Teaching Algorithmic Reasoning via In-context Learning

Hattie Zhou[1*], **Azade Nova**[2], **Hugo Larochelle**[2], **Aaron Courville**[1], **Behnam Neyshabur**[2†], **and Hanie Sedghi**[2†]

[1]**Mila, Université de Montreal**
[2]**Google Research**

Abstract

Large language models (LLMs) have shown increasing in-context learning capabilities through scaling up model and data size. Despite this progress, LLMs are still unable to solve algorithmic reasoning problems. While providing a rationale with the final answer has led to further improvements in multi-step reasoning problems, Anil et al. (2022) showed that even simple algorithmic reasoning tasks such as parity are far from solved. In this work, we identify and study four key stages for successfully teaching algorithmic reasoning to LLMs: (1) formulating algorithms as skills, (2) teaching multiple skills simultaneously (skill accumulation), (3) teaching how to combine skills (skill composition) and (4) teaching how to use skills as tools. We show that it is possible to teach algorithmic reasoning to LLMs via in-context learning, which we refer to as *algorithmic prompting*. We evaluate our approach on a variety of arithmetic and quantitative reasoning tasks, and demonstrate significant boosts in performance over existing prompting techniques. In particular, for long parity, addition, multiplication and subtraction, we achieve an error reduction of approximately 10x, 9x, 5x and 2x respectively compared to the best available baselines.

1 Introduction

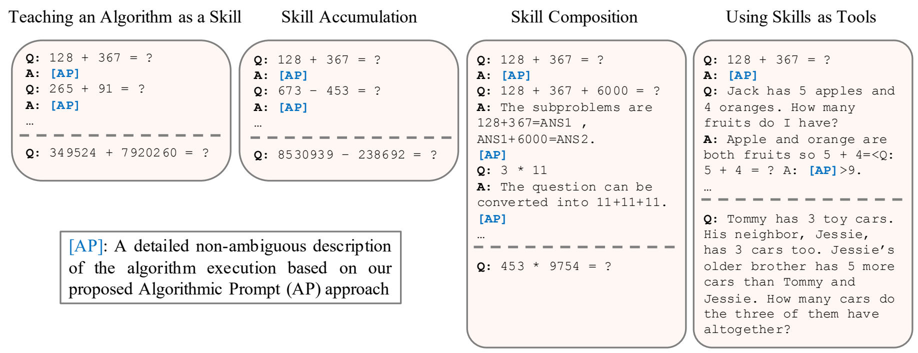
Large language models (LLMs) have shown impressive progress in recent years, driven by the scaling up of models and training data sizes (Kaplan et al., 2020; Wei et al., 2022a; Hoffmann et al., 2022). Despite these advances, LLMs still struggle to perform complex reasoning tasks. In order to generalize in and out-of-distribution (OOD) on many of these reasoning tasks, the model needs to learn the underlying algorithm for solving a task. We refer to this behavior as *algorithmic reasoning* (Kaiser & Sutskever, 2015; Veličković & Blundell, 2021). Since algorithms are input independent by nature, they guarantee good OOD performance when executed properly. In this work, we study how to teach algorithmic reasoning to LLMs via in-context learning. In-context learning (Brown et al., 2020) refers to the ability to learn a task from a few examples being presented within a *prompt*, and provides a powerful platform for specialized skill acquisition without losing the generality of the underlying model. Various prompting strategies have demonstrated significant potential in solving certain types of reasoning problems (Jung et al., 2022; Zhou et al., 2022; Wei et al., 2022b; Kojima et al., 2022).

We identify and explore four key stages for teaching algorithms as skills to LLMs (Figure 1). This setup is reminiscent of how similar skills are taught to children in school. We begin by studying the shortcomings of existing approaches and propose ways to alleviate them. We focus

[*]Work done while interning at Google Research

[†]Equal advising

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on MATH-AI.



```
Q: 128 + 367 = ?
A: [AP]
Q: 265 + 91 = ?
A: [AP]
…
- - - - - - - - - - - -
Q: 349524 + 7920260 = ?
```

```
Q: 128 + 367 = ?
A: [AP]
Q: 673 - 453 = ?
A: [AP]
…
- - - - - - - - - - - -
Q: 8530939 - 238692 = ?
```

```
Q: 128 + 367 = ?
A: [AP]
Q: 128 + 367 + 6000 = ?
A: The subproblems are
128+367=ANS1 ,
ANS1+6000=ANS2.
[AP]
Q: 3 * 11
A: The question can be
converted into 11+11+11.
[AP]
…
- - - - - - - - - - - -
Q: 453 * 9754 = ?
```

```
Q: 128 + 367 = ?
A: [AP]
Q: Jack has 5 apples and
4 oranges. How many
fruits do I have?
A: Apple and orange are
both fruits so 5 + 4=<Q:
5 + 4 = ? A: [AP]>9.
…
- - - - - - - - - - - -
Q: Tommy has 3 toy cars.
His neighbor, Jessie,
has 3 cars too. Jessie's
older brother has 5 more
cars than Tommy and
Jessie. How many cars do
the three of them have
altogether?
```

[AP]: A detailed non-ambiguous description of the algorithm execution based on our proposed Algorithmic Prompt (AP) approach

Figure 1: The four learning stages investigated in this work (from left to right): (i) Teaching an algorithm as a skill (Section 3) (ii) Skill Accumulation, i.e., teaching multiple skills simultaneously (Section 4) (iii) Skill Composition, i.e. the ability to learn a complex skill through building upon simpler ones (Section 5) (iv) Using Skills as Tools to solve problems (Section 6).

on arithmetic algorithms such as addition, subtraction and multiplication as they have been widely benchmarked (Saxton et al., 2019; Hendrycks et al., 2021) and famously fail at out-of-distribution generalization even for best performing models on the MATH benchmark (Lewkowycz et al., 2022). While one can avoid learning these algorithms by the use of external tools (Cobbe et al., 2021), this approach cannot scale to higher levels of abstraction where a model needs to use "soft algorithms" and certain steps must be flexibly applied in different situations.

2 Algorithmic prompting

Nye et al. (2021) proposed the idea of a *scratchpad*, where the model is trained on and outputs the intermediate computations used in solving a task, and showed that it can greatly improve performance on multi-step computation problems. This was taken further by Wei et al. (2022b) to the in-context learning setting, where they showed that providing rationales in the prompts significantly increases the model's ability to solve similar reasoning problems. They refer to this approach as *chain-of-thought*. We hypothesize that these existing methods fail to sufficiently constrain the model's interpretation of the prompting information, and result in unexpected and undesirable model behaviors on tasks that require precise algorithmic reasoning. Thus, we push the limits of rationale-based prompting by drastically increasing the amount of detail included in the rationales, while specifying the steps of an algorithm within this information. We refer to this strategy as *algorithmic prompting*, and evaluate its performance in the context of the four capabilities identified in Figure 1.

As a motivating example, consider the carry calculation in the simple addition algorithm. If the model only sees a few examples of digit sums and their respective carries (e.g. $8 + 7$ has carry of 1, $2 + 6$ has carry of 0), the model could conclude different rules of carry that would explain the training examples but do not generalize to new situations. However, by explicitly providing the rules of carry in the prompt, such as including the step $C = (8 + 7 - ((8 + 7)\%10))/10 = (15 - 5)/10 = 1$, we can remove ambiguity and increase the probability that the model will adhere to these patterns.

3 Teaching algorithms as skills

For all the experiments in this section and in the rest of the paper, we use the Codex model (Chen et al., 2021) `code-davinci-002` from OpenAI. This model has a maximum context length of 8000 tokens. Task examples are sampled uniformly at each length. All results are sampled once using a temperature of 0 and default settings for other hyperparameters.

Two-number addition: We begin our analysis by studying the two-number addition task and exploring the effectiveness of various prompting strategies. We present an algorithmic prompt for addition and compare its performance against the few-shot, chain-of-thought, instruction-only, and scratchpad methods. An illustration of these prompting strategies for addition is shown in Figure 4, and the prompt used for instruction-only is shown in Section B.2. For all addition experiments, we restrict prompt examples to answers up to 5 digits in length, and evaluate on questions up to 19 digits

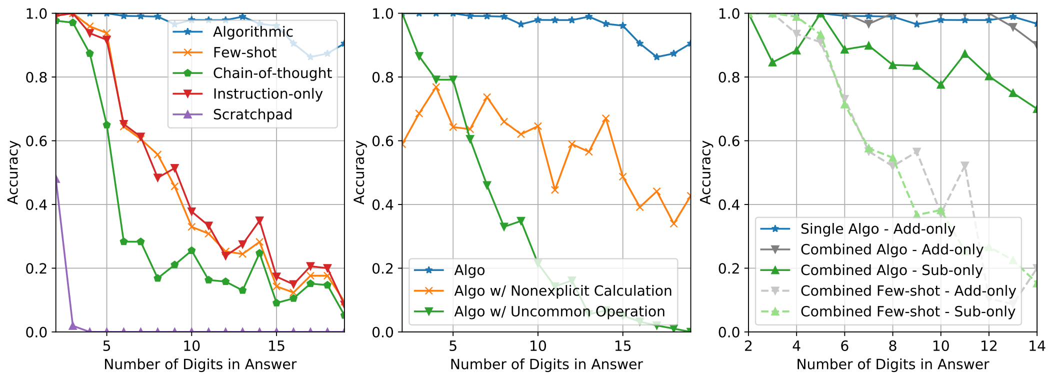


Figure 2: We use "Algo" to refer to the algorithmic prompt. **Left:** Accuracy on addition questions using various prompting strategies. The algorithmic prompt significantly outperforms other baselines and achieves superior length generalization. **Middle:** Addition accuracy of the algorithmic prompt and its two less-detailed variants. We see the necessity of non-ambiguous explanations within the algorithmic prompt. **Right:** Accuracy on addition and subtraction questions using a combined prompt. There is minimal degradation in performance for addition-only questions using the combined prompt compared to the single algorithm addition-only prompt.

in length[3]. The dataset contains 2000 randomly sampled questions, uniformly across the length of the final answer. Similar settings are employed for all other experiments in this paper.

Figure 2 shows the performance of the algorithmic prompt against existing methods. We see that algorithmic prompt achieves near perfect performance and OOD generalization on addition, while existing methods have decreasing performance as the length of the answer increases. Interestingly, chain-of-thought and scratchpad prompts did not improve performance over simple few-shot, suggesting that insufficient rationales may even hinder the model's natural abilities.

We further compare the algorithmic prompt to two less-detailed variants. One version (*nonexplicit calculation*) omits the explicit equation showing how the carry value is derived. The second version (*uncommon operation*) requires the model to index the correct digit for a given step. The indexing of a digit at a variable position is a more uncommon operation than the indexing of the digit at the same position each time. In our final addition prompt, we introduce a mechanism that allows the model to avoid the indexing operation by copying the unprocessed digits over to each step and always taking the last digit. Figure 2 illustrates the relative gains that come from the disambiguation of these two aspects of the algorithm.

In Section A.3, we show that algorithmic prompting leads to similar performance boosts for subtraction, multiplication, and parity, illustrating that the results are not specific to the addition problem.

4 Skill accumulation

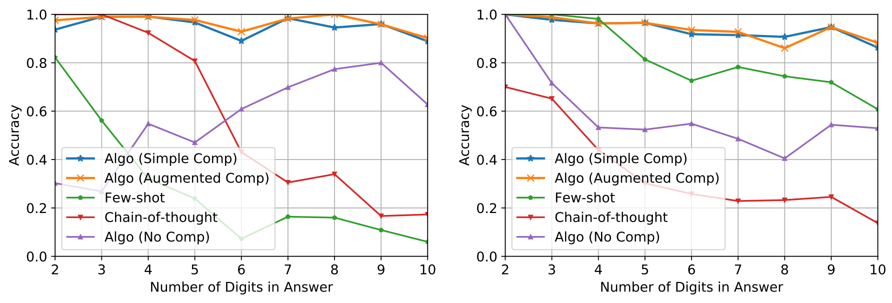
So far we have demonstrated the ability to teach single-algorithms through in-context learning. In this section, we study the model's ability to simultaneously learn multiple algorithms and choose the applicable one when solving problems. To do so, we teach both addition and subtraction algorithms within the same prompt and see how the performance compares to the single-algorithm prompt. For a discussion on the differences between the addition and subtraction algorithm, see Section A.3.

To succeed at this task, the model needs to demonstrate the ability to follow different processing paths when the question is addition or subtraction. In Figure 2, we see that the model is able to effectively execute the correct algorithm based on the individual questions. Comparing the performance on addition-only questions to the addition prompt from Section 3, we see that there is minimal change in performance despite having an extra other algorithm present in the prompt.

5 Skill composition

In this section, we explore the model's ability to learn multiple algorithms that build on top of each other. This is a desirable property because it enables the model to learn a more complex algorithm

[3]The length of 19 is chosen because this is the level after which the model begins to run out of context.



(a) Multi-number addition (b) Multiplication-as-addition

Figure 3: Performance on compositions of skills. "Algo" indicates algorithmic prompting. "Simple Comp" refers to a simple composition strategy where previously taught algorithms are transferred as is. "Augmented Comp" adjusts the previously taught algorithm to match the format of the new task. This simulates a version where the full prompt specializes to the new task. "No Comp" uses only the part of the "Simple Comp" prompt that describes the new task. This simulates the comparison to learning a new skill from scratch without first learning its stepping stones. We observe that the composed algorithmic templates demonstrate better generalization than the baseline methods. Note that for multiplication, we evaluate the algorithmic methods on a harder task than the few-shot baselines, since we force the model to convert the question into the addition of a number n times, while for other baselines we simply perform $1 \times n$-digit multiplication directly.

without having to relearn simpler sub-components of that algorithm. This setting allows for the modularization of complex algorithms. To establish a framework for skill composition, we explore two extensions to the addition algorithm: 1) adding multiple numbers together, and 2) solving multiplication as an addition problem (e.g. by converting $3 * 7$ into $7 + 7 + 7$). The ability to add multiple numbers builds on top of the ability to add two numbers together, which we already established. Solving multiplication as addition further builds on the addition of multiple numbers. An illustration can be found in Figure 9.

In-context skill composition is limited by the context length of current models. Unlike the previous experimental results, these composition tasks include a number of questions that were incomplete for the algorithmic prompt. To separate out the issue of context length from the ability of the model to follow an algorithm, we report performance in Figure 3 on only the questions for which the algorithmic prompt could fit into context, and use these examples for all baselines. Figure 3 shows that the algorithmic prompt significantly outperforms few-shot and chain-of-thought baselines. Moreover, we observe that there is minimal difference between the simple composition and augmented composition strategies, and that the no composition approach performs much worse than its composed counterparts. In Section A.4, we discuss a strategy to move past the context length limitations and evaluate performance inclusive of questions that exceed the context length limit.

6 Using skills as tools

In this section, we study the behavior of the model when using a given algorithm as a step in solving a larger reasoning problem. We leverage the grade school math word dataset (GSM8k), which consists of high-quality mathematical reasoning problems presented as natural language questions (Cobbe et al., 2021)[4]. Solving GSM8k requires a logical reasoning aspect and an arithmetic computation aspect. To study the ability to use the addition algorithm while solving GSM8k questions, we filter for a subset of GSM8k whose solutions consist of only addition steps, and randomly select 50 examples from this subset to create a hard version that contains large numbers in the questions[5].

We introduce a way to side-step context-length limitations through a dialogue-like interaction between models loaded with different prompts. In this case, we teach the model doing the logical reasoning steps to call on another model when it needs to perform addition calculations. This involves teaching the model to output specific tokens to indicate when a separate model should be consulted. Then, we send the question to the second model loaded with the addition prompt, which executes the algorithm and returns the answer back to the first model. The first model would then continue with the rest of the answer. For an illustration of this, see Figure 12. The performance on the GSM8k-Hard

[4]An example of an GSM8k question is shown in Figure 11.

[5]An example of an GSM8k-Hard question and its augmented chain-of-thought solution is shown in Figure 12.

Table 1: Performance on GSM8k-Hard addition dataset with or without algorithmic tool use. We see that the overall performance is doubled when we call on a second model loaded with the algorithmic addition prompt to perform addition calculations, demonstrating the potential of leveraging in-context algorithmic skills as a form of tool-use. Moreover, we observe that this performance gain comes directly from more accurate addition accuracy. Finally, we identify the issue of interference caused by the use of specific tokens in the chain-of-thought reasoning output, which is shown through the decreased logical accuracy in the method with algorithmic calls.

Method	**Overall Accuracy**	**Logical Accuracy**	**Addition Accuracy**
Chain-of-thought w/ Algo call	55.8%	57.7%	98.4%
Chain-of-thought wo/ Algo call	27.4%	70.6%	61.9%

dataset is shown in Table 1. Surprisingly, we find that having tokens indicating when to call on the second model leads to interference with the logical reasoning abilities of the first model. We conjecture that the use of tokens disrupts the flow of the natural language rationale, and that the informal mathematical reasoning abilities of the model is sensitive to such disruptions. Thus, we suspect that finetuning with the tokens in the answer can alleviate this interference issue. Nonetheless, despite the decrease in logical accuracy, the method that leverages the algorithmic tool use still achieves double the accuracy as the baseline method.

References

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. *arXiv preprint arXiv:2207.04901*, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. Maieutic prompting: Logically consistent reasoning with recursive explanations, 2022.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. 2022. URL `https://openreview.net/forum?id=6p3AuaHAFiN`.

Aitor Lewkowycz, Anders Andreassen, David Martin Dohan, Ethan S Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. 2022. URL `https://arxiv.org/abs/2206.14858`.

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

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557*, 2019.

Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022a.

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

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

A Appendix

A.1 Algorithmic prompting

Table 2: Comparison of different prompting strategies studied in this work. The number of ★ indicates the level to which each strategy exhibit the given characteristic. We see that algorithmic prompt includes both qualities of natural language explanation and explicit intermediate computations.

Prompt Strategy	**Input-target pairs**	**Natural language rationale**	**Intermediate computations**	**Multiple rationales possible**
Few-shot	★★★	-	-	-
Chain-of-thought	★★★	★★★	★	★★★
Scratchpad	★★★	-	★★	-
Instruction-only	-	★★★	-	★★★
Algorithmic	★★★	★★	★★★	★

A.2 Two-number addition

Baselines: We mainly compare the proposed algorithmic prompt to few-shot and chain-of-thought baselines in our experiments. The *few-shot* baseline refers to the simple approach of presenting examples of question and answer pairs with no additional explanation. The *chain-of-thought* baseline provides a rationale along with the final answer in the few-shot examples. In order to generate the rationale for various tasks, we follow the method introduced in Kojima et al. (2022) and use the phrase "let's think step by step" to get a model-generated rationale for the few-shot examples.

```
Input:
128+367
Target:
<scratch>
1 2 8 + 3 6 7 , C: 0
1 2 + 3 6 , 5 C: 1
1 + 3 , 9 5 C: 0
, 4 9 5 C: 0
4 9 5
</scratch>4 9 5.
```

Scratchpad

```
Input:
128+367
Target:
<scratch>
[1,2,8] has 3 digits.
[3,6,7] has 3 digits.
[1,2,8] + [3,6,7] , C=0 , 8+7+0=15 , A->5 , C->1
[1,2] + [3,6] , A=[5] , C=1 , 2+6+1=9 , A->9 , C->0
[1] + [3] , A=[9,5] , C=0 , 1+3+0=4 , A->4 , C->0
[] + [] , A=[4,9,5] , C=0 , END
</scratch>
4 9 5
```

Detailed scratchpad w/
comma-delim

```
Problem: 128+367=?
Explanation:

Let's think step by
step.

128+367=128+300+67=428+6
7=495.

The final Answer is 495.
```

Chain-of-thought

```
Problem: 128+367=
...
Length of FN is 3. FN=[1,2,8]. Length of SN is 3.
SN=[3,6,7]. FN[3]=8. SN[3]=7. C[3]=0. Since 8+7+0=15,
15%10=5. Length of A is 1. Thus A=[5]. Since (15-5)/10=1,
C[2]=1.
Length of FN is 2. FN=[1,2]. Length of SN is 2. SN=[3,6].
FN[2]=2. SN[2]=6. C[2]=1. Since 2+6+1=9, 9%10=9. Length of
A is 2. Thus A=[9,5]. Since (9-9)/10=0, C[1]=0.
...
There are no more digits and C[0]=0. Thus the process is
complete. The final Answer is [4,9,5].
```

Algorithmic prompt

Figure 4: Examples of the two-number addition prompt using different techniques.

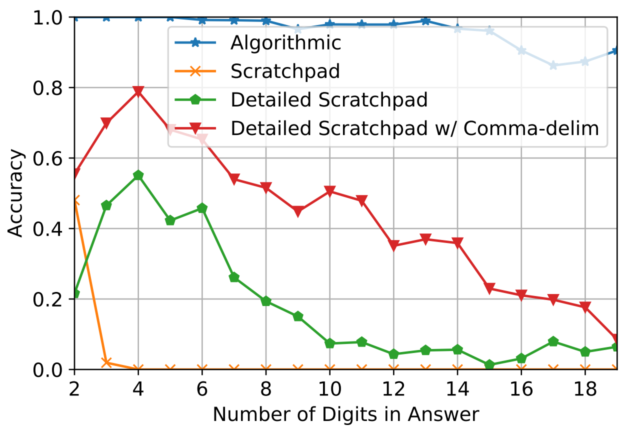


Figure 5: Accuracy on addition questions of increasing length for different versions of scratchpad prompting. The number of digits in answer plotted in the x-axis refers to the length of c for questions of the form $a + b = c$. Accuracy is measured over 2000 total examples sampled uniformly over the length of c. The max length for examples in the prompt is 5. Using scratchpad-style output as a prompt leads to abysmal performance, but adding a few extra details to the scratchpad format leads to non-trivial generalization.

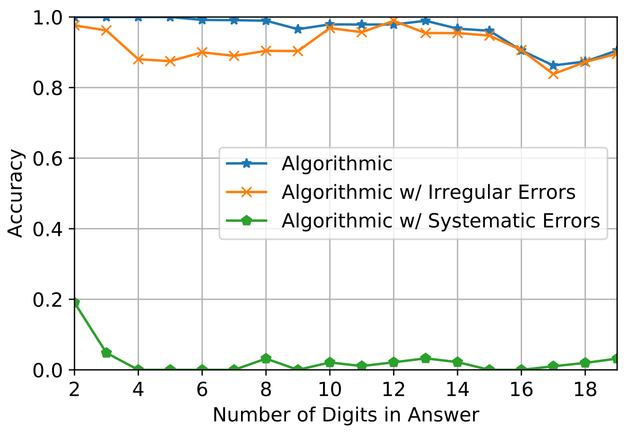


Figure 6: Errors are introduced to the algorithmic prompt in the digit value of the second number in the equation. *Irregular errors* are introduced to a minority subset of steps in the algorithmic examples, while *systematic errors* are introduced to all steps of the examples. We see that irregular errors have a minor impact on the performance, and systematic errors completely destroy the model's ability to solve this task. This suggests that the model is following the algorithm as it is specified in-context, rather than loosely mimicking the format of the algorithm.

Figure 5 compares the performance of scratchpad and detailed scratchpad. We conjecture that the abysmal performance of scratchpad as a few-shot prompt is due to the structure of the solution format being sufficiently regimented to move the model away from its memorized solutions, but not clear enough for the model to extract the true underlying rules and adapt them to new examples. With detailed scratchpad, we add more intermediate steps to illustrate how the values of the answer (A) and the carry (C) is derived. An additional version converts the numbers from space-delimited to comma-delimited, as we observed that comma is a more effective deliminator for Codex. We find that the additional information leads to a significant boost in results.

Is the model actually learning the algorithm through in-context learning? Min et al. (2022) have shown that it is not necessary to provide the correct question-answer pairings in the few-shot prompt, suggesting that the model does not rely on the demonstrations themselves to figure out the right way to solve the given task. However, in order to claim that we are teaching algorithms in-context, we would like to understand whether the model is actually following the algorithm as it is prescribed in the prompt. To do so, we validate that 1) mistakes in the intermediate output steps lead to mistakes in the final answer, and 2) errors in the prompt significantly impact performance.

We first look at the errors that the model makes. We find that for every addition question where the final answer was correct, *all* intermediate steps were also correct. Next, we analyze the performance of the model when we introduce errors into the algorithmic steps in the prompt. We introduce errors into second digit of the calculation step ($\text{digit}_1 + \textit{digit}_2 + \text{carry} = \text{answer}$), and keep all other elements the same as before. We consider two types of errors: *irregular errors* where only a subset of the steps contain an error, and *systematic errors* where all of the steps presented in the prompt contain an error. With irregular errors, the model still has a chance of extrapolating the correct rule based on the unchanged steps. With systematic errors, the model should *not* derive the correct rule if it was truly learning from context, rather than simply mapping to the output format and overriding the individual steps with what it has learned from its pretraining. Figure 6 shows that there is a small degradation in performance with irregular errors, whereas the accuracy drops to near 0% with systematic errors, thus confirming the expected behavior of a model that is actually learning in-context. This is in contrast to the findings in which providing shuffled targets (Min et al., 2022) or wrong patterns in chain-of-thought (Madaan & Yazdanbakhsh, 2022) do not materially impact model's performance. Thus, algorithmic prompting differs from other approaches and constrains the model's behavior towards what is being taught in-context.

A.3 Teaching other algorithms using algorithmic prompt

To validate that the performance of algorithmic prompt is not specific to two-number addition, we evaluate model performance on three other algorithms: subtraction, multiplication, and parity.

Subtraction: We follow a similar strategy as addition for subtraction. Different from addition-only questions, the ordering of the two numbers matter for subtraction. To see this, consider the examples $43 - 250 = -207$ and $543 - 250 = 293$. When we process the digits from right to left, the answer depends on whether the first number is greater than or less than the second number in absolute value. Thus, subtraction requires a different – albeit similar – algorithm to addition. For a sense of the relative complexity of the two settings, note that the subtraction algorithm we use runs in $2n$ steps, while the addition algorithm runs in n steps. We use 6 shots of up to 5-digits in answer length in the prompt.

Multiplication: Multiplication requires $O(n^2)$ steps if we use a similar strategy as the addition algorithm (which takes $O(n)$ steps). Inspired by this complication, we use multiplication to explore whether the model's existing zero-shot or few-shot capabilities can be leveraged in conjunction with algorithmic prompting to reduce the complexity of the required instructions. Instead of using single-digit multiplication in each step, we perform direct calculations for single-digit $\times$ n-digit numbers. Instead of doing n^2 single-digit calculations for two n-digit numbers, we now only need to perform n steps of $1\times$n-digit multiplication.

To choose a reasonable value of n for this experiment, we evaluate the model's zero-shot accuracy in $1 \times n$-digit multiplication (shown in Figure 7). We see that after $n = 3$, the zero-shot performance falls drastically. Thus, we restrict to $n \leq 3$ and restrict the problems we consider to have at least one of the two numbers in $a \times b$ be less than 1000. If one of the numbers has more than 3 digits, we break it down into groups of ≤ 3 digits and add the resulting sub-components appropriately. We use 2 shots of up to 6-digits in answer length in the prompt.

Parity: The task of calculating parity of a given binary list was extensively studied in Anil et al. (2022), and despite its simplicity this problem is far from being solved. Similar to (Anil et al., 2022) we investigate the parity problem as an example of length generalization. We use an algorithmic prompt for parity problem and compare its performance to that of scratchpad few-shot prompt as discussed in Anil et al. (2022)(Figure 9) as well as few-shot prompting of Codex. Figure 8 captures the performance of these three approaches on lists of varying sizes. We use 2 shots of up to 8 list length in the prompt. Each point in Figure 8 represents the average over 100 random samples, and the same examples are used for all methods. We observe that algorithmic prompt significantly outperforms both baselines and while the baselines' performance reaches random chance (50%) around length 5, algorithmic prompt keeps accuracy of around 80% to lists of up to 30 digits. B.3, B.4 depict the prompts used for these three methods.

The performance of subtraction, multiplication, and parity are shown in Table 3, and we see that we are able to effectively teach these algorithms and significantly outperform the best available baselines.

Table 3: Performance on addition, subtraction, multiplication, and parity tasks. For addition we use the few-shot baseline and evaluate at length 19. For subtraction we use the few-shot baseline and evaluate at length 14. For multiplication we use the chain-of-thought baseline and evaluate at length 7. These lengths are chosen based on the maximum task length that could fit into context for the algorithmic prompt. For parity we evaluate at length 20 which is the longest instance reported in Anil et al. (2022).

Method	Addition	Subtraction	Multiplication	Parity
Algorithmic prompt	90.5%	65.6%	79.7%	95.0%
Best available baseline	9.5%	16.7%	5.5%	50.0%

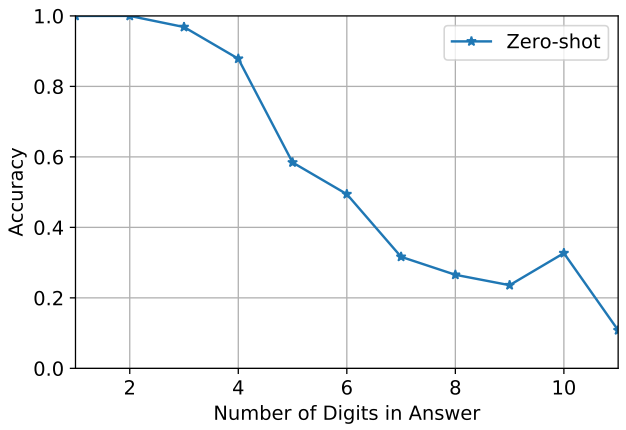


Figure 7: Zero-shot accuracy of one-digit multiplication, where the other number can have up to 11 digits. We see that the accuracy starts to drop after 3 digits in the answer.

A.4 Additional results for Skill Composition

This section includes additional details and figures on skill composition from Section 5. In Figure 9, we provide an illustrative demonstration of the change in the prompt when going from two-number addition to multi-number addition to multiplication-as-addition, showing the progression in complexity.

In order to move past context length limitations, we further experiment with a second-pass strategy, where we keep only the last completed algorithmic step in the model's output, and perform a second inference pass using the original prompt and the last output step. This simple approach benefits

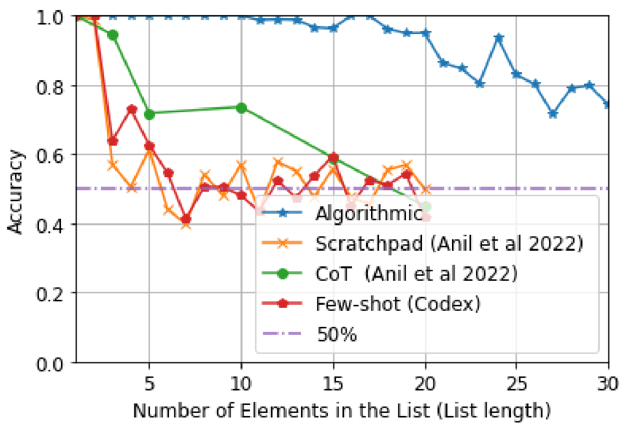
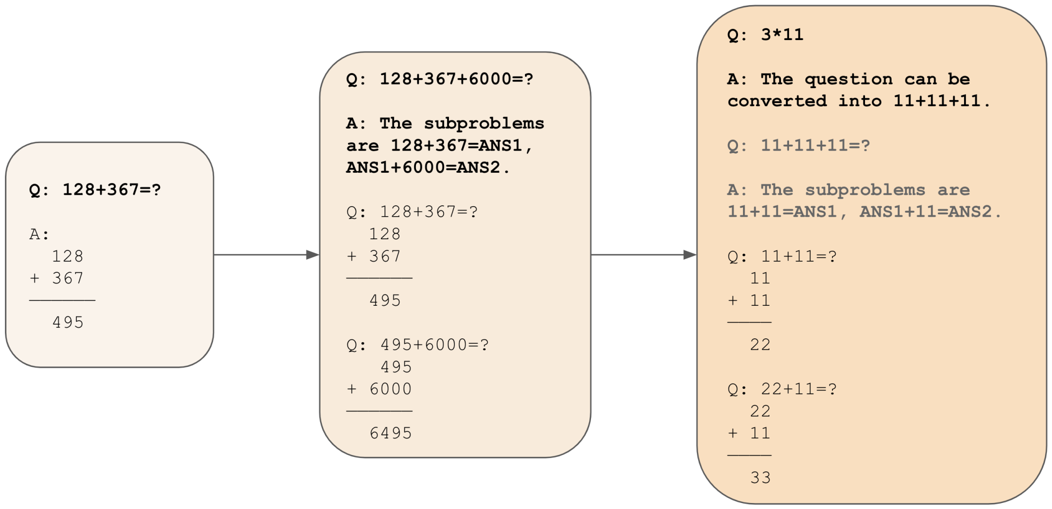


Figure 8: Investigating the performance of algorithmic prompt on parity problem and comparing it to scratchpad few-shot prompt of Anil et al. (2022) as well as few-shot prompting OpenAI's Codex. Each point on the plot corresponds to 100 random samples of a binary list of the same length. B.3, B.4 depict the prompts used for these three methods. The number of examples used in the prompt is two.



```
Q: 128+367=?

A:
  128
+ 367
-----
  495
```

```
Q: 128+367+6000=?

A: The subproblems
are 128+367=ANS1,
ANS1+6000=ANS2.

Q: 128+367=?
  128
+ 367
-----
  495

Q: 495+6000=?
   495
+ 6000
-----
  6495
```

```
Q: 3*11

A: The question can be
converted into 11+11+11.

Q: 11+11+11=?

A: The subproblems are
11+11=ANS1, ANS1+11=ANS2.

Q: 11+11=?
  11
+ 11
----
  22

Q: 22+11=?
  22
+ 11
----
  33
```

Figure 9: Illustration of the tasks and algorithmic prompting strategies considered for skill composition in Section 5

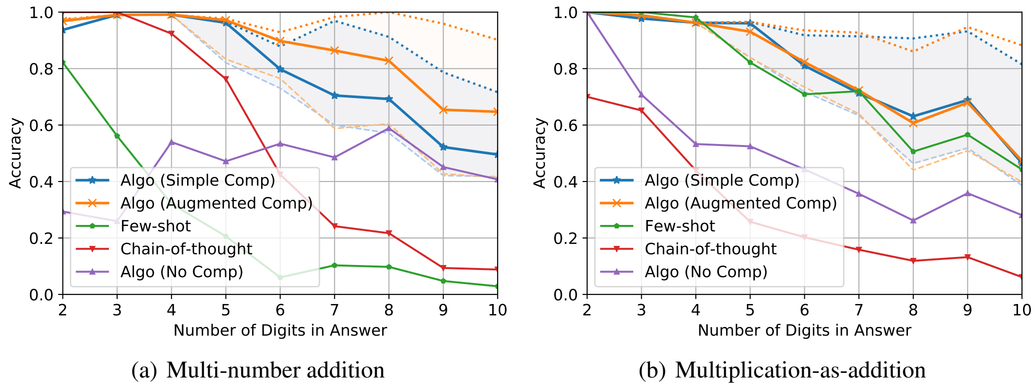


Figure 10: Performance on compositions of skills. Due to length of the algorithmic output for this task, a number of the longest examples exceed the context length limit for Codex. We employ a second pass strategy to get a final answer for the incomplete questions, where we keep in-context only the last completed state from the first pass. The dotted lines consider only questions for which the model completes the output within one pass, and provide an upper bound on performance. The dashed lines consider all incomplete questions as false, and provide an lower bound on performance. We observe that although the algorithmic prompting methods are suffering from having to do a second pass for the longer samples in this task, they still demonstrate better generalization than the baseline methods. Note that for multiplication, we evaluate the algorithmic methods on a harder task than the few-shot baselines, since we force the model to convert the question into the addition of a number n times, while for other baselines we simply perform $1 \times n$-digit multiplication directly.

from the fact that all relevant state variables are outputted in each step of the algorithm. We report performance on the entire dataset using the second pass strategy in Figure 10.

A.5 Additional results for Tool Use

This section includes additional details and figures for tool use in Section 6. In this section, we use GSM8k as a test bed for studying how the model can leverage a learned algorithm (instead of an external calculator) inside a broader reasoning process.

Q: Tommy has 3 toy cars. His neighbor, Jessie, has 3 cars too. Jessie's older brother has 5 more cars than Tommy and Jessie. How many cars do the three of them have altogether?
A: Tommy and Jessie have 3+3=6 cars. Jessie's brother has 5+6=11 cars. Altogether, they have 6+11=17 cars. The answer is 17.

Figure 11: An example question and answer pair from GSM8k with chain-of-thought rationale.

```
Q: June liked to collect cat stickers. She
had a total of 768835 in her collection. Her
twin sister Bonnie also collected cat
stickers and had 63212920 in her collection.
For their birthday, their grandparents gave
them 250148 cat stickers each. How many
combined stickers did they have?
--------------------------------------------
A: June had
<ALGO_START>768835+250148=<CALL_ADD_ALGO>101
8983 stickers. Bonnie had
<ALGO_START>63212920+250148=<CALL_ADD_ALGO>6
3463068 stickers. Together, they had
<ALGO_START>1018983+63463068=<CALL_ADD_ALGO>
64482051 stickers. The answer is 64482051.
```

Figure 12: An example of the GSM8k-Hard question, where the answer is shown to contain the tokens indicating when a second model should be consulted.

B Prompt Examples

B.1 Algorithmic prompt for addition

For addition prompts, we use 3-shot with the examples $128 + 367$, $9980 + 29$, and $802 + 7145$ in order. For conciseness, we may include only subsets of the prompt questions in the prompt examples.

Problem: 128+367=
```
Explanation:
The first number is 128, FN=[1,2,8].  The second number is 367,
SN=[3,6,7].  Since FN [1,2,8] has 3 digits, SN [3,6,7] has 3 digits, thus
the maximum number of digits is 3.  In each subsequent step, we remove
one number from the end of FN and one from the end of SN.
Length of FN is 3.  FN=[1,2,8].  Length of SN is 3.  SN=[3,6,7].  FN[3]=8.
SN[3]=7.  C[3]=0.  Since 8+7+0=15, 15>10, 15%10=5.  Length of A is 1.
Thus A=[5].  Since (15-5)/10=1, C[2]=1.
Length of FN is 2.  FN=[1,2].  Length of SN is 2.  SN=[3,6].  FN[2]=2.
SN[2]=6.  C[2]=1.  Since 2+6+1=9, 9<10, 9%10=9.  Length of A is 2.  Thus
A=[9,5].  Since (9-9)/10=0, C[1]=0.
Length of FN is 1.  FN=[1].  Length of SN is 1.  SN=[3].  FN[1]=1.
SN[1]=3.  C[1]=0.  Since 1+3+0=4, 4<10, 4%10=4.  Length of A is 3.  Thus
A=[4,9,5].  Since (4-4)/10=0, C[0]=0.
There are no more digits and C[0]=0.  Thus the process is complete.
Since there are no more operators, the problem is complete.  The final
Answer is [4,9,5].
```

B.2 Instruction-only prompt for addition

Compared to the algorithmic prompt, the instruction-only prompt also provides detailed explanations for the addition algorithm, but does not demonstrate the algorithm on running examples.

```
The following are instructions for solving addition problems in the form
of x + y = z, where x, y, and z are positive integers.
We will use the standard algorithm for addition.  We align the numbers x
and y on the least significant digit, which is the ones digit.  Starting
from right to left, we go from the least significant digit to the most
significant digit and add the corresponding digits from each number.
When the sum of the two digits is greater than 9, a carry of 1 is
included in the sum of the next digits.  When there is only one digit
available from the two numbers, only that digit along with any carry
is included in the sum.  When all the digits are processed, only the
remaining carry if any shall be included in the sum.
For x + y = z where x = int(str(abc)), y = int(str(defg)), we can solve z
with the following steps:
1) c+g=w', w=w'%10
2) b+f+((w'-w)/10)=v', v=v'%10
3) a+e+((v'-v)/10)=u', u=u'%10
4) d+((u'-u)/10)=t', t=t'%10
5) s=(t'-t)/10
Thus, z = int(str(stuvw)).
The answer should be in the form below:
Q: What is abc+defg=?
A: abc
+defg
-----
stuvw
The answer is stuvw.
```

B.3 Algorithmic prompt for parity

```
Q: What is the parity on the list a=[1, 1, 0, 1, 0]?
A: We initialize s=
a=[1, 1, 0, 1, 0].  The first element of a is 1 so b=1.  s = s + b = 0 +
1 = 1.  s=1.
a=[1, 0, 1, 0].  The first element of a is 1 so b=1.  s = s + b = 1 + 1 =
0.  s=0.
a=[0, 1, 0].  The first element of a is 0 so b=0.  s = s + b = 0 + 0 = 0.
s=0.
a=[1, 0].  The first element of a is 1 so b=1.  s = s + b = 0 + 1 = 1.
s=1.
a=[0].  The first element of a is 0 so b=0.  s = s + b = 1 + 0 = 1.  s=1.
a=[] is empty.  Since the list a is empty and we have s=1, the parity is
1.
```

B.4 Scratchpad prompt for parity (Anil et al., 2022)

```
Q: What is the parity on the list a=[1, 1, 0, 1, 0]?
A: [1, 0, 0, 1, 1], the parity is 1.
Q: What is the parity on the list a=[0, 1, 1, 0, 0, 0, 0, 0]?
A: [0, 1, 0, 0, 0, 0, 0, 0] , the parity is 0.
```

B.5 Chain-of-thought prompt for multi-number addition

Q: 9980+29=
```
A: Let's think step by step.
9980+29=10009
So, 9980+29=10009.  The answer is 10009.
```
Q: 802+7145+6=
```
A: Let's think step by step.
802+7145=7947
7947+6=7953
```