agent is based on the division algorithm presented in Macbeth (2018). To divide by n, the algorithm uses ungrouping rules to mechanically split the number into n equal partitions. First, the algorithm checks if there are n occurrences of the most significant symbol. If there are, that symbol is removed and added to the quotient. If not, the most significant symbol is ungrouped. This process repeats until there are fewer I's left in the dividend than there are in the divisor. At that point, the quotient is output, and whatever is left of the divisor is the remainder.

For instance, to divide $MDCXII$ by three the algorithm begins by splitting M into DD. At this point, there are three D's in the dividend, so they are removed and D is added to the quotient. Next, the agent sees that there are not enough C's, and so C is ungrouped into LL. There are still not enough L's, so the first L is split into $XXXXX$. Now, since there are more than three X's, they can be removed from the dividend and added to the quotient. The algorithm will continue the above until it adds the final group of three Is to the quotient.

The agent is equipped with a piece of working memory representing all unprocessed digits. First, the agent ungroups the dividend into I's. Each I forms the heading of a column. Then, the agent copies down all of the most significant symbols in the divisor, if the symbols match up with the I's, then the agent continues to the next step. If not, the agent adds the entire line into its working memory and begins ungrouping onto the line below. If, at any point, the agent fills up an entire row, it marks that row and moves on to the next. Once it has ungrouped every symbol from the previous step, it moves on to the next symbol in the dividend, and the process repeats.

After dividing $MDCXII$ by three, the agent has produced a grid reading:

```
MDCXII
III
M
DDD ✓
C
LL
XXXXX ✓
XXXXXXX ✓
XXXX ✓
```

```
XX
VVVV ✓
V
IIIII ✓
III ✓
I

DXXXVII
```

The lines with a ✓ indicate that the agent has marked that line for inclusion in the final answer. Printing only those lines reveals that the quotient is written vertically and that each symbol in the quotient is written three times:

```
DDD
XXXXX
XXXXXXX
XXXX
VVVV
IIIII
III
```

This algorithm is simpler than long division and performs very well for small divisors. However, since dividing by n requires writing at least n symbols, the scaling behavior of the algorithm is poor, and performing it is tedious. Additionally, though the agent cannot make mistakes, this algorithm requires lining up dozens of symbols while keeping precise track of which ones have been used, so errors seem likely if the algorithm were to be implemented on paper.

3 Results

Experimental Setup To determine the overall efficiency of each algorithm, both were run for every combination of dividends between 500 and 3999 and divisors between 2 and 100. Some selected graphs have been shown.

3.1 Results

The results are presented in figures one through four.

As can be seen, in almost every metric, Roman numeral long division obtained a lower complexity than MacBeth division. As expected, for smaller divisors, MacBeth division uses fewer symbols. This makes sense given that the algorithm is designed by reasoning about Roman numerals as literal collections of things. As those collections get larger, the algorithm becomes less efficient. Long division, however, displays no correlation between divisor value and total symbols written, other than a slight benefit for larger divisors over smaller ones.

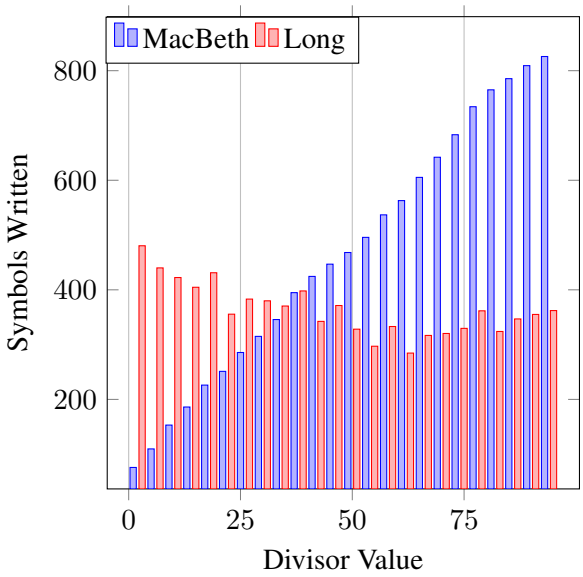

Figure 1: The count of symbols written for divisor value, averaged over intervals of 4

For facts recalled, the numbers are much closer. This is because MacBeth division only ever needs to recall ungrouping facts, while long division must recall several multiplication facts in the beginning, and then many more ordering steps for each step during the summation, in addition to any simplifications it must perform during these processes.

When measured against the input length, long division seems to grow logarithmically while MacBeth's algorithm grows linearly. This makes intuitive sense: it should be roughly twice as difficult to divide a collection of things into ten groups than into five. The logarithmic scaling of Roman numerals seems to indicate the agents are not considering Roman numerals as collections of objects. Instead, Roman numeral long division exhibits the same kind of scaling as it would in Arabic numerals[1]. This indicates that additive notational systems do not limit the user to treating their numerals as collections of objects. Further, the scaling behavior shows that the same kind of abstract reasoning that enables Arabic numerals to efficiently divide numbers can also be implemented in Roman numerals.

[1] this is because long division is similar to a binary search for the quotient

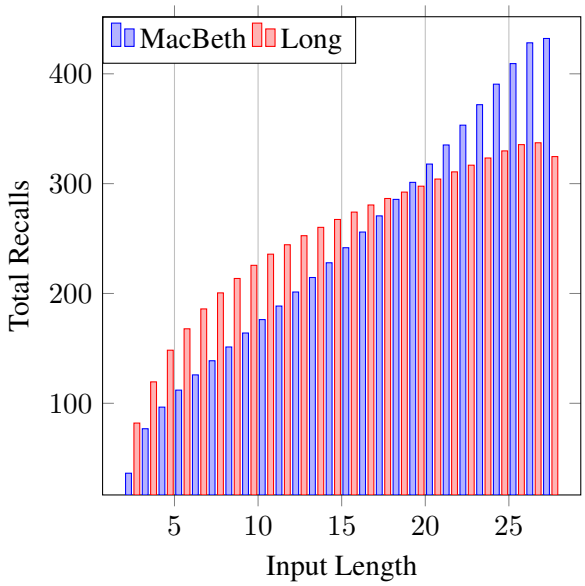

Figure 2: The average count of all facts recalled for each input length

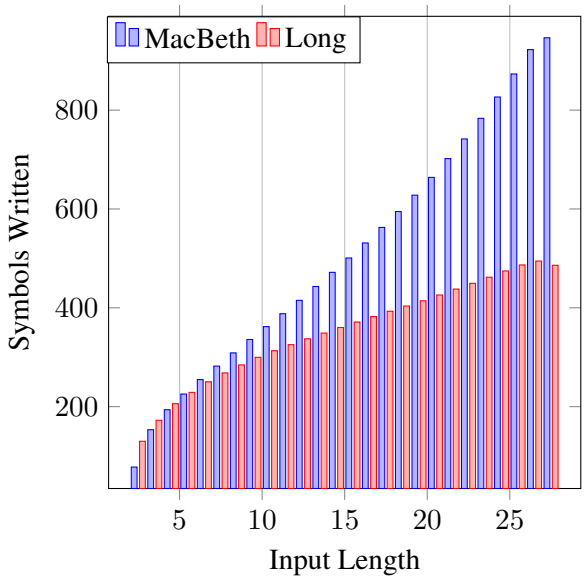

Figure 3: The average count of symbols written for each input length

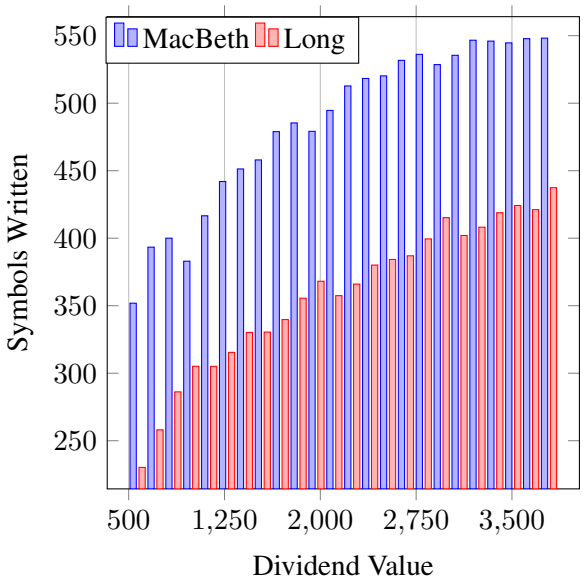

Figure 4: The count of symbols written for dividend value, averaged over intervals of 140

4 Limitations and Discussion

There are two major limitations of my argument. First, there is no comparison to Arabic numerals, and second, the comparisons themselves are far from objective.

Developing these grid-based agents was not the original method considered for this report. Initially, they were more simple. Instead of using a grid, I implemented each step of the algorithm on strings and then estimated the mental difficulty of that portion of the calculation. Each function returned the result of that portion of the algorithm, as well as a couple of measures based on how I imagined that particular step would be implemented on paper. Although the actual development of the system was successful, the choice of complexity measurements was not satisfactory. There were several instances where very slight changes to the methods of measurement resulted in massive changes in eventual value. These changes were slight enough that any measurements produced were arbitrary to the point of being unreproducible.

The second agent I tried came from a different perspective. As I developed the first agent, I continuously developed minor simplifications for each step of the calculation. As I memorized the Roman times table, for instance, I found myself instinctively able to tell which symbol would exceed the divisor without the need to perform any computation which sped up the preprocessing step. As the grouping rules became instinctive, I found that

most of the repeated addition could be skipped as well, as I could estimate how many times I needed to add a number, and then check that the next addition would exceed the dividend.

As such, I decided to develop a small game. At each step of the algorithm, the game would compute the correct answer, and then provide me with a prompt. It would only move to the next step after I responded with the correct answer. The game would log the inputs to each function and the time they took me to perform. From this game, the data I generated was related to some genuine complexity, namely, how difficult I found the operation. Though my measurements now better reflected the mental effort of each step, I decided that there was little interest in Milton Rosenbaum complexity.

Finally, I settled on the grid-based agent. Even with more reproducible agents, there are still an overwhelming number of arbitrary, yet significant choices. For long division, the agent is exceedingly verbose. It rewrites the divisor each time the product is calculated, inflating the number of symbols written. This may be reasonable since the agent would need to "reconsider" the numeral that it is multiplying by. However, this is also captured by the get operations that the agent would need to perform to manipulate the numerals. Additionally, the agent always rewrites a numeral during simplification, even if the number is already in simplest form. Since it would have to do some reasoning to determine that the numeral is already in simplest form, again this seems reasonable. However, removing this requirement would only further benefit the agent's complexity scores.

The same kinds of choices presented themselves for MacBeth division. Its first version rewrote the entire dividend on each line and was less sophisticated in the order of symbols it chose to ungroup. These two choices, though closer to the original algorithm, caused the agent to write upwards of ten times as many symbols as the long division agent, in some cases. Again, arbitrary choices that did not impact my intuitions of symbolic complexity caused massive changes in the measurements my agents produced.

Even within a single notation system, the measured complexity of an algorithm can change dramatically for arbitrary reasons. This fact challenges our ability to make meaningful comparisons of cognitive difficulty across different notation systems. The task of predicting mental complexity is a well-established field of cognitive science, and as such far outside my area of knowledge. The other major work in this area, Schlimm and Neth (2008), relies on established means of relating symbolic manipulation to mental effort. Any comparison I would be capable of generating between Roman numerals and their Arabic counterparts would undoubtedly present the same randomness. As such, I did not develop an Arabic agent.

Since long division is nearly the same algorithm for both notational systems, it is reasonable to expect that the inefficiency of each algorithm depends on the relative compactness of their numerals, which also determines the number of steps that the agent needs to make. For Roman numerals between 1 and 3999, the length of the numeral is on average the base 3.4 logarithm of the number[2]. This implies that a Roman numeral takes, on average three times as many symbols to display and that the agent will require three times the steps to complete the algorithm. I expect that any reasonable comparison will show that Roman numeral division requires roughly nine times as many symbols to perform as the Arabic numeral equivalent. Any comparison I could make that disagreed with that figure would count as evidence against the validity of the comparison, rather than evidence for Roman numerals.

5 Conclusion

By comparing two different approaches to division — one based on mechanical manipulation and another derived from abstract numerical reasoning — I have shown that Roman numerals can benefit from the same kind of mathematical thinking that makes Arabic numerals effective for calculation.

The experimental results show that Roman numeral long division algorithm outperforms MacBeth division across many metrics, particularly as the size of numbers increases. While MacBeth's algorithm performs adequately for small divisors, its requirement to physically partition numbers makes it increasingly inefficient for larger values.

These findings challenge the common assumption that Roman numerals are inherently unsuited for complex mathematical operations. While the encoding itself is less efficient than Arabic numerals, the calculations themselves are often more

[2] calculated by converting each number from 1 to 3999 into a Roman numeral, and then computing the average

simple. This suggests that the limitations often attributed to Roman numerals may be more related to our familiarity with Arabic numerals, or the specific domains that they are employed.

However, there are limitations between comparisons of symbolic complexity across different numerical systems. The arbitrary nature of many implementation choices in our grid-based agent model highlights the difficulty of objectively measuring symbolic complexity.

While Roman numerals may never be as computationally efficient as Arabic numerals, they are capable of supporting genuine mathematical thought and reasoning. This suggests that the relationship between notation systems and mathematical thinking is more complex than previously assumed, opening up new questions about how different representations shape our ability to reason about numbers.

References

Herbert Bruderer. 2019. Calculating with roman numerals is not so difficult. *Communications of the ACM*.

Michael Detlefsen, Douglas K. Erlandson, J. Clark Heston, and Charles M. Young. 1976. Computation with roman numerals. *Archive for History of Exact Sciences*, 15(2):141–148.

Danielle Macbeth. 2018. Logical form, mathematical practice, and frege's begriffsschrift. *Annals of Pure and Applied Logic*, 169(12):1419–1436. Logic Colloquium 2015.

Dirk Schlimm. 2018. Numbers through numerals. the constitutive role of external representations. In Sorin Bangu, editor, *Naturalizing Logico-Mathematical Knowledge: Approaches From Psychology and Cognitive Science*, pages 195–217. Routledge.

Dirk Schlimm and Hansjörg Neth. 2008. Modeling ancient and modern arithmetic practices: Addition and multiplication with arabic and roman numerals. In B. C. Love, K. McRae, and V. M. Sloutsky, editors, *Proceedings of the 30th Annual Conference of the Cognitive Science Society.*, pages 2097–2102. Cognitive Science Society.

A Appendix

The entire output of $MCCXI$ divided by $XVIII$.

```
MCCXI
I XVIII

  XVIII
  LXXVVVV
     X X
V LXXXX

  XVIII
  CLXXX

X CLXXX

  XVIII
  DCCLLLL
     C C
L DCCCC

  XVIII
  MDCCC

C MDCCC

DCCCC                    L

DCCCC                    L
DCCCCCLXXX               LX
MD
MLXXX                    LX
MCLLXXXXXX               LXX
  C L
MCCLX                    LXX

MLXXX                    LX
MLLXXXXXX                LXV
 C L
MCLXX                    LXV
MCLXXXVIII               LXVI

MCLXXXVIII               LXVI
MCLXXXXVVIIIIII          LXVII
  CL   X V
MCCVI                    LXVII
MCCXVVIIII               LXVIII
    X
MCCXXIIII                LXVIII

MCCVI                    LXVII
```

MCCXI
XVIII
VVIIIIIIIII
IIIIIIIIIIIIIIIII
M
DD
CCCCCCCCCCC
LLLLLLLLLLLLLLLLL ✓
LLLLLL
XXXXXXXXXXXXXXXXXX ✓
XXXXXXXXXXXX
VVVVVVVVVVVVVVVVV ✓
VVVVVVVV
IIIIIIIIIIIIIIIIII ✓
IIIIIIIIIIIIIIIIIII ✓
IIIII

LXVII