Theoretical Barriers of Modern Tokenizer on Symbolic and Arithmetic Computation in Language Models

Anonymous ARR Submission

Abstract

Tokenization is the first—and often underappreciated—layer of computation in language models. While Chain-of-Thought (CoT) prompting enables transformer models to approximate recurrent computation by externalizing intermediate steps, we show that the success of such reasoning is fundamentally bounded by the structure of tokenized inputs. This work presents a theoretical and empirical investigation into how tokenization schemes, particularly subword-based methods like byte-pair encoding (BPE), impede symbolic computation by merging or obscuring atomic reasoning units. We introduce the notion of *Token Awareness* to formalize how poor token granularity disrupts logical alignment and prevents models from generalizing symbolic procedures. Through systematic evaluation on arithmetic and symbolic tasks, we demonstrate that token structure dramatically affect reasoning performance, causing failure even with CoT, while atomically-aligned formats unlock strong generalization, allowing small models (e.g., GPT-4o-mini) to outperform larger systems (e.g., o1) in structured reasoning. Our findings reveal that symbolic reasoning ability in LLMs is not purely architectural, but deeply conditioned on token-level representations. Full code, prompts, and results are available at Anonymous GitHub.

1 Introduction

Inductive reasoning and arithmetic computation, such as counting, addition, and pattern generalization, are foundational components of symbolic and algorithmic intelligence. These abilities have long been studied across disciplines—from their cognitive development in humans (Wynn, 1990; De Bruijn, 1964) to their formal characterization in logic and computability theory (Boolos et al., 2002; Cooper, 2017). In theoretical computer science, arithmetic primitives like counting and addition have been analyzed in terms of circuit complexity (Jerrum, 1995), computational depth (Fischer et al., 1968), and machine models such as counter automata (Ibarra et al., 2002). Even simple operations—e.g., counting from `1` to `n`—are known to require a depth complexity that grows with input length, imposing strict lower bounds on any computational model (Fischer et al., 1968).

Transformers (Vaswani, 2017), including both autoregressive (GPT-style) (Gregor et al., 2014; Achiam et al., 2023) and non-autoregressive (BERT-style) (Devlin, 2018; Liu et al., 2022) variants, are inherently limited to *constant-depth* computation (Zhang et al., 2024; Delétang et al., 2022; Li et al., 2024a). In neural models, all computation occur within model's latent space $\mathcal{H}$, where the hidden state $\boldsymbol{h}$ encodes intermediate computational representations. Unlike recurrent architectures—where hidden states evolve over time via recursive updates $\boldsymbol{h}_t = g_\theta(\boldsymbol{h}_{t-1})$—Transformers update $\boldsymbol{h}$ only across a fixed number of layers, independent of input length. As a result, a standard Transformer can process (or reason over) its hidden states only a constant number of times, limiting its computational depth and situating it at the lower end of the Chomsky hierarchy (Delétang et al., 2022). This architectural bottleneck fundamentally restricts Transformer-based models—ranging from task-specific expert systems to large-scale LLMs—from solving even basic arithmetic operations such as *counting*, which require iterative updates to internal state and growing depth with input length.

Chain of Thought (CoT)(Wei et al., 2022) revolutionizes the reasoning paradigm by shifting the locus of computation from the latent space $\mathcal{H}$ to the textual output space $\mathcal{O}$(Zhang et al., 2024). By externalizing intermediate reasoning steps into natural language "thoughts," CoT enables transformer-based models to tackle fundamental computational tasks that would otherwise exceed their architectural capacity. These include basic arithmetic and symbolic reasoning operations such as counting,

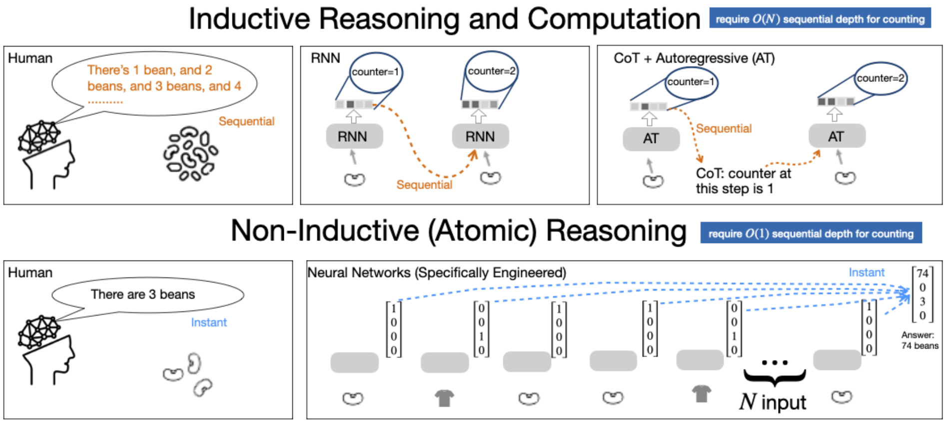

Figure 1: Illustration of inductive reasoning as performed by humans, RNNs, and LLMs with CoT, respectively.

addition, and sequence manipulation. Theoretical studies (Li et al., 2024a; Zhang et al., 2024; Feng et al., 2024) demonstrate that CoT-augmented language models, under idealized assumptions, possess an *upper bound* capacity to simulate computations of arbitrary complexity—thereby extending the class of tasks solvable beyond what standard Transformers can achieve.

Despite extensive theoretical analyses and guarantees on the upper bound of computational abilities (Zhang et al., 2024; Chang and Bisk, 2024), actual model performance remains far below these limits. As LLMs scale from millions to billions of parameters (Achiam et al., 2023), improvements on fundamental tasks such as counting have been marginal—GPT-4, for instance, still struggles to count the number of "r"s in a word. While recent work has explored contributing factors like training data (Allen-Zhu and Li, 2023) and positional encoding (Chang and Bisk, 2024), one of the most basic components—tokenization—has received surprisingly little attention. In particular, modern byte pair encoding (BPE) (Sennrich, 2015) merges multiple characters into single tokens for efficiency, often degrading arithmetic reasoning due to information loss during tokenization. Even OpenAI's latest o1[1] model, which integrates Monte Carlo Tree Search (MCTS) for improved reasoning, achieves only 50% accuracy on long string arithmetic tasks involving 30–40 characters.

In this work, we systematically investigate how tokenization choices can substantially constrain the theoretical reasoning and arithmetic capabilities of neural models. Our approach is model-agnostic, allowing us to evaluate even closed-source LLMs with undisclosed tokenization schemes. Leveraging extensive experiments with Chain of Thought (CoT)—which has been theoretically shown to achieve Turing completeness under idealized conditions (Li et al., 2024b)—we demonstrate that the choice of tokenization plays a critical role in unlocking a model's full computational potential and bridging the gap between theoretical guarantees and practical performance. Neglecting this factor can lead to performance degradations of up to 80%. Moreover, we find that the impact of tokenization is model-dependent: some tokens disproportionately hinder performance on counting tasks, even when the underlying task remains fixed.

2 Neural Networks and Arithmetic: A Revisit

Training neural networks for arithmetic computation. Arithmetic operations—including counting, matching, and bracket balancing—are foundational for symbolic reasoning and more complex algorithmic tasks (Chang and Bisk, 2024). Early studies on training neural networks (NNs) for such tasks focused on architectures capable of handling variable-length inputs. Since multi-layer perceptrons (MLPs) (Rosenblatt, 1958) are inherently limited to fixed-size inputs, initial progress came through recurrent neural networks (RNNs).

Rodriguez et al. (1999) trained early RNNs to recognize the *regular* language $\mathtt{a}^n\mathtt{b}^n$, which requires the network to implicitly count occurrences of `a` and `b`. Of the 50 networks trained, 8 successfully generalized to longer sequences, highlighting RNNs' capacity for basic arithmetic generalization. Building on this, Suzgun et al. (2019) showed that LSTMs could perform more complex *dynamic counting* via bracket pairing tasks, leveraging gating and cell-state mechanisms to maintain multiple

[1]https://openai.com/o1

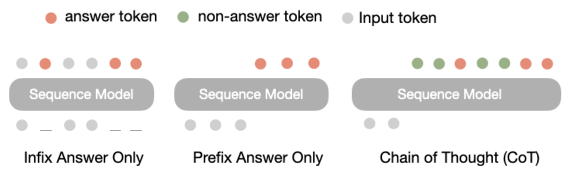

Figure 2: CoT vs Answer Only Generation Models.

counters—capabilities that standard RNNs lacked.

Theory of Model Architectures with Composability. Delétang et al. (2022) systematically investigated arithmetic capabilities across modern architectures, including RNNs, LSTMs, and Transformers. Their findings confirmed that while LSTMs exhibit computational behavior aligned with counter machines, Transformers consistently fail at even basic counting tasks. Chang and Bisk (2024) extended this analysis to newer architectures such as Mamba (Gu and Dao, 2023) and RWKV (Peng et al., 2023), revealing that these models also underperform on arithmetic tasks outside their training distributions—often performing worse than classic RNNs in generalization.

Recent studies (Weiss et al., 2018; Ackerman and Cybenko, 2020) have further validated the computational capabilities of both RNNs and LSTMs, particularly for tasks requiring symbolic or arithmetic reasoning. In contrast, Transformers—lacking inherent recurrence—are restricted to TC^0 complexity in their inductive reasoning capacity (Li et al., 2024a), placing them at the lower bound of the Chomsky hierarchy (Sanford et al., 2024; Li et al., 2024a; Delétang et al., 2022). As a result, they are fundamentally incapable of solving even basic algorithmic tasks, such as arithmetic pattern induction or sequence manipulation, without incorporating explicit inductive biases (Chang and Bisk, 2024).

3 Theoretical Limits of Answer-Only Models for Arithmetic and Symbolic Computation.

Transformer-based models (Vaswani, 2017) without Chain-of-Thought (CoT) prompting (Fig. 2 left 2) are inherently constrained by their *fixed architectural depth* and lack of *recurrence*. Let $\boldsymbol{\mathcal{X}}$ be the input token space, $\boldsymbol{\mathcal{H}}$ the hidden state space, and $\boldsymbol{\mathcal{O}}$ the output token space. For a Transformer with L layers and input sequence $\boldsymbol{x}_{1:n} \in \boldsymbol{\mathcal{X}}^n$, the hidden representation at layer ℓ and position t is given by $\boldsymbol{h}_t^{(\ell)} = \text{Layer}_\ell\left(\boldsymbol{h}_{1:n}^{(\ell-1)}\right)$, with $\boldsymbol{h}_t^{(0)} = \text{Embed}(\boldsymbol{x}_t)$. The output token $\boldsymbol{o}_t \in \boldsymbol{\mathcal{O}}$ is then computed as $\boldsymbol{o}_t = \text{Softmax}\left(W\boldsymbol{h}_t^{(L)}\right)$.

This process applies a fixed sequence of transformations to each input $\boldsymbol{x}_t$, with the number of computational steps bounded above by $L = \mathcal{O}(1)$. Since L does not scale with the input length n, the *depth complexity* of computation in such models is constant, i.e., $\text{Depth}_{\text{Transformer}} = \mathcal{O}(1)$. As established in complexity theory (Zhang et al., 2024; Li et al., 2024a; Chang and Bisk, 2024), this places answer-only Transformers in the class TC^0—constant-depth circuits with polynomial size and threshold gates—incapable of performing even simple arithmetic functions such as parity, addition, or comparison over unbounded inputs.

Formally, consider a function $f : \boldsymbol{\mathcal{X}}^n \to \boldsymbol{\mathcal{O}}^m$ defined by a task such as computing $\texttt{sum}(\boldsymbol{x}_{1:n})$. Such tasks require a computation of depth $\Omega(\log n)$ for associative operations and $\Omega(n)$ for sequentially dependent operations (e.g., counting, carry propagation, or string reversal) (Fischer et al., 1968). Since L is constant in Transformers and all transformations are composed in parallel across tokens, such models fail to meet the depth requirement: $\text{Depth}_{\text{task}}(f) > \text{Depth}_{\text{Transformer}}$ implies that the Transformer cannot compute f.

Furthermore, Transformers lack a mechanism to store and evolve intermediate computational states over time. In recurrent models, hidden states $\boldsymbol{h}_t$ are recursively defined as $\boldsymbol{h}_t = g_\theta(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$, allowing the system to simulate Turing-complete behavior (Zhang et al., 2024; Li et al., 2024b). In contrast, Transformers treat all inputs simultaneously through attention-based aggregation without iterative update: $\boldsymbol{h}_t^{(L)} = f_{\text{attn}}(\boldsymbol{x}_{1:n})$, disallowing symbolic loop constructs or dynamic memory—key components in arithmetic computation.

When constrained to generate only answer tokens $\boldsymbol{o}_{1:m} \in \boldsymbol{\mathcal{O}}^m$ without emitting intermediate reasoning steps (Fig 2), the model's total computational budget is tightly bound by m, which is typically small. Let $\boldsymbol{o}_{1:m} = f(\boldsymbol{x}_{1:n})$ be the model's prediction. Then, for a computation that requires $T(n)$ steps, with $T(n) \gg m$, the model must either (1) compress computation into fixed layers—violating the task's depth complexity—or (2) memorize input-output mappings—an approach that does not generalize beyond training.

This reliance on shallow function approximation implies that such models can only succeed by memorizing specific input-output pairs, not by executing general algorithms. Empirically, these models exhibit sharp performance degradation on arithmetic tasks outside their training distribution (Chang and Bisk, 2024).

If the maximum number of distinct computation traces a model can represent with d-dimensional hidden states and p-bit numerical precision: $|\mathcal{H}| \leq 2^{dp}$. Therefore, the number of unique state transitions is also bounded above by 2^{dp}, which is finite and insufficient for representing the $\mathcal{O}(n)$-length trajectories required for tasks such as binary addition or bracket matching.

In summary, models limited to generating only final answer tokens without CoT or external recurrence simulation (Fig 2).

4 CoT under Ideal Assumptions Enables General Arithmetic Computation

Transformer-based LLMs (Achiam et al., 2023; Touvron et al., 2023; Bai et al., 2023), though powerful, are fundamentally bounded by their *fixed architectural depth* (Li et al., 2024a; Zhang et al., 2024), limiting their ability to perform arithmetic operations that require sequential, stateful updates. Chain-of-Thought (CoT) prompting (Wei et al., 2022), however, offers a mechanism to simulate recurrence, transforming the depth-limited Transformer into a theoretically Turing-complete system under ideal assumptions.

4.1 Inductive Arithmetic Requires Depth

Arithmetic reasoning, in both human cognition and formal computation, often involves *inductive updates* across time or space (Fig 1). For instance, computing the cumulative sum of a digit sequence $\boldsymbol{x}_{1:n} = (7, 3, 2, \cdots)$ requires maintaining an accumulator that evolves as $\boldsymbol{s}_t = \boldsymbol{s}_{t-1} + \boldsymbol{x}_t$ over $t = 1$ to n. In recurrent neural networks (RNNs), this is naturally represented as $\boldsymbol{h}_t = g_\theta(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$, where the hidden state $\boldsymbol{h}_t$ stores intermediate quantities such as partial sums, carries, or flags. The computation depth required for such tasks is $\mathcal{O}(n)$, aligning with results from counter machine theory (Fischer et al., 1968).

Transformers, in contrast, lack temporal recurrence. Their hidden states $\boldsymbol{h}_t$ are updated via a fixed sequence of layers, independent of sequence length. As a result, their total reasoning depth is $\mathcal{O}(1)$ per token. Because all $\boldsymbol{x}_t$ are processed in parallel, the Transformer cannot simulate stepwise updates required for arithmetic unless all logic is memorized or encoded through exponentially wide circuits (Li et al., 2024a).

This explains why arithmetic tasks—such as computing $\texttt{sum}(\boldsymbol{x}_{1:n})$, $\texttt{reverse}(\boldsymbol{x}_{1:n})$, or $\texttt{count}_{\texttt{token}}(\boldsymbol{x}_{1:n})$—are infeasible for answer-only Transformers without inductive bias (Chang and Bisk, 2024; Delétang et al., 2022). These tasks require *depth-sensitive* computation, where each output depends on a chain of intermediate results not recoverable from input alone.

4.2 Chain-of-Thought Simulates Recurrent Computation

Chain-of-Thought (CoT) reasoning allows a Transformer to externalize its hidden state through intermediate tokens. Instead of directly mapping $\boldsymbol{x}_{1:n} \to \boldsymbol{y}$, the model generates a sequence of *thought tokens* $\boldsymbol{o}_{1:k}$:

$$\boldsymbol{x}_{1:n} \Rightarrow (\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_k) \Rightarrow \boldsymbol{y}.$$

Each $\boldsymbol{o}_t$ can encode intermediate computations (e.g., partial sums, loop counters, flags) that are later re-ingested through the embedding layer, reinitializing the next latent state: $\boldsymbol{h}_{t+1} = f_\theta(\texttt{Embed}(\boldsymbol{o}_t), \boldsymbol{x}_{t+1})$.

This iterative reasoning cycle approximates the recurrence in RNNs:$\boldsymbol{h}_{t-1} \Rightarrow \boldsymbol{o}_t \Rightarrow \boldsymbol{h}_t$, where $\boldsymbol{o}_t$ encodes sufficient information from $\boldsymbol{h}_{t-1}$ to resume and advance computation. Under ideal assumptions—namely unlimited CoT token budget and precise token-to-state fidelity—this externalization loop can simulate unbounded depth, making CoT+autoregressive models *Turing complete* (Zhang et al., 2024; Li et al., 2024b).

5 Tokenization as a Barrier to Chain-of-Thought Computation

Despite the theoretical promise of Chain-of-Thought (CoT) prompting to approximate Turing-complete computation under ideal assumptions (Zhang et al., 2024; Li et al., 2024a), empirical failures persist even in state-of-the-art models such as GPT-4. These failures are particularly evident in arithmetic and symbolic tasks that require precise reasoning over fine-grained units (e.g., digits, letters, or symbols), where large language models often yield incorrect results for inputs of even moderate length (e.g., computing the number of `rs` in `Strawberry`). This discrepancy highlights a critical limitation: CoT effectiveness is inherently bounded not just by model architecture, but also by the *expressiveness* of the underlying language, which is in turn shaped by the tokenizer.

We treat tokenization as a black-box preprocessor $\boldsymbol{x}_{1:n} \xrightarrow{\mathcal{T}} \boldsymbol{t}_{1:m}$, where $\boldsymbol{x}_{1:n}$ is a raw input string over characters and $\boldsymbol{t}_{1:m}$ is the resulting sequence of tokens from vocabulary $\boldsymbol{\mathcal{V}}$. Modern tokenizers such as byte-pair encoding (BPE) (Sennrich, 2015) are designed to optimize compression and training efficiency, not fidelity of semantic or syntactic granularity. However, for CoT reasoning to succeed in arithmetic and symbolic computation, the token space $\boldsymbol{\mathcal{V}}$ must be able to *express and preserve intermediate state information*.

5.1 Expressiveness and the Token-to-Thought Mapping

We define the expressiveness of a language $\mathcal{L} = (\boldsymbol{G}, \boldsymbol{\mathcal{V}})$—with grammar $\boldsymbol{G}$ and vocabulary $\boldsymbol{\mathcal{V}}$—as the number of unique semantically meaningful sequences $\boldsymbol{S}_{\mathcal{L}}$ it can generate:

$$\text{Expressiveness}(\mathcal{L}) := |\boldsymbol{S}_{\mathcal{L}}|.$$

For a CoT process to emulate Turing-complete computation, it must support a recurrent approximation: $\boldsymbol{h}_{t-1} \to (\boldsymbol{o}_1, \ldots, \boldsymbol{o}_k) \to \boldsymbol{h}_t$, where latent state $\boldsymbol{h}_{t-1} \in \boldsymbol{\mathcal{H}}$ is decoded into intermediate natural language tokens $\boldsymbol{o}_{1:k} \in \boldsymbol{\mathcal{V}}^k$, which are then re-embedded and fed back to reconstruct $\boldsymbol{h}_t$.

This implies the need for a high-fidelity vector-to-token mapping:

$$\phi : \boldsymbol{\mathcal{H}} \longrightarrow \boldsymbol{\mathcal{V}}^* \quad \text{and} \quad \psi : \boldsymbol{\mathcal{V}}^* \longrightarrow \boldsymbol{\mathcal{H}},$$

such that the composed transformation $\psi \circ \phi(\boldsymbol{h}_{t-1}) \approx \boldsymbol{h}_t$ retains sufficient computational state to perform stepwise updates. Tokenization introduces two major obstacles to this cycle, degrading the effective CoT expressiveness.

5.2 Damage Type I: Information Hiding via Token Granularity

The first form of damage is semantic obfuscation. Suppose the reasoning task requires operating over atomic units (e.g., characters, digits), but tokenization merges these into opaque multi-character tokens: `Strawberry` $\to$ [`Straw`, `berry`]. Now, let $\boldsymbol{t}_i \in \boldsymbol{\mathcal{V}}$ denote a token for which the model lacks fine-grained internal features (e.g., how many `r`'s are present). We define the token awareness function as:

$$\text{TokenAware}(\boldsymbol{t}_i, \texttt{prop}) := \mathbb{I}\left[\texttt{prop} \in \text{Emb}(\boldsymbol{t}_i)\right],$$

where `prop` denotes a property (e.g., digit count, lexical features), and $\text{Emb}(\boldsymbol{t}_i)$ is the token embedding. When $\text{TokenAwareness}(\boldsymbol{t}_i, \texttt{prop}) = 0$, reasoning that relies on `prop` (e.g., "count the number of 3's") will fail. Thus, even if the CoT reasoning process is intact, its input signal is corrupted at the encoding layer.

5.3 Damage Type II: Limited CoT Expressiveness via Token Bottleneck

The second, more subtle limitation occurs during the CoT process itself. The latent state $\boldsymbol{h}_{t-1}$ stores accumulated reasoning. To externalize this into thought tokens $\boldsymbol{o}_{1:k}$, we require that:

$$\forall \boldsymbol{h}_{t-1} \in \boldsymbol{\mathcal{H}}, \exists \boldsymbol{o}_{1:k} \in \boldsymbol{\mathcal{V}}^k \text{ such that } \phi(\boldsymbol{h}_{t-1}) = \boldsymbol{o}_{1:k},$$

but when $\boldsymbol{\mathcal{V}}$ is coarse (e.g., BPE with token merges) or lacks the necessary expressive forms (e.g., missing digits, variable names, or operations), this surjection fails. Let $\boldsymbol{S}_{\boldsymbol{h}}$ be the space of latent states and $\boldsymbol{S}_{\boldsymbol{o}}$ be the expressible token sequences. Then CoT fidelity is bounded as:

$$\text{Fidelity}(\text{CoT}) \leq \frac{|\phi(\boldsymbol{S}_{\boldsymbol{h}}) \cap \boldsymbol{S}_{\boldsymbol{o}}|}{|\boldsymbol{S}_{\boldsymbol{h}}|}.$$

Low expressiveness $\Rightarrow$ low overlap $\Rightarrow$ critical reasoning steps cannot be externalized.

This mismatch becomes catastrophic in arithmetic tasks where thought tokens must verbalize structured computations (e.g., carrying, intermediate sums). Without expressive enough $\boldsymbol{\mathcal{V}}$, many $\boldsymbol{h}_{t-1}$ are untranslatable, rendering the CoT step ineffective. The model is thus forced to either truncate reasoning or approximate it via memorized heuristics, both of which degrade generalization.

5.4 Formal Failure Case: CoT under BPE Tokenizer

Assume the model is asked to compute a symbolic function $f : \boldsymbol{x}_{1:n} \to \boldsymbol{y}$ (e.g., reverse digits). The optimal CoT process proceeds via:

$$\boldsymbol{h}_0 \to \boldsymbol{o}_{1:k_1} \to \boldsymbol{h}_1 \to \boldsymbol{o}_{k_1+1:k_2} \to \cdots \to \boldsymbol{y},$$

but if $\boldsymbol{o}_{1:k_i} \notin \boldsymbol{S}_{\mathcal{L}}$ due to token constraints, then $\boldsymbol{h}_{i+1}$ will be misaligned, i.e., $\boldsymbol{h}_{i+1} \not\approx g(\boldsymbol{h}_i)$. Over time, errors compound, and f becomes uncomputable.

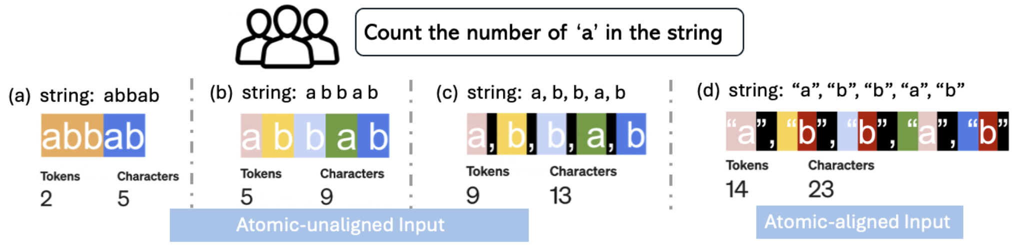

Figure 3: Four types of string formatting to manipulate tokenization in counting. Examples in the figure are tokenized using the GPT-4o tokenizer. Each string-token type is labeled as (a), (b), (c), and (d) in the diagram. Note that changing the format does not alter the fundamental nature or difficulty of the counting task.

5.5 Quantifying Tokenization Effects on Symbolic Computation

To complement our theoretical findings, we introduce a general, model-agnostic framework to evaluate how tokenization impacts symbolic and arithmetic reasoning in LLMs. While many models are closed-source, we treat LLMs as black boxes and isolate tokenization as the key variable influencing performance.

Let $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}^n$ be a character-level input string and $\mathcal{T}$ a tokenizer mapping it into a token sequence $\boldsymbol{t}_{1:m} = \mathcal{T}(\boldsymbol{x})$, where each $\boldsymbol{t}_i \in \boldsymbol{\mathcal{V}}$ comes from a fixed vocabulary. The LLM $\mathcal{M}$ then performs a symbolic task by computing an output $\boldsymbol{y} = \mathcal{M}(\boldsymbol{t}_{1:m})$. Our goal is to determine how mismatches between task granularity and token structure affect the model's ability to solve $f : \boldsymbol{\mathcal{X}}^n \rightarrow \boldsymbol{\mathcal{Y}}$.

We base our input manipulation on three typical properties of modern BPE-like tokenizers: (1) Common substrings of 2–4 characters are merged into single tokens. (2) Delimiters (e.g., spaces, commas) are usually merged with adjacent tokens. (3) Adding repeated delimiters can break these merges and force token boundaries. These patterns let us construct inputs that vary tokenization while keeping the underlying symbolic task fixed.

Input Design. For a symbolic function f (e.g., digit sum, string reversal, pattern matching), we generate two sets of inputs:

- **Atomic-aligned inputs** x^{atomic}: token boundaries align with units required for the task.
- **Merged-token inputs** x^{merged}: intentionally merged to obscure symbolic units within tokens.

If the model lacks internal awareness of sub-token structure (e.g., characters inside a token), then symbolic reasoning that depends on those units will fail—even if CoT prompting is used.

Quantifying Degradation. We define the tokenization damage as the average accuracy drop:

$$\Delta_{\text{tok}} := \left|\mathbb{E}_{\boldsymbol{x}}\left[\mathcal{A}(\boldsymbol{x}^{\text{atomic}}) - \mathcal{A}(\boldsymbol{x}^{\text{merged}})\right]\right|,$$

where $\mathcal{A}(\cdot)$ is model accuracy.

A high Δ_{tok} indicates that the model relies on token structure and cannot generalize symbolic logic across inconsistent tokenizations.

6 Experiments

6.1 Settings

We evaluate the impact of tokenization and input formatting on symbolic reasoning capabilities of large language models (LLMs). We focus on three fundamental symbolic tasks: Arithmetic counting, Sorting and Sorting.

Each task operates on inputs drawn from controlled domains (letters, digits, or words), and varies in input length and tokenization strategy. Despite differing surface forms, all tasks share a symbolic core that requires composition, memory, and manipulation of atomic units. Importantly, the *task identity* remains unchanged across formatting conditions.

To isolate the role of tokenization, we disable tool use in all models and treat LLMs as black-box functions $\mathcal{M} \circ \mathcal{T}$, where $\mathcal{T}$ is the tokenizer and $\mathcal{M}$ is the model.

For counting tasks, we test four competitive LLMs: GPT-4o-mini, Claude 3.5 Sonnet, Qwen Turbo, and OpenAI o1. For sorting and reversing, we focus on GPT-4o-mini due to its consistent performance and API accessibility. Each experiment

String-Token Type	Counting letter a						Counting letter b					
	len ∈ [10-20]		len ∈ [20-30]		len ∈ [30-40]		len ∈ [10-20]		len ∈ [20-30]		len ∈ [30-40]	
	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT
pure string BPE tokens (a)	30.10	45.70	15.10	9.10	6.40	2.00	33.20	47.70	14.00	9.40	3.80	2.70
" "-deliminated token (b)	46.20	58.40	16.10	24.90	7.50	10.90	45.90	63.70	17.60	34.00	5.60	18.60
", "-deliminated token (c)	56.00	55.40	19.40	38.60	10.20	28.10	63.60	69.30	32.80	56.10	13.90	42.30
precise-item token (d)	50.70	96.80	15.80	81.60	7.90	56.10	58.30	96.50	30.20	90.00	12.60	70.80
Δ_{tok} [max]	25.90	41.10	4.30	72.50	3.80	54.10	30.40	48.80	18.80	80.60	10.10	68.10

Table 1: Results of counting as and bs in string consisting of letter a and b, using GPT-4o-mini API. Numbers indicate the average accuracy (%) over 1000 random generated instances.

String-Token Type	Counting letter e						Counting letter z					
	len ∈ [10-20]		len ∈ [20-30]		len ∈ [30-40]		len ∈ [10-20]		len ∈ [20-30]		len ∈ [30-40]	
	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT	no-CoT	CoT
pure string BPE tokens (a)	26.60	55.20	19.80	12.20	11.40	2.10	31.10	59.10	11.70	22.10	4.60	7.30
" "-deliminated token (b)	41.00	52.90	23.90	28.20	13.00	16.00	45.30	63.90	16.60	46.20	6.80	29.50
", "-deliminated token (c)	45.50	64.20	27.40	44.20	18.00	27.60	56.20	73.60	28.20	55.60	13.90	41.90
precise-item token (d)	60.10	97.70	32.50	89.30	15.30	70.70	60.60	98.40	30.60	93.80	13.30	74.80
Δ_{tok} [max]	33.50	44.80	12.70	77.10	6.60	68.60	29.50	39.30	18.90	71.70	9.30	67.50

Table 2: Results of counting es and zs in string consisting of letter e and z, using GPT-4o-mini model. Numbers indicate the average accuracy (%) over 1000 random generated instances.

Method/Length		Letter			Letter+Digit			Digit		
		Str	List	Δ_{tok}	Str	List	Δ_{tok}	Str	List	Δ_{tok}
CoT	**5-10**	24.6	**32.0**	7.4	30.1	**35.7**	5.6	56.2	**84.4**	27.8
	10-15	3.3	8.7	5.4	5.4	**10.1**	4.7	7.9	**33.0**	25.1
	15-20	0.4	1.1	0.7	0.7	2.7	2.0	0.8	4.8	4.0
SCoT	**5-10**	28.0	**35.1**	7.1	31.1	**38.5**	7.4	64.7	**84.8**	20.1
	10-15	10.4	**12.6**	2.2	10.6	**15.6**	5.0	15.6	**34.6**	19.0
	15-20	2.4	3.4	1.0	2.6	4.7	2.1	3.2	8.6	5.4

Table 3: Performance on **sorting tasks** using GPT-4o mini with Chain-of-Thought (CoT) and Supervised Chain-of-Thought (SCoT) across different input types, length ranges and tokenization types.

Method/Length		Random			Word			High-freq Word		
		Str	List	Δ_{tok}	Str	List	Δ_{tok}	Str	List	Δ_{tok}
CoT	**5-10**	46.0	**70.0**	24.0	39.1	**56.5**	17.4	54.2	**66.6**	12.4
	10-15	8.6	**38.1**	29.5	11.7	**22.4**	10.7	13.6	**25.9**	12.3
	15-20	2.5	**20.1**	17.6	1.5	**8.0**	6.5	2.3	**9.0**	6.70
	20-25	0.3	**9.6**	9.3	0.5	**2.2**	1.7	0.6	**2.4**	1.8
	25-30	0.4	**4.7**	4.3	0.1	**0.7**	0.6	**0.5**	0.4	0.1
SCoT	**5-10**	50.2	**72.1**	21.9	51.1	**68.0**	16.9	59.1	**72.3**	23.2
	10-15	35.8	**56.9**	21.1	29.9	**52.8**	22.9	33.6	**56.5**	22.9
	15-20	24.9	**44.4**	19.5	18.7	**38.9**	20.2	21.5	**43.9**	22.4
	20-25	18.6	**31.3**	12.7	13.6	**30.1**	16.5	12.3	**32.6**	20.3
	25-30	12.6	**23.8**	11.2	7.2	**21.4**	14.2	8.5	**23.8**	15.3

Table 4: Performance comparison on **reversing tasks**, using simlar settings as Table 3.

uses 1,000 randomly generated input instances per length bucket. Input lengths are task-specific: for counting, we use lengths in $[10, 20]$, $[20, 30]$, and $[30, 40]$; for sorting, lengths are in $[5, 10]$, $[10, 15]$; and for reversing, in $[5, 30]$ (5-step increments). All models use identical prompts. We evaluate with and without CoT reasoning, and for some tasks apply supervised CoT (SCoT) (Zhang et al., 2025) to control for CoT quality. Evaluation measures exact-match accuracy.

6.2 Tokenizer Sensitivity in Symbolic Tasks

Across all tasks, we observe a consistent phenomenon: model performance varies dramatically with tokenization format, even when the underlying symbolic function f remains fixed. Following are detailed analysis across all tasks conducted:

CoT grants compositional logic

Enabling Chain-of-Thought (CoT) significantly boosts performance, particularly for tasks that require sequential or compositional logic. This improvement is most pronounced when input length increases, suggesting that CoT enables models to simulate recurrent computation by externalizing intermediate state. In tasks where CoT is not used, performance plateaus or drops sharply as sequence length grows, reflecting the fixed-depth constraint of answer-only Transformers.

Symbolic Reasoning is sensitive to input token structure

Experiments show that tokenization plays a critical role in determining symbolic generalization. For a fixed task function f, changes in tokenization alone—without altering task semantics—can yield over 70% variance in accuracy. This phenomenon is captured quantitatively by the tokenization degradation gap Δ_{tok}, which consistently reaches high values across all experiments. In Table 1, for ex-

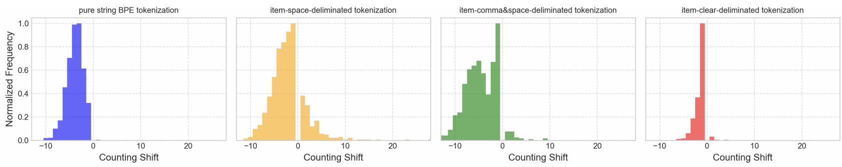

Figure 4: Distribution of shifts from the correct count.

ample, switching from raw BPE inputs (type a) to atomic-aligned inputs (type d) improves accuracy by $\Delta_{\text{tok}} = 54.1\%$ for counting `a`, and similar gains are observed in Table 2 for letters `e` and `z`.

The results in Tables 4 and 3 further reveal that symbolic reasoning ability is not only sensitive to tokenizer (controlled by using string vs. list), but also to the type of atomic unit being processed. Tasks involving digits consistently yield significantly higher performance than those involving letters or words, even when the overall structure of the task and input formatting are matched.

For instance, in sorting tasks (Table 3), CoT performance on digit sequences reaches up to 84.8% accuracy for lengths 5–10 in list format, with a corresponding Δ_{tok} of 27.8. In contrast, performance drops to 35.1% on letter sequences under identical conditions, with a much smaller Δ_{tok} of 7.1. This discrepancy persists across input lengths and holds under both CoT and SCoT prompting. A similar trend is observed in reversing tasks (Table 4): digit and high-frequency word sequences achieve the highest absolute accuracies and largest gains from structured formatting, suggesting that token content—i.e., whether the model processes compact numerical symbols or open-vocabulary lexical tokens—has a significant effect on symbolic generalization.

These results validate our theoretical claim that symbolic reasoning over atomic units cannot emerge reliably unless tokenization preserves unit-level structure. When input tokens merge multiple semantic units (e.g., letters or digits), the model cannot apply symbolic operations like comparison or increment at the proper resolution. This leads to brittle reasoning and reliance on memorization. In contrast, atomic-aligned formats ensure that symbolic computation is recoverable from token-level patterns, enabling models to generalize even across longer inputs or different domains (letters vs. digits vs. words).

Overall, the combination of CoT and precise tokenization unlocks the model's latent arithmetic and symbolic capabilities. However, without either component, performance degrades—even if the model architecture is unchanged. These findings confirm that CoT grants access to general symbolic reasoning under ideal token granularity, and that a high Δ_{tok} is a strong indicator that a model's performance hinges on token alignment rather than true generalization.

6.3 Error Shifts Reveal BPE-Induced Counting Failures

We define error shifts as the difference between the model-predicted and true counts on failed instances. As shown in Figure 4, GPT-4o mini exhibits a strong bias toward negative shifts across all tokenization types, indicating systematic undercounting. With pure BPE tokenization, shifts are exclusively negative—likely due to the model's inability to parse individual characters within merged tokens (e.g., "abaa"), often resulting in zero counts for target symbols (see Appendix Figure 10).

When delimiter-separated formats (types (b)–(d)) are used, some positive shifts appear, likely caused by overcounting or retrieval inconsistencies. Yet with fully atomic-aligned tokens (type (d)), errors narrow to a band between –1 and –3, reflecting smaller arithmetic missteps rather than structural confusion. This confirms that BPE introduces larger, systematic errors, whereas cleaner tokenization mitigates extreme deviations.

7 Conclusion

We have demonstrated that tokenization is a critical bottleneck in the symbolic reasoning ability of language models. Even with Chain-of-Thought prompting, coarse or misaligned token structures prevent models from accurately performing arithmetic and structured symbolic tasks. Our theoretical framework and empirical findings jointly show that both token format and token type (e.g., digits vs. letters) significantly affect generalization. Aligning tokenization with atomic reasoning units enables smaller models to rival or surpass larger ones, highlighting the need to treat tokenization design as a core component of model capabilities—not merely a preprocessing step.

Limitations

Our experiments were conducted on GPT-4o Mini, Claude 3.5 Sonnet and Qwen Turbo. While all models demonstrated strong patterns and consistent evidence showing that certain types of tokenization significantly improve counting performance, we did not extend our testing to other open-source LLMs such as LLaMA, Mistral. This was primarily due to budget and time constraints, as well as preliminary findings that these models exhibited weaker instruction-following abilities compared to GPT and Claude, making the evaluation process more challenging. However, we believe our research remains robust despite these limitations, as mainstream model training and design principles are largely universal, and the patterns observed are likely generalizable to other LLMs.

Additionally, our experiments did not explore extreme context lengths, such as counting instances with more than several hundred tokens. We found that such cases often led to instability due to the accumulation of long CoT steps. We aim to further investigate this aspect as LLMs improve in handling long-context retrieval and generation.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Joshua Ackerman and George Cybenko. 2020. A survey of neural networks and formal languages. *arXiv preprint arXiv:2006.01338*.

Zeyuan Allen-Zhu and Yuanzhi Li. 2023. Physics of language models: Part 3.1, knowledge storage and extraction. *arXiv preprint arXiv:2309.14316*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

George S Boolos, John P Burgess, and Richard C Jeffrey. 2002. *Computability and logic*. Cambridge university press.

Yingshan Chang and Yonatan Bisk. 2024. Language models need inductive biases to count inductively. *arXiv preprint arXiv:2405.20131*.

S Barry Cooper. 2017. *Computability theory*. Chapman and Hall/CRC.

Nicolaas Govert De Bruijn. 1964. Polya's theory of counting. In *Applied combinatorical mathematics*, pages 144–184. Eindhoven University of Technology.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. 2022. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. 2024. Towards revealing the mystery behind chain of thought: a theoretical perspective. *Advances in Neural Information Processing Systems*, 36.

Patrick C Fischer, Albert R Meyer, and Arnold L Rosenberg. 1968. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283.

Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. 2014. Deep autoregressive networks. In *International Conference on Machine Learning*, pages 1242–1250. PMLR.

Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Oscar H Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan, and Richard A Kemmerer. 2002. Counter machines and verification problems. *Theoretical Computer Science*, 289(1):165–189.

Mark Jerrum. 1995. The computational complexity of counting. In *Proceedings of the International Congress of Mathematicians: August 3–11, 1994 Zürich, Switzerland*, pages 1407–1416. Springer.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. 2024a. Chain of thought empowers transformers to solve inherently serial problems. *arXiv preprint arXiv:2402.12875*.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. 2024b. Chain of thought empowers transformers to solve inherently serial problems.

Puyuan Liu, Xiang Zhang, and Lili Mou. 2022. A character-level length-control algorithm for non-autoregressive sentence summarization. *Advances in Neural Information Processing Systems*, 35:29101–29112.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. 2023. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*.

Paul Rodriguez, Janet Wiles, and Jeffrey L Elman. 1999. A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Clayton Sanford, Daniel Hsu, and Matus Telgarsky. 2024. Transformers, parallel computation, and logarithmic depth. *arXiv preprint arXiv:2402.09268*.

Rico Sennrich. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M. Shieber. 2019. Lstm networks can perform dynamic counting.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision rnns for language recognition.

Karen Wynn. 1990. Children's understanding of counting. *Cognition*, 36(2):155–193.

Xiang Zhang, Muhammad Abdul-Mageed, and Laks V. S. Lakshmanan. 2024. Autoregressive + chain of thought = recurrent: Recurrence's role in language models' computability and a revisit of recurrent transformer.

Xiang Zhang, Juntai Cao, Jiaqi Wei, Chenyou You, and Dujian Ding. 2025. Why does your cot prompt (not) work? theoretical analysis of prompt space complexity, its interaction with answer space during cot reasoning with llms: A recurrent perspective.

Appendix

A Supervised Chain of Thought

Naive Chain of Thought (CoT), which uses a generic "think step by step" prompt for all tasks, poses significant challenges for models in determining the correct steps, especially for complex, multi-step reasoning tasks. To mitigate this confounding factor, we follow previous work and employ Supervised CoT (Zhang et al., 2025), as the derivation of steps is not the focus of our research and should not affect performance due to incorrect CoT steps. Below, we define Supervised CoT and explain its application in counting tasks.

A.1 Definition

The search space for solving a task can be viewed as a combination of the prompt space and the answer space. When instructed to perform tasks step by step, language models must devise a step template which is used to determine the actions at each step. This template is crucial for solving tasks, as it specifies what information is processed and how it is computed at each CoT step. However, for a given task, there are numerous ways to perform a "step-by-step" approach, each computing different elements per step. Finding the optimal set of steps is challenging yet essential, as it directly influences the ability to find solutions in the answer space (Zhang et al., 2025).

Supervised CoT provides human supervision in determining the step template. Rather than asking the model to develop its own plan for each step, humans identify the "recurrent" procedure in the computation and explicitly instruct the model to follow a specific step template. This approach allows the CoT to bypass the need to search for optimal steps, focusing instead on finding solutions within the answer space under optimal step guidance.

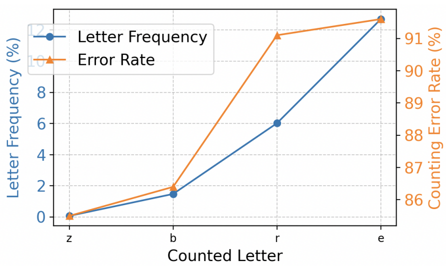

Figure 5: Counting accuracy (Orange) with respect to target letter frequency (Blue) in Human Natural Language.

A.2 Supervised CoT and Counting

In inductive counting, which relies on CoT to compute the counter value recurrently(Figure 1), it is crucial that each step of CoT accurately extracts and outputs the counter value in text. This output is necessary for the value to be recurrently processed through "string-vector" conversion. Therefore, rather than simply prompting the model with "determine the number of a in the given string" using the generic instruction "think step by step," we specifically instruct the model to print out a counter value at each step. We explicitly define the step template to ensure the model follows the optimal CoT steps, preventing deviations or the use of suboptimal steps.

Experiments. We demonstrate the significant performance gap between Supervised and Unsupervised CoT. Specifically, we observe that supervision not only helps the model accurately extract the counter but also ensures it follows the correct steps (e.g., an incorrect step would be outputting whether the current letter is the target, rather than extracting the counter value). Even when Unsupervised CoT identifies the correct steps (i.e., extracting the counter into text), we still notice more frequent errors during the extraction process compared to Supervised CoT, which imposes strict constraints on what to extract at each step. The comparison between Supervised and Unsupervised CoT is presented in Table 5, showing a clear dominance of Supervised CoT, with accuracy gains observed in nearly all cases.

B Comprehensive Experiments on the Relationship Between Letter Frequency and Symbolic Reasoning Performance

Our results in counting experiments show consistently higher counting accuracy for the letter `b` compared to `a` across all proper counting settings (CoT enabled, non-BPE tokenization), as shown in Table 1 and Figure 8 left. We hypothesized this difference stems from varying letter frequencies in natural language affecting token-embedding sensitivity.

To further investigate this hypothesis, we compared counting performance between the most frequent letter `e` (12.7%) and least frequent letter `z` (0.07%) in English. Results in Table 2 show `z` significantly outperforming `e`, mirroring the pattern seen with `b` (1.5%) versus `a` (8.2%). The accuracy advantage for lower-frequency letters ranges from 3-14% (Figure 8).

Our results reveal that lower-frequency tokens carry less embedded information from training, making them easier to track through the attention mechanism. In contrast, common letters like `a` and `e` may encode more complex linguistic information, potentially interfering with counting tasks.

To verify these results beyond the letter pairs `a`, `b` and `e`, `z`, we selected another set of letters with significantly different frequencies in *human languages*, according to Wikipedia: `z` (0.07%), `b` (1.48%), `r` (6.02%), and `e` (12.70%). We generated counting instances of lengths between 80 and 100—ensuring that each letter appears more than 20 times on average—by uniformly sampling one of the four letters to form each string (e.g., `zrrbeez`). We then performed counting for each letter in the generated strings. As shown in Table 7, a consistent trend was observed across tokenization types (b)–(d) (excluding (a), as pure BPE was previously shown not to yield meaningful counting results). Specifically, rare tokens consistently outperformed more frequent tokens in natural language, with performance improvements ranging from 6% to 12%. Figure 5 visually compares performance and letter frequency, showing an overlap between frequency and error rate. We suspect that rare letters carry less information in their embeddings, reducing distraction during the attention calculation in the counting process.

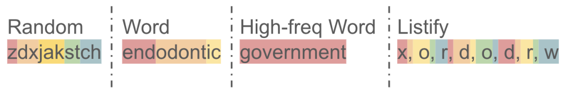

Figure 6: Tokenization patterns of the GPT-4o tokenizer when processing four different input types for reversing task: (a) random character strings, (b) regular dictionary words, (c) high-frequency English words, and (d) listified random strings with explicit delimiters.

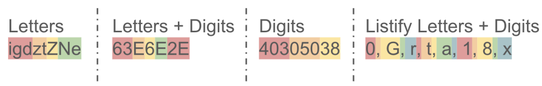

Figure 7: Tokenization patterns of the GPT-4o tokenizer across diverse input compositions for sorting task: (a) random letter strings composed solely of alphabetic characters, (b) mixed random strings containing both letters and digits, (c) random digit strings composed exclusively of numerical characters, and (d) listified mixed strings with explicit delimiters separating letter and digit combinations.

C Tokenization in Different LLMs

Figure 10 and Figure 11 illustrate the tokenization of input binary strings with difference lengths across various LLMs. We investigate both language models and multi-modal models, observing nearly identical tokenization behaviors across most tested models (except GPT-4o series). Therefore, in Figure 6 and Figure 7, we use the GPT-4o series models to further demonstrate tokenization

String-token Type	Counting a		Counting b	
	Unsupervised-CoT	Supervised CoT	Unsupervised-CoT	Supervised CoT
(b)	8.40	**10.90**	**20.70**	18.60
(c)	24.00	**28.10**	29.30	**42.30**
(d)	34.90	**56.10**	42.70	**70.80**

Table 5: Counting experiments in the length range of 30-40 comparing Supervised CoT and Unsupervised CoT. The bolded font indicates the better performance in the pairwise comparison between Supervised and Unsupervised CoT.

string-token type	len ∈ [10-20]		len ∈ [20-30]		len ∈ [30-40]	
	count a	count b	count a	count b	count a	count b
(a)	86.30	86.20	62.40	65.20	50.60	54.40
(b)	90.60	94.00	80.40	87.50	76.10	79.60
(c)	94.90	97.70	92.80	97.90	91.40	94.20
(d)	93.00	94.20	87.80	91.00	87.30	89.80

Table 6: Counting results on strings with letter a and b, using Claude 3.5 Sonnet API. All results are using supervised CoT (Zhang et al., 2025), with same prompt for GPT-4o mini. Numbers indicate the average accuracy (%) over 1000 random generated instances.

String-Token Type	len ∈ [80, 100]			
	z	b	r	e
(b)	14.50	13.60	8.90	8.40
(c)	36.00	36.60	28.30	24.30
(d)	61.60	60.20	54.10	51.90
Letter Frequency				
percentage	0.07	1.48	6.02	12.70

Table 7: Counting performance of letters that have very different letter frequency in human language.

patterns in more complex cases. These include random letter strings, random number sequences, dictionary words, high-frequency words, and mixed digit-letter strings. This analysis provides insight into how tokenization varies across different input types and structural formats.

Since pure strings may be tokenized differently due to the varying byte pair encoding (BPE) schemes used by each tokenizer. When a token-level delimiter is introduced, we consistently observe that the delimiter is combined with the adjacent letter, aligning with our previous assumptions. Additionally, some models handle the initial token differently, resulting in the first letter being treated as a standalone token (e.g., in the Grok model) or being combined with a quotation delimiter (e.g., in GPT-4o mini). In summary, our string design effectively allows us to manipulate modern LLMs to tokenize identical counting instances into different, desired tokens.

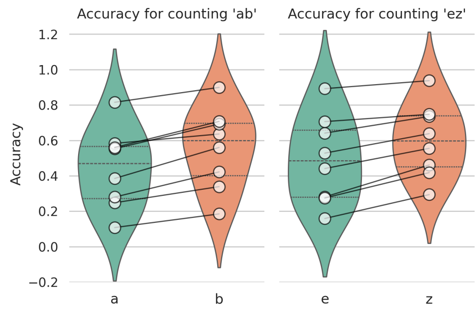

Figure 8: Pairwise comparison of counting accuracy for different letters in strings. The left plot shows the distribution of accuracy for a and b in ab strings, with each dot representing the average accuracy for a in a given CoT case (e.g., spaced-string in the [10,20] range), connected to the corresponding accuracy for b in the same setting. The right plot illustrates a similar case for e and z in ez strings. Note: The y-axis limit exceeds [0,1] as the distribution is calculated based on variance and mean, with larger variance pushing the upper bound of the confidence interval beyond the maximum value.

D Prompt Template

The set of prompts we use for counting experiments (base, unsupervised CoT, supervised CoT) are shown in Figure 12. The set of prompts we use for reversing and sorting experiments (unsupervised CoT, supervised CoT) are shown in Figure 13 and Figure 14 respectively.

E Case Studies: Counting

In this section, we use counting task to showcase our experiment results. The tables present cases for each type of token counted using CoT. As shown, Supervised CoT consistently adheres to a strict step-by-step template, accurately extracting the counter at each step. In contrast, Unsupervised CoT often skips crucial steps or deviates from the optimal method of extracting counters. Additionally, Supervised CoT with type (d) tokenization tends to produce much longer reasoning contexts, yet still achieves the best performance due to the combination of optimal tokenization and supervision.

We also repeated the experiments with Claude

3.5 Sonnet and Qwen Turbo. With Qwen model, we observed similar trends as GPT-4o mini (Table 8). With Claude model, there is a slight exception that type (c) yielded the best results among types (a)-(d), as shown in Table 6. Upon investigation, we suspect this is because type (d) results in longer CoT steps due to the higher number of irrelevant tokens generation, leading to long-context reasoning failures in many cases for this model. We also provide case studies using GPT-4o mini for counting tasks, including examples where CoT led to both correct and incorrect answers. Additionally, we reveal the inferior performance with OpenAI o1 full model when tokenization is not properly done, detailed in Appendix section E.1.

To this end, we are confident that our experimental results can be generalized to other LLMs, given that the training methods and tokenization strategies (as demonstrated in Appendix Section C) are nearly identical, leading to counting being performed in a similar manner across such models.

E.1 OpenAI o1

We evaluate o1 on samples ranging from 30 to 40 letters in length, using pure string (type (a)) to showcase the importance of using proper tokenization. Additionally, since o1 applies inference-time scaling techniques (such as MCTS search and per-step verifying), it implicitly engages in advanced chain-of-thought reasoning. Thus, we rely solely on a straightforward prompt rather than explicitly specifying reasoning steps using Supervised Chain-of-Thought as with other naive models. The final accuray on tested samples for o1 is 50%, which is much lower than using GPT-4o mini with most optimal tokenization techniques (Table 1, 70% in such length range). An example for correct counting is shown in Table 9, and an example for incorrect counting is shown in Table 10. **In conclusion, advanced LLM searching algorithms and inference time scaling techniques do not make up for defect in tokenizer.**

E.2 Qwen Turbo

We evaluate Qwen Turbo using supervised Chain-of-Thought (CoT) prompts. Tables 17 and 18 demonstrate an incorrect counting example using tokenization type (a), Table 19 shows a correct counting example using tokenization type (d).

Notably, Qwen Turbo generates more tokens per CoT step compared to Claude 3.5 Sonnet, which appears to lead to its slightly lower performance.

string-token type	len ∈ [10-20] count a	len ∈ [10-20] count b	len ∈ [20-30] count a	len ∈ [20-30] count b	len ∈ [30-40] count a	len ∈ [30-40] count b
(a)	56.40	62.50	26.20	32.20	16.20	15.90
(b)	75.10	80.00	50.60	54.60	31.10	28.30
(c)	93.40	96.00	81.60	83.50	59.20	57.60
(d)	95.90	96.70	83.20	86.50	68.10	63.50

Table 8: Counting results under the same settings as in Table 6 but using Qwen Turbo API. Numbers indicate the average accuracy (%) over 1000 random generated instances.

This suggests that concise reasoning steps is beneficial for counting accuracy.

E.3 GPT-4o mini

We present three progressive examples that demonstrate the effectiveness of combining Chain-of-Thought (CoT) reasoning with appropriate tokenization strategies:

1. Table 20 shows that using a base prompt with standard BPE tokenization (type (a)) results in a counting error of 3 from the correct value.
2. Table 21 demonstrates that incorporating supervised CoT improves accuracy, reducing the counting error to just 1.
3. Table 22 illustrates that combining supervised CoT with type (d) character-wise tokenization achieves perfect accuracy, matching the gold label exactly.

These examples clearly demonstrate how the synergy between CoT reasoning and appropriate tokenization methods can progressively enhance counting accuracy.

E.4 Claude 3.5 Sonnet

We evaluate Claude 3.5 Sonnet using supervised CoT prompts. We provide example cases demonstrating different tokenization approaches and their outcomes:

- Tables 11 and 12 showcase incorrect counting results using tokenization type (a)
- Tables 13 and 14 demonstrate correct counting using tokenization type (d)
- Tables 15 and 16 illustrate correct counting using tokenization type (c)

To sum up, our analyses show that tokenization type (c) yields superior results compared to type (d). Notably, in Tables 13 and 14, we observe that type

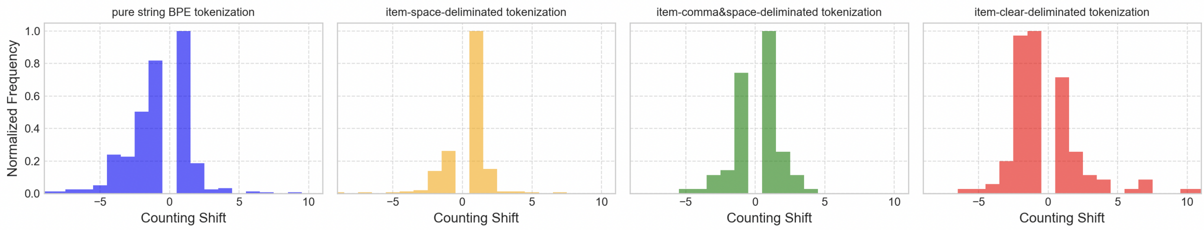

Figure 9: Same error-shifting distribution (as in Figure 4) but for Claude model. Claude 3.5 tend to count *more* than counting *less*, compared to GPT-4o.

(d) tokenization generates excessive and irrelevant content (specifically, index information) which may interfere with the accuracy of the counting process.

F Replication Experiments Note

We have open-sourced the experimental results for every instance of each experiment, in the provided GitHub link, to facilitate future research and analysis by other researchers. All reported experiment numbers are stable, using the same experimental settings and prompts. Specifically, we observe an average variance in accuracy of less than 1% across runs of the same experiments, indicating that they are fully replicable with the same model version used. Note that updates to the API version may cause potential variations in results, which are beyond our control.

Figure 10: Difference in tokenization on *long binary* strings without punctuations across different LLMs.

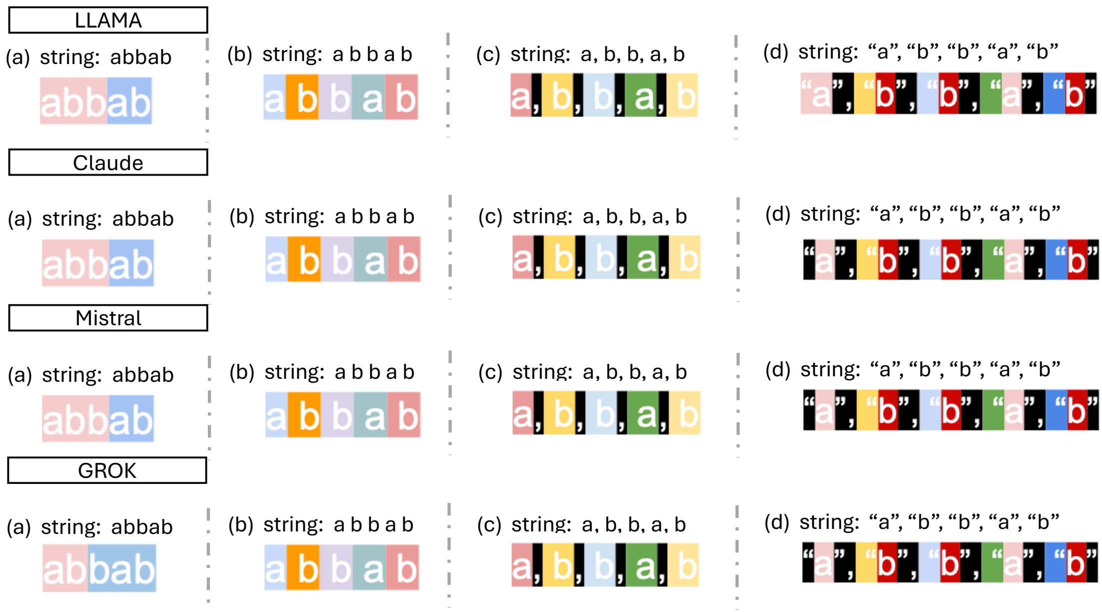

Figure 11: Difference in tokenization on *binary* strings when counting instances are presented in different formats with punctuations to facilitate tokenization, across different LLMs.

Count the number of appearances of ‘{substring}’s in the string below. Directly output ‘Result: ’ followed by the counted number. Do not use bold font in the response.

String: {sample}

Response:

(a) Base prompt template

Determine the number of appearances of ‘{substring}’s in the string below. Think step by step. Directly output ‘Result: ’ followed by the counted number. Do not use bold font in the response.

String: {sample}

Response:

(b) Chain-of-Thought (unsupervised) prompt template

Task: Count the number of occurrences of the substring ‘{substring}’ in the given string step by step. Do not use bold font in the response.

Instructions:
• Iterate through the string and count each occurrence of the substring.
• Use a counter to keep track of how many times the substring appears.
• Output your step-by-step counting process.
• Conclude with: ‘Result: ’ followed by the number.

String: {sample}

Response:

(c) Chain-of-Thought (supervised) prompt template

Figure 12: Different prompt templates used in counting experiments. The templates include a base prompt, an unsupervised chain-of-thought prompt, and a supervised chain-of-thought prompt.

Reverse the string provided below. Think step by step. Output the final result in a dictionary with the key ‘Result’. For instance, if the input string is ‘iamhappy’, the output would be: {‘Result’: ‘yppahmai’}. The input string may contain punctuations, do not add them into reversed string.

Input string: {{string}}

(a) Chain-of-Thought (unsupervised) prompt template

Reverse the string following the instructions below:

1. Initialize an empty string called ‘reversed’
2. For each character in the input string:
   - Remove the first (leftmost) character
   - Add this character to the beginning of ‘reversed’
   - Display the remaining input string
3. Once all characters are processed, ‘reversed’ will contain the reversed string. Return the reversed string in a dictionary format with the key ‘Result’. For example, the result dictionary for input string "hello" will be {‘Result’: ‘olleh’}.

Input String: {{string}}

(b) Chain-of-Thought (supervised) prompt template

Figure 13: Different prompt templates used in reversing experiments. The templates include an unsupervised chain-of-thought prompt, and a supervised chain-of-thought prompt.

Sort the string below in ascending order using insertion sort. Think step by step. Note that lower case characters are greater than upper case characters. Convert the sorted characters into a single string, and store the result in a dictionary with the key ‘Result.’ For example, given the string ‘2CsU4bSc’, the sorted result should be ‘24CSUbcs’ and should be stored as {‘Result’: ‘24CSUbcs’}.

String: {{string}}

(a) Chain-of-Thought (unsupervised) prompt template

Sort the string below in ascending order using insertion sort following the steps below. Note that lower case characters are greater than upper case characters.

1. Start by creating an empty list for sorted characters.
2. For each step:
   - Specify which character you’re currently looking and insert the character to the correct place in sorted list.
   - Display both the original string (showing characters remaining) and sorted list after each step.
3. Conbine the sorted list to a single string. Return the sorted string in a dictionary format with the key ‘Result’. For example, the result dictionary for input list “[‘2’, ‘C’, ‘s’, ‘U’, ‘4’, ‘b’, ‘S’, ‘c’]” will be {‘Result’: ‘24CSUbcs’}.

String: {{string}}

(b) Chain-of-Thought (supervised) prompt template

Figure 14: Different prompt templates used in sorting experiments. The templates include an unsupervised chain-of-thought prompt, and a supervised chain-of-thought prompt.

Model & Experiment:
OpenAI o1, token type (a), length 30-40

Prompt:
Count the number of appearances of 'a' in the string below. Do not use tool.

String: abaaaaaaabaabaaabbaaaaabaabbbbbbbabaaaaa

Response:

Model Response:
Let's count the occurrences of 'a' in the given string:

String:

```
abaaaaaaabaabaaabbaaaaabaabbbbbbbabaaaaa
```

We'll go through each character and count how many times 'a' appears:

```
a (1)
b
a (2)
a (3)
a (4)
a (5)
a (6)
a (7)
a (8)
b
a (9)
a (10)
b
a (11)
a (12)
a (13)
b
b
a (14)
a (15)
a (16)
a (17)
a (18)
b
a (19)
a (20)
b
b
b
b
b
b
b
a (21)
b
a (22)
a (23)
a (24)
a (25)
a (26)
a (27)
```

Number of 'a's counted: **27**

Correct Answer:
26

Table 9: Case Study 1 - Incorrect Prediction with o1 (base prompt)

Model & Experiment:

OpenAI o1, token type (a), length 30-40

Prompt:

Count the number of appearances of ‘b’ in the string below. Do not use tool.

String: babbbbbbbabbababbbaabbbbbbabaabbbbabb

Response:

Model Response:

Let’s count carefully:

The given string is:
`babbbbbbbabbababbbaabbbbbbabaabbbbabb`

Let’s go through it character by character and count every ‘b’:

```
b (1)
a
b (2)
b (3)
b (4)
b (5)
b (6)
b (7)
b (8)
a
b (9)
b (10)
a
b (11)
a
b (12)
b (13)
b (14)
a
a
b (15)
b (16)
b (17)
b (18)
b (19)
b (20)
a
b (21)
a
a
b (22)
b (23)
b (24)
b (25)
a
b (26)
b (27)
```

Counting all the marked ‘b’s, we get a total of **27**.

Correct Answer:

27

Table 10: Case Study 2 - Correct Prediction with o1 (base prompt)

Model & Experiment:

Claude 3.5 Sonnet, token type (a), length 30-40

Prompt:

Task: Count the number of occurrences of the substring ‘a’ in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: babbbababbbaabbbbaabaabaaabbaaababbbabb

Response:

Model Response:

Step 1: Initialize a counter to 0.
Counter = 0

Step 2: Start iterating through the string:
b - not ‘a’, counter remains 0
a - found ‘a’, increment counter
Counter = 1

Step 3: Continue iterating:
b - not ‘a’, counter remains 1
b - not ‘a’, counter remains 1
b - not ‘a’, counter remains 1
b - not ‘a’, counter remains 1
a - found ‘a’, increment counter
Counter = 2

Step 4: Continue iterating:
b - not ‘a’, counter remains 2
a - found ‘a’, increment counter
Counter = 3

Step 5: Continue iterating:
b - not ‘a’, counter remains 3
b - not ‘a’, counter remains 3
b - not ‘a’, counter remains 3
a - found ‘a’, increment counter
Counter = 4

Step 6: Continue iterating:
a - found ‘a’, increment counter
Counter = 5

Step 7: Continue iterating:
b - not ‘a’, counter remains 5
b - not ‘a’, counter remains 5
b - not ‘a’, counter remains 5
b - not ‘a’, counter remains 5
a - found ‘a’, increment counter
Counter = 6

Step 8: Continue iterating:
a - found ‘a’, increment counter
Counter = 7

Table 11: Case Study 3 - Incorrect Prediction with Claude 3.5 Sonnet (CoT prompt)

Step 9: Continue iterating:
b - not 'a', counter remains 7
a - found 'a', increment counter
Counter = 8

Step 10: Continue iterating:
a - found 'a', increment counter
Counter = 9

Step 11: Continue iterating:
b - not 'a', counter remains 9
a - found 'a', increment counter
Counter = 10

Step 12: Continue iterating:
a - found 'a', increment counter
Counter = 11

Step 13: Continue iterating:
b - not 'a', counter remains 11
b - not 'a', counter remains 11
a - found 'a', increment counter
Counter = 12

Step 14: Continue iterating:
a - found 'a', increment counter
Counter = 13

Step 15: Finish iterating through the remaining characters:
b - not 'a', counter remains 13
a - found 'a', increment counter
Counter = 14

Step 16: Iteration complete.

Result: 14

Correct Answer:

17

Table 12: Case Study 3 Continued

Model & Experiment:
Claude 3.5 Sonnet, token type (d), length 30-40

Prompt:
Task: Count the number of occurrences of the substring ’a’ in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: [‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘b’, ‘b’]

Response:

Model Response:
Step 1: Initialize counter to 0
Counter: 0

Step 2: Start iterating through the string
[‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘b’, ‘b’]

Step 3: Found ‘a’ at index 1
Counter: 1

Step 4: Found ‘a’ at index 5
Counter: 2

Step 5: Found ‘a’ at index 7
Counter: 3

Step 6: Found ‘a’ at index 11
Counter: 4

Step 7: Found ‘a’ at index 12
Counter: 5

Step 8: Found ‘a’ at index 17
Counter: 6

Step 9: Found ‘a’ at index 18
Counter: 7

Step 10: Found ‘a’ at index 20
Counter: 8

Step 11: Found ‘a’ at index 21
Counter: 9

Table 13: Case Study 4 - Correct Prediction with Claude 3.5 Sonnet 1 (CoT prompt)

Step 12: Found ‘a’ at index 23
Counter: 10

Step 13: Found ‘a’ at index 24
Counter: 11

Step 14: Found ‘a’ at index 25
Counter: 12

Step 15: Found ‘a’ at index 28
Counter: 13

Step 16: Found ‘a’ at index 29
Counter: 14

Step 17: Found ‘a’ at index 30
Counter: 15

Step 18: Found ‘a’ at index 32
Counter: 16

Step 19: Found ‘a’ at index 36
Counter: 17

Step 20: Finished iterating through the string

Result: 17

Correct Answer:

17

Table 14: Case Study 4 Continued

Model & Experiment:

Claude 3.5 Sonnet, token type (c), length 30-40

Prompt:

Task: Count the number of occurrences of the substring ’a’ in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: b, a, b, b, b, a, b, a, b, b, b, a, a, b, b, b, b, a, a, b, a, a, b, a, a, a, b, b, a, a, a, b, a, b, b, b, a, b, b

Response:

Model Response:

Step 1: Initialize counter to 0
Counter: 0

Step 2: Start iterating through the string
b, - Counter: 0
a, - Found ‘a’. Counter: 1
b, - Counter: 1
b, - Counter: 1
b, - Counter: 1
a, - Found ‘a’. Counter: 2
b, - Counter: 2
a, - Found ‘a’. Counter: 3
b, - Counter: 3
b, - Counter: 3
b, - Counter: 3
a, - Found ‘a’. Counter: 4
a, - Found ‘a’. Counter: 5
b, - Counter: 5
b, - Counter: 5
b, - Counter: 5
b, - Counter: 5
a, - Found ’a’. Counter: 6
a, - Found ’a’. Counter: 7
b, - Counter: 7
a, - Found ’a’.

Table 15: Case Study 5 - Correct Prediction with Claude 3.5 Sonnet 2 (CoT prompt)

Counter: 8
a, - Found ’a’. Counter: 9
b, - Counter: 9
a, - Found ’a’. Counter: 10
a, - Found ’a’. Counter: 11
a, - Found ’a’. Counter: 12
b, - Counter: 12
b, - Counter: 12
a, - Found ‘a’. Counter: 13
a, - Found ’a’. Counter: 14
a, - Found ‘a’. Counter: 15
b, - Counter: 15
a, - Found ‘a’. Counter: 16
b, - Counter: 16
b, - Counter: 16
b, - Counter: 16
a, - Found ‘a’. Counter: 17
b, - Counter: 17
b - Counter: 17

Step 3: Finished iterating through the string

Result: 17

Correct Answer:

17

Table 16: Case Study 5 Continued

Model & Experiment:

Qwen Turbo, token type (a), length 30-40

Prompt:

Task: Count the number of occurrences of the substring 'a' in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: 'Result: ' followed by the number.

String: baaabaaabaaababbbababbaababbabbaabb

Response:

Model Response:

To count the number of occurrences of the substring 'a' in the given string step by step, we will iterate through the string and increment a counter each time we encounter an 'a'.

String: baaabaaabaaababbbababbaababbabbaabb

Step 1: Start with the first character 'b', no 'a' found.
Step 2: Second character is 'a', increment the counter (counter = 1).
Step 3: Third character is 'a', increment the counter (counter = 2).
Step 4: Fourth character is 'a', increment the counter (counter = 3).
Step 5: Fifth character is 'b', no 'a' found.
Step 6: Sixth character is 'a', increment the counter (counter = 4).
Step 7: Seventh character is 'a', increment the counter (counter = 5).
Step 8: Eighth character is 'b', no 'a' found.
Step 9: Ninth character is 'a', increment the counter (counter = 6).
Step 10: Tenth character is 'a', increment the counter (counter = 7).
Step 11: Eleventh character is 'b', no 'a' found.
Step 12: Twelfth character is 'a', increment the counter (counter = 8).
Step 13: Thirteenth character is 'b', no 'a' found.
Step 14: Fourteenth character is 'b', no 'a' found.
Step 15: Fifteenth character is 'b', no 'a' found.
Step 16: Sixteenth character is 'a', increment the counter (counter = 9).
Step 17: Seventeenth character is 'b', no 'a' found.
Step 18: Eighteenth character is 'a', increment the counter (counter = 10).

Table 17: Case Study 6 - Incorrect Prediction with Qwen Turbo (CoT prompt)

Step 19: Nineteenth character is ‘b’, no ’a’ found.
Step 20: Twentieth character is ‘b’, no ‘a’ found.
Step 21: Twenty-first character is ‘a’, increment the counter (counter = 11).
Step 22: Twenty-second character is ‘a’, increment the counter (counter = 12).
Step 23: Twenty-third character is ‘b’, no ’a’ found.
Step 24: Twenty-fourth character is ‘b’, no ‘a’ found.
Step 25: Twenty-fifth character is ‘a’, increment the counter (counter = 13).
Step 26: Twenty-sixth character is ‘b’, no ‘a’ found.
Step 27: Twenty-seventh character is ‘a’, increment the counter (counter = 14).
Step 28: Twenty-eighth character is ‘b’, no ‘a’ found.
Step 29: Twenty-ninth character is ‘b’, no ‘a’ found.
Step 30: Thirtieth character is ‘a’, increment the counter (counter = 15).
Step 31: Thirty-first character is ‘b’, no ‘a’ found.
Step 32: Thirty-second character is ‘b’, no ‘a’ found.
Step 33: Thirty-third character is ‘a’, increment the counter (counter = 16).
Step 34: Thirty-fourth character is ‘b’, no ‘a’ found.
Step 35: Thirty-fifth character is ‘b’, no ‘a’ found.
Step 36: Thirty-sixth character is ‘a’, increment the counter (counter = 17).
Step 37: Thirty-seventh character is ‘b’, no ‘a’ found.
Step 38: Thirty-eighth character is ‘b’, no ‘a’ found.
Step 39: Thirty-ninth character is ‘a’, increment the counter (counter = 18).
Step 40: Fortieth character is ‘b’, no ‘a’ found.
Step 41: Forty-first character is ‘b’, no ‘a’ found.
Step 42: Forty-second character is ‘a’, increment the counter (counter = 19).

Result: 19

Correct Answer:

18

Table 18: Case Study 6 Continued

Model & Experiment:

Qwen Turbo, token type (d), length 30-40

Prompt:

Task: Count the number of occurrences of the substring ‘a’ in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: [‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘a’, ‘b’, ‘b’, ‘a’, ‘a’, ‘b’, ‘b’]

Response:

Model Response:

To count the number of occurrences of the substring ‘a’ in the provided string, we’ll go through it step by step:

- First element: ‘b’ - No match. Counter = 0
- Second element: ‘a’ - Match found. Counter = 1
- Third element: ‘a’ - Match found. Counter = 2
- Fourth element: ‘a’ - Match found. Counter = 3
- Fifth element: ‘b’ - No match. Counter = 3
- Sixth element: ’a’ - Match found. Counter = 4
- Seventh element: ‘a’ - Match found. Counter = 5
- Eighth element: ‘a’ - Match found. Counter = 6
- Ninth element: ‘b’ - No match. Counter = 6
- Tenth element: ‘a’ - Match found. Counter = 7
- Eleventh element: ‘a’ - Match found. Counter = 8
- Twelfth element: ‘a’ - Match found. Counter = 9
- Thirteenth element: ‘b’ - No match. Counter = 9
- Fourteenth element: ‘a’ - Match found. Counter = 10
- Fifteenth element: ‘b’ - No match. Counter = 10
- Sixteenth element: ‘b’ - No match. Counter = 10
- Seventeenth element: ‘b’ - No match. Counter = 10
- Eighteenth element: ‘a’ - Match found. Counter = 11
- Nineteenth element: ‘b’ - No match. Counter = 11
- Twentieth element: ‘a’ - Match found. Counter = 12
- Twenty-first element: ‘b’ - No match. Counter = 12
- Twenty-second element: ‘b’ - No match. Counter = 12
- Twenty-third element: ‘a’ - Match found. Counter = 13
- Twenty-fourth element: ‘a’ - Match found. Counter = 14
- Twenty-fifth element: ‘b’ - No match. Counter = 14
- Twenty-sixth element: ‘a’ - Match found. Counter = 15
- Twenty-seventh element: ‘b’ - No match. Counter = 15
- Twenty-eighth element: ‘b’ - No match. Counter = 15
- Twenty-ninth element: ‘a’ - Match found. Counter = 16
- Thirtieth element: ‘b’ - No match. Counter = 16
- Thirty-first element: ‘b’ - No match. Counter = 16
- Thirty-second element: ‘a’ - Match found. Counter = 17
- Thirty-third element: ‘a’ - Match found. Counter = 18
- Thirty-fourth element: ‘b’ - No match. Counter = 18

Result: 18

Correct Answer:

18

Table 19: Case Study 7 - Correct Prediction with Qwen Turbo (CoT prompt)

Model & Experiment:
GPT-4o mini, token type (a), length 20-30
Prompt:
Count the number of appearances of 'a's in the string below. Directly output 'Result: ' followed by the counted number. Do not use bold font in the response. String: abaaababbbbbbaaaabaaaa Response:
Model Response:
Result: 10
Correct Answer:
13

Table 20: Case Study 8 - Incorrect Prediction with GPT-4o mini (base prompt)

Model & Experiment:

GPT-4o mini, token type (a), length 20-30

Prompt:

Task: Count the number of occurrences of the substring ‘a’ in the given string step by step. Do not use bold font in the response.
Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: abaaababbbbbbaaaabaaaa

Model Response:

1. Start with the string: abaaababbbbbbaaaabaaaa.
2. Initialize a counter at 0.
3. Iterate through each character in the string.

- Index 0: ’a’ (counter = 1)
- Index 1: ’b’ (counter remains 1)
- Index 2: ’a’ (counter = 2)
- Index 3: ’a’ (counter = 3)
- Index 4: ’a’ (counter = 4)
- Index 5: ’b’ (counter remains 4)
- Index 6: ’a’ (counter = 5)
- Index 7: ’b’ (counter remains 5)
- Index 8: ’b’ (counter remains 5)
- Index 9: ’b’ (counter remains 5)
- Index 10: ’b’ (counter remains 5)
- Index 11: ’a’ (counter = 6)
- Index 12: ’a’ (counter = 7)
- Index 13: ’a’ (counter = 8)
- Index 14: ’b’ (counter remains 8)
- Index 15: ’a’ (counter = 9)
- Index 16: ’a’ (counter = 10)
- Index 17: ’a’ (counter = 11)
- Index 18: ’a’ (counter = 12)

4. End of string reached.

Result: 12

Correct Answer:

13

Table 21: Case Study 9 - Incorrect Prediction with GPT 4o mini (CoT prompt)

Model & Experiment:

GPT-4o mini, token type (a), length 20-30

Prompt:

Task: Count the number of occurrences of the substring ‘a’ in the given string step by step. Do not use bold font in the response.

Instructions:

- Iterate through the string and count each occurrence of the substring.
- Use a counter to keep track of how many times the substring appears.
- Output your step-by-step counting process.
- Conclude with: ‘Result: ’ followed by the number.

String: [‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘a’]

Model Response:

Let’s count the occurrences of the substring ‘a’ step by step in the given string:

1. Initialize a counter at 0.
2. Start iterating through the list: [‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘b’, ‘b’, ‘b’, ‘b’, ‘b’, ‘b’, ‘a’, ‘a’, ‘a’, ‘a’, ‘b’, ‘a’, ‘a’, ‘a’, ‘a’].

- Index 0: The element is ‘a’. Increment counter to 1.
- Index 1: The element is ‘b’. Counter remains 1.
- Index 2: The element is ‘a’. Increment counter to 2.
- Index 3: The element is ‘a’. Increment counter to 3.
- Index 4: The element is ‘a’. Increment counter to 4.
- Index 5: The element is ’b’. Counter remains 4.
- Index 6: The element is ‘a’. Increment counter to 5.
- Index 7: The element is ‘b’. Counter remains 5.
- Index 8: The element is ‘b’. Counter remains 5.
- Index 9: The element is ‘b’. Counter remains 5.
- Index 10: The element is ‘b’. Counter remains 5.
- Index 11: The element is ‘b’. Counter remains 5.
- Index 12: The element is ‘b’. Counter remains 5.
- Index 13: The element is ‘a’. Increment counter to 6.
- Index 14: The element is ‘a’. Increment counter to 7.
- Index 15: The element is ‘a’. Increment counter to 8.
- Index 16: The element is ‘a’. Increment counter to 9.
- Index 17: The element is ‘b’. Counter remains 9.
- Index 18: The element is ‘a’. Increment counter to 10.
- Index 19: The element is ‘a’. Increment counter to 11.
- Index 20: The element is ‘a’. Increment counter to 12.
- Index 21: The element is ‘a’. Increment counter to 13.

3. The final count after iterating through all elements is 13.

Result: 13

Correct Answer:

13

Table 22: Case Study 10 - Correct Prediction with GPT-4o mini (CoT prompt)