Chain-of-Thought Tokens are Computer Program Variables

Anonymous ACL submission

Abstract

Chain-of-thoughts (CoT) instructs large language models (LLMs) to generate intermediate steps before reaching the final answer, and has been proven effective to help LLMs solve complex reasoning tasks. However, the inner mechanism of CoT still remains largely unclear. In this paper, we empirically study the role of CoT tokens in LLMs on two compositional tasks: multi-digit multiplication and dynamic programming. While CoT is essential for solving these problems, we find that preserving only tokens that store intermediate results would achieve comparable performance. Furthermore, we observe that storing intermediate results in an alternative latent form will not affect model performance. We also randomly intervene some values in CoT, and notice that subsequent CoT tokens and the final answer would change correspondingly. These findings suggest that CoT tokens function like variables in computer programs, but with potential drawbacks like unintended shortcuts and computational complexity limits between tokens.

1 Introduction

Chain-of-thoughts (CoT) (Wei et al., 2022) is a widely adopted technique that greatly boosts the capability of large language models (LLMs) in reasoning tasks like solving mathematical problems (Shao et al., 2024; Wang et al., 2024) or generating codes (Guo et al., 2025). By requiring language models to generate intermediate steps before reaching the final result, chain-of-thoughts enables LLMs to perform advanced reasoning, and thus significantly outperforms standard supervised learning methods. Various methods have been explored to unlock the ability of chain-of-thought reasoning in LLMs, for example designing prompts (Wei et al., 2022; Khot et al., 2022; Zhou et al., 2022), instruction tuning (Yue et al., 2023; Yu et al., 2023) or reinforcement learning (Havrilla et al., 2024; Wang et al., 2024; Guo et al., 2025).

Recent theoretical studies on the efficacy of chain-of-thoughts (Deng et al., 2024; Li et al., 2024; Chen et al., 2024) reveal that while it is exponentially difficult for language models to solve compositional problems requiring serial computations, CoT could help models solve problems under multinominal complexity. More interestingly, CoT tokens do not need to fall in the “language space”, using latent vectors could also enable language models to perform complex reasoning (Hao et al., 2024), indicating that CoTs are more than mere thought traces. However, the mechanism of how CoT works, and the role of CoT tokens are still not fully explored.

In this paper, we propose the hypothesis that **CoT tokens function like computer program variables**. To be specific, the tokens in CoT store intermediate values that will be used in subsequent computations, and these values are partially mutable to control the final output. As long as the important intermediate values are calculated and stored, the CoT that leads to the final answer could be represented in different forms.

To verify the hypothesis, we conduct empirical study on two types of problems that both require long-chain serial computations: multi-digit multiplication and dynamic programming. By comparing the performance of vanilla prompting with CoT prompting, we confirm that CoT is crucial for these problems. We also find that removing non-result tokens would not bring significant performance drops, which means that tokens storing intermediate values matter more in chain-of-thoughts.

We further explore whether intermediate values could be represented in different forms. We attempt to compress consequent number digits within a single latent vector, and experimental results show that it does not detriment the model performance. This phenomenon indicates that the existence, rather than the form, of intermediate values matters more to language models. However, when the degree of compression exceeds a certain limit of language models' capacity, it would lead to failure in reasoning.

To further confirm that the intermediate values are causally connected with the output, we intervene in some tokens in CoT, replacing them with random values. It can be observed that LLMs will ignore previous steps, and use the intervened value to perform subsequent computations, supporting that CoT tokens are causally related with the final result. We conclude that CoT tokens function like the variables in computer programs.

To sum up, we empirically study the function of CoT tokens, and find that: (1) The role of CoT tokens is similar to variables in computer programs as they store intermediate values used in subsequent computations; (2) The intermediate values could be stored in CoT tokens with different forms; (3) The values in CoT tokens are causally related to the final output and could be intervened like program variables. These findings are helpful in understanding alternative forms of CoT, and could assist in designing more concise CoTs.

2 Preliminary

2.1 Chain-of-Thoughts

Chain-of-thoughts (CoT) (Wei et al., 2022) is a technique commonly used in decoder-only transformers. Given the input text x, CoT attempts to generate intermediate steps z prior to the final answer y. In other words, instead of modeling the probability distribution $P(y|x)$, CoT attempts to model the joint distribution $P(y, z|x) = P(z|x)P(y|x, z)$.

For convenience, we use two special tokens `<COT>` and `</COT>` to separate CoT tokens from the final result in our experiments.

2.2 Compositional Tasks

It has been noticed that LLMs may fail on seemingly trivial problems like multi-digit multiplication. The commonality of these problems is that they need strict multi-hop reasoning to derive correct predictions, which requires language models to perform step-to-step reasoning like human intelligence. In this paper, we choose two representative tasks to study the role of CoT tokens:

Multi-digit Multiplication Calculating the multiplication result of two multi-digit numbers (x, y) requires executing multiple operations based on procedural rules (Dziri et al., 2023). A commonly adopted solution is the long-form multiplication algorithm, which iteratively calculates the digit-wise multiplication result and adds them up to get the final result. We describe the algorithm in Algorithm 1, see Appendix B for prompt and dataset construction details.

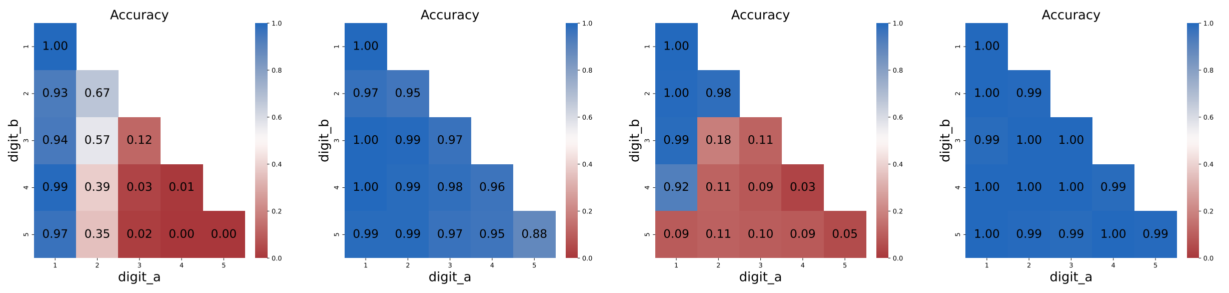
Algorithm 1 Digit-wise multiplication

```
Require: Integer $a$ and $b$
Ensure: Value of $a * b$
  Partial = [ ]
  for Digit $b[i]$ in $b$ do
    carry ← 0
    for Digit $a[i]$ in $a$ do
      x ← $a[i] * b[i]$ + carry
      digit ← x/10
      carry ← x mod 10
    end for
    res ← Combine digits and last carry
    Add res to Partial
  end for
  while Len(Partial) > 1 do
    x ← Partial[0] + Partial[1]
    Partial ← [x] + Partial[2:]
  end while
  return Partial[0]
```

Algorithm 2 Maximum path sum in a grid

```
Require: A $m * n$ matrix $W$
Ensure: Maximum weight sum $s$ on path
  DP ← $m * n$ matrix filled with 0
  for $i$ in range($m$) do
    for $j$ in range($n$) do
      DP[$i$][$j$] = max(DP[$i-1$][$j$], DP[$i$][$j-1$])
        + W[$i$][$j$]
    end for
  end for
  return DP[$m-1$][$n-1$]
```

Dynamic Programming Dynamic programming (DP) is an algorithmic paradigm that breaks down complicated problems into simpler sub-problems, and then recursively solves these sub-problems. In our experiments, we use the "Maximum Path Sum in a Grid" problem: *Given a $m \times n$ grid filled with non-negative numbers where only moving downward and rightward is allowed, find a path from top left to bottom right which maximizes the sum of all numbers along its path.* This is a classic problem that can be solved with dynamic programming



(a) Plain multiplication (b) CoT multiplication (c) Plain DP (d) CoT DP

Figure 1: Comparison on model accuracy between plain prompting and chain-of-thought prompting.

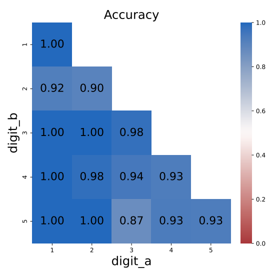


Figure 2: Model performance when non-result tokens are removed from CoT in multi-digit multiplication. Removing these tokens has little impact.

in $O(m \times n)$ time. We describe the algorithm in Algorithm 2, see Appendix C for prompt and dataset construction details.

3 CoT Tokens Store Intermediate Results

Experimental Setup In all of our experiments, We use Qwen-2.5-1.5B (Yang et al., 2024) as the backbone model. On each task, we finetune the model on the corresponding training data and then evaluate whether the generated final answer matches the golden answer. The training stage goes under a learning rate of 1e-5 for 1 epoch. See Appendix D for detailed hyperparameter settings.

3.1 Necessity of Chain-of-Thoughts

We start by examining the effectiveness of CoT by comparing the model performance under direct prompting and CoT settings. As illustrated in Figure 1, training the model with direct prompts faces difficulty starting from 3*3 multiplication problems, and completely fails on larger numbers. In contrast, the model could easily solve multiplication problems with chain-of-thoughts, with near-perfect accuracy.

The same applies to dynamic programming problems. Direct prompting would fail as the number of intermediate states increases, while CoT maintains its competence. These results support the conclusion that chain-of-thoughts is necessary for solving inherent serial problems that require multi-step reasoning, just as previous research suggests (Li et al., 2024; Chen et al., 2024).

3.2 Removing Non-result Tokens

One of the concerns about CoT is whether it could be compressed into a more concise form. An obvious approach is to remove some less important tokens. To be specific, we remove tokens that are neither a number nor a symbol[1], making CoT a sequence consisting purely of intermediate results to solve the task.

Figure 2 shows the model performances after removing these tokens. While the removal would make the CoT unreadable, models finetuned on compressed CoT still achieve satisfying performance. We can infer from this phenomenon that intermediate results are more important than semantic completeness in CoT. In other words, the central function of CoT is to store the sequence of intermediate results.

3.3 Merging Results into Latent Tokens

Another concern about CoT is whether intermediate results should be explicitly recorded. To test this hypothesis, we try to merge some of the intermediate results, and represent the merged results with latent tokens.

Method Design As depicted in Figure 3, we use latent tokens <LAT> to store intermediate results, and each latent token stores the information of a complete number. For simplicity, we use multi-hot vectors $\mathbf{l}$ as the embedding of latent tokens: a one-hot vector $\mathbf{l} = (l_1, l_2, \ldots, l_d) \in \mathbb{R}^d$ consisting of d dimensions could represent a number N of at most

[1]For example word "carry" in multiplication problems, see Appendix D for details

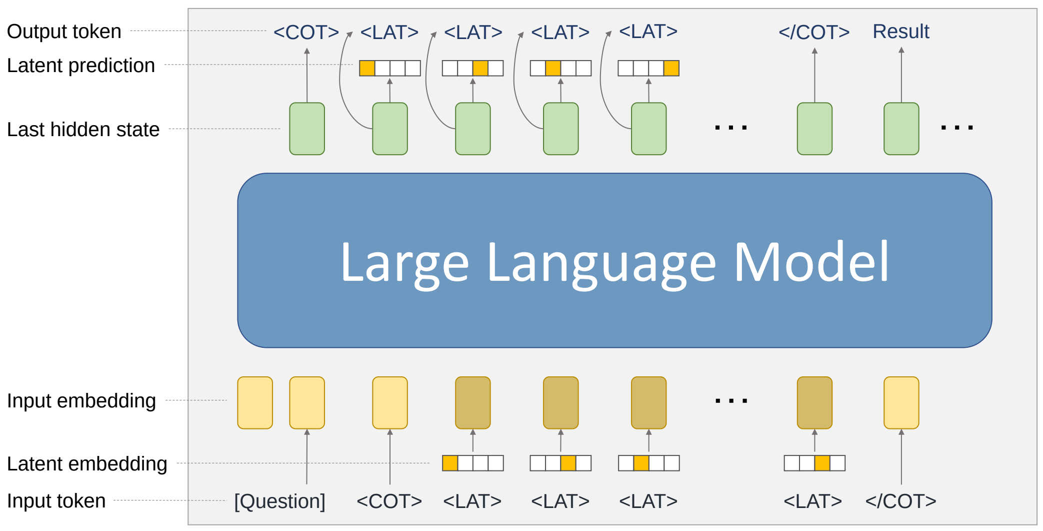
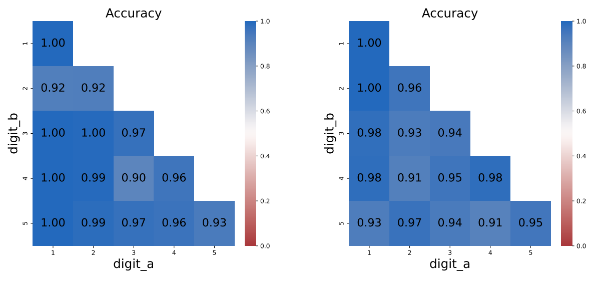


Figure 3: The model structure used to reason with latent tokens. We use one-hot vectors as the latent embedding of latent tokens <LAT>. When the input token is a latent token, we use its projected latent embedding to replace the original input embedding. Correspondingly, a latent output head is added to predict the latent embedding of the next token from the last hidden state.



(a) Multiplication (b) DP

Figure 4: Model performances when merging intermediate results into latent tokens.

n digits, where $d = 10n$.

$$\mathbf{l}_{10k+x} = \begin{cases} 1, \lfloor \frac{N}{10^k} \rfloor \mod 10 = x \\ 0, \lfloor \frac{N}{10^k} \rfloor \mod 10 \neq x \end{cases} \tag{1}$$

We start by setting all values in $\mathbf{l}$ to 0. Assuming that the value of the k-th digit under the little-endian system is x, we set $\mathbf{l}_{10k+x} = 1$. In this way, we could represent a number with a single latent token instead of multiple tokens.

To support reasoning with latent tokens, we augment the Transformer structure by adding an input projection module P_{in} and a latent output head P_{out}. When the input token c_t at position t is a latent token, we feed its latent embedding $\mathbf{l}_t$ to the projection module P_{in}, and use the projected vector as the input embedding; Correspondingly, the last hidden state $\mathbf{h}_t$ is fed to the latent output head P_{out} aside from the default LM head to predict the latent embedding of the next token $\mathbf{l}_{t+1}$.

We use linear layers to implement P_{in} and P_{out}, which can be described as:

$$P_{in}(\mathbf{l}_t) = \mathbf{W}_{in}\mathbf{l}_t + \mathbf{b}_{in} \tag{2}$$
$$P_{out}(\mathbf{h}_t) = \mathbf{W}_{out}\mathbf{h}_t + \mathbf{b}_{out} \tag{3}$$

where $\mathbf{W}_{in}, \mathbf{b}_{in}, \mathbf{W}_{out}, \mathbf{b}_{out}$ are trainable parameters. We randomly initialize these parameters.

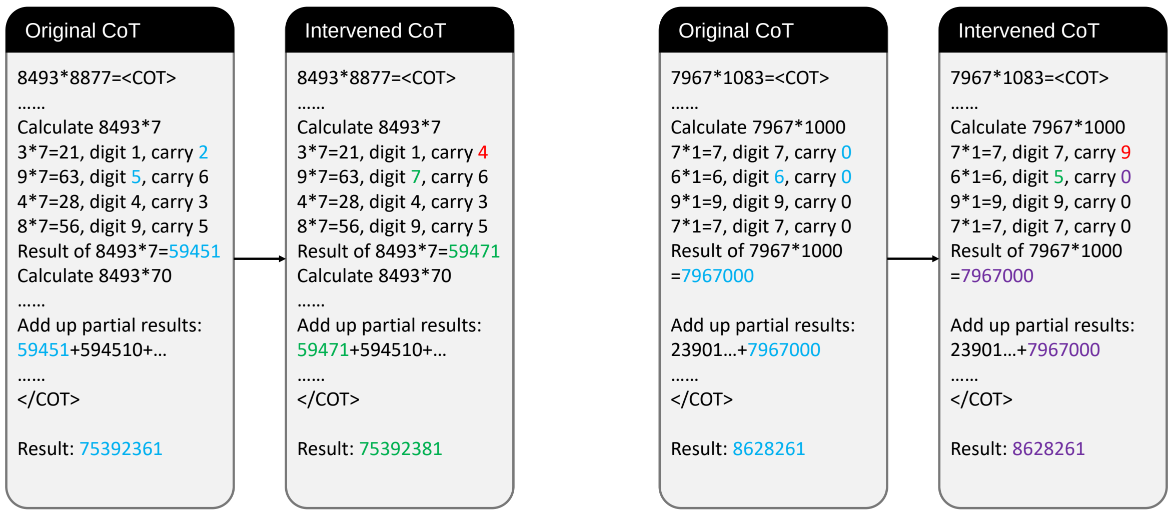
An additional latent loss $\mathcal{L}_{lat}$ is introduced to train the augmented model:

$$\mathcal{L}_{lat} = \frac{1}{N_l} \sum_{c_t=<LAT>} \text{BCE}(\sigma(P_{out}(\mathbf{h}_t), \mathbf{y})) \tag{4}$$

Where N_l is the number of latent tokens, $\mathbf{y}$ is the golden latent embedding, BCE is the binary cross entropy loss function, and σ is the Sigmoid function.

Experimental Setup For multiplication problems, we replace each digit-wise multiplication step with a single latent token and set $d = 20$; For DP problems, we replace each intermediate state with a single latent token and set $d = 50$. We add the latent loss $\mathcal{L}_{lat}$ with the default LM head loss as the final loss for training.

Figure 4 shows the model performances when trained with latent tokens. Surprisingly, merging digit tokens to a single latent token does not detriment the performance: the model retains most of its ability to solve problems. The accuracy of using



```
Original CoT
8493*8877=<COT>
......
Calculate 8493*7
3*7=21, digit 1, carry 2
9*7=63, digit 5, carry 6
4*7=28, digit 4, carry 3
8*7=56, digit 9, carry 5
Result of 8493*7=59451
Calculate 8493*70
......
Add up partial results:
59451+594510+...
......
</COT>

Result: 75392361
```

```
Intervened CoT
8493*8877=<COT>
......
Calculate 8493*7
3*7=21, digit 1, carry 4
9*7=63, digit 7, carry 6
4*7=28, digit 4, carry 3
8*7=56, digit 9, carry 5
Result of 8493*7=59471
Calculate 8493*70
......
Add up partial results:
59471+594510+...
......
</COT>

Result: 75392381
```

```
Original CoT
7967*1083=<COT>
......
Calculate 7967*1000
7*1=7, digit 7, carry 0
6*1=6, digit 6, carry 0
9*1=9, digit 9, carry 0
7*1=7, digit 7, carry 0
Result of 7967*1000
=7967000

Add up partial results:
23901...+7967000
......
</COT>

Result: 8628261
```

```
Intervened CoT
7967*1083=<COT>
......
Calculate 7967*1000
7*1=7, digit 7, carry 9
6*1=6, digit 5, carry 0
9*1=9, digit 9, carry 0
7*1=7, digit 7, carry 0
Result of 7967*1000
=7967000

Add up partial results:
23901...+7967000
......
</COT>

Result: 8628261
```

(a) Successful intervention

(b) Intervention with a shortcut error

Figure 5: Examples of a successful intervention (left) and an intervention with a shortcut error (right). Blue numbers refer to relevant values in the original CoT, red numbers refer to the intervention, green numbers refer to values that change as expected, but purple numbers do not change due to a shortcut error.

latent tokens on multiplication problems is almost identical with the accuracy of using full CoT. On 5*5 multiplication problems, using latent tokens even surpasses the original CoT, suggesting that the form of intermediate results does not matter.[2]

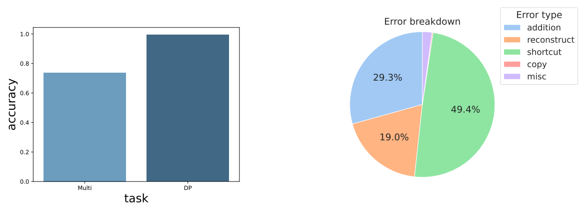
However, it can also be observed that using latent tokens brings disadvantage on DP problems where latent tokens store larger numbers. For example, the accuracy reduces by 9% on 4*5 DP problems. This raises the hypothesis that the computation complexity should not exceed a certain limit, which we will discuss further in Section 4.2.

4 CoT Tokens are Mutable Variables

In the previous section, we find that while CoT is essential for solving complex problems (Section 3.1), the tokens representing intermediate results are more important than others (Section 3.2). Meanwhile, compressing intermediate results into latent tokens would not obviously harm model performance (Section 3.3), indicating that intermediate results could be stored in different forms.

Here, we continue to discuss whether these stored intermediate results are causally related to the final prediction, and how the computation complexity between intermediate results affects model performance.

[2]It may be controversial whether the latent embedding used in this section is equal to vocabulary expansion, which we will discuss in Appendix F.



(a) Success rate

(b) Error breakdown

Figure 6: (a) **Success rate of intervention**. When the intervened output is the same as simulated, we view it as a successful intervention. (b) **Error breakdown**. Shortcut error occupies a large percentage of the errors.

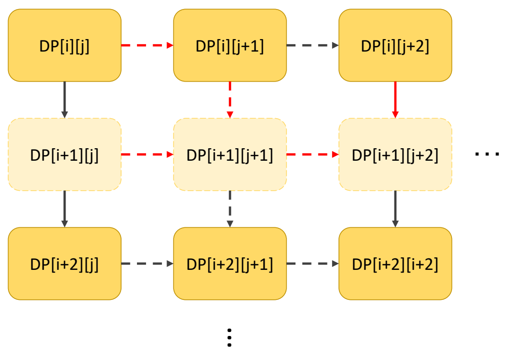


Figure 7: Demonstration of the alternative merging strategy. Each line refers to the compare-then-add state transfer function in the original setup. Nodes corresponding to the dashed boxes will not appear in the new CoT. It will cost at most 3 compare-then-add operations (red lines) to transfer states between new matrix tokens.

4.1 Intervening the Value of CoT Tokens

An inherent problem is that while CoT is essential for reaching the right answer, some of the intermediate results may only be correlational to the output, rather than having causal effects. To address this problem, we perform intervention experiments by replacing intermediate results and observe whether the final result would change as expected.

For multiplication problems, we randomly choose a substep in CoT and replace its result with a different random number; For DP problems, we randomly choose an intermediate state and replace it with a different random number. For simplicity, we perform interventions on 4*4 problems, and only one number is replaced in each data entry. Details are described in Appendix G.

As shown in Figure 6a, the intervention on both tasks achieves a decent success rate, clearly indicating that the intermediate values stored in CoT tokens are causally related to the final answer. We also notice that subsequent reasoning steps will change correspondingly. Take Figure 5a as an example, when we change the carry from 2 to 4, the model generates a result of $8493 * 7 = 59471$ instead of 59451, just as simulated. In other words, tokens in CoT not only store intermediate values, but they are also "variables" that would affect subsequent reasoning steps.

Another interesting observation is that the success rate on multiplication problems is significantly lower than that on DP problems. We investigate the cause of unsuccessful interventions and categorize them into 5 categories. (1) **Addition error** means that the model fails to add up partial multiplication results; (2) **Reconstruction error** means that the partial multiplication result conflicts with digit-wise results; (3) **Copy error** means that partial multiplication results do not correctly appear in the addition step; (4) **Shortcut error** means that the model learns a "shortcut" on certain multiplications (usually when one of the operands is 0 or 1); (5) **Misc error** covers the remaining errors.

Figure 6b illustrates the distribution of error types. Among the 5 types, shortcut error occupies the largest portion. As shown in Figure 5b, while changing the carry from 0 to 9 will affect the next digit as intended, the model does not change its result in the substep $7967 * 1000$. When multiplying a number x by 1, the model seems to be taking a shortcut of directly copying x, rather than collecting the digit-wise multiplication results.

To sum up, language models use the value in CoT tokens like treating program variables, but models may develop shortcut on easy subproblems that leave certain variables unused.

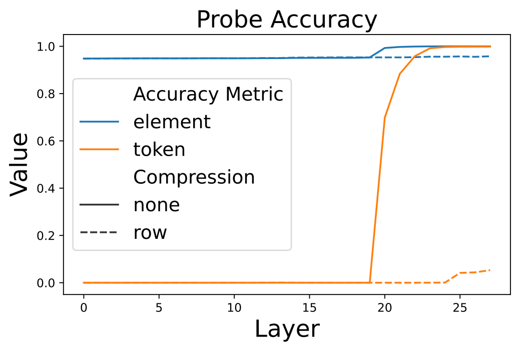


Figure 8: Probing accuracy on different layers. Intermediate variable values can only be probed on late layers, regardless of the overall accuracy.

4.2 Probing the Limit of CoT Tokens

In Section 3.3, we discover that intermediate values can be compressed in latent tokens. This naturally raises the question: to what extent could the values be compressed? To address this problem, we adopt some aggressive compression strategies and use linear probing classifiers to observe how the compression affects the final output.

We choose 5*5 DP problems as the base problem and use the latent token setting in Section 3.3. Specifically, we introduce an alternative strategy that merges two adjacent latent tokens in a row to one latent token (Figure 7). In this way, this strategy yields a 3*3 CoT token matrix instead of a 5*5 matrix. However, the computational complexity between CoT tokens also increases: it would cost up to 3 times as much as in the original case.

For each CoT token <LAT>, we use a linear probe P to probe its latent embedding $\mathbf{l}$ from the hidden states $\mathbf{h}_k$ on different layer k of the previous token. We use a unique probe P_k for each layer:

$$P_k(\mathbf{h}_k) = \mathbf{W}_k \mathbf{h}_k + \mathbf{b}_k \tag{5}$$

where $\mathbf{W}_k$ and $\mathbf{b}_k$ are trainable parameters.

After training the probes on the training set, we evaluate them with two metrics: **element accuracy** evaluates the ratio of correctly predicted individual dimensions, and **token accuracy** evaluates the ratio of correct latent tokens.

Figure 8 shows the result of probing CoT tokens. Aggressively merging CoT tokens will significantly lower both element accuracy and token accuracy, meaning that there exists a computation complexity limit, over which the LLM can no longer correctly calculate the next intermediate variable.

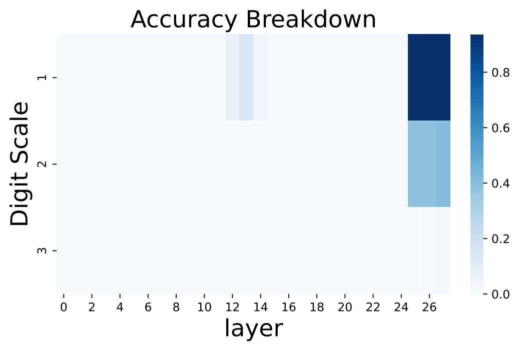


Figure 9: Accuracy breakdown by the scale of target values. When computational complexity between tokens exceeds a limit, the model will fail.

Figure 9 further breaks down the accuracy distribution by the range of values stored in merged latent tokens. We can see that merging latent tokens has little impact on numbers with a digit length of 1 or 2, but would decrease the accuracy to near 0 on larger number values. This phenomenon can be explained as it is easier to calculate small numbers, and thus the model could "afford" the extra computational cost of merging latent tokens.

Another interesting point to notice is that two accuracy curves share a similar pattern: the token accuracy stays at 0 from early layers, and rapidly rises around layer 20. Previous work (Stolfo et al., 2023; Zhu et al., 2025) has concluded that LLMs tend to use early-mid layers to gather and process information from previous tokens, and determine the output only in mid-late layers. We may further assume that the role of layers will not change with computation complexity between CoT tokens.

5 Discussion

Explaining alternative forms of CoT. By viewing CoTs as programs, we can explain alternative CoT forms in a novel way. For example, the success of internalizing CoT steps (Deng et al., 2024; Hao et al., 2024) could be viewed as compressing explicit step tokens into implicit latent tokens that cover all essential intermediate values. And the validity of inserting meaningless filler tokens (Goyal et al., 2024; Pfau et al., 2024) comes from enabling LLMs to store intermediate results in the hidden states of filler tokens. By reserving space for complex reasoning steps and compressing simple reasoning steps, we could design better CoTs in different forms.

Generalization of CoT "programs". From the experiments in previous sections, we can see that CoT tokens store intermediate values, and their values are subsequently used like the way variables function in computer programs. Theoretical proof has been made that Transformer with a recurrent module is Turing complete (Pérez et al., 2021). However, there is also evidence that LLMs may struggle to generalize on compositional problems (Dziri et al., 2023): models trained on easy problems would fail on more complex problems. In real-world settings, the type and complexity of desired programs are unknown, and a general-purpose LLM needs to first determine the type of program to use, or in other words, generate a meta-program first. It would be beneficial to explore the generalization ability of LLMs on different types of program paradigms, like loop, search, etc.

Identification of intermediate variable tokens. It is not surprising that the CoT generated by LLMs is partially redundant and could be shortened. In Section 3.2, we find that preserving value tokens could retain most of the ability of language models. While it is easy to judge whether a token stores intermediate results in multiplication and DP problems, it is harder to identify variable tokens on general tasks: Madaan and Yazdanbakhsh (2022) finds that plain text helps LLMs elicit semantic commonsense knowledge, which may be infused into later CoT tokens. Developing an approach to identifying variable tokens would benefit further CoT compression.

Estimation of computational complexity between variable tokens. Section 4.2 shows that LLMs would fail when the computational complexity between variable tokens exceeds a certain limit. However, it is difficult to estimate the exact complexity limit for LLMs. It is possible to calculate the theoretical bound of ability for finite-precision Transformers (Chen et al., 2024), but how LLMs process semantic information is still largely opaque, and unexpected features may appear (Lindsey et al., 2025). Moreover, LLMs are not guaranteed to solve similar subproblems in the same way, they may take shortcuts (Section 4.1) that would largely affect the computational complexity between variable tokens. We hope that the broader research community could help estimate the computational complexity between variable tokens in different types of questions.

6 Related Work

Chain-of-Thought (CoT) reasoning Chain-of-Thoughts (CoT) (Wei et al., 2022) is a commonly adopted technique in LLMs. Nowadays, CoT refers to a broad range of approaches that require LLMs to generate an intermediate reasoning process before reaching the final answer. Typical approaches include designing the prompt (Wei et al., 2022; Khot et al., 2022; Zhou et al., 2022) and finetuning LLMs on existing chain-of-thoughts (Yue et al., 2023; Yu et al., 2023). Recently, reinforcement learning also reveals its great potential in enabling LLMs to perform complex reasoning without extensive human annotations (Havrilla et al., 2024; Wang et al., 2024; Shao et al., 2024; Guo et al., 2025). While the tokens in CoT can be classified into symbols, patterns, and text, which both contribute to the final answer (Madaan and Yazdanbakhsh, 2022), it seems that LLMs can still perform well with a small amount of CoT tokens (Xu et al., 2025).

Aside from plain text, researchers have also explored alternative forms of CoT. Some works focus on search abilities, like tree-form thought traces (Yao et al., 2023; Xie et al., 2023) and Monte-Carlo Tree Search (MCTS) algorithms (Zhang et al., 2024; Guan et al., 2025). Another line of work attempts to reason in a latent space: Goyal et al. (2024) uses a pause token to help models process extra computation before reaching an answer, and Pfau et al. (2024) shows it is also possible to replace CoT with meaningless filler tokens. On top of this, Deng et al. (2024) tries to train models with gradually shortened CoT, and COCONUT (Hao et al., 2024) proposes the continuous thought paradigm, where the last hidden state of a latent token is used as the next input embedding.

Theoretical analysis on CoT It has been noticed that LLMs face difficulty in compositional problems where combining multiple reasoning steps is strictly required, and it may be an intrinsic drawback of the Transformer structure (Dziri et al., 2023). Feng et al. (2023) explains the phenomenon with the circuit complexity theory, and reaches the conclusion that it is impossible for a constant-depth log-precision transformer to solve certain math problems like linear equations. However, with the help of CoT, the model could solve these problems in polynomial complexity. Li et al. (2024) further extends the conclusion that constant-depth transformers using constant-bit precision could solve any problems solvable by boolean circuits, as long as they are equipped with CoT whose steps are longer than the circuit size. Chen et al. (2024) analyzes the problem with a multi-party autoregressive communication model, and finds that it is exponentially harder for Transformer models to solve composition tasks that require more steps than the model layers, and CoT could make the problem exponentially easier.

In fact, Transformer models are powerful enough to represent finite-state automata (Liu et al., 2022), and could even be Turing-complete (Pérez et al., 2021) to simulate computer programs when equipped with loop modules (Giannou et al., 2023). We hold the belief that these findings could also be extended to chain-of-thoughts reasoning.

7 Conclusion

In this paper, we empirically explore the role CoT tokens play in reasoning. By observing the model performance on multi-digit multiplication problems and dynamic programming, we confirm that CoT is essential for solving these compositional problems. We further find that we could mostly preserve model ability by only using tokens that store intermediate results, and these intermediate results could be stored in different forms.

To validate the causal connection between CoT tokens and model output, we intervene values in CoT and find that both the subsequent reasoning process and the final result would change corresponding to the intervention. The way CoT tokens behave is similar to the function of computer program variables. However, in easy subproblems LLMs would learn shortcuts that are unfaithful to the generated reasoning process, and the intervention would fail under these scenarios. We also train probing classifiers to probe variable values from hidden states on different layers, and find that there exists a computational complexity limit between CoT tokens. Intermediate values could be compressed within a single latent CoT token, but the model would drastically fail when computational complexity exceeds the limit.

Our work conducts preliminary experiments on the function of CoT tokens, and there still exist mysteries like generalization ability, variable identification and complexity limit estimation, which we leave for future explorations.

Limitations

In this paper we empirically demonstrate that an important function of CoT tokens is to store intermediate values, and these values function like program variables. However, currently we are not able to give a theoretical proof on this statement.

Another limitation of our work is that the experiments are conducted on two synthetic tasks with Qwen-2.5-1.5B, as it is difficult to identify and analyze intermediate results in real-world datasets like GSM8K and Math. Future experiments on other problems and models will be beneficial.

References

Lijie Chen, Binghui Peng, and Hongxun Wu. 2024. Theoretical limitations of multi-layer transformer. *arXiv preprint arXiv:2412.02975*.

Yuntian Deng, Yejin Choi, and Stuart Shieber. 2024. From explicit cot to implicit cot: Learning to internalize cot step by step. *arXiv preprint arXiv:2405.14838*.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, and 1 others. 2023. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36:70293–70332.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. 2023. Towards revealing the mystery behind chain of thought: a theoretical perspective. *Advances in Neural Information Processing Systems*, 36:70757–70798.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. 2023. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pages 11398–11442. PMLR.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. 2024. Think before you speak: Training language models with pause tokens. In *The Twelfth International Conference on Learning Representations*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2024. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*.

Alexander Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. 2024. Teaching large language models to reason with reinforcement learning. In *AI for Math Workshop@ ICML 2024*.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. 2024. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, and 8 others. 2025. On the biology of a large language model. *Transformer Circuits Thread*.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. 2022. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*.

Aman Madaan and Amir Yazdanbakhsh. 2022. Text and patterns: For effective chain of thought, it takes two to tango. *arXiv preprint arXiv:2209.07686*.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. 2021. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35.

Jacob Pfau, William Merrill, and Samuel R Bowman. 2024. Let's think dot by dot: Hidden computation in transformer language models. In *First Conference on Language Modeling*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Alessandro Stolfo, Yonatan Belinkov, and Mrinmaya Sachan. 2023. A mechanistic interpretation of arithmetic reasoning in language models using causal mediation analysis. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7035–7052.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. 2023. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36:41618–41650.

Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. 2025. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Longhui Yu, Weisen Jiang, Han Shi, YU Jincheng, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023. Mammoth: Building math generalist models through hybrid instruction tuning. In *The Twelfth International Conference on Learning Representations*.

Di Zhang, Jianbo Wu, Jingdi Lei, Tong Che, Jiatong Li, Tong Xie, Xiaoshui Huang, Shufei Zhang, Marco Pavone, Yuqiang Li, and 1 others. 2024. Llama-berry: Pairwise optimization for o1-like olympiad-level mathematical reasoning. *arXiv preprint arXiv:2410.02884*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

Fangwei Zhu, Damai Dai, and Zhifang Sui. 2025. Language models encode the value of numbers linearly. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 693–709.

A Potential Risks

We mainly conduct experiments on multiplication and DP problems, which are well-defined and structured. For real-world tasks like question answering or chatting, the boundary of “intermediate results” are more vague, and editing these results would lead to unexpected behavior.

B Details on Multiplication Task

Dataset construction For each problem scale of $m \times n$ that multiplies m-digit number a with n-digit number b, we generate 100,000 data entries by randomly sampling a and b. When the scale is small (for example 1×2), we exhaustively generate all number pairs instead. The generated data entries are then divided into train and test splits with a ratio of 90%/10%.

Prompt and CoT Formulation We use simple multiplication expressions as prompts. Figure 10 shows an example prompt for querying the model to perform multiplication.

For convenience, we use `<tool_call>` as the start-of-CoT token `<COT>`, `</tool_call>` as the end-of-CoT token `</COT>`, and `<|fim_middle|>` as the latent token `<LAT>`, which already exist in the tokenizer vocabulary.

We formulate the reasoning process with the algorithm of digit-wise multiplication, whose example is demonstrated in Figure 11. In the compressed CoT setting, we remove all tokens that merely represent text semantics in CoT, namely “Calculate”, “digit”, “carry”, “Result of” “Add up partial results:” and “The final result is:”, whose example is demonstrated in Figure 12.

Prompt example

3773*6821=

Figure 10: Example prompt for the multi-digit multiplication task.

C Details on Dynamic Programming Task

Dataset construction Similar to the multiplication problems, we generate 100,000 data entries for each problem scale of $m \times n$ (whose input matrix has a shape of m rows, n columns), and

Full CoT example

```
3773*6821=<tool_call>Calculate 3773*1
3*1=3, digit 3, carry 0
7*1=7, digit 7, carry 0
7*1=7, digit 7, carry 0
3*1=3, digit 3, carry 0
Result of 3773*1=3773
Calculate 3773*20
3*2=6, digit 6, carry 0
7*2=14, digit 4, carry 1
7*2=14, digit 5, carry 1
3*2=6, digit 7, carry 0
Result of 3773*20=75460
Calculate 3773*800
3*8=24, digit 4, carry 2
7*8=56, digit 8, carry 5
7*8=56, digit 1, carry 6
3*8=24, digit 0, carry 3
Result of 3773*800=3018400
Calculate 3773*6000
3*6=18, digit 8, carry 1
7*6=42, digit 3, carry 4
7*6=42, digit 6, carry 4
3*6=18, digit 2, carry 2
Result of 3773*6000=22638000

Add up partial results: 3773+75460+3018400+22638000
3773+75460+3018400+22638000=79233+3018400+22638000
79233+3018400+22638000=3097633+22638000
3097633+22638000=25735633

The final result is: 3773*6821=25735633</tool_call>

Result: 25735633
```

Figure 11: Example CoT for the multi-digit multiplication task.

Compressed CoT example

```
3773*6821=<tool_call>3773*1
3*1 3 0
7*1 7 0
7*1 7 0
3*1 3 0
3773*1=3773
3773*20
3*2 6 0
7*2 4 1
7*2 5 1
3*2 7 0
3773*20=75460
3773*800
3*8 4 2
7*8 8 5
7*8 1 6
3*8 0 3
3773*800=3018400
3773*6000
3*6 8 1
7*6 3 4
7*6 6 4
3*6 2 2
3773*6000=22638000

3773+75460+3018400+22638000
3773+75460+3018400+22638000=79233+3018400+22638000
79233+3018400+22638000=3097633+22638000
3097633+22638000=25735633

3773*6821=25735633</tool_call>

Result: 25735633
```

Figure 12: Example CoT after compression for the multi-digit multiplication task.

divide them into train and test splits with a ratio of 90%/10%.

To control the value of intermediate states within a reasonable range, we ensure all values x in the input matrix satisfy $1 < x < 100$. In other words, each input value is a 2-digit number.

Prompt formulation We use a matrix whose shape is the same as the input matrix to store intermediate values. The choice of special tokens `<COT>`, `</COT>` and `<LAT>` are the same as those in multiplication problems.

An example of the input prompt is shown in Figure 14, and an example of the full prompt is shown in Figure 15. Notice that we do not have a compressed version of CoT in dynamic programming tasks.

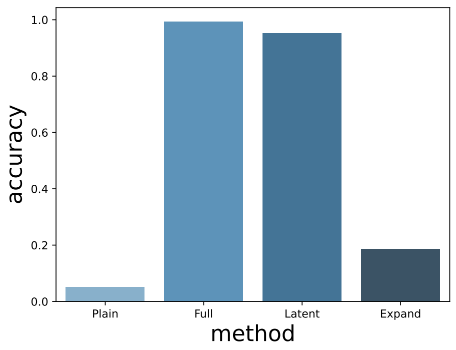


Figure 13: Comparison between expanding the vocabulary and other methods.

D Main Experiment Settings

For all of our experiments, we use Qwen-2.5-1.5B (Yang et al., 2024) from the huggingface model hub as the base model. We use the model according to its license and intended use.

We implement the experiments with the huggingface Transformers library. On each task, we finetune the model on the training set and then evaluate the model on the test set of the corresponding prompt type. We use the full-parameter supervised finetuning setting and do not use parameter-efficient training techniques.

During training, we use the AdamW optimizer with a learning rate of $1e-5$. The weight decay is set to 0 and the gradient clipping threshold is set to 1. We train the model for 1 epoch with a training batch size of 4 by default. For small datasets like 1×2 digit multiplication, we change the epoch to 10 to ensure convergence.

The models on multiplication problems are trained under BFloat16 precision, while models on DP problems are trained under Float32 precision. All experiments are trained on a single NVIDIA A40 40GB GPU. Running experiments on multiplication problems with a size of 5*5 costs about 60 GPU hours, and experiments on DP problems with a size of 5*5 costs about 10 GPU hours.

During evaluation, we evaluate with a batch size of 1. We only check the correctness of the final result during evaluation.

E Latent Experiment Settings

The hyperparameters in latent experiments are the same as the main experiment. For convenience, we use `<|fim_middle|>` as the latent token `<LAT>`.

In multiplication problems, the dimension of latent embeddings is set to 20 (10 for digit results and 10 for carry results). In dynamic programming problems, the dimension of latent embeddings is set to 50 to store values no larger than 100,000. The latent projection module P_{in} and the latent output head P_{out} are trained with the backbone model with the same learning rate. We simply add the latent loss $\mathcal{L}_{lat}$ with the original LM head loss $\mathcal{L}_{lm}$ as the final loss $\mathcal{L} = \mathcal{L}_{lat} + \mathcal{L}_{lm}$.

Figure 16 shows an example of latent CoT in multiplication problems, and Figure 17 shows an example of latent CoT in dynamic programming problems.

F Comparison with Vocabulary Expansion

Our implementation of latent tokens in Section 3.3 is not exactly the same with the implementation in COCONUT (Hao et al., 2024). Instead of using the last hidden state of the previous token as the new input embedding, we manually design multi-hot vectors to encode the value of intermediate results. This raises the question that given the number of possible multi-hot vectors is countable (though very large), is it equal to expanding the vocabulary?

To address the problem, we expand the vocabulary of the base model, giving each intermediate result a unique token named "<x>", where x is the value of the result. We randomly initialize the embeddings of the new token, and train them on 5*5 DP problems. Figure 13 shows the comparison between not using CoT, using full CoT, using latent CoT (Section 3.3) and expanding the vocabulary.

Prompt example

Find a path in the given table from the top-left corner to the bottom-right corner that maximizes the sum of the numbers on it. You can only move rightwards or downwards.

Table:
85 93 45 79 49
28 12 37 57 76
3 22 37 55 68
26 2 57 7 100
87 11 12 67 89

Figure 14: Example Prompt for the dynamic programming task.

Full CoT example

Find a path in the given table from the top-left corner to the bottom-right corner that maximizes the sum of the numbers on it. You can only move rightwards or downwards.

Table:
15 5 59 62 22
41 61 7 12 27
98 60 34 94 24
45 40 12 77 11
56 94 46 34 45

<tool_call>15 20 79 141 163
56 117 124 153 190
154 214 248 342 366
199 254 266 419 430
255 349 395 453 498</tool_call>

Result: 498

Figure 15: Example CoT for the dynamic programming task.

Latent CoT example

```
8493*8877=<tool_call>8493*7
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>|59451
8493*70 <|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>|594510
8493*800
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>|6794400
8493*8000
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>|67944000

59451+594510+6794400+67944000
59451+594510+6794400+67944000=653961+6794400+67944000
653961+6794400+67944000=7448361+67944000
7448361+67944000=75392361

8493*8877=75392361</tool_call>

Result: 75392361
```

Figure 16: Example latent CoT for the multi-digit multiplication task.

Latent CoT example

Find a path in the given table from the top-left corner to the bottom-right corner that maximizes the sum of the numbers on it. You can only move rightwards or downwards.

Table:
15 5 59 62 22
41 61 7 12 27
98 60 34 94 24
45 40 12 77 11
56 94 46 34 45

```
<tool_call><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|>
<|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|><|fim_middle|></tool_call>
```

Result: 498

Figure 17: Example latent CoT for the dynamic programming task.

We can clearly see that expanding the vocabulary falls far behind using full CoT or latent CoT, indicating that the precise value in latent tokens makes core contribution to reasoning.

G Intervention Experiment Details

In the intervention experiments, we randomly substitute a number value in the CoT generated by trained models on the test set. The interventions are performed on the full CoT texts. The substituted number has the same digit length as the original number, but with a different value. To prevent outlier values, we keep the first digit to be the same as the original number when substituting numbers with 2 or more digits.

We choose the number to substitute within the following range:

Multiplication

- x or y in "digit x, carry y" statements;
- A random number x in "Add up partial results:" statements;
- The first partial result x in "$a_1 + \ldots + a_n = x + \ldots$ statements;
- The result x in the "The final result is: $\ldots = x$" statement.

Dynamic programming A random intermediate value in the CoT.

After intervention, we truncate all tokens after the intervened value, and feed the partial CoT into trained models to complement the full CoT and get the final answer.

The detailed breakdown of errors in multiplication problems is shown in Table 1 (1 entry with deformed CoT is excluded):

Type	Count
Total	9999
Success	7383
Error	2616
Addition error	767
Reconstruct error	496
Shortcut error	1291
Copy error	6
Misc error	56

Table 1: Intervention error breakdown in multiplication problems.

H Probing Experiment Details

In the probing experiments, we probe on latent CoT for simplicity. We first collect hidden states of LLMs on different layers, and then train the probe classifiers. The training set of hidden states is collected by running the trained model on the original training set, and so is the test set.

We use a learning rate of $1e-3$ and a gradient clipping threshold of 1. We train the probe classifiers for 4 epochs with a training batch size of 32, and an evaluate batch size of 64.