Recursion of Thought: A Divide-and-Conquer Approach to Multi-Context Reasoning with Language Models

Soochan Lee
Seoul National University
soochan.lee@vision.snu.ac.kr

Gunhee Kim
Seoul National University
SNU-LG AI Research Center
gunhee@snu.ac.kr

Abstract

Generating intermediate steps, or Chain of Thought (CoT), is an effective way to significantly improve language models' (LM) multi-step reasoning capability. However, the CoT lengths can grow rapidly with the problem complexity, easily exceeding the maximum context size. Instead of increasing the context limit, which has already been heavily investigated, we explore an orthogonal direction: making LMs divide a problem into multiple contexts. We propose a new inference framework, called Recursion of Thought (RoT), which introduces several special tokens that the models can output to trigger context-related operations. Extensive experiments with multiple architectures including GPT-3 show that RoT dramatically improves LMs' inference capability to solve problems, whose solution consists of hundreds of thousands of tokens.

1 Introduction

Recently, LMs have become a prominent direction to solve reasoning. Given a question sequence, the models are tasked to predict the following answer sequence. One recent line of research for reasoning with LMs is *chain of thought* (CoT) generation (Nye et al., 2021; Wei et al., 2022; Kojima et al., 2022). In CoT generation, complex reasoning problems are solved by generating intermediate reasoning steps, or chain of thought, before producing the final answer. This allows the problem's complexity to be spread across multiple token generations, making each generation more straightforward given the previous tokens.

Although CoT dramatically increases reasoning accuracy, there is a critical issue that limits its utility: the effective context size of sequence models cannot grow unbounded. Context refers to the set of input tokens that a model is conditioned on when generating output. Practically, all sequence models have a limit on the maximum context length

due to various reasons. For instance, Transformers (Vaswani et al., 2017) suffer from a quadratic computational cost on the context length, and RNNs (Hochreiter and Schmidhuber, 1997) struggle with long-term dependency modeling. Therefore, even the state-of-the-art LMs limit the maximum context length to a few thousand tokens. However, complex real-world problems may take even millions of tokens of reasoning steps to reach the answer.

While there has been extensive research on Transformers with longer contexts (Tay et al., 2020b), we explore an orthogonal direction: divide and conquer. Our new model-agnostic inference framework *Recursion of Thought* (RoT) lets an LM recursively create multiple contexts by producing special tokens. Therefore, even if a problem's solution exceeds the maximum context size, the model can divide it into multiple short contexts. We show the potential of RoT with our new synthetic benchmark consisting of eight arithmetic and algorithmic tasks. One can easily adjust the difficulty of the tasks to produce problems with extremely long (100K+ tokens) reasoning steps. Without any task-specific component, such as a calculator, the models with RoT can easily learn to solve extremely complex problems whose solutions consist of hundreds of thousands of tokens. To the best of our knowledge, no previous work comes close to handling this scale of reasoning procedures. Since RoT is an early exploration in this direction, it needs several improvements to be applied to more practical scenarios. Nonetheless, the impressive experimental results suggest that the multi-context paradigm of RoT might play an important role in future LMs. In the supplementary file, we provide our PyTorch (Paszke et al., 2019) implementation that can *fully* reproduce the experiments.

2 Related Work

Scratchpad (Nye et al., 2021) is one of the earliest approaches demonstrating that fine-tuning lan-

guage models to produce CoT can largely improve reasoning accuracy. In the paper, the authors also mention the confined context size as a major hurdle to scaling their method. More recently, it has been found that sufficiently large pre-trained language models can be induced to produce CoT, by simply tuning the *prompt* (Wei et al., 2022; Kojima et al., 2022). Several concurrent works extend CoT prompting to decompose complex problems into smaller problems (Dua et al., 2022; Zhou et al., 2022; Khot et al., 2022). Although these works also share the principle of divide and conquer like RoT, they mostly focus on improving the reasoning accuracy of relatively small problems whose solutions usually can fit in a single context. On the other hand, we focus on solving problems that the solutions are orders of magnitude longer than the context size. More detailed description of related work can be found in Appendix A.

3 Recursion of Thought (RoT)

3.1 Inference

We start with how an RoT-trained LM performs at test time. RoT is a model-agnostic framework, whose only requirement is that the model can infer $p(x_{i+1}|X_{1:i})$, the probability of the next token x_{i+1} given a sequence $X_{1:i} = [x_1; ...; x_i]$. For recursive context control, we introduce the following special tokens: GO, STOP, and THINK. GO and STOP respectively mark the start and end of a problem sequence. They can be nested inside another GO-STOP pair to indicate a subproblem. THINK initiates a recursion procedure. RoT teaches a model how to use these tokens so that it can perform divide-and-conquer problem-solving. We formulate each inference context of a QA problem, denoted X, as the following concatenation:

$$X = [Q; Q^{\text{sub},1}; A^{\text{sub},1}; \ldots; Q^{\text{sub},N}; A^{\text{sub},N}; A] \tag{1}$$

where Q and A are the main question and answer sequence, and $Q^{\text{sub},*}$ and $A^{\text{sub},*}$ are those of the *top-level* subproblems. Although a subproblem can have smaller, lower-level subproblems recursively, only the top-level subproblems remain in an RoT context. During inference, a model is given Q and tasked to generate the rest. Questions (Q and $Q^{\text{sub},*}$) start with a GO token, and answers (A and $A^{\text{sub},*}$) end with a STOP token. In the base cases, contexts do not have ($Q^{\text{sub},*}$, $A^{\text{sub},*}$) pairs.

Figure 1 presents an example of solving 408 + 351 for better understanding. The pseudocode and more detailed illustrations can be found in Appendix B and E. RoT starts by initializing the context X with the original question Q (i.e., GO 4 0 8 + 3 5 1 = in Figure 1). Then, similar to CoT, the model solves multiple subproblems (generating $Q^{\text{sub},*}$ and $A^{\text{sub},*}$) before producing the final answer. However, there is a key difference: instead of producing a sub-answer directly, the model outputs the THINK token. This special token triggers a recursive process that separates the sub-question in a new context. If the new context is a base case (i.e., X^2, X^4, and X^5), the answer is produced directly. Otherwise, the model recursively solves more subproblems. If enough subproblems are solved, the model generates the final answer ending with a STOP. Once an answer is returned to the previous context, it replaces the THINK token, and the generation continues.

For tail recursion, where the last subquestion's answer becomes the final answer, we additionally introduce the TAIL token. If TAIL is used in the place of a GO token in the last subquestion $Q^{\text{sub},N}$, its answer $A^{\text{sub},N}$ is treated as the final answer A.

3.2 Training

Currently, we train RoT in a supervised manner, using ground truth (GT) intermediate steps that include when to output the special tokens. The GTs are constructed following the standard procedures developed for humans. For example, the procedures for arithmetic problems are borrowed from elementary school math. More details can be found in Appendix H. We leave training RoT with less supervision as a future work.

Each training example is constructed as a pair of a ground truth context sequence X and the corresponding target sequence Y. An example and the pseudocode for creating a target sequence are presented in Figure 3 and Algorithm 2 in Appendix D. Overall, Y is a copy of X except for the parts corresponding to Q and $A^{\text{sub},*}$. Since the question Q is always given in a context, Q is replaced by special PAD tokens, which are excluded from the loss function. Each subproblem's answer $A^{\text{sub},n}$ is replaced by a THINK token followed by several PADs that fill in the rest to make sure $|X| = |Y|$. This way, the model is trained to output THINK instead of the first token of $A^{\text{sub},n}$. Since the whole $A^{\text{sub},n}$ will be returned from the recursive process
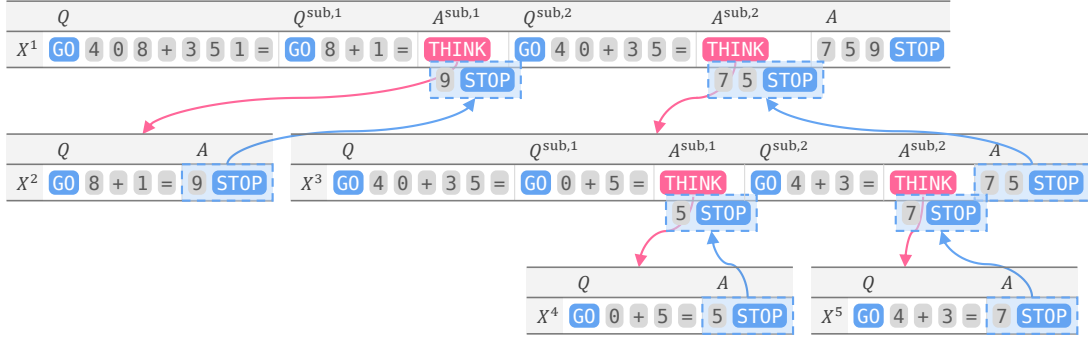
Figure 1: An example of the Recursion of Thought inference solving 408+351. Each table represents an inference context X^k in order of creation. For each context, the model is given Q and tasked to generate the rest. The model outputs the THINK token when it needs to generate $A^{\text{sub},*}$, the answer of a subproblem. The THINK token triggers a recursive process and is later replaced by the answer returned from the process.

and replace the THINK during inference, we do not need a training signal for the rest of $A^{\text{sub},n}$.

Given a pair (X, Y), the training objective is defined as follows:

$$\mathcal{L} = - \sum_i I[y_{i+1} \neq \texttt{PAD}] \log p(y_{i+1}|X_{1:i}) \quad (2)$$

where I is the indicator function that excludes PADs from training. Its form is almost identical to the standard LM objective: $\mathcal{L}_{\text{LM}} = -\sum_i \log p(x_{i+1}|X_{1:i})$, which is to predict the next token given previous tokens. Therefore, any sequence model is trained in the standard way, i.e., end-to-end via stochastic gradient descent.

4 Experiments

4.1 Baselines

We compare RoT with two baselines. The first one is to output an answer directly from a question, which we call *Without Thought* (WT). The other one is to generate all the intermediate steps before the answer without recursion (Nye et al., 2021), which we refer to as *Chain of Thought* (CoT; not to be confused with the CoT *prompting* (Wei et al., 2022)). We construct the ground truths for CoTs by unraveling the same recursive process which we design for RoT, into a single context sequence (see Appendix F for examples). Therefore, the number of tokens to generate while solving a problem is the same for both CoT and RoT (if we do not count the THINK tokens). However, the sizes of the individual contexts of CoT are far longer than those of RoT due to the recursively nested subproblems, limiting the range of solvable problems. Refer to Appendix M for a more detailed analysis of the context sizes. For a fair comparison, we *train* these

baselines and do not use any prompting technique. When evaluating, we consider a problem to be correctly solved only if all the intermediate steps and the answer are correct.

4.2 The Reasoning Problems

To evaluate the reasoning capabilities, we test four basic arithmetic tasks and four algorithmic tasks: addition, subtraction, multiplication, division, longest common subsequence, longest palindromic subsequence, 0-1 knapsack, and matrix chain multiplication. The details can be found in Appendix G. We choose these tasks because we can easily increase the problem difficulty while being able to get ground truth solutions. Therefore, we can test problems whose solution contains hundreds of thousands of tokens. All problems are formulated in pure sequence modeling, without any external programs (e.g., calculator) involved.

4.3 Experiments with GPT-3

Using the OpenAI API, we fine-tune GPT-3 for each reasoning task in §4.2 for 10K steps with a batch size of 256. The results are presented in Figure 2a, and the technical details are described in Appendix I. Each point in the graphs represents one experiment at a certain problem difficulty. We report the accuracy on a test set of 1K unique problems randomly sampled as explained in Appendix G. To the best of our knowledge, the problems at this scale (e.g., 48-digit addition/subtraction and 16-digit multiplication/division) have never been solved by any LM without the help of external programs. For reference, Minerva (Lewkowycz et al., 2022) achieves around 80% accuracy on 10-digit addition and 20% on 18-digit addition.
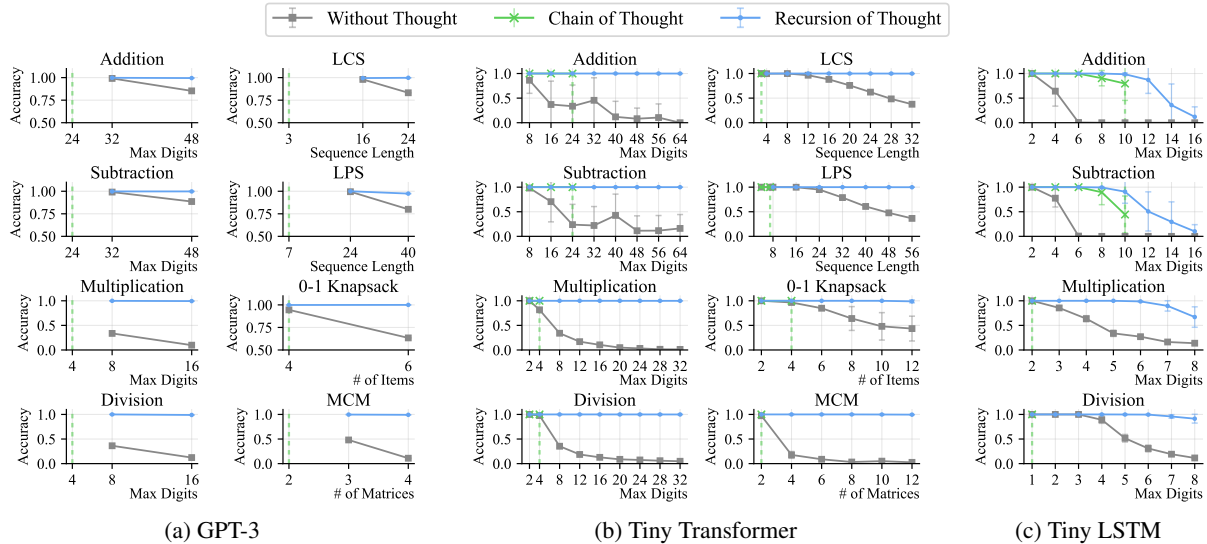
Figure 2: Comparison of the thought types. In each graph, the x-axis is the problem difficulty, while the y-axis is the reasoning accuracy. Each point represents an independent experiment. The green vertical lines indicate the maximum problem difficulty that CoT can handle without exceeding the maximum context size.

Results. Even WT fine-tuning cannot make GPT-3 deal with such a level of complexity, while CoT is not applicable due to the context limit of 2048. The green dotted lines mark the maximum difficulty that can be handled by CoT under the context limit. On the other hand, RoT enables the GPT-3 to achieve near-perfect scores in every experiment. As presented in Appendix M, solving each problem requires up to tens of thousands of tokens. Without any architectural change, RoT makes GPT-3 handle these extremely complex problems.

4.4 Experiments with Tiny Language Models

Recent research on reasoning has been mostly focused on extremely large pre-trained LMs. In this section, we show an interesting result that RoT can make even tiny models, without any pre-training, perform convoluted reasoning procedures. We test the two basic sequence model architectures: a Transformer (Vaswani et al., 2017) with 536K parameters and an LSTM (Hochreiter and Schmidhuber, 1997) with 272K parameters. These models are more than a million times smaller than the recent 540B-parameter PaLM (Chowdhery et al., 2022). The context limit is set to 2048 for the Transformer and 512 for the LSTM.

By virtue of their small sizes, we conduct far more extensive experiments than GPT-3, which are presented in Figure 2b and Figure 2c. For each experiment, we train a randomly initialized model and evaluate it on a test set of 30K unique problems. We repeat each experiment eight times and report

the average and standard deviation of the accuracies. With the tiny Transformer, we experiment to the extent that even humans would find daunting. For example, we test addition/subtraction up to 64 digits and multiplication/division up to 32 digits. Note that a 32-digit number cannot even fit into the 64-bit integer datatype.

Throughout the experiments, we observe consistent patterns: (i) WT's accuracy drops most quickly as the problem difficulty increases, (ii) CoT achieves near-perfect accuracy, but it can only be applied to simple problems due to the context limit, (iii) RoT achieves near-perfect accuracy and can be scaled up to extremely complex problems. Despite the small sizes, RoT makes the Transformers master all types of extremely complex problems. We do not test more difficult problems mainly because the evaluation becomes too costly, not because RoT is incapable of learning them.

5 Conclusion

We explored the novel idea of making LMs produce special tokens to create multiple contexts. Following the principle of divide and conquer, LMs with RoT can solve extremely complex problems that have never been handled by any LM. We believe the core idea of utilizing multiple contexts has a great potential and can play an essential role in future language models.

Limitations

Although RoT remarkably improves LMs' reasoning capability, we currently rely on supervised training to teach RoT. To apply RoT to a wider range of tasks, it would be crucial to reduce the expensive supervision. Parallel to our work, Khot et al. (2022) use prompting techniques to induce LMs to decompose problems. However, prompting has other drawbacks. First, lengthy prompts should be added for each inference, causing additional computational overhead. And more critically, it is hard to guarantee high accuracy. To achieve reasonable accuracy in the tasks in our experiments, each subproblem should be solved at extremely high accuracy (e.g., > 99.9%) since each problem may contain hundreds or thousands of subproblems. We have tested several prompting techniques with GPT-3, but could not get satisfactory accuracy. Therefore, we conclude that solely relying on prompting cannot be a solution to this problem. As one possible approach, we may combine RoT with the RL-based methodologies that are developed for reducing supervision of Neural Programmer-Interpreters (Li et al., 2017; Fox et al., 2018; Pierrot et al., 2019).

Another limitation of this work is that the experiments are performed on somewhat synthetic tasks. Since our goal is to enable LMs to solve reasoning problems whose intermediate steps are orders of magnitude longer than the context limit, we need a dataset with such complex problems. However, no currently available dataset meets this requirement. For example, the Long-Range Arena benchmark (Tay et al., 2020a) has at most 16K-token sequences and focuses on problems with long inputs and short outputs. On the other hand, we tackle problems that require generating 100K+ tokens to solve. Gathering natural language data at this scale is extremely challenging and costly. Therefore, we currently resort to arithmetic and algorithmic problems since it is easy to scale them up and generate ground-truth solutions. In the future, we hope to see new datasets and benchmarks that cover natural language reasoning at this scale.

Interestingly, RoT cannot facilitate length generalization, e.g., training on 8-digit multiplication with RoT cannot make a model generalize to 16-digit multiplication. We believe this problem is rooted in a more fundamental limitation of the Transformer architecture (Hahn, 2020), orthogonal to RoT. Fortunately, since RoT is a model-agnostic framework, we would be able to apply RoT to more advanced architectures to come in the future, which might be capable of length generalization.

Ethics Statement

Since the problem types in our experiments are pure arithmetic or algorithmic tasks, we do not find any ethical concerns directly related to our work. If RoT is applied to more general problems, the training data should meet ethical standards to ensure the non-toxic behavior of the model.

Acknowledgements

References

Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek B Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Oliveira Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern,

Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. 2022. Successive prompting for decomposing complex questions. *ArXiv*, abs/2212.04092.

Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. 2018. Parametrized hierarchical procedures for neural programming. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9:1735–1780.

Tushar Khot, H. Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *ArXiv*, abs/2210.02406.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *ArXiv*, abs/2205.11916.

Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. *ArXiv*, abs/2206.14858.

Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. 2017. Neural program lattices. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *ArXiv*, abs/2112.00114.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Thomas Pierrot, Guillaume Ligner, Scott E. Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. 2019. Learning compositional neural programs with recursive tree search and planning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14646–14656.

Scott E. Reed and Nando de Freitas. 2016. Neural programmer-interpreters. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020a. Long range arena: A benchmark for efficient transformers. *ArXiv*, abs/2011.04006.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020b. Efficient transformers: A survey. *ACM Computing Surveys*, 55:1 – 28.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903.

Denny Zhou, Nathanael Scharli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Huai hsin Chi. 2022. Least-to-most prompting enables complex reasoning in large language models. *ArXiv*, abs/2205.10625.

A Extended Related Work

Chain of Thought.   Scratchpad (Nye et al., 2021) fine-tunes LMs to generate CoT before the final answer. It demonstrates its effectiveness in 8-digit addition, polynomial evaluation, and Python program execution. Instead of fine-tuning, it is found that we can elicit large pre-trained LMs to produce CoT with appropriate prompting. For example, CoT prompting (Wei et al., 2022) adds several QA exemplars with CoT before the main question, encouraging the model to generate final answers in a similar manner. Compared to the few-shot CoT prompting of Wei et al. (2022), Kojima et al. (2022)'s zero-shot CoT prompting is even simpler; after a question, they start the answer with "Let's think step by step," and then let the model finish the rest. Minerva (Lewkowycz et al., 2022) utilizes these prompting techniques with a specially curated scientific pre-training dataset to achieve remarkable results on various reasoning benchmarks.

Prompting Language Models to Divide and Conquer Reasoning Problems.   Based on CoT prompting (Wei et al., 2022), several concurrent works demonstrate that decomposing problems into smaller subproblems can effectively improve reasoning accuracy. Successive prompting (Dua et al., 2022) induces a model to alternate between generating a question and answering the question until the final answer is produced. Similarly, least-to-most prompting (Zhou et al., 2022) makes a model start from the easiest subproblem and progressively solve more complex ones on top of the previous results. Decomposed prompting (Khot et al., 2022) is a modular approach that the subproblems are solved by different modules depending on the problem type. It also supports recursive decomposition. These works are all closely related to our work. Our work is unique in that we deal with far more complex problems that consist of thousands of subproblems. In this case, the individual subproblems should be solved with almost perfect accuracy, or the overall accuracy drops significantly. We empirically find that such a level of accuracy is hard to achieve by simply prompting a pre-trained LM.

Neural Programmer-Interpreter (NPI).   Unlike language models, NPI (Reed and de Freitas, 2016) interacts with its environment through a series of program execution. It consists of an LSTM core, an encoder for each domain, and a memory of program embeddings. At every time step, the LSTM core takes a program embedding, arguments, and an observation of its environment to produce the next program embedding and corresponding arguments. Cai et al. (2017) combine NPI with recursion and show that recursion plays a critical role in generalization. Since NPI requires full execution traces for training, there are multiple works to relax this requirement using reinforcement learning (Li et al., 2017; Fox et al., 2018; Pierrot et al., 2019).

B   RoT Inference Algorithm

Algorithm 1 Recursion of Thought Inference

Require: A sequence model $\mathcal{M}$ trained for Recursion of Thought, a question sequence Q
 1: **function** RoT($\mathcal{M}, Q$)
 2:     $X \leftarrow Q$ ▷ Initialize context with Q
 3:     $i_{\text{ans}} \leftarrow |X| + 1$ ▷ Start of answer
 4:     $t \leftarrow false$ ▷ Tail recursion
 5:     **while** True **do**
 6:         $x \leftarrow \mathcal{M}(X)$ ▷ Generate next token
 7:         $X \leftarrow [X; x]$
 8:         **if** $x = $ STOP **then**
 9:             **return** $X_{i_{\text{ans}}:|X|}$
10:         **else if** $x = $ GO **then**
11:             $i_{\text{go}} \leftarrow |X|$ ▷ Mark last GO
12:         **else if** $x = $ TAIL **then**
13:             $i_{\text{go}} \leftarrow |X|$
14:             $t \leftarrow true$ ▷ Mark tail recursion
15:         **else if** $x = $ THINK **then**
16:             $Q^{\text{sub}} \leftarrow X_{i_{\text{go}}:|X|-1}$
17:             $A^{\text{sub}} \leftarrow $ RoT($\mathcal{M}, Q^{\text{sub}}$)
18:             **if** t **then**
19:                 **return** A^{sub}
20:             **end if**
21:             $X \leftarrow [X_{1:|X|-1}; A^{\text{sub}}]$
22:   ▷ Replace THINK with A^{sub}
23:             $i_{\text{ans}} \leftarrow |X| + 1$
24:         **end if**
25:     **end while**
26: **end function**

C   Training Batch Distribution

We use the same problem distribution for both training and evaluation since out-of-distribution generalization is not within the scope of this paper. That is, when teaching 6-digit multiplication to the model, both training and test sets are all examples of 6-digit multiplication. The problem distributions are elaborated in Appendix G. Another important detail regarding the training of RoT is that each training example in a batch is a context, not a whole problem. Since RoT generates multiple contexts per problem, often a large portion of contexts can be a duplicate (mostly the base cases). Therefore, to build a training batch for RoT, we first sample a top-level problem and find the set of unique RoT contexts from the problem. Out of the unique contexts, we randomly sample one context as a training example. We find this simple technique works well, and we do not need a more sophisticated method, such as the adaptive curriculum learning in Reed and de Freitas (2016).

D Target Sequence

Algorithm 2 Creating the target sequence

Require: Context $X = [Q; Q^{\text{sub},1}; A^{\text{sub},1}; \ldots; Q^{\text{sub},N}; A^{\text{sub},N}; A]$
1: $Y \leftarrow \underbrace{\text{PAD} \ldots \text{PAD}}_{|Q|}$
2: **for** n in $1 \ldots N$ **do**
3: $\quad Y \leftarrow [Y; Q^{\text{sub},n}]$
4: $\quad Y \leftarrow [Y; \text{THINK} \underbrace{\text{PAD} \ldots \text{PAD}}_{|A^{\text{sub},n}|-1}]$
5: **end for**
6: $Y \leftarrow [Y; A]$
7: **return** Y

	Q		$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$	A
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 = 9	STOP	GO 4 0 + 3 5 = 7	5 STOP	7 5 9 STOP
Y^1	PAD $\cdots$ PAD	GO 8 + 1 = THINK PAD	GO 4 0 + 3 5 = THINK PAD PAD	7 5 9 STOP

Figure 3: The target sequence Y^1 for X^1 in Figure 1.

E A Step-by-Step Illustration of RoT Inference

In this section, we provide a step-by-step illustration of the example in Figure 1. Here we assume an ideal model fully trained for RoT.

Step 1

The context is initialized with the question Q.

	Q
X^1	GO 4 0 8 + 3 5 1 =

Step 2

The model generates the first subquestion $8 + 1$.

	Q	$Q^{\text{sub},1}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =

Step 3

Instead of immediately producing the answer, the model outputs the THINK token.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	THINK

Step 4

The `THINK` token triggers the creation of a new context. The new context is initialized with the subproblem starting from the last `GO` of X^1, i.e., $8 + 1$.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	`THINK`

	Q
X^2	`GO` 8 + 1 =

Step 5

Since the subproblem is a base case, the model outputs the answer 9 immediately.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	`THINK`

	Q	A
X^2	`GO` 8 + 1 =	9 `STOP`

Step 6

The answer is returned and replaces the `THINK` token.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	9 `STOP`

Step 7

The model generates the next subproblem, which is to add the remaining digits. Then, it produces `THINK` to find its answer.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	9 `STOP`	`GO` 4 0 + 3 5 =	`THINK`

Step 8

The THINK token creates a new context X^3 for solving $40 + 35$.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	THINK

	Q
X^3	GO 4 0 + 3 5 =

Step 9

Since $40 + 35$ is not a base case, the model recursively produces more subproblems. In this case, the first subproblem is to add the last digits, i.e., 0 and 5. Then it outputs the THINK token to solve the subproblem.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	THINK

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^3	GO 4 0 + 3 5 =	GO 0 + 5 =	THINK

Step 10

The new context X^4 is created to solve $0 + 5$.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	THINK

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^3	GO 4 0 + 3 5 =	GO 0 + 5 =	THINK

	Q	A
X^4	GO 0 + 5 =	5 STOP

Step 11

The answer is returned to X^3 and replaces the `THINK` token.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	9 `STOP`	`GO` 4 0 + 3 5 =	`THINK`

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$
X^3	`GO` 4 0 + 3 5 =	`GO` 0 + 5 =	5 `STOP`

Step 12

The model generates the next subproblem.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	9 `STOP`	`GO` 4 0 + 3 5 =	`THINK`

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^3	`GO` 4 0 + 3 5 =	`GO` 0 + 5 =	5 `STOP`	`GO` 4 + 3 =	`THINK`

Step 13

X^5 is created to solve the subproblem $4 + 3$. Since this is a base case, the model produces the answer directly.

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^1	`GO` 4 0 8 + 3 5 1 =	`GO` 8 + 1 =	9 `STOP`	`GO` 4 0 + 3 5 =	`THINK`

	Q	$Q^{\text{sub},1}$	$A^{\text{sub},1}$	$Q^{\text{sub},2}$	$A^{\text{sub},2}$
X^3	`GO` 4 0 + 3 5 =	`GO` 0 + 5 =	5 `STOP`	`GO` 4 + 3 =	`THINK`

	Q	A
X^5	`GO` 4 + 3 =	7 `STOP`

Step 14

The answer from X^5 replaces the `THINK` token in X^3.

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	THINK

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$
X^3	GO 4 0 + 3 5 =	GO 0 + 5 =	5 STOP	GO 4 + 3 =	7 STOP

Step 15

Since all subproblems are solved in X^3, the answer 75 is generated and returned to X^1.

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	THINK

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$	A
X^3	GO 4 0 + 3 5 =	GO 0 + 5 =	5 STOP	GO 4 + 3 =	7 STOP	7 5 STOP

Step 16

The answer of X^3 replaces the `THINK` token in X^1.

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	7 5 STOP

Step 17

Since the subproblems in X^1 are all solved, the model produces the final answer.

	Q	$Q^{\mathrm{sub},1}$	$A^{\mathrm{sub},1}$	$Q^{\mathrm{sub},2}$	$A^{\mathrm{sub},2}$
X^1	GO 4 0 8 + 3 5 1 =	GO 8 + 1 =	9 STOP	GO 4 0 + 3 5 =	7 5 STOP

A
7 5 9 STOP

F Examples of CoT Training Data

If we solve the example of 408+351 in Figure 1 with RoT, the following five contexts are produced.

- X^1: GO 4 0 8 + 3 5 1 = GO 8 + 1 = 9 STOP GO 4 0 + 3 5 = 7 5 STOP 7 5 9 STOP

- X^2: GO 8 + 1 = 9 STOP

- X^3: GO 4 0 + 3 5 = GO 0 + 5 = 5 STOP GO 4 + 3 = 7 STOP 7 5 STOP

- X^4: `GO 0 + 5 = 5 STOP`

- X^5: `GO 4 + 3 = 7 STOP`

The CoT context of the same problem is:

- X^{CoT}: `GO 4 0 8 + 3 5 1 = GO 8 + 1 = 9 STOP GO 4 0 + 3 5 = GO 0 + 5 STOP GO 4 + 3 STOP 7 5 STOP 7 5 9 STOP`

In a slightly more complicated example of 34×5, the RoT contexts are as follows:

- X^1: `GO 3 4 * 5 = GO 4 * 5 = 2 0 STOP GO 3 * 5 = 1 5 STOP TAIL 1 5 0 + 2 0 = THINK`

- X^2: `GO 4 * 5 = 2 0 STOP`

- X^3: `GO 3 * 5 = 1 5 STOP`

- X^4: `GO 1 5 0 + 2 0 = GO 0 + 0 = 0 STOP GO 1 5 + 2 = 1 7 STOP 1 7 0 STOP`

- X^5: `GO 0 + 0 = 0 STOP`

- X^6: `GO 1 5 + 2 = GO 5 + 2 = 7 STOP 1 7 STOP`

- X^7: `GO 5 + 2 = 7 STOP`

The corresponding CoT context is:

- X^{CoT}: `GO 3 4 * 5 = GO 4 * 5 = 2 0 STOP GO 3 * 5 = 1 5 STOP TAIL 1 5 0 + 2 0 = GO 0 + 0 = 0 STOP GO 1 5 + 2 = GO 5 + 2 = 7 STOP 1 7 STOP 1 7 0 STOP`

Notice that the CoT context consists of all the corresponding RoT contexts as its subsequences. The number of tokens to generate is identical to that of RoT if we do not count the `THINK` tokens. Even in these simple examples, however, the context size of CoT is far longer than that of RoT. For much more complex problems, such as 8-digit multiplication or 0-1 Knapsack, the CoT context size can be orders of magnitude larger than RoT. See Appendix M for more details on the distribution of context sizes.

G Problem Specifications

G.1 The Arithmetic Problems

For arithmetic tasks, we test addition, subtraction, multiplication, and division on non-negative integers. For subtraction, we add a constraint that the first operand is not less than the second one, to enforce non-negative answers. For division, we let the output include both a quotient and a remainder, separated by a special token `R`, e.g., `GO 7 ÷ 3 = 2 R 1 STOP`.

As briefly mentioned in §4.2, naively sampling the operands from a uniform distribution makes the operands extremely biased towards large numbers. For example, the probability of sampling a 2-digit number from the 6-digit space is less than 0.01%. Thus, we define a variation of the log-uniform distribution (often called the reciprocal distribution) to sample the operands. As a result, we obtain roughly the same proportion of operands for each number of digits.

The probability density of a log-uniform distribution is proportional to the reciprocal of the value. By definition, zero is not in the support of a log-uniform distribution, and samples are overly concentrated to the first few values in the sampling range. Therefore, we slightly extend the log-uniform distribution by introducing an offset parameter δ. To sample an integer in range $[\alpha, \beta)$ with offset δ, we first uniformly sample a real number r in range $[\log(\alpha + \delta), \log(\beta + \delta)]$. Then, r is transformed to $\lfloor \exp(r) - \delta \rfloor$. We denote the extended log-uniform distribution $U_{\log}(\alpha, \beta, \delta)$. As δ gets larger, the samples are more dispersed to larger numbers. In the experiments, we set $\delta = 3$.

Additionally, we introduce several other sampling details for division problems. Assume that we independently sample two numbers a and b for the dividend and the divisor. In about half of the cases, the

Addition	Subtraction	Multiplication	Division
$1330 + 121163$	$376776 - 35241$	9466×176175	$620261 \div 155034$
$114780 + 4356$	$10638 - 100$	179×516	$111730 \div 1176$
$638 + 2$	$109033 - 52649$	5509×133	$28268 \div 1$
$35 + 77$	$85137 - 3098$	6783×2	$588137 \div 25571$
$114261 + 354$	$22355 - 2824$	6×80285	$180330 \div 739$
$3 + 13792$	$7 - 1$	37275×19258	$879975 \div 97772$
$10151 + 7$	$652781 - 78853$	168484×154	$111461 \div 905026$
$22 + 1399$	$64914 - 3114$	3331×40	$42338 \div 14003$
$363356 + 450475$	$13041 - 1422$	349×158	$108 \div 384103$
$73 + 11$	$28293 - 4540$	17988×262130	$60002 \div 7479$
$179895 + 4128$	$11553 - 3576$	8140×1670	$131467 \div 131290$
$3 + 10$	$656291 - 2795$	51×5	$890679 \div 62$
$1 + 141972$	$93 - 42$	16497×158	$228 \div 131108$
$57612 + 18403$	$55972 - 1782$	74×10	$892 \div 124$
$9 + 1621$	$84587 - 51$	216×13414	$15 \div 964156$
$3370 + 381$	$273269 - 5867$	621×2	$369044 \div 28364$
$678 + 8854$	$274405 - 14$	2×5951	$457 \div 46$
$422 + 10348$	$51926 - 9$	189486×13080	$14687 \div 730$
$118 + 582$	$4272 - 229$	552792×763	$200361 \div 1049$
$1343 + 408534$	$223267 - 377$	77×3	$19715 \div 965179$
$24 + 9251$	$14857 - 1994$	179090×469029	$98 \div 7$
$315 + 652424$	$914771 - 836$	1037×258	$406 \div 9$
$355 + 4434$	$3035 - 2963$	8×769974	$47345 \div 122$
$22 + 834928$	$30 - 12$	47765×7254	$391613 \div 1631$
$3028 + 357$	$149 - 4$	5608×18164	$892642 \div 3898$
$777 + 1355$	$89057 - 6$	21437×12	$241554 \div 1901$
$154874 + 81059$	$296410 - 9$	15007×15	$116475 \div 12908$
$64936 + 216852$	$45 - 3$	539860×427	$488317 \div 197443$
$3 + 340939$	$78906 - 3$	3583×9754	$7519 \div 325$
$3 + 984775$	$56560 - 29960$	13×66	$3560 \div 847611$
$50581 + 1183$	$98 - 6$	266394×185	$9711 \div 1385$
$415 + 943$	$16551 - 920$	3988×12	$44540 \div 103$
$110 + 49$	$25606 - 194$	5514×57	$19721 \div 58$
$15 + 17058$	$45 - 37$	5×1712	$59544 \div 24$
$36278 + 100$	$129443 - 70196$	17×430178	$333057 \div 333057$
$6 + 23516$	$221 - 54$	227×127	$25719 \div 5142$
$1462 + 848$	$11010 - 818$	20888×54	$7544 \div 46$
$1002 + 2773$	$47759 - 67$	96×232801	$45 \div 410$
$135 + 178346$	$10 - 8$	175×1050	$195659 \div 2047$
$22672 + 162038$	$1439 - 153$	146×166	$412572 \div 16$

Table 1: 40 randomly selected samples of each type of 6-digit arithmetic problems.

dividend a would be less than the divisor b, so the quotients will be zero for those cases. To ensure a diverse range of quotients, we sample the divisor b from $U_{\log}(1, 10^N, 3)$, the quotient c from $U_{\log}(0, 10^N/b, 3)$, and the remainder r from $U_{\log}(0, b, 3)$. The dividend is calculated from these values: $a = b \times c + r$. This way, we can sample division problems with a diverse range of quotients and remainders.

Table 1 presents 40 problem samples for each 6-digit problem type. Several properties of our sampling scheme can be observed in the table. First, each number ranges over diverse numbers of digits. Second, the division problems are mostly non-trivial, i.e., the quotients are not concentrated at zero.

G.2 The Algorithmic Problems

G.2.1 Longest Common Subsequence (LCS)

The question of an LCS problem is two number sequences joined by the `LCS` token, and the answer is the corresponding LCS and its length separated by `;`. Here is an example of a length-4 LCS problem:

- Q: `GO` `1` `2` `3` `4` `LCS` `2` `4` `6` `8` `=`

- A: `2` `4` `;` `2` `STOP`

For a length-N LCS problem, we sample two sequences of length N. Each character of the sequences is randomly sampled from 0-9 with equal probability.

G.2.2 Longest Palindromic Subsequence (LPS)

The question of a length-N LPS problem starts with the `LPS`, followed by a sequence of length N. Similar to LCS, the answer contains the corresponding LPS and its length separated by `;`. The following is an example of a length-8 LPS problem:

- Q: `GO LPS 4 1 2 5 3 2 6 1 =`

- A: `1 2 3 2 1 ; 5 STOP`

The sequence of an LPS problem is sampled in the same way as done for the LCS problem.

G.2.3 0-1 Knapsack

Each item in a 0-1 Knapsack problem is represented by its value and weight. For instance, `1 2 & 3 4` represents an item with a value of 12 and a weight of 34. The question part of a 0-1 Knapsack problem is a sequence consisting of the `KNAPSACK` token, a list of items separated by `,`, the token `@`, and the capacity of the knapsack. The answer part starts with a list of items to include, then `$`, and finally the total value. The following is an example of a 3-item knapsack problem.

- Q: `GO KNAPSACK 5 & 1 2 , 2 5 & 1 5 , 1 9 & 1 8 @ 4 0 =`

- A: `2 5 & 1 5 , 1 9 & 1 8 $ 4 4 STOP`

In this example, given a knapsack of capacity 40, the last two are selected with a total value of 44.

For a fixed number of items, we uniformly sample each item's value and weight from the integers of range [1, 99].

G.2.4 Matrix Chain Multiplication (MCM)

The cost of multiplying many matrices is very sensitive to the order of multiplication. Matrix chain multiplication is the task of finding the best order with the minimum cost. Here, the cost is defined to be the total number of element multiplications. In the example of three matrices A, B, and C, whose shapes are 4×2, 2×8, and 8×3 respectively, the cost of computing $(AB)C$ is $4 \times 2 \times 8 + 4 \times 8 \times 3 = 160$, while another order $A(BC)$ costs only $2 \times 8 \times 3 + 4 \times 2 \times 3 = 72$. In the question of an MCM problem, the sizes of the matrices are enumerated, and the answer contains the order and the total cost separated by `;`. The example above is represented as the following sequences.

- Q: `GO MCM 4 × 2 , 2 × 8 , 8 × 3 =`

- A: `4 × 2 , ( 2 × 8 , 8 × 3 ) ; 7 2 STOP`

Given a fixed number of matrices, we sample the sizes of matrices from the range [1, 99].

G.2.5 Sorting

Although not included in the main text, we test the problem of sorting multi-digit numbers. The results are presented in Appendix N. The problem difficulty is defined by the maximum number of terms. For a sorting problem of at most N terms, we first uniformly sample the number of terms from $[2, N]$. Then we sample each term from $U_{\log}(0, 1000, 5)$. The following is an example of the sorting problem.

- Q: `GO SORT 1 3 9 , 1 6 0 , 4 3 4 , 7 9 6 , 4 1 =`

- A: `4 1 , 1 3 9 , 1 6 0 , 4 3 4 , 7 9 6 STOP`

H   Details of the Recursive Reasoning Procedures

In this section, we elaborate on the procedures to recursively solve the arithmetic problems. Specifically, we present the algorithms to produce the subproblems of a problem. Therefore, for a set of randomly sampled questions, we can generate ground truth contexts using these algorithms. For better understanding, we present the key parts of our Python code, the `thought` methods. For each problem, we create a child class the `Problem` class and implement `thought` static method. The method takes a set of arguments for a problem and returns the list of direct subproblems. Each subproblem is represented by a problem class, problem arguments, and recursion type (whether it is a tail recursion or not). We use named tuple `T` to group this information:

```python
from collections import namedtuple
T = namedtuple('Thought', ['prob_cls', 'args', 'type'], defaults=[''])
```

For instance, `T(Mul, (3, 4))` represents a regular subproblem of 3×4, and `T(Add, (12, 340), 'tail')` represents a subproblem of $12 + 340$ which should be performed as a tail recursion. Once the `thought` method returns a list of `T`s, we can recursively find more subproblems for each subproblem.

H.1   Addition

The core idea of our recursive procedure for addition is to first add the last digits and then add the rest. If the sum of the last digits is greater than or equal to 10, we insert another subproblem for adding the carry right after adding the last digits.

```python
class Add(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        left, right = args

        # Base cases
        if left < 10 and right < 10:
            return []

        l_last, r_last = left % 10, right % 10
        thoughts = [T(Add, (l_last, r_last))]

        l_rest, r_rest = left // 10, right // 10
        if l_last + r_last >= 10:
            thoughts.append(T(Add, (l_rest, 1)))
            l_rest += 1

        if l_rest > 0 and r_rest > 0:
            thoughts.append(T(Add, (l_rest, r_rest)))

        return thoughts
```

Figure 1 in the main draft is an example with no carry, and the following is another example of 27+65 with a carry.

- X^1: GO 3 1 7 + 6 5 = GO 7 + 5 = 1 2 STOP GO 3 1 + 1 = 3 2 STOP GO 3 2 + 6 = 3 8 STOP 3 8 2 STOP

- X^2: GO 7 + 5 = 1 2 STOP

- X^3: GO 3 1 + 1 = GO 1 + 1 = 2 STOP 3 2 STOP

- X^4: GO 1 + 1 = 2 STOP

- X^5: GO 3 2 + 6 = GO 2 + 6 = 8 STOP 3 8 STOP

- X^6: GO 2 + 6 = 8 STOP

639

H.2 Subtraction

Similar to addition, we first subtract the last digits and solve the rest recursively. When subtracting the last digits x and y, we always borrow 10 for x to prevent a negative result. The borrowing of 10 is easy for a sequence model: just put `1` before x. Therefore, the base cases of subtraction are when $a \leq 19$ and $b \leq 9$. If the subtraction result of the last digits is smaller than 10, i.e., the borrow is actually needed, we subtract 1 from the rest of the first operand m.

```python
class Sub(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        left, right = args

        # Base cases
        if left <= 19 and right <= 9:
            return []

        l_last = left % 10 + 10
        r_last = right % 10
        thoughts = [T(Sub, (l_last, r_last))]
        l_rest, r_rest = left // 10, right // 10
        if l_last - r_last < 10:
            thoughts.append(T(Sub, (l_rest, 1)))
            l_rest -= 1
        if r_rest > 0:
            thoughts.append(T(Sub, (l_rest, r_rest)))

        return thoughts
```

Here is an example of 432-216:

- X^1: GO 4 3 2 − 2 1 6 = GO 1 2 − 6 = 6 STOP GO 4 3 − 1 = 4 2 STOP GO 4 2 − 2 1 = 2 1 STOP 2 1 6 STOP

- X^2: GO 1 2 − 6 = 6 STOP

- X^3: GO 4 3 − 1 = GO 1 3 − 1 = 1 2 STOP 4 2 STOP

- X^4: GO 1 3 − 1 = 1 2 STOP

- X^5: GO 4 2 − 2 1 = GO 1 2 − 1 = 1 1 STOP GO 4 − 2 = 2 STOP 2 1 STOP

- X^6: GO 1 2 − 1 = 1 1 STOP

- X^7: GO 4 − 2 = 2 STOP

Notice that the final answer and the questions of each subproblem can be easily constructed from the previous sequence.

H.3 Multiplication

The base cases of multiplication are (i) when either operand is 0 or 1, or (ii) when both operands are less than 10. If one of the operands is 0, then the answer is zero; when one of them is 1, then the answer is just a copy of the other operand. For the cases where both operands are less than 10, we just let the model memorize them, which is similar to an elementary school math curriculum.

There are two types of non-base cases. For the simpler case, where the second operand is less than 10, we first split the first operand into the last digit and the rest. We then multiply each of them with the second operand and combine the results. Otherwise, we split the second operand into the last digit and the rest. The first operand is multiplied to each of them, and the results are summed.

```python
class Mul(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        left, right = args
```

```python
 5
 6          # Base cases
 7          if left <= 1 or right <= 1:
 8              return []
 9          if left <= 9 and right <= 9:
10              return []
11
12          thoughts = []
13          if right < 10:
14              thoughts.append(T(Mul, (left % 10, right)))
15              thoughts.append(T(Mul, (left // 10, right)))
16
17              a1 = (left % 10) * right
18              a2 = (left // 10) * right
19              thoughts.append(T(Add, (a2 * 10, a1), 'tail'))
20          else:
21              a1 = left * (right % 10)
22              thoughts.append(T(Mul, (left, right % 10)))
23
24              a2 = left * (right // 10)
25              thoughts.append(T(Mul, (left, right // 10)))
26
27              thoughts.append(T(Add, (a2 * 10, a1), 'tail'))
28          return thoughts
```

Here are some example contexts of multiplication:

- X^1: `GO` 4 3 * 2 1 = `GO` 4 3 * 1 = 4 3 `STOP` `GO` 4 3 * 2 = 8 6 `STOP` `TAIL` 8 6 0 + 4 3 = `THINK`

- X^2: `GO` 4 3 * 1 = 4 3 `STOP`

- X^3: `GO` 4 3 * 2 = `GO` 3 * 2 = 6 `STOP` `GO` 4 * 2 = 8 `STOP` `TAIL` 8 0 + 6 = `THINK`

- X^4: `GO` 3 * 2 = 6 `STOP`

- X^5: `GO` 4 * 2 = 8 `STOP`

- X^6: `GO` 8 0 + 6 = `GO` 0 + 6 = 6 `STOP` 8 6 `STOP`

- X^7: `GO` 0 + 6 = 6 `STOP`

- X^8: `GO` 8 6 0 + 4 3 = `GO` 0 + 3 = 3 `STOP` `GO` 8 6 + 4 = 9 0 `STOP` 9 0 3 `STOP`

- X^9: `GO` 0 + 3 = 3 `STOP`

- X^{10}: `GO` 8 6 + 4 = `GO` 6 + 4 = 1 0 `STOP` `GO` 8 + 1 = 9 `STOP` 9 0 `STOP`

- X^{11}: `GO` 6 + 4 = 1 0 `STOP`

- X^{12}: `GO` 8 + 1 = 9 `STOP`

Notice that we use tail recursion in X^1 and X^3.

H.4 Comparison

Comparison is used as a subroutine during division. The procedure for comparison consists of three steps:

1. Compare the numbers of digits.

2. If the numbers of digits are the same, compare the most significant digits.

3. If the most significant digits are identical, compare the remaining digits recursively.

We find that the sequence models can perform the first step without an explicit subproblem. Therefore, we only add intermediate steps for the second and third steps.

```python
class Compare(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        left, right = args

        # Base cases
        if left < 10 and right < 10:
            return []

        thoughts = []
        digit_l, digit_r = len(str(left)), len(str(right))
        if digit_l == digit_r:
            # Compare the first digits
            l_first, r_first = int(str(left)[0]), int(str(right)[0])
            thoughts.append(T(Compare, (l_first, r_first)))
            if l_first == r_first:
                # Compare the rest
                l_rest = int(str(left)[1:])
                r_rest = int(str(right)[1:])
                thoughts.append(T(Compare, (l_rest, r_rest)))

        return thoughts
```

The following is an example of comparing 153 and 159.

- X^1: GO 1 5 3 VS 1 5 9 = GO 1 VS 1 = EQ STOP GO 5 3 VS 5 9 = LT STOP LT STOP

- X^2: GO 1 VS 1 = EQ STOP

- X^3: GO 5 3 VS 5 9 = GO 5 VS 5 = EQ STOP GO 3 VS 9 = LT STOP LT STOP

- X^4: GO 5 VS 5 = EQ STOP

- X^5: GO 3 VS 9 = LT STOP

H.5 Division

Solving division is the most challenging among the four basic arithmetic operations since the procedure is basically trial and error, searching for the correct quotient. Nonetheless, the following process is a recursive version of the elementary school division.

The base case is when the dividend is less than or equal to the divisor. If the dividend is smaller than the divisor, the quotient is 0, and the remainder is the dividend. If the dividend is equal to the divisor, then the quotient is 1, and the remainder is 0. Both cases can be handled relatively easily by neural sequence models. To determine whether it is one of these cases, we always perform the comparison as the first subproblem.

If it is not a base case, we check whether the dividend is smaller than 10 times the divisor. If the dividend is smaller, we subtract the divisor from the dividend and recursively divide the result with the divisor. The final answer is attained by simply adding 1 to the quotient of the smaller division.

To explain the other case, where the dividend is greater than 10 times the divisor, let us call the dividend a and the divisor b. First, we split the a into the last digit x and the remaining digits m. Then, we divide m with the divisor b, i.e., we are solving the one-digit-smaller subproblem first. Since we define the division operation to return both a quotient and a remainder, the quotient $q_1 = m/b$ and the remainder $r_1 = m \bmod b$ from the subproblem are added to the context. Next, we *concatenate* the remainder and x, which is numerically computing $r \times 10 + x$, and divide it again with b. Let the quotient and the remainder of this operation q_2 and r_2. Then, the quotient of the final answer is $q_1 \times 10 + q_2$, while the remainder is simply r_2.

```python
class Div(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        left, right = args
```

```
 5            thoughts = [T(Compare, (left, right))]
 6
 7            # Base cases
 8            if left <= right:
 9                return thoughts
10
11            thoughts.append(T(Compare, (left, right * 10)))
12            if left <= right * 10:
13                diff = left - right
14                thoughts.append(T(Sub, (left, right)))
15                thoughts.append(T(Div, (diff, right)))
16            else:
17                thoughts.append(T(Div, (left // 10, right)))
18                left_remainder = (left // 10) % right * 10 + left % 10
19                thoughts.append(T(Div, (left_remainder, right)))
20            return thoughts
```

The following is an example of $76 \div 29$.

- X^1: GO 7 6 ÷ 2 9 = GO 7 6 VS 2 9 = GT STOP GO 7 6 VS 2 9 0 = LT STOP GO 7 6 - 2 9 = 4 7 STOP GO 4 7 ÷ 2 9 = 1 R 1 8 STOP 2 R 1 8 STOP

- X^2: GO 7 6 VS 2 9 = GO 7 VS 2 = GT STOP GT STOP

- X^3: GO 7 VS 2 = GT STOP

- X^4: GO 7 6 VS 2 9 0 = LT STOP

- X^5: GO 7 6 - 2 9 = GO 1 6 - 9 = 7 STOP GO 7 - 1 = 6 STOP GO 6 - 2 = 4 STOP 4 7 STOP

- ...

- X^9: GO 4 7 ÷ 2 9 = GO 4 7 VS 2 9 = GT STOP GO 4 7 VS 2 9 0 = LT STOP GO 4 7 - 2 9 = 1 8 STOP GO 1 8 ÷ 2 9 = 0 R 1 8 STOP 1 R 1 8 STOP

- X^{10}: GO 4 7 VS 2 9 = GO 4 VS 2 = GT STOP GT STOP

- X^{11}: GO 4 VS 2 = GT STOP

- X^{12}: GO 4 7 VS 2 9 0 = LT STOP

- X^{13}: GO 4 7 - 2 9 = GO 1 7 - 9 = 8 STOP GO 4 - 1 = 3 STOP GO 3 - 2 = 1 STOP 1 8 STOP

- ...

- X^{17}: GO 1 8 ÷ 2 9 = GO 1 8 VS 2 9 = LT STOP 0 R 1 8 STOP

- X^{18}: GO 1 8 VS 2 9 = GO 1 VS 2 = LT STOP LT STOP

- ...

H.6 Longest Common Subsequence (LCS)

Given sequences A and B, the algorithm starts by comparing the last characters of the two sequences. If the last two characters are the same, we find LCS of the subsequences without the last characters, i.e., LCS of $A_{:-1}$ and $B_{:-1}$. Otherwise, we compute the LCSs of the cases where the last character of either side is removed and return the better one. In the following code, LCS._answer is the subroutine that finds the LCS of two sequences. Equal returns TRUE if the two arguments are the same, or FALSE otherwise.

```
1   class LCS(Problem):
2       @staticmethod
3       def thought(args) -> list[T]:
4           l, r = args
5           if len(l) == 0 or len(r) == 0:
6               return []
7
8           thoughts = [T(Equal, (l[-1], r[-1]))]
9           if l[-1] == r[-1]:
10              thoughts.append(T(LCS, (l[:-1], r[:-1])))
11              return thoughts
12
13          lcs1_args = (l[:-1], r)
14          lcs2_args = (l, r[:-1])
15          lcs1 = LCS._answer(lcs1_args)
16          lcs2 = LCS._answer(lcs2_args)
17          thoughts.extend([
18              T(LCS, lcs1_args),
19              T(LCS, lcs2_args),
20              T(Compare, (len(lcs1), len(lcs2)))
21          ])
22          return thoughts
```

The following is an example of finding the LCS of `123` and `234`.

- X^1: `GO` 1 2 3 LCS 2 3 4 = `GO` EQUAL 3 , 4 = FALSE `STOP` `GO` 1 2 LCS 2 3 4 = 2 ; 1 `STOP` `GO` 1 2 3 LCS 2 3 = 2 3 ; 2 `STOP` `GO` 1 VS 2 = LT `STOP` 2 3 ; 2 `STOP`

- X^2: `GO` EQUAL 3 , 4 = FALSE `STOP`

- X^3: `GO` 1 2 LCS 2 3 4 = `GO` EQUAL 2 , 4 = FALSE `STOP` `GO` 1 LCS 2 3 4 = ; 0 `STOP` `GO` 1 2 LCS 2 3 = 2 ; 1 `STOP` `GO` 0 VS 1 = LT `STOP` 2 ; 1 `STOP`

- ...

- X^{21}: `GO` 1 2 3 LCS 2 3 = `GO` EQUAL 3 , 3 = TRUE `STOP` `GO` 1 2 LCS 2 = 2 ; 1 `STOP` 2 3 ; 2 `STOP`

- ...

- X^{23}: `GO` 1 VS 2 = LT `STOP`

H.7 Longest Palindromic Subsequence (LPS)

The overall algorithm for LPS is similar to LCS. The base cases are when the sequence length is less than 3. If it is not a base case, we first check if the characters at both ends of the sequence are the same. If they are the same, we find the LPS of the subsequence excluding them. Otherwise, we compare the cases where one of the end characters is excluded.

```
1   class LPS(Problem):
2       @staticmethod
3       def thought(args) -> list[T]:
4           # Base cases
5           if len(args) == 1:
6               return []
7           elif len(args) == 2:
8               return [T(Equal, args)]
9
10          thoughts = [T(Equal, (args[0], args[1]))]
11          if args[0] == args[-1]:
12              sub_lps = LPS._answer(args[1:-1])
13              thoughts.extend([
14                  T(LPS, args[1:-1]),
15                  T(Add, (len(sub_lps), 2))
16              ])
```

```
17              else:
18                  lps1_args = args[:-1]
19                  lps2_args = args[1:]
20                  lps1 = LPS._answer(lps1_args)
21                  lps2 = LPS._answer(lps2_args)
22                  thoughts.extend([
23                      T(LPS, lps1_args),
24                      T(LPS, lps2_args),
25                      T(Compare, (len(lps1), len(lps2)))
26                  ])
27              return thoughts
```

The following is an example of LPS.

- X^1: `GO` LPS 1 2 3 2 = `GO` EQUAL 1 , 2 = FALSE `STOP` `GO` LPS 1 2 3 = 1 ; 1 `STOP` `GO` LPS 2 3 2 = 2 3 2 ; 3 `STOP` `GO` 1 VS 3 = LT `STOP` 2 3 2 ; 3 `STOP`

- X^2: `GO` EQUAL 1 , 2 = FALSE `STOP`

- X^3: `GO` LPS 1 2 3 = `GO` EQUAL 1 , 3 = FALSE `STOP` `GO` LPS 1 2 = 1 ; 1 `STOP` `GO` LPS 2 3 = 2 ; 1 `STOP` `GO` 1 VS 1 = EQ `STOP` 1 ; 1 `STOP`

- ...

- X^{10}: `GO` LPS 2 3 2 = `GO` EQUAL 2 , 2 = TRUE `STOP` `GO` LPS 3 = 3 ; 1 `STOP` `GO` 1 + 2 = 3 `STOP` 2 3 2 ; 3 `STOP`

- ...

- X^{14}: `GO` 1 VS 3 = LT `STOP`

H.8   0-1 Knapsack

The base cases are when there is only one item. In this case, we simply compare the item's weight and the knapsack's capacity, to determine whether the item should be included. If it is a non-base case, we compare two possibilities: (i) include the first item, or (ii) exclude the first item. We recursively compute the subproblems and find the case with the best value.

```
1   class LPS(Problem):
2       @staticmethod
3       def thought(args) -> list[T]:
4           items, capacity = args
5           value, weight = items[0]
6
7           # Base case
8           if len(items) == 1:
9               return [T(Compare, (weight, capacity))]
10
11          # When excluding the current item
12          items_max, value_max = Knapsack._answer((items[1:], capacity))
13          thoughts = [
14              T(Knapsack, (items[1:], capacity)),
15              T(Compare, (weight, capacity)),
16          ]
17
18          # When including the current item
19          if weight <= capacity:
20              items_sub, value_sub = Knapsack._answer(
21                  (items[1:], capacity - weight))
22              value_incl = value_sub + value
23              thoughts.extend([
24                  T(Sub, (capacity, weight)),
25                  T(Knapsack, (items[1:], capacity - weight)),
26                  T(Add, (value_sub, value)),
27                  T(Compare, (value_incl, value_max)),
28              ])
29
```

```
30              return thoughts
```

The following is an example of a 0-1 knapsack problem with three items and a knapsack capacity of 10.

- X^1: `GO` KNAPSACK 3 & 9 , 4 & 2 , 9 & 5 @ 1 0 = `GO` KNAPSACK 4 & 2 , 9 & 5 @ 1 0 = 4 & 2 , 9 & 5 $ 1 3 `STOP` `GO` 9 VS 1 0 = LT `STOP` `GO` 1 0 − 9 = 1 `STOP` `GO` KNAPSACK 4 & 2 , 9 & 5 @ 1 = $ 0 `STOP` `GO` 0 + 3 = 3 `STOP` `GO` 3 VS 1 3 = LT `STOP` 4 & 2 , 9 & 5 $ 1 3 `STOP`

- X^2: `GO` KNAPSACK 4 & 2 , 9 & 5 @ 1 0 = `GO` KNAPSACK 9 & 5 @ 1 0 = 9 & 5 $ 9 `STOP` `GO` 2 VS 1 0 = LT `STOP` `GO` 1 0 − 2 = 8 `STOP` `GO` KNAPSACK 9 & 5 @ 8 = 9 & 5 $ 9 `STOP` `GO` 9 + 4 = 1 3 `STOP` `GO` 1 3 VS 9 = GT `STOP` 4 & 2 , 9 & 5 $ 1 3 `STOP`

- ...

- X^{11}: `GO` 9 VS 1 0 = LT `STOP`

- X^{12}: `GO` 1 0 − 9 = 1 `STOP`

- X^{13}: `GO` KNAPSACK 4 & 2 , 9 & 5 @ 1 = `GO` KNAPSACK 9 & 5 @ 1 = $ 0 `STOP` `GO` 2 VS 1 = GT `STOP` $ 0 `STOP`

- ...

- X^{17}: `GO` 0 + 3 = 3 `STOP`

- X^{18}: `GO` 3 VS 1 3 = LT `STOP`

H.9 Ternary Addition and Multiplication

Ternary addition and multiplication arise as subproblems while solving MCM, which will be explained in the next section. They are simple extensions of addition and multiplication to three integers.

```python
class TernaryAdd(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        a1, a2, a3 = args
        return [
            T(Add, (a1, a2)),
            T(Add, (a1 + a2, a3), 'tail')
        ]


class TernaryMul(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        a1, a2, a3 = args
        return [
            T(Mul, (a1, a2)),
            T(Mul, (a1 * a2, a3), 'tail')
        ]
```

H.10 Matrix Chain Multiplication (MCM)

Given N matrices, the $N - 1$ subproblems are defined for each possible binary split. For the multiplication of four matrices $ABCD$, there are three possible binary splits: $A(BCD)$, $(AB)(CD)$, and $(ABC)D$. For each binary split, the total cost is the sum of (i) the minimum cost of computing the first group, (ii) the minimum cost of computing the second group, and (iii) the cost of multiplying the two matrices resulting from each group. Once we get the total costs of each binary split, we return the best split with the minimum cost. The following code implements this procedure.

```python
class MCM(Problem):
    @staticmethod
    def thought(args) -> list[T]:
        mats, min_order, min_cost = args

        # Base cases
        if len(mats) == 1:
            return []

        if min_order is None:
            # Top-level problem
            l_mats, r_mats = mats[:1], mats[1:]
        else:
            # Middle of recursion
            l_mats, r_mats = mats

        l_args = (l_mats, None, None)
        r_args = (r_mats, None, None)
        l_order, l_cost = MCM._answer(l_args)
        r_order, r_cost = MCM._answer(r_args)
        agg_cost = l_mats[0][0] * r_mats[0][0] * r_mats[-1][1]
        thoughts = [
            T(MCM, l_args),
            T(MCM, r_args),
            T(TernaryMul, (l_mats[0][0], r_mats[0][0], r_mats[-1][1])),
            T(TernaryAdd, (l_cost, r_cost, agg_cost)),
        ]

        cost = l_cost + r_cost + agg_cost
        if min_cost is not None:
            thoughts.append(T(Compare, (cost, min_cost)))
        if min_cost is None or cost < min_cost:
            min_cost = cost
            min_order = l_order, r_order

        if len(r_mats) > 1:
            new_l_mats = l_mats + (r_mats[0],)
            new_r_mats = r_mats[1:]
            thoughts.append(
                T(MCM, ((new_l_mats, new_r_mats), min_order, min_cost), 'tail'))

        return thoughts
```

The following is an example of a three-matrix MCM.

- X^1: GO MCM 3 × 9 , 9 × 4 , 4 × 5 = GO MCM 3 × 9 = 3 × 9 ; 0 STOP GO MCM 9 × 4 , 4 × 5 = 9 × 4 , 4 × 5 ; 1 8 0 STOP GO 3 * 9 * 5 = 1 3 5 STOP GO 0 + 1 8 0 + 1 3 5 = 3 1 5 STOP TAIL MCM 3 × 9 , 9 × 4 | 4 × 5 ACC 3 × 9 , (9 × 4 , 4 × 5) ; 3 1 5 = THINK

- ...

- X^{32}: GO MCM 3 × 9 , 9 × 4 | 4 × 5 ACC 3 × 9 , (9 × 4 , 4 × 5) ; 3 1 5 = GO MCM 3 × 9 , 9 × 4 = 3 × 9 , 9 × 4 ; 1 0 8 STOP GO MCM 4 × 5 = 4 × 5 ; 0 STOP GO 3 * 4 * 5 = 6 0 STOP GO 1 0 8 + 0 + 6 0 = 1 6 8 STOP GO 1 6 8 VS 3 1 5 = LT STOP (3 × 9 , 9 × 4) , 4 × 5 ; 1 6 8 STOP

- ...

H.11   Sorting

Among several sorting algorithms, we choose merge sort for our experiments with CoT and RoT. Note that WT is not relevant to the sorting algorithm since it produces the answer directly. The merge sort algorithm is simple: (i) split the given sequence into two equally sized subsequences, (ii) sort each subsequence, and (iii) merge the two sorted sequences. Since the final merge operation is quite complicated, we define the merge as a problem type.

```python
 1  class Merge(Problem):
 2      @staticmethod
 3      def thought(args) -> list[T]:
 4          l, r = args
 5          if len(l) == 0 or len(r) == 0:
 6              return []
 7
 8          thoughts = [T(Compare, (l[0], r[0]))]
 9          if l[0] < r[0] and len(l) > 1:
10              thoughts.append(T(Merge, (l[1:], r)))
11          elif l[0] >= r[0] and len(r) > 1:
12              thoughts.append(T(Merge, (l, r[1:])))
13          return thoughts
14
15
16  class MergeSort(Problem):
17      @staticmethod
18      def thought(args) -> list[T]:
19          if len(args) < 2:
20              return []
21
22          l_len = (len(args) + 1) // 2
23          l = args[:l_len]
24          r = args[l_len:]
25          return [
26              T(MergeSort, l),
27              T(MergeSort, r),
28              T(Merge, (tuple(sorted(l)), tuple(sorted(r))), 'tail')
29          ]
```

I Fine-Tuning GPT-3 for Recursion of Thought

Using the OpenAI API, we fine-tune GPT-3 for Recursion of Thought. The goal is to learn 16-digit addition, 16-digit subtraction, 8-digit multiplication, and 8-digit division simultaneously. GPT-3's fine-tuning API takes a dataset where each example is a prompt-completion pair in plain text. It is converted to tokens by a special tokenizer for GPT, which we cannot control. This API is not directly compatible with RoT due to several reasons.

- There is no special tokens such as `GO`, `THINK`, and `STOP`.

- The input and target sequences have to be the same. However, they are different in RoT due to the `THINK` token. Once `THINK` is produced, the RoT framework triggers the recursion process to find the subproblem's answer and replace the `THINK` token with it. Therefore, the `THINK` token appears in the target sequences, but never in the input sequences.

Moreover, the way that GPT-3 tokenizes numbers hinders the learning of arithmetic reasoning rules. GPT-3 tokenizes a multi-digit number into a set of two-digit or three-digit numbers. For example, the text `1234567` is converted to the sequence of tokens `123` `45` `67`. Under this tokenization scheme, the relationship between the numbers becomes obscured. As an example, the tokens `7`, `17`, `27`, ..., `997` all have 7 as their last digit. Since there is no direct way for a model to know that they share the same digit, it is crucial to use each digit as a token. We believe that OpenAI needs to correct this tokenization of GPT-3 for numbers.

Luckily, we can mimic the RoT procedures with the API by using several tricks. First, we replace the special tokens with plain lower-case words, e.g., `GO` → go and `STOP` → stop, which are included in the vocabulary of GPT-3. Second, we add a space before each token to make sure that the GPT tokenizer separates each token. We also add space before each digit to prevent the tokenizer from grouping a number into 2-to-3-digit tokens. Finally, to simulate the behavior of the `THINK` and `STOP` tokens, we derive multiple examples from each context, one for each `THINK` or `STOP` output.

As an example, context X^3 in Figure 1 is converted to the following JSON lines for GPT-3 as follows:

X^3	`GO 4 0 + 3 5 =` `GO 0 + 5 = 5 STOP` `GO 4 + 3 = 7 STOP` `7 5 STOP`
Y^3	`PAD ×7` `GO 0 + 5 = THINK PAD` `GO 4 + 3 = THINK PAD` `7 5 STOP`

$$\Downarrow$$

```
1  {"prompt": "_go_4_0_+_3_5_=", "completion": "_go_0_+_5_=_think"}
2  {"prompt": "_go_4_0_+_3_5_=_go_0_+_5_=_5_stop", "completion": "_go_4_+_3_=_think"}
3  {"prompt": "_go_4_0_+_3_5_=_go_0_+_5_=_5_stop_go_4_+_3_=_7_stop", "completion": "_7_5_stop"}
```

In the case of Without Thought (WT), each problem is simply converted into a single example:

X	`GO 4 0 + 3 5 =` `7 5 STOP`
Y	`PAD ×7` `7 5 STOP`

$$\Downarrow$$

```
1  {"prompt": "_go_4_0_+_3_5_=", "completion": "_7_5_stop"}
```

In both cases of RoT and WT, we fine-tune GPT-3 for 10K steps with a batch size of 256. Among the several variants of GPT-3, we use Ada which is offered at the lowest cost. Note that RoT produces multiple contexts for each problem, and each RoT context is converted to multiple training examples. For this reason, the GPT-3 fine-tuned for RoT encounters much fewer problems during training, although the numbers of training steps are the same.

J Training Details of the Tiny Models

In all experiments, we use a batch size of 256 and Adam optimizer (Kingma and Ba, 2015) with a learning rate of 0.001, i.e., the default learning rate in PyTorch. We train the Transformers for 500K steps and decay the learning rate by half every 50K steps. Since the LSTMs converge slower than the Transformers, we train them for 800K steps and decay the learning rate by half every 100K steps. At every 20K steps, we evaluate the model on a test set of 30K problems, and if a model reaches a perfect accuracy of 1.0, we do not train the model further. The models can be trained on a single GPU with 12GB memory.

K Efficient Evaluation of RoT

At the problem scales that RoT is tested, solving a single problem can require hundreds of thousands of tokens. Therefore, we have to develop an efficient evaluation algorithm. Since the models are deterministic, we do not need to evaluate the same context multiple times. Therefore, given a test set, we first find all unique contexts that should appear while solving the test set. Note that there would be a lot of duplicate contexts, especially for the base cases. These unique contexts can be evaluated in parallel. We evaluate each context in a similar manner to the training. For each context, we construct the target sequence, as we described in §3.2, and check if the model outputs the target sequence correctly, given the GT context. In the case of GPT-3, we convert the contexts and target sequences as described in Appendix I to feed into the API. Once every context is individually evaluated, the next step is to aggregate the results. In this aggregation stage, a (sub)problem is marked correct only if its subproblems are all marked correct. This way, we efficiently evaluate RoT with minimal cost.

L Failure Case Analysis of GPT-3 with RoT

In this section, we provide some failure cases of RoT in the GPT-3 experiments for further insights. Although the model achieves an accuracy of over 99% in almost every case, there are several problems where it makes mistakes. Nonetheless, only a few characters are wrong in each context.

The most common type of error is repeating the same character a wrong number of times. The following is the examples of this error type.

Prompt	go 1 6 4 3 4 vs 1 0 1 1 6 = go 1 vs 1 = eq stop
Completion	go 6 4 3 4 vs 1 1 6 = think
Model Output	go 6 4 3 4 vs 1 1 1 6 = think

Prompt	go 4 8 0 9 4 4 9 5 1 8 2 5 5 9 1 * 4 4 3 0 2 5 3 6 0 1 7 = go 4 8 0 9 4 4 9 5 1 8 2 5 5 9 1 * 7 = 3 3 6 6 6 1 4 6 6 2 7 7 9 1 3 7 stop go 4 8 0 9 4 4 9 5 1 8 2 5 5 9 1 * 4 4 3 0 2 5 3 6 0 1 = 2 1 3 0 7 0 8 1 0 4 7 0 8 0 9 6 0 5 1 7 0 3 1 9 1 stop
Completion	tail 2 1 3 0 7 0 8 1 0 4 7 0 8 0 9 6 0 5 1 7 0 3 1 9 1 0 + 3 3 6 6 6 1 4 6 6 2 7 7 9 1 3 7 = think
Model Output	tail 2 1 3 0 7 0 8 1 0 4 7 0 8 0 9 6 0 5 1 7 0 3 1 9 1 0 + 3 3 6 6 6 6 1 4 6 6 2 7 7 9 1 3 7 = think

Prompt	go 1 0 5 4 6 2 0 * 1 5 = go 1 0 5 4 6 2 0 * 5 = 5 2 7 3 1 0 0 stop go 1 0 5 4 6 2 0 * 1 = 1 0 5 4 6 2 0 stop
Completion	tail 1 0 5 4 6 2 0 0 + 5 2 7 3 1 0 0 = think
Model Output	tail 1 0 5 4 6 2 0 0 0 + 5 2 7 3 1 0 0 = think

Another common case is incorrectly counting the number of digits while comparing two numbers. When comparing numbers following our recursive procedure, the model has to immediately return the answer if the numbers of digits are different. In the following example, the first number has seven digits, and the second one has six digits.

Prompt	go 1 0 6 9 0 3 4 vs 8 9 1 5 0 6 =
Completion	gt stop
Model Output	go 1 vs 8

Although the model should directly output the answer, i.e., `gt stop`, it decides that the numbers of digits are the same and moves on to the next step, which is to compare the most significant digits. This error is caused by the specific recursive procedure of our choice. Since we assume that the model would easily figure out the difference in the number of digits, we did not add explicit steps to count the digits. Although the model detects the difference in most cases, it turns out to be a relatively challenging operation. A possible solution is to teach the model to explicitly count the number of digits and compare them.

M   Context Length Distribution

In Figure 4, we present the distributions of context lengths for each problem type. We compare the context lengths of RoT and CoT. For each configuration, we randomly sample 10K contexts from the training distribution and plot the histogram of their lengths. The graphs show that the context sizes of CoT are many orders of magnitude larger than RoT. In theory, the total number of tokens to generate for each problem is identical in both RoT and CoT (if we do not count the THINK tokens). However, RoT's context sizes are much smaller since it utilizes multiple contexts.

Another advantage of RoT is the utilization of dynamic programming. Since we can easily cache the duplicate computations of RoT as explained in Appendix K, we can drastically reduce the amount of token generation if there is a redundant structure in the problem. The amount of tokens to generate for each problem is plotted in Figure 5. The benefit is especially prominent in algorithmic problems. For example, finding the LCS of two 32-digit sequences results in more than 10^{18} tokens if we naively use CoT or RoT. If we use dynamic programming with RoT, we can efficiently solve the same problem with much less cost.
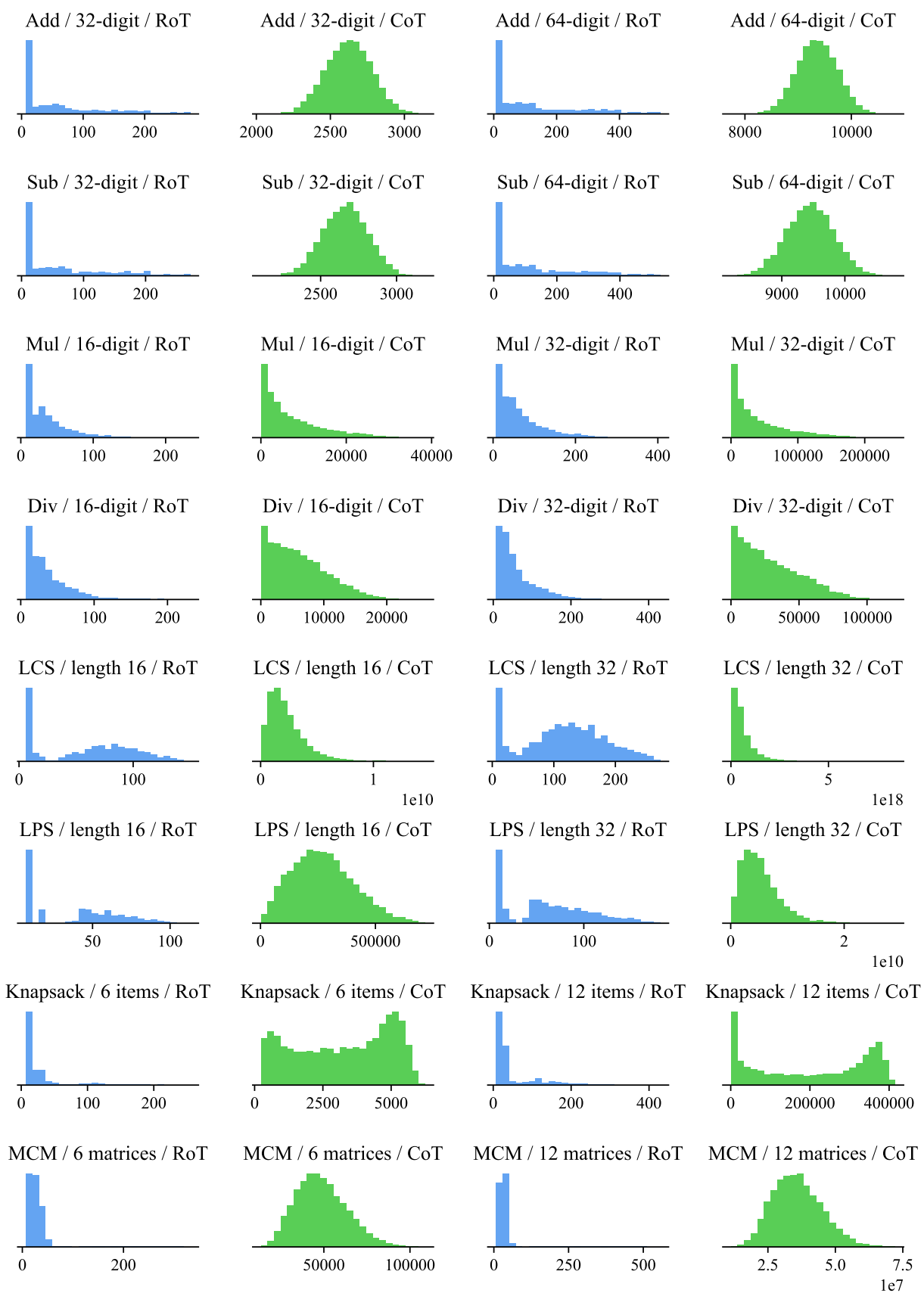
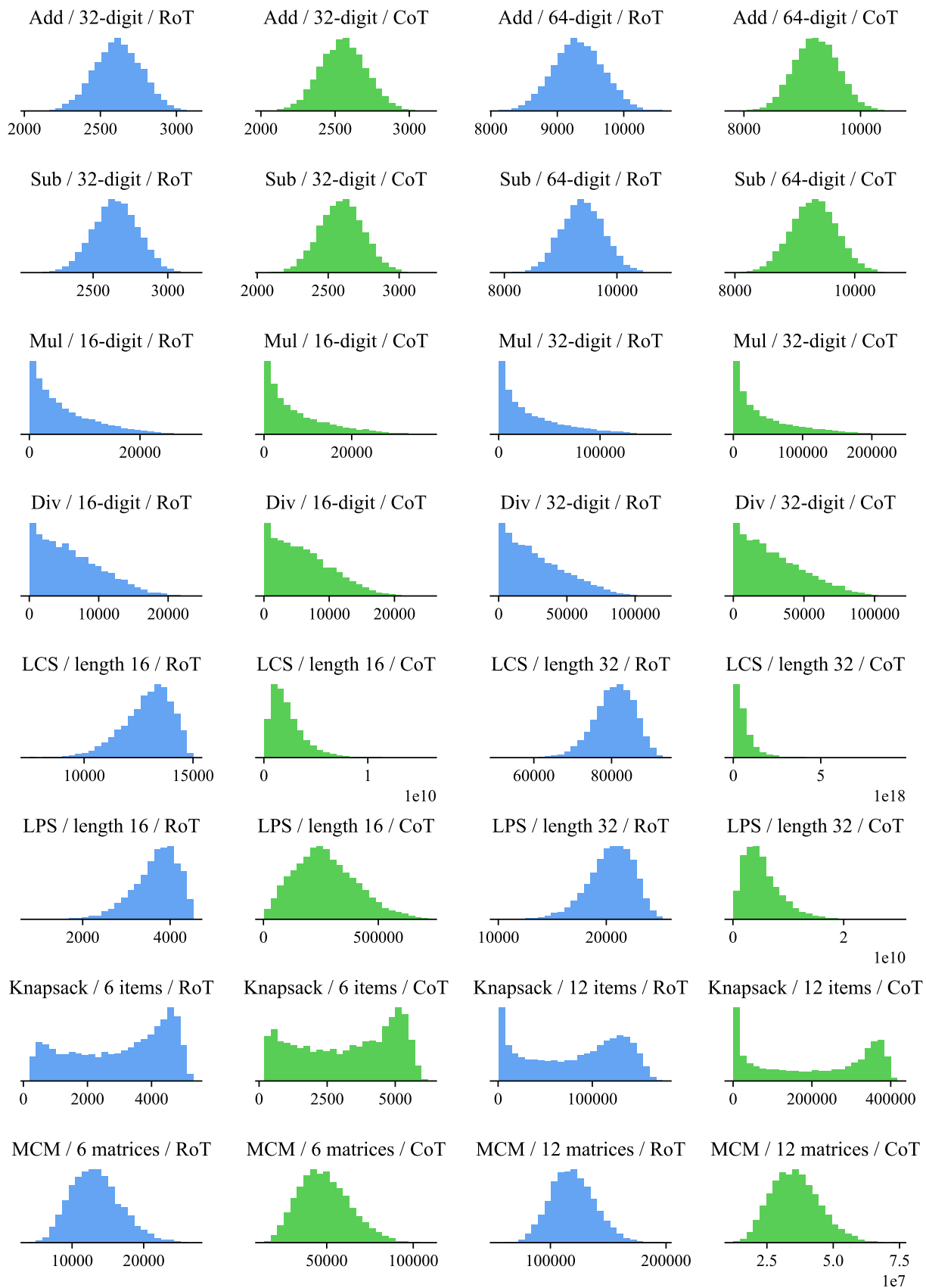Figure 4: The distributions of context lengths.

Figure 5: The distribution of the total number of tokens to produce in order to solve each problem. RoT can utilize dynamic programming to reduce redundant computations.
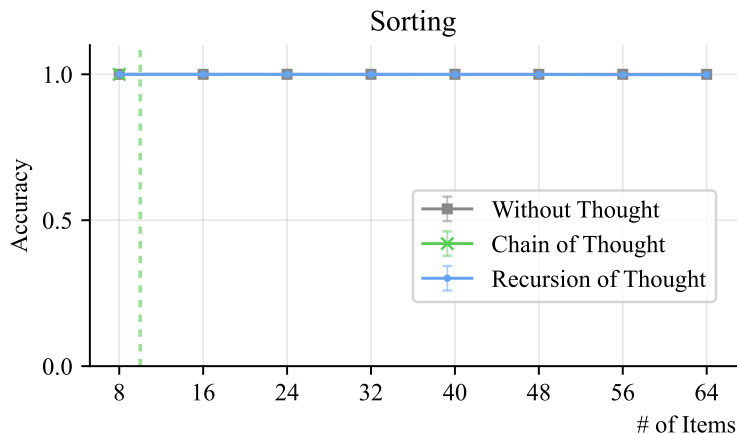
Figure 6: Sorting experiment with the tiny Transformer.

Problem	Difficulty	WT	CoT	RoT
Addition	32-digit	0.991	–	0.998
	48-digit	0.853	–	0.995
Subtraction	32-digit	0.991	–	0.998
	48-digit	0.886	–	0.998
Multiplication	8-digit	0.337	–	0.999
	16-digit	0.098	–	0.994
Division	8-digit	0.363	–	1.000
	16-digit	0.123	–	0.989
LCS	length 16	0.980	–	0.995
	length 24	0.832	–	0.998
LPS	length 24	0.995	–	1.000
	length 40	0.800	–	0.974
0-1 Knapsack	4 items	0.945	–	0.999
	6 items	0.634	–	1.000
MCM	3 matrices	0.481	–	0.997
	4 matrices	0.110	–	0.992

Table 2: The exact values of the GPT-3 experiments in Figure 2a.

N   Transformers Are Powerful Sorting Machines

In fact, the first algorithmic task that we tested is sorting since it has been widely used as a benchmark for algorithmic reasoning (Reed and de Freitas, 2016; Cai et al., 2017; Pierrot et al., 2019). However, we find that Transformers are incredibly good at sorting, even in the WT setting. Figure 6 shows the sorting experiment. For CoT and RoT, we train the merge sort algorithm. Interestingly, WT easily achieves a perfect score in sorting 64 three-digit numbers. Also, the training converges much faster than RoT. The Transformer architecture, more specifically the attention mechanism, seems to be perfectly suited for the sorting operation.

O   The Exact Values of Figure 2

Table 2-5 show the exact values of the graphs in Figure 2. Except for the GPT-3 experiments in Table 2, we report the average and the standard deviation of eight runs. Each GPT-3 experiment is done only once.

Problem	Difficulty	WT	CoT	RoT
Addition	8-digit	0.863 ± 0.265	1.000 ± 0.000	1.000 ± 0.000
	16-digit	0.370 ± 0.475	1.000 ± 0.000	1.000 ± 0.000
	24-digit	0.336 ± 0.430	1.000 ± 0.000	1.000 ± 0.000
	32-digit	0.455 ± 0.458	−	1.000 ± 0.000
	40-digit	0.119 ± 0.316	−	1.000 ± 0.000
	48-digit	0.082 ± 0.216	−	1.000 ± 0.000
	56-digit	0.105 ± 0.277	−	1.000 ± 0.000
	64-digit	0.000 ± 0.000	−	1.000 ± 0.001
Subtraction	8-digit	0.982 ± 0.006	1.000 ± 0.000	1.000 ± 0.000
	16-digit	0.705 ± 0.411	1.000 ± 0.000	1.000 ± 0.000
	24-digit	0.238 ± 0.412	1.000 ± 0.000	1.000 ± 0.000
	32-digit	0.221 ± 0.385	−	1.000 ± 0.000
	40-digit	0.426 ± 0.433	−	1.000 ± 0.000
	48-digit	0.114 ± 0.303	−	1.000 ± 0.000
	56-digit	0.116 ± 0.307	−	1.000 ± 0.000
	64-digit	0.161 ± 0.282	−	1.000 ± 0.000
Multiplication	2-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	4-digit	0.817 ± 0.023	1.000 ± 0.000	1.000 ± 0.000
	8-digit	0.340 ± 0.032	−	1.000 ± 0.000
	12-digit	0.169 ± 0.015	−	1.000 ± 0.000
	16-digit	0.104 ± 0.016	−	1.000 ± 0.000
	20-digit	0.048 ± 0.020	−	1.000 ± 0.000
	24-digit	0.033 ± 0.017	−	0.999 ± 0.001
	28-digit	0.014 ± 0.006	−	0.999 ± 0.001
	32-digit	0.012 ± 0.001	−	0.999 ± 0.000
Division	2-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	4-digit	0.978 ± 0.008	1.000 ± 0.000	1.000 ± 0.000
	8-digit	0.354 ± 0.029	−	1.000 ± 0.000
	12-digit	0.186 ± 0.009	−	1.000 ± 0.000
	16-digit	0.128 ± 0.011	−	1.000 ± 0.000
	20-digit	0.087 ± 0.012	−	1.000 ± 0.000
	24-digit	0.075 ± 0.005	−	1.000 ± 0.000
	28-digit	0.059 ± 0.007	−	0.999 ± 0.000
	32-digit	0.048 ± 0.008	−	0.999 ± 0.000

Table 3: The exact values of the Transformer experiments in Figure 2b (arithmetic problems).

Problem	Difficulty	WT	CoT	RoT
LCS	length 3	1.000 ± 0.000	1.000 ± 0.000	–
	length 4	0.997 ± 0.008	–	1.000 ± 0.000
	length 8	0.999 ± 0.002	–	1.000 ± 0.000
	length 12	0.965 ± 0.025	–	1.000 ± 0.000
	length 16	0.880 ± 0.035	–	1.000 ± 0.000
	length 20	0.759 ± 0.043	–	1.000 ± 0.000
	length 24	0.622 ± 0.038	–	1.000 ± 0.000
	length 28	0.484 ± 0.043	–	0.999 ± 0.000
	length 32	0.375 ± 0.030	–	0.999 ± 0.000
LPS	length 4	1.000 ± 0.000	1.000 ± 0.000	–
	length 7	1.000 ± 0.000	1.000 ± 0.000	–
	length 8	1.000 ± 0.000	–	1.000 ± 0.000
	length 16	0.999 ± 0.001	–	1.000 ± 0.000
	length 24	0.950 ± 0.019	–	1.000 ± 0.000
	length 32	0.788 ± 0.019	–	1.000 ± 0.000
	length 40	0.608 ± 0.023	–	1.000 ± 0.000
	length 48	0.477 ± 0.030	–	0.999 ± 0.001
	length 56	0.365 ± 0.029	–	0.998 ± 0.000
0-1 Knapsack	2 items	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	4 items	0.966 ± 0.006	1.000 ± 0.000	1.000 ± 0.000
	6 items	0.849 ± 0.007	–	1.000 ± 0.000
	8 items	0.640 ± 0.242	–	1.000 ± 0.000
	10 items	0.481 ± 0.279	–	1.000 ± 0.000
	12 items	0.435 ± 0.252	–	0.988 ± 0.029
MCM	2 matrices	0.973 ± 0.009	1.000 ± 0.000	1.000 ± 0.000
	4 matrices	0.177 ± 0.069	–	1.000 ± 0.000
	6 matrices	0.088 ± 0.029	–	1.000 ± 0.000
	8 matrices	0.033 ± 0.025	–	1.000 ± 0.000
	10 matrices	0.051 ± 0.032	–	0.998 ± 0.001
	12 matrices	0.026 ± 0.011	–	0.996 ± 0.002

Table 4: The exact values of the Transformer experiments in Figure 2b (algorithmic problems).

Problem	Difficulty	WT	CoT	RoT
Addition	2-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	4-digit	0.642 ± 0.305	1.000 ± 0.001	1.000 ± 0.000
	6-digit	0.005 ± 0.008	0.997 ± 0.005	0.999 ± 0.000
	8-digit	0.000 ± 0.000	0.905 ± 0.155	0.999 ± 0.001
	10-digit	0.000 ± 0.000	0.795 ± 0.341	0.986 ± 0.024
	12-digit	0.000 ± 0.000	−	0.871 ± 0.275
	14-digit	0.000 ± 0.000	−	0.358 ± 0.430
	16-digit	0.000 ± 0.000	−	0.120 ± 0.202
Subtraction	2-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	4-digit	0.776 ± 0.179	1.000 ± 0.000	1.000 ± 0.000
	6-digit	0.006 ± 0.001	1.000 ± 0.000	1.000 ± 0.000
	8-digit	0.000 ± 0.000	0.896 ± 0.252	0.994 ± 0.016
	10-digit	0.000 ± 0.000	0.443 ± 0.377	0.908 ± 0.236
	12-digit	0.000 ± 0.000	−	0.507 ± 0.398
	14-digit	0.000 ± 0.000	−	0.295 ± 0.406
	16-digit	0.000 ± 0.000	−	0.101 ± 0.137
Multiplication	2-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	3-digit	0.855 ± 0.044	−	1.000 ± 0.000
	4-digit	0.636 ± 0.061	−	1.000 ± 0.000
	5-digit	0.338 ± 0.063	−	1.000 ± 0.000
	6-digit	0.270 ± 0.030	−	0.987 ± 0.008
	7-digit	0.162 ± 0.025	−	0.896 ± 0.105
	8-digit	0.138 ± 0.025	−	0.670 ± 0.208
Division	1-digit	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
	2-digit	1.000 ± 0.000	−	1.000 ± 0.000
	3-digit	1.000 ± 0.001	−	1.000 ± 0.000
	4-digit	0.891 ± 0.072	−	1.000 ± 0.000
	5-digit	0.516 ± 0.077	−	0.998 ± 0.004
	6-digit	0.308 ± 0.069	−	0.996 ± 0.007
	7-digit	0.192 ± 0.028	−	0.958 ± 0.036
	8-digit	0.115 ± 0.015	−	0.914 ± 0.090

Table 5: The exact values of the LSTM experiments in Figure 2c.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
Limitations

☐ A2. Did you discuss any potential risks of your work?
Not applicable. Left blank.

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Abstract & Section 1

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

We include the source code used in our experiments.

☒ B1. Did you cite the creators of artifacts you used?
We did not use any external artifcats.

☒ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
We did not use any external artifcats.

☐ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Not applicable. Left blank.

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.

☑ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
We provide documentation in README.md inside the source code.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
Appendix C, G, H

C ☑ Did you run computational experiments?

Section 4

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
Section 4, Appendix I, J

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
Appendix J

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Section 4, Appendix O

☐ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
Not applicable. Left blank.

D   ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
No response.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
No response.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
No response.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
No response.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
No response.