# Investigating Symbolic Capabilities of Large Language Models

Neisarg Dave[1,*], Daniel Kifer[2], C. Lee Giles[1] and Ankur Mali[3]

[1]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[2]*Department of Computer Science and Engineering, The Pennsylvania State University, USA*
[3]*Department of Computer Science and Engineering, University of South Florida, USA*

## Abstract

Prompting techniques have significantly enhanced the capabilities of Large Language Models (LLMs) across various complex tasks, including reasoning, planning, and solving math word problems. However, most research has predominantly focused on language-based reasoning and word problems, often overlooking the potential of LLMs in handling symbol-based calculations and reasoning. This study aims to bridge this gap by rigorously evaluating LLMs on a series of symbolic tasks, such as addition, multiplication, modulus arithmetic, numerical precision, and symbolic counting. Our analysis encompasses eight LLMs, including four enterprise-grade and four open-source models, of which three have been pre-trained on mathematical tasks. The assessment framework is anchored in Chomsky's Hierarchy, providing a robust measure of the computational abilities of these models. The evaluation employs minimally explained prompts alongside the zero-shot Chain of Thoughts technique, allowing models to navigate the solution process autonomously. The findings reveal a significant decline in LLMs' performance on context-free and context-sensitive symbolic tasks as the complexity, represented by the number of symbols, increases. Notably, even the fine-tuned GPT3.5 exhibits only marginal improvements, mirroring the performance trends observed in other models. Across the board, all models demonstrated a limited generalization ability on these symbol-intensive tasks. This research underscores LLMs' challenges with increasing symbolic complexity and highlights the need for specialized training, memory and architectural adjustments to enhance their proficiency in symbol-based reasoning tasks.

## Keywords

Large Language Models, Symbolic Tasks, Chomsky's Hierarchy

## 1. Introduction

Large language models are driving revolutionary advancements in artificial intelligence. They are transforming how we interact with and interpret data across various modalities. These models have surpassed human performance in generating text, images, and videos. LLM-augmented systems are revolutionizing traditional expert systems such as search, retrieval, theorem proving, symbolic reasoning, programming, and drug discovery. Amid these advancements, exploring the symbolic capabilities of LLMs, particularly in logical reasoning, mathematical computation, and formal language manipulation, remains a crucial area of investigation.

A key aspect of leveraging LLM capabilities is developing effective prompting techniques. Prompts are carefully designed inputs that guide the model's behavior, enabling it to perform specific tasks or exhibit desired behaviors. These techniques include few-shot and zero-shot learning, where models are prompted with examples or instructions to help them generalize to new tasks without extensive retraining. Other techniques include Chain-of-Thought [1], Tree-of-Thought [2], Graph-of-Thought [3] etc. While LLMs have gained prominence, other neural network models have also been studied for their symbol manipulation abilities. This research can be broadly categorized into two areas: directly learning to solve logical and mathematical problems, and research conducted from the perspective of formal methods and automata theory. Recurrent Neural Networks (RNNs) are highly efficient at

learning regular grammars, with Second Order RNNs particularly adept at encoding stable states. Memory-augmented models, such as stack-RNN, tape-RNN, and Neural Turing Machines (NTMs), have demonstrated their ability to handle more complex languages. Additionally, transformers, with their attention mechanisms, have been explored for their potential in managing formal languages and symbolic computations. However, RNN-based models often fail to generalize on tasks such as math word problems and logical reasoning.

In this work, we address the symbol manipulation capabilities of Large Language Models (LLMs). We investigate fundamental mathematical operations such as addition, multiplication, and counting. The inherent complexity of these tasks is derived from Chomsky's Hierarchy. Specifically, we present the following research questions:

**RQ1:** Can LLMs apply simple symbolic rules, such as addition and multiplication, stably and repeatedly?

**RQ2:** How precisely can LLMs preserve the construction of a group of symbols and respect the order of operations?

In this study, we first discuss task complexity and the computational requirements necessary to solve these tasks. We then estimate the number of bits required to encode such a machine and compare it to the knowledge tuple encoding in LLMs as described by Deletang et al. [4]. Additionally, we estimate the number of parameters needed for a neural network to solve the given tasks using both methods. Finally, we provide experimental evidence of the performance of LLMs. We evaluate four enterprise LLMs and four open-source LLMs on three essential skills: addition, multiplication, and counting, and design 5 tasks to answer $RQ1$ and $RQ2$. Utilizing Hindu-Arabic numerals and both uppercase and lowercase English alphabets as symbols, we construct multi-digit numbers from single digits. This approach allows us to assess the precision of the LLMs by increasing the number of digits in the numbers, both before and after the decimal point.

## 2. Related Work

Early research demonstrated that recurrent neural networks (RNNs) could learn and encode deterministic finite state automata, enabling them to perform symbolic tasks such as rule-based reasoning and grammatical inference. Omlin and Giles [5, 6, 7] demonstrated the extraction, insertion, and refinement of deterministic finite automata in recurrent neural networks. They also explored higher-order recurrent networks and their applications in grammatical inference [8]. However, RNNs have been shown to struggle with tasks extending beyond regular grammars. For instance, RNNs require auxiliary data structures like stacks or tapes to process inputs governed by context-free or context-sensitive grammars. Mali et al. [9, 10] and Stogin et al. [11] advanced this work by providing theoretical bounds for the expressiveness of RNNs and memory-augmented RNN models. Dave et al. [12] empirically showed that second-order connections encode states more stably for formal languages in comparison to first-order RNNs like LSTM and GRU. Symbolic tasks encompass a range of applications, including the manipulation of formal languages and the execution of mathematical operations [13]. Mathematics provides a structured framework for symbolic evaluation, aligning well with the principles of neurosymbolic AI, where symbol manipulation and operations are central.

Saxton et al. [14] created a dataset of math questions across various topics and observed that LSTM models barely memorized the question-answer pairs and could not generalize. Dave et al. [15] used the same dataset to create distractors for math multiple-choice questions. Mistry et al. [16] summarized the specialized architectures developed to solve arithmetic tasks, highlighting gaps in the robustness, compositionality, and interpretability of these models. The development of large language models (LLMs) has spurred significant interest in their mathematical and symbolic capabilities. Research has increasingly focused on training LLMs specifically for mathematical tasks, aiming to enhance their proficiency in symbolic reasoning. LLM models like Deepseek [17], LLemma [18], and Metamath [19] are specially trained with math datasets. However, studies have shown that neural networks, including LLMs, often fail to genuinely learn to solve mathematical and symbolic tasks. Dziri et al. [20] investigated the limitations of LLMs across three problems: multiplication, puzzle solving, and

dynamic programming. They tested GPT-3, GPT-3.5, and GPT-4 models using zero-shot, few-shot, and fine-tuning techniques, revealing a significant decline in performance as the sample size increased. Similarly Shakarian et. al. [21] also reported performance of ChatGPT [22] declines with the increase in the number of addition and subtraction operations required to solve math word problems (MWPs). Furthermore, they also observe that LLMs perform much better when asked to show work. Additionally, Frieder et al. [23] demonstrated that GPT-4's mathematical proficiency was far below graduate-level standards. Previous research on the mathematical proficiency of large language models (LLMs) has primarily focused on a broad set of math problems and reported findings based on static datasets such as GSM 8K [24] and MATH [25]. This approach does not uncover the underlying inconsistencies in LLMs' understanding of mathematics. In this work, we examine the consistency and precision of LLMs as the complexity of the tasks increases.

Through our analysis, we assess the performance of enterprise and open-source LLMs trained on text and mathematical datasets, testing them on five symbolic tasks. These tasks fall into context-free and context-sensitive categories on Chomsky's hierarchy. We discuss the bit complexity of encoding symbolic tasks as an automaton and a set of knowledge tuples. Our experiments show that while enterprise LLMs, with their higher parameters and curated datasets, outperform open-source models, all models exhibit similar performance trends that align with the knowledge-tuple encoding of symbolic tasks. Fine-tuning these models on symbolic tasks has little effect on their performance, indicating that LLMs do not learn the rules of symbol manipulation but rather encapsulate relationships in terms of tuples. The contrast in the number of parameters required to encode symbolic tasks suggests a need for developing larger models capable of learning automata rather than merely storing information.

## 3. Background

Ideally, a model should either encode mathematical tasks as knowledge tuples or learn to simulate a machine that can solve such tasks efficiently. In this section, we examine the parameters required by models using both approaches. We relate tasks to their complexity, derived from Chomsky's hierarchy, with detailed definitions provided in the appendix. Furthermore, prior work by Allen-Zhu et al. [26] has studied the bit capacity of large language models. Following their setup, we use the conversion of 2 bits per parameter to estimate the number of parameters required.

### 3.1. Addition of Sequence of Numbers

A finite state machine can add two numbers. Addition of sequence of numbers can be represented by production rules $A \rightarrow tB; A \rightarrow t$, where $A$ is the sequence of numbers, $t$ is the sum of partial sequence and $B$ is the remaining sequence. It is clear from the construction that a non-deterministic PDA can solve this problem. At each step of operation, the nPDA computes the sum of two numbers of arbitrary but finite number of digits. Let the the number of digits be $n$ and $m$. Then the number of digits in the sum of two numbers is $max(n, m) + 1$. There are two ways a LLM can encode the the rules of addition : 1) Encoding the states and stack in the internal parameters of the model, 2) Encoding addition rules for each pair of numbers. Encoding the nPDA requires $log_2(\#(STATES) + MAXLEN(stack))$ bits which is considerable lower than encoding specific addition rules for numbers of arbitrary number of digits.

**Proposition 3.1.** *For the addition of two base $p$ numbers with finite digits $n$ and $m$, respectively, where $n \geq m$, the encoding of their sum requires at most $(2n + m + 1)\log_2 p$ bits. This account includes the possibility of a carryover in the addition, which may increase the length of the resulting number by one digit.*

*Proof.* Consider two numbers $A$ and $B$ represented in base $p$, where $A$ has $n$ digits and $B$ has $m$ digits with $n \geq m$. Each digit in these numbers requires $\log_2 p$ bits to encode due to the need to distinguish between $p$ different values.

**Step 1: Encoding Individual Numbers.**

- The number $A$ requires $n \log_2 p$ bits.
- The number $B$ requires $m \log_2 p$ bits.

**Step 2: Maximum Size of Sum.** When adding $A$ and $B$, the maximum number of digits in the sum, $C = A + B$, could be $n + 1$ (considering the possibility of a carryover from the most significant digit).

**Step 3: Encoding the Sum.** The sum $C$ hence requires at most $(n + 1) \log_2 p$ bits.

**Step 4: Total Encoding Requirement.** Adding the encoding requirements together, the total number of bits needed is:

$$n \log_2 p + m \log_2 p + (n + 1) \log_2 p = (2n + m + 1) \log_2 p.$$

This calculation confirms that the sum of the encoding requirements for the numbers and their potential maximum sum, considering carryover, matches and proves the proposition. $\square$

**Corollary 3.2.** *Consider a sequence of $N$ numbers, each of base $10$ and with at most $D$ digits. To completely memorize all possible addition operations within this sequence, a Large Language Model (LLM) must store at least $\eta_A (3D_{max} - 2) \log_2 10$ bits of information, where:*

$$D_{max} = \lfloor 1 + \log_{10} \left( (10^D - 1) \cdot N \right) \rfloor,$$

*and*

$$\eta_A = \frac{\lambda(\lambda + 1)}{4}, \qquad \lambda = N(10^D - 1)$$

*representing the total number of distinct pairwise addition operations that could be required to solve the sequence. Here, $D_{max}$ represents the maximum number of digits in any number or partial sum within the given sequence, and $\eta_r$ quantifies the total number of unique addition rules required for comprehensive solution strategies.*

Table 1 displays the maximum size of sequences that can be effectively solved by LLMs through the memorization of all necessary addition rules.

## 3.2. Multiplication

Multiplicating two numbers of arbitrary digits is a context-sensitive task. It requires a finite state machine augmented with a finite tape. For the input expression of form $a \times b$ on the input tape, the tape pointer reads the digits of $a$ from right to left. For digit $d$ with place value $v$ in $a$, the pointer moves right to the output part of the tape and increments it $d * v * b$ times. The pointer moves back to the digit, sets it to <BLANK>, and moves to the digit on the left.

**Proposition 3.3.** *An automaton, which includes a finite state controller and a finite length tape, capable of computing the multiplication of an $n$-digit number and an $m$-digit number, requires encoding with*

$$\log_2 6 + 2 \cdot O(n + m) \cdot \log_2 p \text{ bits.}$$

*Proof.* To compute the multiplication of two numbers, each represented by $n$ and $m$ digits respectively, consider an automaton designed as follows:

**State Design:** The finite state controller is structured with six distinct states to manage the computation:

1. Two states are dedicated to reading the digits of the numbers, specifically one state for each number ($n_1$ and $n_2$).
2. One state is allocated for writing the output of the multiplication.
3. Three additional states facilitate the movement of the tape head to the appropriate positions corresponding to the digits of the two numbers and the resulting output.

**Tape Complexity:** The tape must accommodate the digits of both numbers and the resultant product. The maximal length of the product of two numbers, each with $n$ and $m$ digits, is $n + m$ digits. Therefore, the tape's length is approximately $O(2 \cdot (n + m))$ to account for both input numbers and their maximal output size.

**Bit Encoding:** The number of bits required to encode the state machine is determined by the logarithm of the number of states, giving $\log_2 6$ for the states. The tape encoding requires $2 \cdot O(n + m) \cdot \log_2 p$ bits, where $\log_2 p$ bits are needed to encode each digit of the input and output numbers, assuming the numbers are in base $p$.

Combining these components, the total encoding requirement for the automaton is thus:

$$\log_2 6 + 2 \cdot O(n + m) \cdot \log_2 p \text{ bits.}$$

This completes the proof. $\qquad\square$

**Proposition 3.4.** *Given two integers $a$ and $b$, where $a$ is an $n$-digit number and $b$ is an $m$-digit number in base $p$, a multiplication rule that expresses the relationship $a \times b = \gamma$ can be encoded as a knowledge tuple $(a, b, \gamma)$. This tuple can be stored using at most $2(n + m) \log_2 p$ bits.*

*Proof.* The multiplication of $a$ and $b$ results in $\gamma$, where $\gamma$ can have at most $n + m$ digits (considering the maximum carry-over in base $p$):

- Each digit of $a$ and $b$ can be encoded using $\log_2 p$ bits because each digit represents a value from 0 to $p - 1$.
- Therefore, $a$ requires $n \cdot \log_2 p$ bits and $b$ requires $m \cdot \log_2 p$ bits.
- The product $\gamma$ can be as large as $n + m$ digits, thus requiring at most $(n + m) \cdot \log_2 p$ bits.

The knowledge tuple $(a, b, \gamma)$ encapsulates the complete multiplication expression and therefore combines the storage requirements of $a$, $b$, and $\gamma$. Thus, the total bit requirement is the sum of the bits needed to store $a$, $b$, and $\gamma$:

$$2(n + m) \log_2 p \text{ bits,}$$

assuming the maximum possible size for $\gamma$ and ignoring potential savings from compressing common information between $a$, $b$, and $\gamma$. This completes the proof. $\qquad\square$

**Corollary 3.5.** *Given all possible tuples $(a, b, \gamma)$ where $a$ and $b$ are numbers in base 10 with a maximum of $D_{max}$ digits, these tuples can be encoded in a Large Language Model (LLM) using at most*

$$4D_{max} \cdot 10^{2D_{max}} \cdot \log_2 10 \text{ bits,}$$

*where $D_{max}$ is the maximum number of digits in any number among $a$, $b$, and $\gamma$.*

*Proof.* To encode each of the tuples $(a, b, \gamma)$ where $a$ and $b$ are up to $D_{\max}$ digits:

1. The total number of possible values for $a$ or $b$ is $10^{D_{\max}}$, since each digit can range from 0 to 9.
2. The product $\gamma = a \times b$ can have at most $2D_{\max}$ digits (considering the worst-case scenario of multiplication).
3. Thus, the number of possible tuples $(a, b, \gamma)$ is $10^{D_{\max}} \times 10^{D_{\max}} \times 10^{2D_{\max}} = 10^{4D_{\max}}$.
4. Each tuple then requires encoding that can be estimated by calculating the total number of bits to represent each digit of $a$, $b$, and $\gamma$ in binary:

$$4D_{\max} \cdot \log_2 10 \text{ bits per digit} \times 10^{2D_{\max}} \text{ possible tuples.}$$

This provides upper bound on the bit requirement to encode tuples in an LLM. $\qquad\square$

### 3.3. Symbolic Counting

Counting the frequency of a character $c$ in a given string $s$ is well solved using an automaton equipped with a counter and a register. Let $V$ be the set of all possible characters s.t. $c \in V$ and $s \in V^*$. First, the automaton reads $c$ and stores it in the register. The finite state controller then reads characters $c' \in s$ step by step and increments the counter when $c' = c$.

**Proposition 3.6.** *An automaton capable of counting the frequency of a character in a given string can be encoded in at most $log_2 3 + log_2|V| + \lfloor 1 + \log_p N_{max} \rfloor * \log_2 p$ bits, where $N_{max}$ is the maximum length of an input string in base $p$*

*Proof.* **Step 1 : Encoding the states**: The above discussed automaton can function with 3 states. The automaton starts in state $q0$, reads the character $c$ into the register, and moves to the next state ($q1$). The automaton now reads the string, one character at a time. If the input character matches the character in the register, the automaton moves into state $q2$. Otherwise, it remains in $q1$. The states $\{q0, q1, q2\}$ can be encoded in $log_2 3$ bits.
    **Step 2 : Encoding the register** : Since the register holds one character $c \in V$, it can be encoded in $log_2|V|$ bits.
    **Step 3: Encoding the counter**: The maximum value the counter would need to hold is the length of the longest string. Thus counter can be encoded in $\lfloor 1 + \log_p N_{max} \rfloor * \log_2 p$ bits. □

    LLMs can mimic this behavior by encoding knowledge tuples $(c, s, n)$

**Proposition 3.7.** *The knowledge tuple $(c, s, n)$, where $n$ is the frequency of $c$ in $s$ can be encoded in atmost $(N_{max} + 1)log_2|V| + \lfloor 1 + \log_p N_{max} \rfloor * \log_2 p$ bits.*

**Corollary 3.8.** *An LLM can encode the entire sample space of $(c, s, n)$ knowledge tuples of maximum string length $N_{max}$ (in base $10$) in atmost $|V|^{N_{max}+1}\Delta$, where*

$$\Delta = (N_{max} + 1)log_2|V| + \lfloor 1 + \log_{10} N_{max} \rfloor * \log_2 10$$

    Encoding the entire space of possible string combinations is a gargantuan task. It's very inefficient, even for massive LLMs. Especially when counting can be reduced to the addition of a sequence of ones and zeros, where 1 represents a character match and 0 represents no match. This reduces symbolic counting into a two-step process:
    **Step 1**: Using attention mechanism to reduce the character matching problem to a binary sequence that represents a match. This can be encoded in atmost $N_{max} + (N_{max} + 1)log_2|V|$ bits.
    **Step 2** : Perform sum of sequence on the sequence of 1s and 0s. There will be total of $2N_{max}$ knowledge tuples to encode addition in form $(n, 1, n+1)$ and $(n, 0, n)$. These can be encoded in almost $2N_{max} * (1 + 2\lfloor 1 + \log_{10} N_{max} \rfloor \log_2 10)$ bits.
    Thus, the bits required by LLMs to encode symbolic counting are:

$$N_{max} + (N_{max} + 1)log_2|V| + 2N_{max} * (1 + 2\lfloor 1 + \log_{10} N_{max} \rfloor \log_2 10)$$

### 3.4. Parameter Estimation

Based on the estimates discussed above, we now find out the computational capacity of large language models with respect to the tasks of addition, multiplication, and counting. Table 1 reflects the bound on input size for given tasks for a model of 7B and 180B parameters, respectively. These bounds are much closer to the performance that we observe in our experiments. Most LLMs are trained on multiple tasks and are primarily focused on modeling language rather than symbolic tasks. Hence, our results show that LLM performance degrades for sample sizes that are way smaller than the bounds discussed.
    Table 2 shows the number of parameters required for the two encoding systems for a fixed input size on the given tasks. It is observed that, if neural network models could simulate the machines to solve the task, they would require considerably less number of parameters. Current LLMs have parameters more aligned with the knowledge tuple encoding system.

| Task | Size Parameter | 7B | 180B |
|---|---|---|---|
| Sum of Sequence | Sequence Length | 363 | 1660 |
| Multiplication | $\#digits(n_1) + \#digits(n_2)$ | 8 | 9 |
| Symbolic Counting | String Length | 1e8 | 2.5e9 |

**Table 1**
Maximum input size of the sample for the three tasks for 7B and 180B parameter models. The tasks are encoded as knowledge tuples.

| Task | Size Parameter Name | Parameter Value | Machine Encoding | Knowledge Tuple Encoding |
|---|---|---|---|---|
| Sum of Sequence | Sequence Length | 100 | 8 | 407M |
| Multiplication | $\#digits(n_1) + \#digits(n_2)$ | 8 | 28 | 2B |
| Symbolic Counting | String Length | 1000 | 11 | 30,929 |

**Table 2**
Comparison of the number of parameters required in the model for machine encoding vs knowledge tuple encoding.

# 4. Experimental Setup

Fundamental mathematics operations, like addition and multiplication, are well understood as symbolic operations. Numbers can be represented as composite symbols composed of a pre-defined set of symbols, i.e., $0 - 9$ numerals. We analyze the performance of large language models on varying difficulty of the tasks modeled as the input size. All models are tested on 100 samples for each difficulty level. We keep a consistent seed for data generation to ensure a fair comparison.

## 4.1. LLM Models

We conduct our experiments on eight LLM models, including four enterprise models and four open-source models. The enterprise models are GPT-3.5 (gpt-3.5-turbo-0125), GPT-4 (gpt-4-0125-preview), Gemini (gemini-pro-1.0), and Claude (claude-3-haiku-20240307). The open-source models analyzed are Llamma 2 (13B), Llemma (7B), Deepseek (7B), and MetaMath (7B). All open source models except Llamma-2 13B are pre-trained on math tasks. Llemma 7B is Llamma-7B based model fine tuned on math datasets. MetaMath is fine-tuned on Llemma-7B. Deepseek is trained on math datasets like OpenWebMath.

**Model query parameters** We keep the maximum token size of 4096 for all models. All models are queried with temperature of 0 to maximize determinism in the generated text. Models are not given access to any external API's, python interpretter, documents or databases, to ensure analysis of standalone LLM performance. No examples are provided in the prompt, and single prompt query is used for each input sample. Zero-shot COT method is used to allow models to navigate the problems without any help.

**Prompting** All prompts are composed of two components: *system prompt* and *user prompt*. Gemini API does not have separate fields for system and user prompts, hence we concatenate both. System prompt is composed of the following components:

1. **Identity** - e.g. for sum of sequence, the prompt stats with *"You are number crunching machine..."*
2. **Input type** - the input type can be *sequence of numbers*, *arithmetic expression* etc.
3. **Command** - instruction on what to do with the input. e.g. *calculate the sum*
4. **Constraint** [Optional] - Any extra guideline or constraint for the model, e.g. *Treat the upper and lower case characters as separate*
5. **Zero-Shot COT** - To help models present the intermediate thoughts and reasoning without providing any example, we use Zero-shot-COT method. This is done by appending *Let's think step by step*, at the end of the prompt.

The user prompt has two parts : 1) *input identifier* and 2) *input sequence*

## 4.2. Symbolic Tasks

| Task | Categorization by Chomsky's Hierarchy | Machine |
|---|---|---|
| Sum of Sequence | Sum of sequence od multi-digit numbers can be constructed as: $A \rightarrow tB; A \rightarrow t$ Hence it is a context free task | Pushdown Automata |
| Multiplication | Multiplication of two multi-digit numbers is a context-sensitive task as it requires a bounded tape to perform the operation. | Linearly Bounded Automata |
| Symbolic Counting | Counting the matches of a character in a string requires a counter with a FSM, | Counter Automata |
| Modular-10 Arithmetic | A modular arithmetic expression with brackets require a stack to store intermediate results and is thus context-free task | Pushdown Automata |
| Decimal Arithmetic | Decimal Arithmetic requires a tape to perform multiplication and thus is context-sensitive. | Linearly Bounded Automata |

**Table 3**
Categorization of Symbolic Tasks in Chomsky's Hierarchy

Table 3 shows the anchoring of symbolic tasks in Chomsky's Hierarchy. Sections 3.1, 3.2 and 3.3 provide the machine encoding for addition, multiplication and counting tasks, respectively. Modulo-10 arithmetic simplifies addition and multiplication to single digit numbers, but add the complexity of brackets placing them in context-free category. Decimal Arithmetic provides a perspective on the precision handling capability of LLMs, sourcing its theoretical framework from addition and multiplication tasks. We define various tasks based on complexity as follows:

1. **Sum of Sequence** The model is tasked with computing the sum of the sequence of numbers. The numbers in the sequence are sampled uniformly in the range $10-99$. Task difficulty is modeled as the length of input sequences. We test the sequence of sizes $5, 10, 20, 50, 100, 200$, and $500$.

2. **Modulo-10 Arithmetic** While Deletang et al. [4] used modulo-5 arithmetic to benchmark RNN and Transformer models. We follow a similar approach and analyze LLMs' performance on modulo-10 arithmetic expressions. The expressions are composed of the following operations: addition, subtraction, multiplication, and brackets. The difficulty of the task is modeled as the length of expression. We test LLMs with expressions of length $5, 10, 20, 50, 100, 200, 500, 1000$, and $2000$.

3. **Decimal Arithmetic** We test the precision handling capacity of LLMs by introducing decimal numbers in arithmetic expressions. The difficulty in task is modeled as the number of digits in the fractional part of decimal, while keeping one digit in whole number part. We test the expressions with $1, 2, 3, 4, 5$, and $6$ fractional digits,

4. **Multiplication** This tests LLM models on the multiplication of 2 whole numbers. The difficulty in task is modeled as the sum of number of digits in multiplier and the multiplicand. Multiplication with combined digits of $2, 4, 6, 8, 10$, and $12$ are tested.

5. **Symbolic Counter** The LLMs are tested on their ability to count the number of character matches in a string. The difficulty of task is modeled as the length of string. We test LLMs on string lengths on $5, 10, 20, 50, 100, 200, 500, 1000$, and $2000$

## 4.3. Fine Tuning

We fine-tune GPT3.5 model on sum of sequence and symbolic counting tasks. For sum of sequence we fine-tune specifically for sequence lengths $5, 10, 20, 50$ and $100$ with $300$ samples for each seq length.
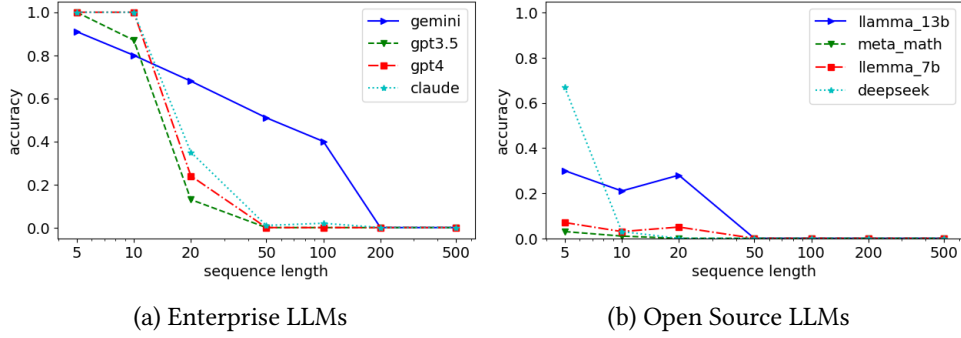
(a) Enterprise LLMs                (b) Open Source LLMs

**Figure 1:** Performance of LLMs on Sum of Sequence task



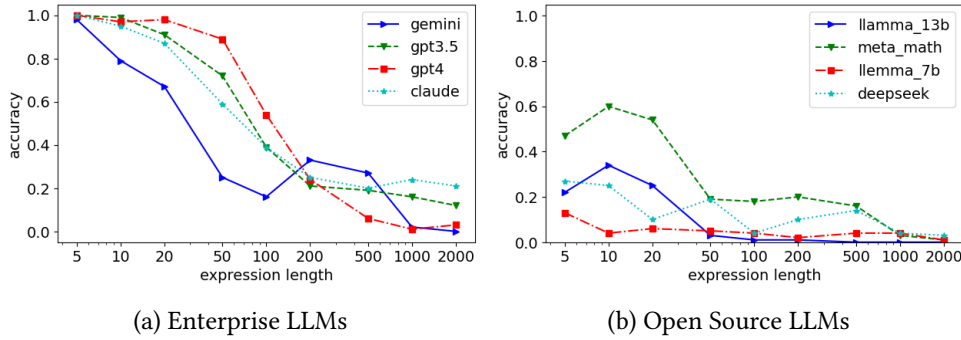(a) Enterprise LLMs                (b) Open Source LLMs

**Figure 2:** Performance of LLMs on Modulo 10 Arithmetic task

The validation set is composed of 50 samples for each length. Finally the model is tested on all sequence lengths i.e, 5, 10, 20, 50, 100, 200, and 500 with 100 samples for each length. Similarly, for the symbolic counting tasks, the model is fine-tuned with strings of length 50, 100, 200, and 500, with 300 for each length. The model is tested for all lengths with 100 samples each.

# 5. Results and Discussion

Figures 1 - 6 show the performance of LLMs on the five symbolic tasks. In all cases, we can see that performance starts degrading as the difficulty of input samples increases. By comparing these results with tables 1 and 2 we can see that performance starts degrading for far simpler examples than anticipated. This indicates that LLMs have not learned to solve symbolic problems but rather have a notion of input-output pairings. From the figures, we can see that while GPT3.5 and Claude Haiku models perform very similarly in all tasks, Gemini is able to sustain performance longer than the other models on the sum of sequence task while it is usually behind other models in the rest of the tasks. GPT4 outperforms other models in modulo-10 arithmetic, decimal arithmetic, and multiplication tasks while lagging behind in other tasks. Claude Haiku performs fairly consistently in all tasks, being the smallest of all enterprise models. Out of all models, Lemma 7B is the worst-performing model across all tasks. Deepseek and MetaMath perform slightly better due to their math pre-training.

In computational tasks, the difficulty is often represented by a parameter that increases the operations in the corresponding state machine without altering the accepted transitions or the number of states. Typically, longer strings are considered more difficult. However, this parameterization does not change the machine's construction, so a neural network capable of emulating the machine should produce correct outputs regardless of input length. While memory and precision constraints provide an upper bound on the input length, modern computers have bounds significantly higher than those used in this study. For instance, in multiplication, increasing the number of digits in the operands necessitates more single-digit multiplications and additions. This requirement, however, does not alter the fundamental
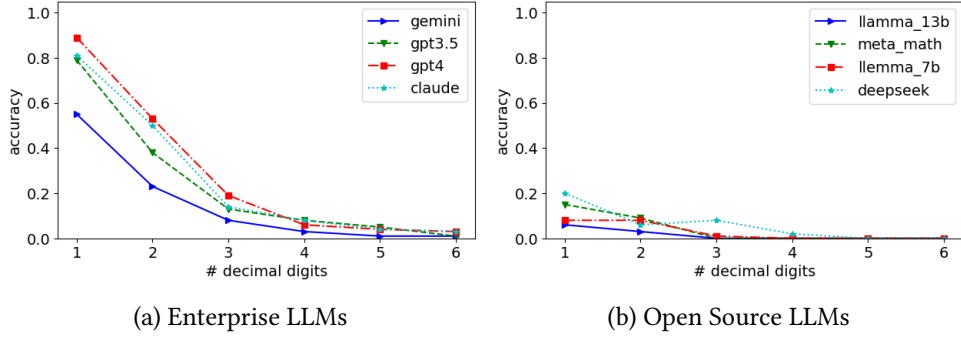
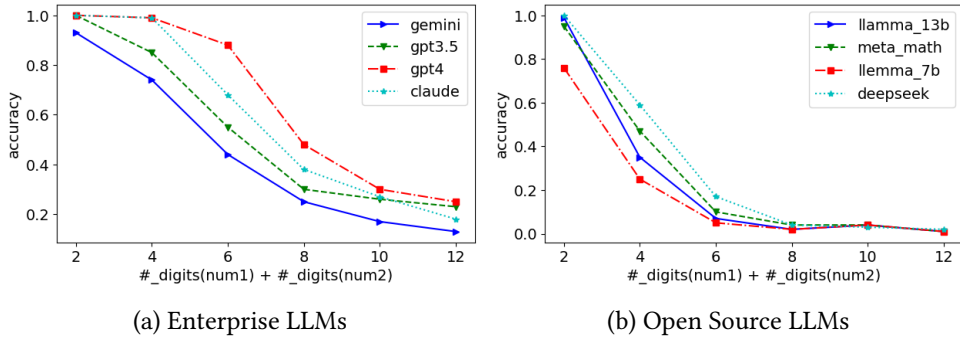**Figure 3:** Performance of LLMs on Decimal Arithmetic task



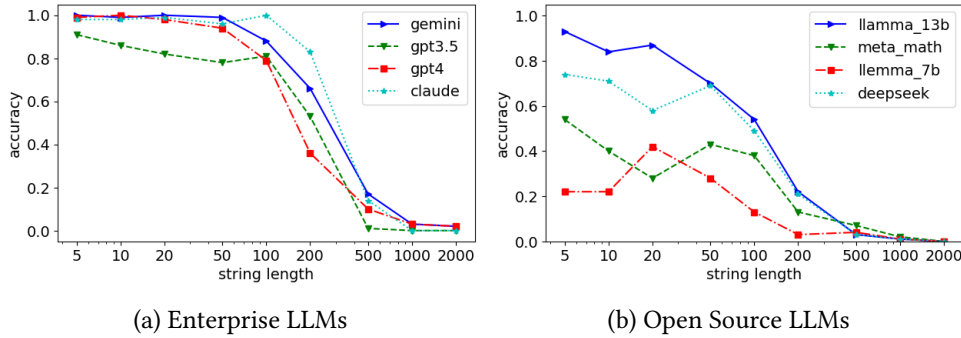**Figure 4:** Performance of LLMs on Multiplication task



**Figure 5:** Performance of LLMs on Symbolic Counter task

operations of multiplication or addition. We represent this complexity with a single parameter: the combined number of digits in both operands. Our experiments indicate that large language models (LLMs) fail to maintain accuracy as the number of digits in the operands increases.

**Enterprise and Open Source Models** Multiplication is the only task where open-source models and enterprise models start with similar performances. In symbolic counting, the Llamma 13B model performs similarly to enterprise models for strings of length 5. In all cases, the performance of open-source models degrades much faster than that of enterprise models. In the other three tasks, open-source models perform far below enterprise models, even for simpler cases. An argument could be made that open-source models are smaller than the enterprise models tested in this work. This further indicates that performance is highly dependent on the models' size, making the knowledge-tuple encoding hypothesis much stronger.

**Math trained models** Out of the three math-trained models, Deepseek gives the best overall performance. However, even math-trained models are not able to sustain their performance on larger examples. Symbolic counting is the only task where the Llamma 13B model outperforms math-trained
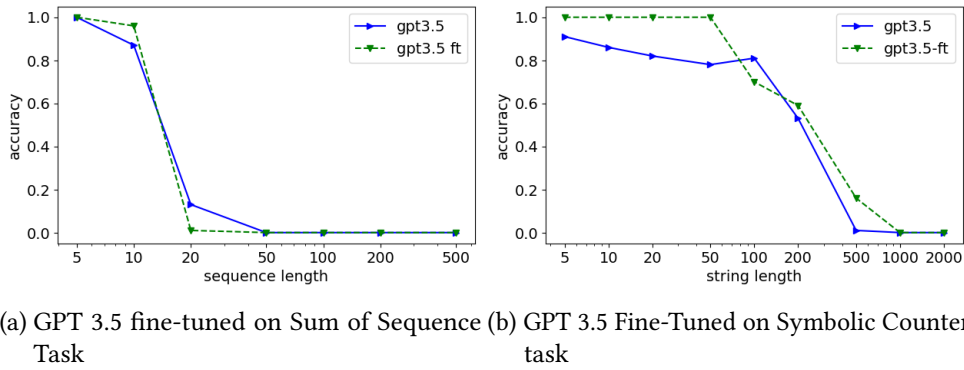
(a) GPT 3.5 fine-tuned on Sum of Sequence
Task

(b) GPT 3.5 Fine-Tuned on Symbolic Counter
task

**Figure 6:** Effect of fine-tuning on GPT 3.5 performance on symbolic tasks

models. All math-trained models have the number of parameters of Llamma 13B.

**Effect of fine tuning** Figure 6 shows the comparison of vanilla and fine-tuned GPT3.5 models on the sum of sequence and symbolic counting tasks. We can observe that fine-tuning has helped performance in smaller examples but made larger examples a little worse. Overall, all fine-tuning and closing performances follow vanilla performance, and there is no improvement in the generalization of the tasks.

## 6. Conclusion

In this work, we analyze the performance of enterprise and open-source LLMs trained on text and mathematical datasets on five symbolic tasks. The tasks occupy context-free and context-sensitive categories on Chomsky's hierarchy. We discussed the bit complexity of encoding symbolic tasks as an automaton and set of knowledge tuples. Our experiments show that although enterprise LLMs perform better than open-source models catering to higher parameters and curated dataset, all models show similar trends in performance and align with knowledge-tuple encoding of symbolic tasks. Fine tuning on these tasks as little effect on the performance of LLMs. This clearly shows that LLMs are not learning the rules of symbol manipulation but rather encapsulate relationships in terms of tuples. The contrast in the number of parameters required to encode symbolic tasks calls for a push in the direction of large models capable of learning automaton rather than storing information.

## References

[1] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Advances in neural information processing systems 35 (2022) 24824–24837.

[2] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, K. Narasimhan, Tree of thoughts: Deliberate problem solving with large language models, Advances in Neural Information Processing Systems 36 (2024).

[3] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, et al., Graph of thoughts: Solving elaborate problems with large language models, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 38, 2024, pp. 17682–17690.

[4] G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, et al., Neural networks and the chomsky hierarchy, arXiv preprint arXiv:2207.02098 (2022).

[5] C. W. Omlin, C. L. Giles, Extraction of rules from discrete-time recurrent neural networks, Neural networks 9 (1996) 41–52.

[6] C. L. Giles, C. W. Omlin, Extraction, insertion and refinement of symbolic rules in dynamically driven recurrent neural networks, Connection Science 5 (1993) 307–337.

[7] C. W. Omlin, C. L. Giles, Constructing deterministic finite-state automata in recurrent neural networks, Journal of the ACM (JACM) 43 (1996) 937–972.

[8] C. W. Omlin, C. L. Giles, Training second-order recurrent neural networks using hints, in: D. Sleeman, P. Edwards (Eds.), Machine Learning Proceedings 1992, Morgan Kaufmann, San Francisco (CA), 1992, pp. 361–366. URL: https://www.sciencedirect.com/science/article/pii/B9781558602472500516. doi:https://doi.org/10.1016/B978-1-55860-247-2.50051-6.

[9] A. Mali, A. Ororbia, D. Kifer, L. Giles, On the computational complexity and formal hierarchy of second order recurrent neural networks, arXiv preprint arXiv:2309.14691 (2023).

[10] A. Mali, A. Ororbia, D. Kifer, L. Giles, Recognizing long grammatical sequences using recurrent networks augmented with an external differentiable stack, in: International Conference on Grammatical Inference, PMLR, 2021, pp. 130–153.

[11] J. Stogin, A. Mali, C. L. Giles, A provably stable neural network turing machine with finite precision and time, Information Sciences 658 (2024) 120034. URL: https://www.sciencedirect.com/science/article/pii/S0020025523016201. doi:https://doi.org/10.1016/j.ins.2023.120034.

[12] N. Dave, D. Kifer, C. L. Giles, A. Mali, Stability analysis of various symbolic rule extraction methods from recurrent neural network, in: ICLR 2024 Workshop on Bridging the Gap Between Practice and Theory in Deep Learning, 2024.

[13] A. Mali, A. G. Ororbia, D. Kifer, C. L. Giles, Recognizing and verifying mathematical equations using multiplicative differential neural units, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, 2021, pp. 5006–5015.

[14] D. Saxton, E. Grefenstette, F. Hill, P. Kohli, Analysing mathematical reasoning abilities of neural models, arXiv preprint arXiv:1904.01557 (2019).

[15] N. Dave, R. Bakes, B. Pursel, C. L. Giles, Math multiple choice question solving and distractor generation with attentional gru networks., International Educational Data Mining Society (2021).

[16] B. Mistry, K. Farrahi, J. Hare, A primer for neural arithmetic logic modules, Journal of Machine Learning Research 23 (2022) 1–58.

[17] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu, et al., Deepseek llm: Scaling open-source language models with longtermism, arXiv preprint arXiv:2401.02954 (2024).

[18] Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, A. Q. Jiang, J. Deng, S. Biderman, S. Welleck, Llemma: An open language model for mathematics, arXiv preprint arXiv:2310.10631 (2023).

[19] L. Yu, W. Jiang, H. Shi, J. Yu, Z. Liu, Y. Zhang, J. T. Kwok, Z. Li, A. Weller, W. Liu, Metamath: Bootstrap your own mathematical questions for large language models, arXiv preprint arXiv:2309.12284 (2023).

[20] N. Dziri, X. Lu, M. Sclar, X. L. Li, L. Jiang, B. Y. Lin, S. Welleck, P. West, C. Bhagavatula, R. Le Bras, et al., Faith and fate: Limits of transformers on compositionality, Advances in Neural Information Processing Systems 36 (2024).

[21] P. Shakarian, A. Koyyalamudi, N. Ngu, L. Mareedu, An independent evaluation of chatgpt on mathematicalword problems (mwp), in: CEUR Workshop Proceedings, volume 3433, CEUR-WS, 2023.

[22] OpenAI, Introducing chatgpt, 2022. URL: https://openai.com/index/chatgpt/.

[23] S. Frieder, L. Pinchetti, , R.-R. Griffiths, T. Salvatori, T. Lukasiewicz, P. Petersen, J. Berner, Mathematical capabilities of chatgpt, in: A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Advances in Neural Information Processing Systems, volume 36, Curran Associates, Inc., 2023, pp. 27699–27744.

[24] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al., Training verifiers to solve math word problems, arXiv preprint arXiv:2110.14168 (2021).

[25] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, J. Steinhardt, Measuring

mathematical problem solving with the math dataset, arXiv preprint arXiv:2103.03874 (2021).

[26] Z. Allen-Zhu, Y. Li, Physics of language models: Part 3.3, knowledge capacity scaling laws, 2024. URL: https://api.semanticscholar.org/CorpusID:269005957.

[27] N. Chomsky, On certain formal properties of grammars, Information and Control 2 (1959) 137–167. URL: https://www.sciencedirect.com/science/article/pii/S0019995859903626. doi:https://doi.org/10.1016/S0019-9958(59)90362-6.

[28] N. Chomsky, Three models for the description of language, IRE Transactions on Information Theory 2 (1956) 113–124. doi:10.1109/TIT.1956.1056813.

# A. Chomsky's Hierarchy

Chomsky [27] [28] established a hierarchy (see Figure 7) delineating formal grammars according to their complexity and the corresponding computational devices capable of recognizing languages generated by these grammars. These grammars are classified from Type-3 to Type-0, with Type-0 representing the highest level of complexity.
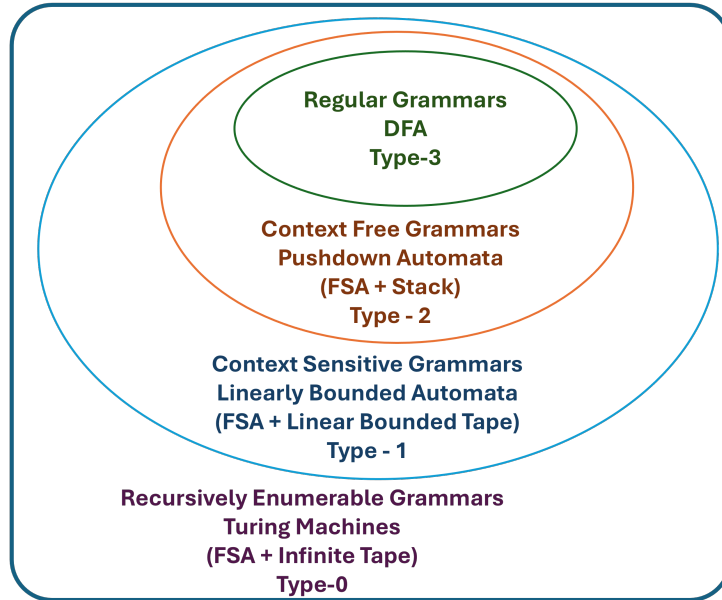


**Figure 7:** Chomsky's Hierarchy

**Definition A.1.** A **Deterministic Finite (State) Automaton** is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where $Q$ is the set of all states, $q_0 \in Q$ is the initial state. $\Sigma$ is the set of alphabets such that all input strings are a subset of $\Sigma^*$. $\delta : Q \times \Sigma \to Q$ are a set of transition rules. $F \subseteq Q$ are accepting states.

A DFA reads input one symbol at time and updates its state determined by the transition rules $\delta$. The input string is accepted if the DFA reaches a state $q \in F$ at the end of the input.

**Definition A.2.** A **Non-Deterministic Finite (State) Automaton** is a 5-tuple similar to DFA with a non-deterministic transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is a power set of all automaton states.

All non-deterministic FSA can be reduced to their deterministic counterparts without the loss of expressivity. Finite State Automata can accept all regular languages.

**Definition A.3.** A **Finite State Transducer** (FST) is a finite state machine with an output tape. Formally it is a 6-tuple, $(Q, \Sigma, \Gamma, q_0, F, \delta)$ where $\Lambda$ is the set of output alphabet, and $\delta : Q \times \{\Sigma \cup \epsilon\} \to Q \times \{\Lambda \cup \epsilon\}$.

**Definition A.4.** A **Pushdown Automaton** is a 7-tuple, $(Q, \Sigma, S, \delta, q_0, z, F)$, where $S$ is a set of stack symbols and $z \in S$ is initial stack symbol. Transition function is defined as $\delta : Q \times \{\Sigma \cup \epsilon\} \times S \to Q \times S^*$

**Definition A.5.** A **Pushdown Transducer** is a pushdown automata with an output tape. Specifically it is a 8-tuple, $(Q, \Sigma, \Lambda, S, \delta, q_0, z, F)$, where $\Lambda$ is a set of output symbols. All other symbols carry the same meaning as a pushdown automata. The transition function is defined as $\delta : Q \times \{\Sigma \cup \epsilon\} \times S \to Q \times \{\Lambda \cup \epsilon\} \times S^*$

**Definition A.6.** A **Turing Machine** is a 7-tuple, $(Q, \Gamma, \beta, \Sigma, \delta, q_0, F)$. $\Gamma$ is the set of all tape alphabets. $\beta \in \Gamma$ is blank symbol, which is default on any on-empty cell on the tape. Input symbols are part of tape symbols, $\Sigma \subseteq \Gamma$ $b$, and transition function is defined as $\delta : Q \times \Gamma \to Q \times \Gamma \times \{N, L, R\}$. Here $\{N, L, R\}$ are *no shift*, *left shift* and *right shift* of the tape pointer respectively.

A FST can generate all regular languages and similarly, a pushdown transducer can generate all context free languages. A linearly bounded automaton is a Turing Machine with a finite size tape.

## B. Symbolic Tasks and State Machines

The five symbolic tasks discussed in this paper—sum of sequences, modular arithmetic, decimal arithmetic, multiplication, and symbolic counting—can be accomplished using three fundamental operations: addition, multiplication, and counting. In this section we will discuss how machines categorized in chomsky's hierarchy can be used for these operations.

### B.1. Addition

The addition of two $n$-digit numbers can be carried out by a two-state finite state transducer (FST). Figure 8a depicts a construction that reads the digits of the two numbers from left to right and outputs the result one digit at a time, from the least significant to the most significant digit. To add a sequence of numbers, we add a stack to FST of figure 8a and make it non-deterministic (figure 8b).
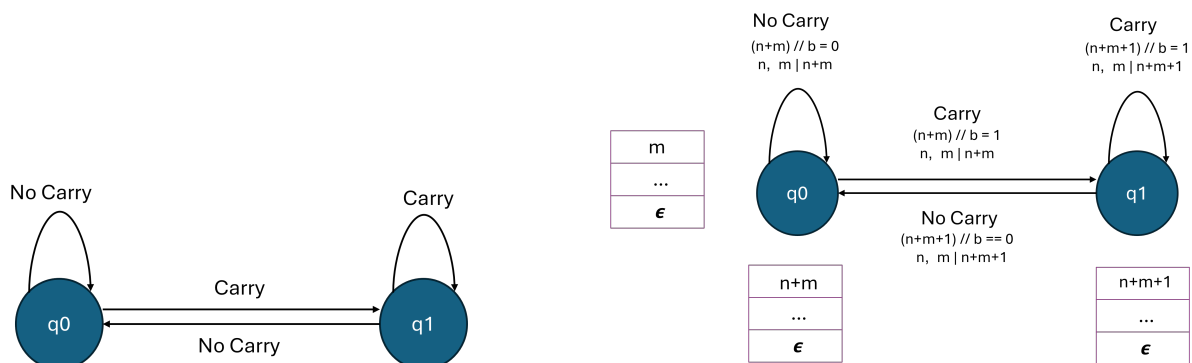


(a) A two-state FST (Finite State Transducer) can add two numbers by reading the digits of each number from right to left, performing addition one digit at a time. The two states represent whether we need to add a carry to the sum or not.

(b) To add a sequence of numbers, the finite state machine depicted in Figure 8a is augmented with a stack to store intermediate addition results. A non-deterministc pushdown transducer is required to ensure that the results are popped from the stack in the correct order.

### B.2. Multiplication

The multiplication of two multi-digit numbers necessitates a tape for storing partial products. As depicted in Figure 9, a linearly bounded automaton (LBA) efficiently performs this task. It can solve the multiplication of two multi-digit numbers by first calculating partial products and then combining them to yield the final product. The number of digits in the partial products and the final product is

constrained by the number of digits in the given numbers. Consequently, a bounded tape is sufficient for evaluating the product.
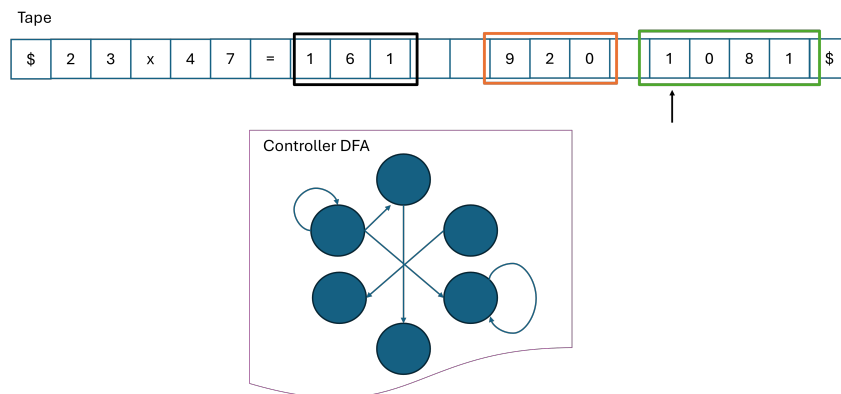


**Figure 9:** The deterministic finite automaton (DFA) controller maneuvers the pointer along the tape to read symbols and execute partial multiplications. Subsequently, these partial results are aggregated to yield the final product.

## B.3. Counting

Counting the occurrences of a specific character in a string can be achieved using a counter machine. In Chomsky's hierarchy, counter machines are more complex than finite state machines but less complex than pushdown automata. Figure 10 shows a construction of a counter machine that can keep the occurrences of a character in a given string. We also take help from a register to store the character to match.
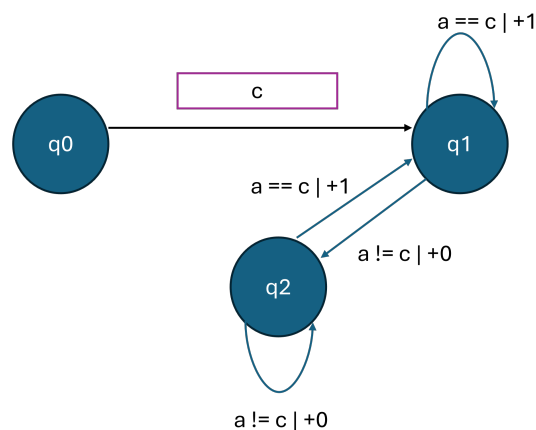


**Figure 10:** The initial state $q0$ reads the character $c$ to match in a register. States $q1$ and $q2$ compare $c$ with the inputs from the string ($a$). The counter increments when $a == c$. The final output is the state of the counter.

## C. Bit Encoding of a Number

**Proposition C.1.** *Given a number $N$ in base $p$, the model requires*

$$\left\lfloor 1 + \log_p(N) \right\rfloor \cdot \log_2 p \text{ bits}$$

*to encode it in symbolic form.*

*Proof.* The number of digits $d$ required to represent a number $N$ in base $p$ can be determined by the formula:

$$d = \left\lfloor 1 + \log_p(N) \right\rfloor.$$

This expression arises because the largest number representable with $d$ digits in base $p$ is $p^d - 1$, and thus:

$$p^{d-1} \leq N < p^d.$$

Taking the logarithm base $p$ of all parts of this inequality results in:

$$d - 1 \leq \log_p(N) < d.$$

Adding 1 to each part and then taking the floor function provides the correct number of digits:

$$d = \left\lfloor 1 + \log_p(N) \right\rfloor.$$

Each digit in base $p$ requires $\log_2 p$ bits for encoding since each digit can take any of $p$ different values, and $\log_2 p$ bits are sufficient to uniquely identify each value. Therefore, the total number of bits required to encode $N$ is the product of the number of digits and the number of bits per digit:

$$\left\lfloor 1 + \log_p(N) \right\rfloor \cdot \log_2 p.$$

This completes the proof. $\square$

## D. Prompt Examples

### GPT 4 on Decimal Arithmetic

**System Prompt**
You are an arithmetic solver for expressions with decimal numbers. Solve the given expression and report the result. Let's think step by step.

**User Prompt**
Expression : (7.6*((-5.7*(4.8+7.5))+(0.3)))

**Output**
Let's solve the expression step by step:
1. First, solve the innermost parentheses: (4.8 + 7.5 = 12.3)
   So, the expression becomes: (7.6 * ((-5.7 * 12.3) + 0.3))
2. Next, multiply (-5.7 ) by (12.3): (-5.7 * 12.3 = -70.11)
   So, the expression becomes: (7.6 * (-70.11 + 0.3))
3. Add (-70.11) and (0.3): (-70.11 + 0.3 = -69.81)
   So, the expression becomes: (7.6 * -69.81)
4. Finally, multiply (7.6) by (-69.81): (7.6 * -69.81 = -530.356\)
Therefore, the result of the given expression is (-530.356).

**Correct Answer**: -530.556
**Correct Output : False**

### Gemini on Sum of Sequence

**System Prompt**
You are number crunching machine. Given a sequence of numbers, calculate the sum of these numbers. Let's think step by step.

**User Prompt**
Sequence of numbers to sum : [96, 93, 72, 45, 66, 22, 28, 30, 22, 65, 18, 69, 48, 33, 39, 21, 66, 41, 50, 51]

**Output**
Sequence of numbers to sum: [96, 93, 72, 45, 66, 22, 28, 30, 22, 65, 18, 69, 48, 33, 39, 21, 66, 41, 50, 51]
Step 1: 96 + 93 + 72 + 45 + 66 + 22 + 28 + 30 + 22 + 65 + 18 + 69 + 48 + 33 + 39 + 21 + 66 + 41 + 50 + 51 = 895
Therefore, the sum of the given sequence of numbers is 895.

**Correct Answer**: 975
**Correct Output : False**

# Llamma 13B on Modulo 10 Arithmetic

**System Prompt**
You are a modulo 10 arithmetic solver. Evaluate the given arithmetic expression and give answer in modulus 10.  Let's think step by step

**User Prompt**
Expression : (((8-(-6--9))-((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)))+(((-9+-9)+(8-(-8)))*((9)*((7-2)*(1)))))

**Output**
You are a modulo 10 arithmetic solver. Evaluate the given arithmetic expression and give answer in modulus 10 Let's think step by step Expression : (((8-(-6--9))-((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)))+(((-9+-9)+(8-(-8)))*((9)*((7-2)*(1))))) Modulus 10 =? Please help me to solve this expression.

Answer:
The given expression is quite complex, but we can break it down step by step and evaluate it modulo 10.

Step 1: Evaluate the innermost expressions first.

(((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)) + ((-9+-9)+(8-(-8)))*((9)*((7-2)*(1))))

Step 2: Evaluate the first term inside the parentheses.

(((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9)))))

Step 3: Evaluate the second term inside the parentheses.

(((-9+-9)+(8-(-8)))*((9)*((7-2)*(1))))

Step 4: Evaluate the product of the two terms.

(((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)) + ((-9+-9)+(8-(-8)))*((9)*((7-2)*(1))))

Step 5: Evaluate the final result modulo 10.

(((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)) + ((-9+-9)+(8-(-8)))*((9)*((7-2)*(1)))) % 10

The result of the expression is:
(((-9*(((-2-(-0))-(9+-1))-(3-(-3*-9))))*(7*1)) + ((-9+-9)+(8-(-8)))*((9)*((7-2)*(1)))) % 10
= (((-9*(-2-(-0))+9*(-1))+(-9+8))*9*6) % 10
= (((-18-9)+8)*9*6) % 10
= ((-27)+8)*9*6) % 10
= (-20*9+8*9) % 10
= (-180+72) % 10
= 152 % 10
Therefore, the result of the given expression modulo 10 is 152.

**Correct Answer**: 9
**Correct Output : False**

# GPT 3.5 on Symbolic Counting

**System Prompt**
You are a symbolic counter whose job is to count the frequency of a given character in the given string. Treat the upper- and lower-case characters as separate characters. Let's think step by step.

**User Prompt**
string :
iLGUgIkcFuSeRnzArycDvEMQRxNTYNajEHthrMGhZhZtCscnVGiBmfTVsJuVvLGGtTiJhUHTfyQFkzQyaTSvYo
VBfVuudSJucOUJhufnBUfFilakBDSqZmyimeIZeBIDhpZpuBQvzGyfyaTWieTsbSlmdCryxzDDpbtirbFEdhGwi
veYmOfeBxAaIBGzjTPguQMLQijd, character to count : w

**Output**
Step 1: Identify the character to count - in this case, it is 'w'.

Step 2: Count the frequency of the character 'w' in the given string.

Let's start counting:

Frequency of 'w' in the given string:
- Lowercase 'w' appears 4 times.
- Uppercase 'W' does not appear in the string.

Therefore, the total frequency of 'w' in the given string is 4.

**Correct Answer**: 1
**Correct Output : False**

# Claude Haiku on Multiplication

**System Prompt**
You are a multiplication engine. Given a multiplication expression of type 'num1 x num2', find the multiplication of num1 and num2. Let's think step by step

**User Prompt**
Expression : 4 x 898

**Output**
Okay, let's think through this step-by-step:
1. The expression given is "4 x 898".
2. We need to find the multiplication of the two numbers, 4 and 898.
3. To do this, we can use the standard multiplication algorithm.
4. First, we multiply 4 by each digit of 898, starting from the ones place:
   - 4 x 8 = 32
   - 4 x 9 = 36
   - 4 x 0 = 0
5. Next, we add up the partial products:
   - 32 + 36 + 0 = 68
6. Therefore, the result of 4 x 898 is 3,592.

**Correct Answer**: 3592
**Correct Output : True**

# Deepseek on Decimal Arithmetic

**System Prompt**
You are an arithmetic solver for expressions with decimal numbers. Solve the given expression and report the result. Let's think step by step.

**User Prompt**
Expression : (((9.0*(-8.2))-(4.7-5.6))*8.3)

**Output**
You are an arithmetic solver for expressions with decimal numbers. Solve the given expression and report the result. Let's think step by step Expression : (((9.0*(-8.2))-(4.7-5.6))*8.3)

## Arithmetic Operations with Decimal Numbers
Arithmetic operations with decimal numbers are similar to arithmetic operations with whole numbers. However, we need to be careful about the placement of the decimal point. Let's review the basic arithmetic operations:

1. Addition: When adding decimal numbers, we align the decimal points and add the numbers as if they were whole numbers. The decimal point in the answer goes directly below the decimal points in the numbers being added. For example:

2. Subtraction: When subtracting decimal numbers, we align the decimal points and subtract as if they were whole numbers. The decimal point in the answer goes directly below the decimal points in the numbers being subtracted. For example:

3. Multiplication: When multiplying decimal numbers, we ignore the decimal points and multiply as if they were whole numbers. Then, we count the number of digits after the decimal points in the numbers being multiplied and add them together. The number of digits in the answer is equal to the sum of the number of digits in the numbers being multiplied. For example:

4. Division: When dividing decimal numbers, we move the decimal point in the divisor to the right until it becomes a whole number. Then, we move the decimal point in the dividend the same number of places to the right. After that, we divide as if they were whole numbers. For example:

## Solve (((9.0*(-8.2))-(4.7-5.6))*8.3)
First, we need to solve the expression inside the parentheses: 4.7 - 5.6 = -0.9 Then, we can simplify the expression: ((9.0 * -8.2) - (-0.9)) * 8.3 = (-73.8 + 0.9) * 8.3 = -72.9 * 8.3 = -605.07
The answer is -605.07.

**Correct Answer**: -605.07
**Correct Output : True**

# Metamath on Modulo 10 Arithmetic

**System Prompt**
You are a modulo 10 arithmetic solver. Evaluate the given arithmetic expression and give answer in modulus 10. Let's think step by step

**User Prompt**
Expression : ((((3)*((9+(((5)--7)*6))-(9)))+(-7-2)-((-6)*(2)))

**Output**
You are a modulo 10 arithmetic solver. Evaluate the given arithmetic expression and give answer in modulus 10 Let's think step by step Expression : ((((3)*((9+(((5)--7)*6))-(9)))+(-7-2)-((-6)*(2))) We can simplify the expression using the order of operations (PEMDAS). First, we simplify the innermost parentheses: ((((3)*((9+(((5)--7)*6))-(9)))+(-7-2)-((-6)*(2))) = ((((3)*((9+((-2)*6))-(9)))+(-7-2)-((-6)*(2))) = ((((3)*((9+(-12))-(9)))+(-7-2)-((-6)*(2))) = ((((3)*((-3)-9))+(-7-2)-((-6)*(2))) = ((((3)*(-12))+(-7-2)-((-6)*(2))) = ((-36)+(-7-2)-((-6)*(2)) = ((-36)-9)-((-6)*(2)) = (-45)-((-6)*(2)) = (-45)-(-12) = (-45)+12 = 7 Now, we give the answer in modulus 10. Since 7 is already in modulus 10, the answer is $\boxed{7}$.The answer is: 7

**Correct Answer**: 9
**Correct Output : False**

# Llemma 7B on Multiplication

**System Prompt**
You are a multiplication engine. Given a multiplication expression of type 'num1 x num2', find the multiplication of num1 and num2. Let's think step by step

**User Prompt**
Expression : 4 x 4

**Output**
You are a multiplication engine. Given a multiplication expression of type 'num1 x num2' find the multiplication of num1 and num2. Lets think step by step Expression : 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x 4 x

**Correct Answer**: 16
**Correct Output : False**