What Algorithms can Transformers Learn? A Study in Length Generalization

Hattie Zhou*[,2], Arwen Bradley[1], Etai Littwin[1], Noam Razin*[,3], Omid Saremi[1], Josh Susskind[1], Samy Bengio[1], and Preetum Nakkiran[1]

[1]*Apple*
[2]*Mila, Université de Montréal*
[3]*Tel Aviv University*

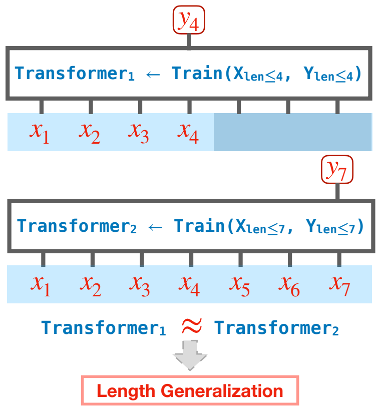
Abstract

Large language models exhibit surprising emergent generalization properties, yet also struggle on many simple reasoning tasks such as arithmetic and parity. In this work, we focus on length generalization, and we propose a unifying framework to understand when and how Transformers can be expected to length generalize on a given task. First, we show that there exist algorithmic tasks for which standard decoder-only Transformers trained from scratch naturally exhibit strong length generalization. For these tasks, we leverage the RASP programming language (Weiss et al., 2021) to show that the correct algorithmic solution which solves the task can be represented by a simple Transformer. We thus propose and give evidence for the RASP-Generalization Conjecture: Transformers tend to learn a length-generalizing solution if there exists a short RASP-L program that works for all input lengths. We then leverage our insights to give new scratchpad formats which yield strong length generalization on traditionally hard tasks (such as parity and addition). Overall, our work provides a novel perspective on the mechanisms of length generalization and the algorithmic capabilities of Transformers.

1 Introduction

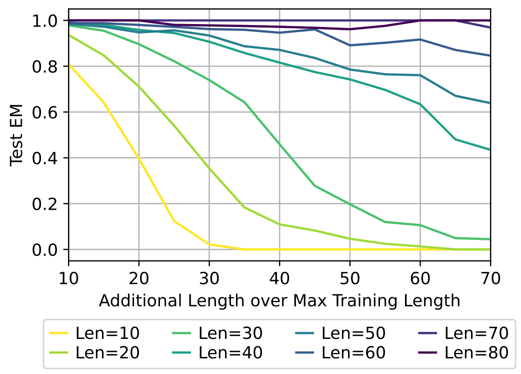
In this work, we aim to understand the factors that determine a standard decoder-only Transformer's ability to length generalize on algorithmic tasks. Length generalization evaluates the model on problems that are longer (and harder) than seen in the training set, and is used as a proxy for whether the model has learned the correct problem-solving strategy for the given task. There is currently scattered evidence regarding the length generalization capabilities of Transformers. Standard Transformers trained from scratch on addition and other arithmetic tasks exhibit little to no length generalization (Nye et al., 2021; Nogueira et al., 2021; Lee et al., 2023), and even models finetuned from pretrained LLMs struggle on simple algorithmic tasks (Anil et al., 2022). On the other hand, length generalization can occur for particular architectural choices and scratchpad formats (Jelassi et al., 2023; Kazemnejad et al., 2023). As a starting point of our work, we show that there exist algorithmic tasks where Transformers trained from scratch generalize naturally far outside of the training distribution. Why then do Transformers exhibit strong length generalization on certain tasks and not others, and what are the mechanisms behind generalization when it occurs? In the following sections, we will propose a unifying framework to predict cases of successful length generalization and describe the possible underlying mechanisms. See Appendix A for additional related works.

*Work done while interning at Apple.



y_4
Transformer₁ ← Train($X_{len\le4}$, $Y_{len\le4}$)
x_1 x_2 x_3 x_4
y_7
Transformer₂ ← Train($X_{len\le7}$, $Y_{len\le7}$)
x_1 x_2 x_3 x_4 x_5 x_6 x_7
Transformer₁ ≈ Transformer₂
Length Generalization

(a) Transformer length generalization



(b) Length generalization on counting task

Figure 1: **(a)** For a Transformer to length-generalize on an algorithmic task, the *same Transformer* must work on all lengths of the task, analogous to how the *same program* can solve a task on all inputs. For some tasks, it is not even clear if such a Transformer exists, and so length generalization is likely to fail. **(b)** Length generalization for the counting task, measured by exact match accuracy (EM). Transformers are trained on sequences of varying length, and tested at different levels of out-of-distribution over the maximum training length. Models trained on sequences up to length 60 or more exhibit near perfect length generalization up to length 150. Medians over 20 random seeds.

Preview of Length Generalization. We begin by introducing a simple task that exhibits strong length generalization. The task is "counting": given a prompt $\boxed{\texttt{SoS}}\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\texttt{>}}$ for numbers a, b, the model must count from a to b inclusive, and terminate with "EoS". An example is: $\boxed{\texttt{SoS}}\boxed{2}\boxed{5}\boxed{\texttt{>}}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{\texttt{EoS}}$. We train a Transformer with learned positional embeddings on count sequences of lengths up to 50, with random a and b points in $[0, 155]$. This trained model then length-generalizes near perfectly when prompted to count sequences of length 100 (see Figure 1b).

Possible Mechanisms. Intuitively, if the Transformer model learns a function that applies the correct transformation for inputs of any length in the context, then we can expect it to length generalize. For the count task, length generalization is possible if the model somehow learns a correct algorithm to solve the count task. One such algorithm is as follows. To predict the next token:

1. Search for the most-recent `SoS` token, and read the following two numbers as a, b.
2. Read the previous token as x. If `(x=='>')`, output a. If `(x==b)`, output `EoS`.
3. Otherwise, output $(x + 1)$.

This program applies to sequences of all lengths. Thus, if the model ends up learning this program from short sequences, then it will automatically length-generalize to long sequences. We will show that a Transformer can *easily represent* the above program, uniformly for all input lengths. To reason about what is "simple" to represent for Transformers, we leverage the RASP programming language (Weiss et al., 2021; Lindner et al., 2023), which is essentially an "assembly language for Transformers." (We describe RASP, and our extension of it RASP-L, in Section 3). We then propose the following toy model of learning.

Toy Model (RASP Learning). *For symbolic tasks, Transformers tend to learn the shortest RASP-L program which fits the training set, if one exists. Thus, if this minimal program exists and correctly length-generalizes, then so will the Transformer.*

We find this toy model to be a useful conceptual tool for predicting when Transformers will length-generalize. In Section 2, we introduce the RASP-Generalization conjecture, which predicts that Transformers are likely to length-generalize on tasks which can be solved by a short RASP-L program — and unlikely otherwise. We support this conjecture with experiments across various algorithmic tasks in Section 3. We apply our insights in Section 4 to modify the input and scratchpad formats of certain "hard" tasks (such as parity and decimal addition), to make them possible to solve with

RASP-L programs. These modifications significantly improve length generalization of Transformers, as expected by our conjecture.

2 Main Conjecture

In our main conjecture, and throughout the paper, we consider Transformers trained to completion on their training distribution. Training details are provided in Appendix B. A key ingredient we develop is a restriction of RASP which we call RASP-L, which we describe in more detail in Section 3.

RASP-Generalization Conjecture. *A decoder-only autoregressive Transformer is likely to length-generalize when trained to completion on an algorithmic task if the following conditions hold.*

1. ***Realizability.*** *The true next-token function for the task can be represented by a single causal Transformer which works on all input lengths.*
2. ***Simplicity.*** *This representation is "simple", meaning it can be written in RASP-L (a learnable subset of RASP defined in Section 3).*
3. ***Diversity.*** *The training data is sufficiently diverse, such that there does not exist any shorter RASP-L program which agrees with the task in-distribution but not out-of-distribution.*

Remarkably, the above features are empirically correlated with: (1) generalization to longer lengths out-of-distribution, and (2) faster train optimization in-distribution.

The first condition of realizability is actually quite stringent, because it requires a *single Transformer* to be able to solve the task at *all lengths*. This is not always possible: for example, next-token functions which require $\Omega(n^3)$ computation time on inputs of length n provably cannot be represented by a Transformer, however large (by the Time Hierarchy Theorem e.g. Arora & Barak (2009)).

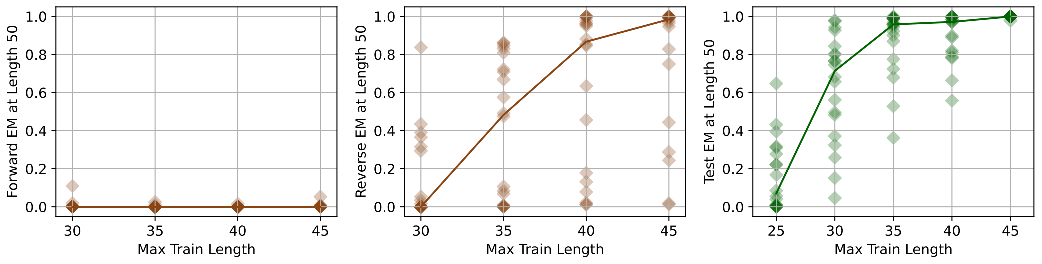
3 RASP-L: What Algorithms Can Transformers Learn?

To apply our conjecture, we need to determine if a given task can be solved in RASP-L. RASP-L is a fairly minor modification to RASP (Weiss et al., 2021), which we describe in Appendix G.4. We briefly describe RASP here; see Weiss et al. (2021) and Lindner et al. (2023) for more details.

The original RASP language can be thought of as a domain-specific-language for specifying Transformer weights, in human-readable form. Importantly, RASP was designed for the computational model of Transformers, so short RASP programs define functions which are "easy to represent" for Transformers. In Appendix G.4.2 we provide a small library of useful functionality built on the RASP-L core. This also serves as a representative sample of functions that are "easy" to implement RASP-L, to build intuition about programming causal Transformers.

Intuition. The key intuition about RASP is that it only allows parallelizable operations, because Transformers are an inherently *parallel* model of computation. This makes performing inherently-sequential computation, such as iterating through each input symbol and updating an internal state, tricky if not impossible to write in RASP. This is why loops are not allowed in RASP: because a Transformer has only constant depth, and cannot directly simulate an arbitrary number of loop iterations. The one way of bypassing this limitation is to exploit the *autoregressive* inference procedure: since the model is called iteratively at inference time, this effectively provides an "outer-loop" that can enable sequential computation which uses the input context as the state. This is exactly what scratchpads enable, as we elaborate in Section 4.

Experimental validation. To test this conjecture, we evaluate Transformers trained from scratch on 3 tasks that have simple RASP-L programs and 3 tasks that do not. The three easy tasks are: count, mode, and copy with unique tokens. We show in Appendix G.4 that the correct algorithms to solve these tasks have short RASP-L representations. The three hard tasks are: addition, parity, and copy with repeated tokens. These tasks do not have simple RASP-L solutions. Appendix D provides a detailed description of the tasks and experimental results, as well as intuition for why the difficult tasks do not permit simple RASP-L solutions. Indeed, we show in Figure 5 that Transformers exhibit strong length generalization on the set of easy tasks, and no length generalization on the hard tasks.



(a) Forward-order addition with index hints

(b) Reverse-order addition with index hints

(c) Parity with scratchpad

Figure 2: **Length generalization on addition and parity.** Plot shows 20 individual trials per length, as well as their median (solid line). **(a)** Shows generalization performance for forward addition with index hints on hard carry examples of length 50. No length generalization is observed. **(b)** Shows the generalization of reverse addition with index hints on hard carry examples of length 50. Most runs start to length generalize perfectly on 50 digit addition once the training length is greater than 40. **(c)** Shows generalization performance for parity with scratchpad, on length 50 inputs. Most runs start to generalize perfectly on 50 digit addition once the training length is greater than 35.

4 Application: Why Scratchpads Help

A number of works have observed that training or prompting with scratchpad-type approaches significantly improves reasoning performance (Wang et al., 2019; Nye et al., 2021; Wei et al., 2022; Anil et al., 2022; Zhou et al., 2022b,a). Our RASP conjecture provides a natural way to understand when and why scratchpads can improve generalization. For operations that are difficult for a Transformer to learn naturally, we can change the format of both the question and the solution such that they permit a shorter RASP-L solution (or any RASP-L solution at all).

Scratchpad helps generalization on addition. One obstacle to implementing the naive addition algorithm in RASP-L is the non-causality of the carry operation (the first token of the answer depends on the carries of all subsequent output tokens); we describe this in-detail in Appendix D. To address this non-causality, we can format the prompt and output in reverse-order. For example, $\boxed{5}\boxed{4}\boxed{+}\boxed{3}\boxed{7}\boxed{>}\boxed{9}\boxed{1}$ becomes $\boxed{5}\boxed{4}\boxed{+}\boxed{3}\boxed{7}\boxed{>}\boxed{1}\boxed{9}$. In reverse order addition, each output digit only needs to reference the previous output digit to get the carry value for the current step. In the example $\boxed{5}\boxed{4}\boxed{+}\boxed{3}\boxed{7}\boxed{>}\boxed{1}\boxed{9}$, to know that 1 should be added to 5+3, the model can look at the previous output and see that there is a carry of 1. Thus, previous tokens in the output act like a built-in scratchpad. In forward order addition, however, determining the carry at each step requires a more complex operation involving all remaining digit sums. In the example $\boxed{5}\boxed{4}\boxed{+}\boxed{3}\boxed{7}\boxed{>}\boxed{9}\boxed{1}$, the step 5+3 needs to also compute 4+7 in order to know that a carry should be added. In the next output step, 4+7 needs to be computed again (see Figure 9 for an illustration).

Another operation which is required for addition, but difficult for Transformers, is precisely accessing and manipulating positional indices (e.g. to query the summand digits for the current output digit). To address the indexing challenge, we can leverage induction heads (Olsson et al., 2022), which are easy to represent in RASP-L, to simplify the addition algorithm for a Transformer. We do so by adding "index hints" to the prompt and answer: $\boxed{5}\boxed{4}\boxed{+}\boxed{3}\boxed{7}\boxed{>}\boxed{9}\boxed{1}$ becomes $\boxed{a}\boxed{5}\boxed{b}\boxed{4}\boxed{+}\boxed{a}\boxed{3}\boxed{b}\boxed{7}\boxed{>}\boxed{a}\boxed{9}\boxed{b}\boxed{1}$. This enables us to get the corresponding digits for each sum step by calling `induct` on its index hint, b, thus sidestepping the need to precisely access and manipulate positional indices. The complete RASP-L program for reverse-order addition with index hints is given in Appendix G.4.7.

In Figure 2, we evaluate addition on a "hard" carry setting, where we constrain the examples to have the longest possible carry chain for the given length. For example, a hard carry instance of length 3 is $381 + 619 = 1000$, which requires the model to compute the carry over a chain of 3 digit positions. We find that index hints allow reverse addition to length generalize near perfectly on up to 10 additional digits, whereas the harder forward addition setting still exhibits no generalization. We show in Figure 8 that index hints allow both forward and reverse addition to generalize in the standard "easy" carry setting where the two numbers are sampled randomly and independently. Lastly, in Appendix E, we show the a well-designed scratchpad can also lead to strong length generalization on parity, and an intuitive scratchpad for the mode task actually leads to worse generalization performance due to increased complexity of the corresponding RASP-L program. Our results on addition and parity demonstrate the first instance (to our knowledge) of strong length generalization for Transformer models trained from scratch on these tasks.

References

Emmanuel Abbe, Samy Bengio, Aryo Lotfi, and Kevin Rizk. Generalization on the unseen, logic reasoning and degree curriculum, 2023.

Cem Anil, Yuhuai Wu, Anders Johan Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Venkatesh Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=zSkYVeX7bC4.

Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages, 2020.

Satwik Bhattamishra, Arkil Patel, Varun Kanade, and Phil Blunsom. Simplicity bias in transformers and their ability to learn sparse boolean functions, 2023.

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.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention, 2022.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Antonia Creswell and Murray Shanahan. Faithful reasoning using large language models, 2022.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy, 2023.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jian, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, et al. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2023.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures, 2021.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.

Samy Jelassi, Stéphane d'Ascoli, Carles Domingo-Enrich, Yuhuai Wu, Yuanzhi Li, and François Charton. Length generalization in arithmetic transformers, 2023.

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

Andrej Karpathy. nanogpt. https://github.com/karpathy/nanoGPT, 2023.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers, 2023.

Jeonghwan Kim, Giwon Hong, Kyung-min Kim, Junmo Kang, and Sung-Hyon Myaeng. Have you seen that number? investigating extrapolation in question answering models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021.

Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers, 2023.

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.

Yuxuan Li and James McClelland. Representations and computations in transformers that support generalization on structured tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=oFC2LAqS6Z.

David Lindner, János Kramár, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. *arXiv preprint arXiv:2301.05062*, 2023.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata, 2023.

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

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks, 2021.

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.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads. *Transformer Circuits Thread*, 2022. https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

Santiago Ontañón, Joshua Ainslie, Vaclav Cvicek, and Zachary Fisher. Making transformers solve compositional tasks, 2022.

Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=R8sQPpGCv0.

Anian Ruoss, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Róbert Csordás, Mehdi Bennani, Shane Legg, and Joel Veness. Randomized positional encodings boost length generalization of transformers. Toronto, Canada, 2023. Association for Computational Linguistics.

Abulhair Saparov, Richard Yuanzhe Pang, Vishakh Padmakumar, Nitish Joshi, Seyed Mehran Kazemi, Najoung Kim, and He He. Testing the general deductive reasoning capacity of large language models using ood examples, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

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

Haoyu Wang, Mo Yu, Xiaoxiao Guo, Rajarshi Das, Wenhan Xiong, and Tian Gao. Do multi-hop readers dream of reasoning chains? In *Proceedings of the 2nd Workshop on Machine Reading for Question Answering*. Association for Computational Linguistics, 2019. URL https://aclanthology.org/D19-5813.

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*, 2022.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers, 2021.

Sean Welleck, Peter West, Jize Cao, and Yejin Choi. Symbolic brittleness in sequence models: on systematic generalization in symbolic mathematics. In *AAAI*, 2022. URL https://arxiv.org/pdf/2109.13986.pdf.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks, 2023.

Shizhuo Dylan Zhang, Curt Tigges, Stella Biderman, Maxim Raginsky, and Talia Ringer. Can transformers learn to solve problems recursively?, 2023.

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*, 2022a.

Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning, 2022b.

A Additional Related Works

Large language models (LLMs) have shown impressive abilities in natural language generation, reading comprehension, code-synthesis, instruction-following, commonsense reasoning, and so on (Brown et al., 2020; Chen et al., 2021; Chowdhery et al., 2022; Lewkowycz et al., 2022; Gunasekar et al., 2023; Touvron et al., 2023). However, when evaluated in controlled studies, Transformers often struggle with systematic generalization (Nogueira et al., 2021; Ontañón et al., 2022; Dziri et al., 2023; Wu et al., 2023; Saparov et al., 2023). It is thus not clear how to reconcile Transformers' seemingly-impressive performance in some settings with their fragility in others.

Our paper is related to the line of work that seeks to understand the capabilities and limitations of Transformer models when it comes to algorithmic reasoning (Kaiser & Sutskever, 2015; Veličković & Blundell, 2021). Recent studies have focused on length generalization on algorithmic tasks as a measure of how well language models can learn to reason (Nogueira et al., 2021; Kim et al., 2021; Anil et al., 2022; Lee et al., 2023; Dziri et al., 2023; Welleck et al., 2022; Liu et al., 2023). In this paper, we also focus on simple tasks like arithmetic and study length generalization on the standard Transformer architecture. Related to this, Lee et al. (2023) study how well transformers trained from scratch can learn simple arithmetic tasks, and finds that no length generalization is observed. Nogueira et al. (2021) find that partial length generalization on addition is observed only when models reach 3B parameters and when the addition questions are presented in reverse order. Jelassi et al. (2023) study models trained on addition and find strong generalization performance when using a few examples of longer sequences. However, they required non-standard architectures (non-causal Transformers) and training procedures (artificial padding) to observe this, and still find that the model does not generalize to unseen length that is in between the minimum and maximum lengths seen during training.

Moreover, our work contributes to the study of what makes for effective scratchpads. Other papers have also found that using positional tokens in the prompt can help with length generalization (Nogueira et al., 2021; Li & McClelland, 2023). However, these works do not provide a framework for understanding why these tricks are helpful. A number of papers also study how chain-of-thought style prompting helps with reasoning performance (Wei et al., 2022; Zhou et al., 2022b,a; Creswell & Shanahan, 2022; Madaan & Yazdanbakhsh, 2022), but these focus on in-context learning and do not study the effect of training models on these formats.

Other papers also aim to understand the limits of what Transformers can learn and represent. Bhattamishra et al. (2020) and Delétang et al. (2023) study the ability of Transformers to represent and generalize on families of formal languages. Zhang et al. (2023) evaluate the ability of transformer models to emulate the behavior of structurally recursive functions from input-output examples. Liu et al. (2023) study how shallow Transformers can simulate recurrent dynamics representable by finite-state automata. Both works identify shortcut solutions that become brittle on out-of-distribution samples. Bhattamishra et al. (2023) suggest that Transformer models have an inductive bias towards learning functions with low sensitivity, such as sparse boolean functions, but focus on the in-distribution setting. Abbe et al. (2023) also propose a simplicity bias in Transformers, but use "minimum-degree" as their notion of function simplicity. However, they only consider functions with fixed input dimension rather than programs on arbitrary input lengths.

Lastly, there have been many other approaches to improving length generalization in Transformers. These include studying how various training hyperparameters and design choices influence compositional generalization (Furrer et al., 2021; Ontañón et al., 2022), and designing better positional embeddings (Press et al., 2022; Ontañón et al., 2022; Kazemnejad et al., 2023; Ruoss et al., 2023).

B Additional Experimental Details

A *Transformer* (Vaswani et al., 2017) refers to a decoder-only causal Transformer architecture with constant depth, width, and fixed setting of weights, along with any computable positional embedding scheme[2]. As a technical point, we allow the transformer weights to take values in the extended real

[2]This is a technical detail: we consider position encoding schemes which can be uniformly generated, i.e. there exists a Turing machine which on input (i, n), produces the positional embedding vector for index i out of n total.

line $\mathbb{R} \cup \{\pm\infty\}$, to allow saturating the softmax at arbitrary context lengths. We consider only greedy sampling throughout, since our tasks are deterministic.

We train all of our models to convergence on the train distribution where possible. For all tasks, the length of training examples is sampled uniformly from length 1 up to the max training length. We train Transformer models from scratch and use learned positional embedding on all tasks. At train time, we "pack the context", filling the Transformer's context window with multiple independent samples of the task, and we randomly shift the Transformer along its context window. This procedure of packing and shifting the context mirrors standard practice in LLM training on real data (Karpathy, 2023; Brown et al., 2020), but is typically not done in prior works using synthetic tasks. It is an important detail: packing and shifting the context allows all positional embeddings to be trained, and encourages the transformer to treat all positions symmetrically. At test time, we evaluate examples without packing and shifting. We measure the exact match (EM) on all outputs, which is 1 if the entire output sequence is correct, and 0 otherwise.

For all experiments, we tokenize every character individually, including digits of a number. We train in the online setting, where each batch is sampled iid from the train distribution instead of from a finite train set — this avoids overfitting issues, and is closer to the training of modern LLMs. Unless otherwise specified, we evaluate test performance on $5\times$ the batch size number of samples. Unless otherwise specified, we run each experiment on 20 random seeds and report the median of the runs. We select hyperparameters for each task based on what is required to fit the training set. Hyperparameter details can be found in Table 1.

Count. For the count task, we train with an alphabet size of 155 and evaluate on test sequences up to 150 in length. Given the nature of the task, we enumerate all possible sequences of each test length at evaluation time. The alphabet is ordered such that the sequence between any two tokens in the alphabet is clearly defined. At train time, the length of each example is sampled uniformly between 1 and the maximum training length.

Mode. For the count task, we train on an alphabet of 52 tokens. Each example consist of 5 unique tokens sampled randomly from the alphabet. The length of each example is sampled uniformly between 1 and the maximum training length, and the sequence is randomly sampled from the 5 selected tokens. If there is a tie for the most frequent element, we randomly select a token from one set and changes it to a token from the other set, thus ensuring that there is one unique answer.

Copy. For the copy task with unique tokens, we train on an alphabet size of 100. The length of each example is sampled uniformly between 1 and the maximum training length, and the sequence is randomly sampled from the alphabet without replacement. For the copy task with repeat tokens, we use the same sampling procedure, but now on an alphabet size of 2.

Addition. For the addition task, we sample the length of each of the two numbers independently, from 1 up to the maximum training length. We then pad the two numbers with 0 in the front such that they have the same length. We pad the numbers with an extra 0 to allow for the potential of an extra digit in the answer due to carry.

Parity. For the parity task, we sample the length of each parity sequence from 1 up to the maximum training length. We then sample randomly from $\{1, 0\}$ a sequence of the given length. We note that the definition of length we use is based on the sequence length and not based on the number of 1s in the sequence.

C Counterfactual Analysis on Count

In this section, we probe whether models trained on count actually learn the count algorithm that we intuitively want. To reiterate, one simple algorithm that solves the count task is as follows. To predict the next token:

1. Search for the most-recent `SoS` token, and read the following two numbers as a, b.
2. Read the previous token as x. If `(x=='>')`, output a. If `(x==b)`, output `EoS`.
3. Otherwise, output $(x + 1)$.

Table 1: Experimental hyperparameters. All experiments use AdamW optimizer and cosine learning rate schedule. Count and Copy use weight decay of 0.1 and grad clip of 0. Parity and Mode use weight decay of 0.1 and grad clip of 1. Addition uses weight decay of 0 and grad clip of 1.

Task	Model Size	Train Iter	Context Len	Batch Size	Learning Rate
Count	6 layer; 8 head; 64 emb	10000	256	128	1e-3 to 1e-5
Mode	6 layer; 8 head; 512 emb	10000	256	128	1e-3 to 1e-6
Copy	6 layer; 8 head; 512 emb	100000	512	128	1e-4 to 1e-6
Addition	6 layer; 8 head; 512 emb	30000	512	64	1e-4 to 0
Parity	6 layer; 8 head; 512 emb	10000	512	256	1e-3 to 1e-6

Since there is no easy way to formally check if the Transformer model learns this exact algorithm internally, we employ the simple heuristic of running the model on counterfactual examples. The allows us to stress test the models behavior and see if it performs the expected algorithmic steps in an input-agnostic way. To do so, we performance inference on randomly generated input sequences that are designed to test the model in four ways:

1. The model should always output the start token in the prompt (a) as the first output. (Figure 3a)
2. The model should always output `EoS` following a token that matches the ending token in the prompt (b). (Figure 3b)
3. In all other settings, the model should increment the previous token by 1. (Figure 4a)
4. The model should not output `EoS` prematurely. (Figure 4b)

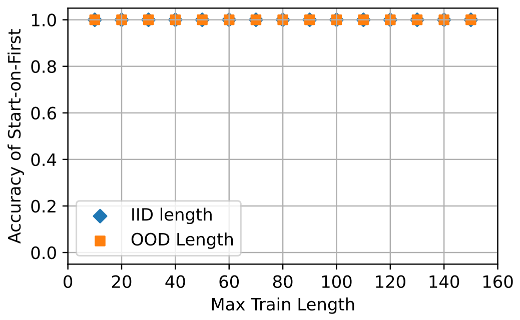
We create the counterfactual dataset by sampling start and end tokens of varying distances, then generate a sequence of random tokens of the length specified by the distance between the start and end token. We then pass this sequence through a trained model and look at its predictions at each token position. The goal of the four proposed tests on random sequences is to probe whether the model learned the expected algorithmic behavior rather than learning something that would strongly depend on statistics of the training distribution. We sample examples for in-distribution lengths and out-of-distribution lengths based on the training distribution of each model. For simplicity, we choose 1 model with strong length generalization performance from each maximum training length setting. The performance on each test is shown in Figure 3 and Figure 4.

We see that for the start-on-first test and the increment-by-1 test, all models exhibit near perfect performance both in- and out-of-distribution. For the end-on-last test, we see that models trained with shorter lengths do not learn to robustly output `EoS` on long test sequences once the ending condition is met. However, on models trained on longer sequences (and has better length generalization), this behavior is more robust. Lastly, when we measure the percentage of `EoS` which are correct, we see that models that do not have strong generalization also fails to output `EoS` only at the appropriate time. This failing is observed on both in-distribution and out-of-distribution lengths. This suggests that the failures of length generalization can be attributed to prematurely outputting an `EoS` token before the full length of the sequence is outputted. Overall, we observe strong correspondence between the model's behavior and what we would expect from the correct algorithm. This lends credence to the intuition that the model learns the correct RASP-L program and generalizes because of it.

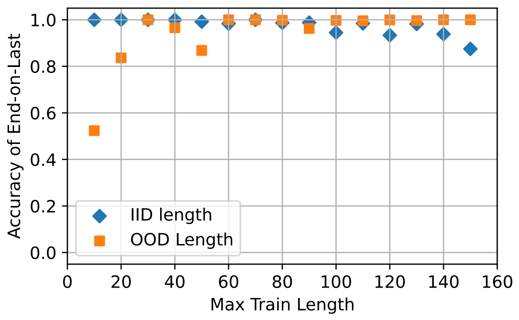
D Case Studies on RASP-L Algorithms

In this section, we experimentally evaluate 3 tasks that have simple RASP-L programs and 3 tasks that do not. We show that RASP-L representability is correlated with length generalization performance. The three easy tasks we consider are: count, mode, and copy with unique tokens. We provide the RASP program for these tasks in Appendix G.4 (Listings 3, 4, and 5, respectively). Detailed training procedures and hyperparameters are provided in Appendix B.

Count. We described the count task in the Introduction and showed results in Figure 1b. This task can be solved by a RASP-L program that essentially translates the pseudocode in the Introduction (defails in Listing 3). We find that models trained on count can generalize near perfectly to double

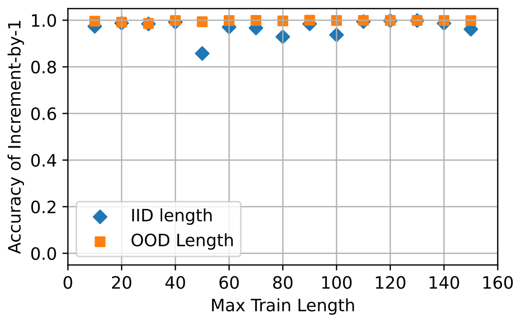


(a) Counterfactual test of starting on first token

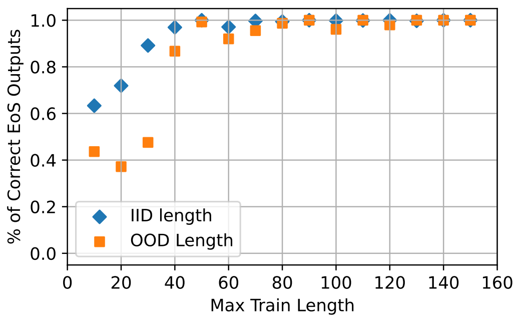


(b) Counterfactual test of ending on last token

Figure 3: We measure performance of models trained on the count task on counterfactual tests designed to evaluate whether the model simulates the correct generalizing algorithm on random sequences far out-of-distribution. We see that models always output the start token as the first output, but do not always output the EoS token once the ending token has been outputted.



(a) Counterfactual test of increment-by-1



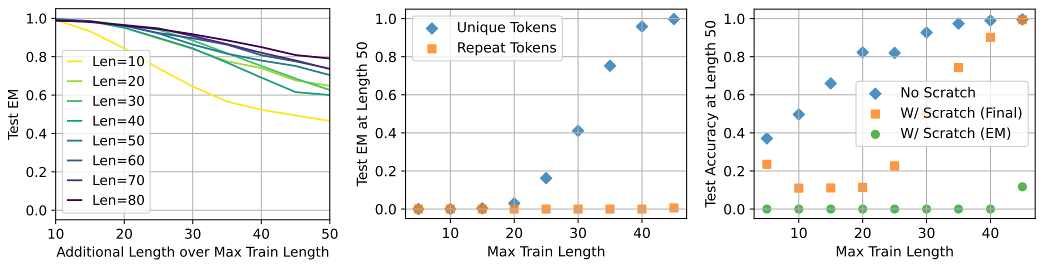
(b) Counterfactual test of the percentage of EoS outputs which are correct

Figure 4: We measure performance of models trained on the count task on counterfactual tests designed to evaluate whether the model simulates the correct generalizing algorithm on random sequences far out-of-distribution. We see that models almost always increments the previous token by 1, no matter what the previous sequence is. However, it sometimes output the EoS token prematurely, especially on lengths longer than seen in training. This likely explains failures of length generalization observed in Figure 1b.

the training lengths. It is crucial that our training distribution contain samples ranging from length 1 to maximum training length, which adds diversity as we scale up training length. These factors are necessary in preventing shortcut programs from being learned: no generalization is observed if we train on sequences of all the same length.

Mode. The mode task identifies the most frequent element in a sequence. We constrain the sequences such that the answer is unique. An example is: `a b b c b a c b > b`. Figure 5a shows the results on mode, when training and testing on random sequences from an alphabet of 52 symbols. We find that models trained on mode generalizes strongly to sequence lengths much longer than the maximum training lengths. Interestingly, increasing training set complexity here does not result in huge improvements in length generalization: Even models trained on sequence length up to 10 can achieve a median test accuracy of 50% of sequences of length 60.

Copy with unique tokens. The copy task repeats the prompt sequence in the output. We constrain the sequences to have all unique tokens in the prompt. An example is: `a c d b > a c d b`. Figure 5b shows the results on copy with unique tokens. For models trained on sequence length up to 40, we find that they can generalize perfectly to length of 50. Intuitively, this task is easy because we



(a) Length generalization on mode (b) Copy task (c) Mode task (w/ scratchpad)

Figure 5: **(a)** Length generalization performance for the mode task. All models generalize perfectly to 10 or more additional tokens at test time. **(b)** Performance on copy tasks for test sequences of length 50, and varying train lengths. Copying unique tokens makes length generalization much easier, since it can be solved by an induction head. **(c)** Performance on mode task for test sequences of length 50, and varying training lengths. The scratchpad hurts in this setting (both final answer accuracy and exact match), since it is not computable by a short RASP-L program.

can leverage something called an "induction head" (Olsson et al., 2022). Induction heads work by identifying a previous instance of the current token, find the token that came after it, and predict the same completion to the current token. Olsson et al. (2022) found that induction heads are reliably learned even by simple Transformers, and conjectured them to be a component of what enables in-context learning in LLMs. Induction heads are simple to implement in RASP-L, as the `induct` function. Thus, the next token can be generated by simply using an induction head on the current token, since all tokens are unique. This is exactly what the RASP-L program does, in Listing 5.

Next, we identify three tasks that do not admit simple RASP-L solutions: addition, parity, and copy with repeating tokens. We discuss reasons why Transformer models struggle to generalize on these tasks by highlighting the operations these algorithms require, but which are unnatural for a Transformer to represent.

Addition & Parity. Both these tasks have been studied as difficult tasks for Transformers: models trained from scratch show little to no length generalization on addition (Nye et al., 2021; Lee et al., 2023) and parity (Bhattamishra et al., 2020; Chiang & Cholak, 2022; Ruoss et al., 2023; Delétang et al., 2023), and even pretrained LLMs cannot solve these tasks robustly (Brown et al., 2020; Chowdhery et al., 2022; Anil et al., 2022) without careful prompting (Zhou et al., 2022b).

Indeed, addition is also difficult to write in RASP-L. To see why, consider the standard addition program shown in Appendix G.4.6. This algorithm requires the carry value to be propagated from least- to most-significant digit, but this cannot be easily simulated due to causal masking. Moreover, the `for`-loop iteration is not allowed in RASP-L. A particularly challenging aspect is the index-related operations. The standard addition algorithm requires index-arithmetic (e.g. finding the middle of the prompt sequence) and precise indexing operations (e.g. look up the corresponding summand digits for the current output digit). It is nontrivial to construct positional embeddings that allow the model to precisely access the n^{th} element in a sequence for any $n \in \mathbb{N}$, in a numerically stable way. It is even more difficult to construct embeddings that allow index-arithmetic, e.g. transforming the encoding of n to the encoding of $\lfloor n/2 \rfloor$, in a way that works globally for all $n \in \mathbb{N}$. Such operations are forbidden in RASP-L, as they are typically not learned in a global, length-generalizing way. Due to these obstructions, we are unable to derive a RASP-L program for this task.

Similarly, parity without any scratchpad requires operations that are forbidden under RASP-L. Since a Transformer cannot update its state sequentially, it must solve parity with parallel operations only. Intuitively, this requires taking the sum of the entire sequence, then determining the parity of the sum. This cannot naturally be computed in a numerically stable way for an arbitrarily large sums. Moreover, we cannot expect to learn a 'sum' operation which generalizes to numbers larger than the training sequences. Indeed, many works have shown that a Transformer cannot even fit the training set of parity sequences over some minimal length (Bhattamishra et al., 2020; Chiang & Cholak, 2022).

Under our experimental settings, we find that *no* length generalization is observed for both addition and pairity tasks: test performance does not exceed random chance when evaluated on examples 5 or more tokens longer than the training set.

Copy with repeating tokens. For this task, we constrain the sequences to consist only of 2 possible tokens. An example is: `a a b a > a a b a`. Since the tokens are no longer unique, the induction-head is no longer helpful. Instead, the model must perform precise index-arithmetic, which is difficult for the same reason that indexing is difficult in addition. We show in Figure 5b that models completely fail to generalize to longer lengths on this task.

E Why Scratchpads Help

In this section, we provide more examples of leveraging the RASP-Generalization Conjecture to understand when and how scratchpads can improve generalization.

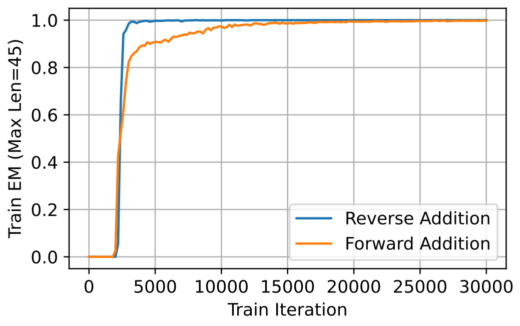
Scratchpad helps generalization on parity. The natural algorithm for parity is to iterate through all input tokens, updating some internal state along the way. Unfortunately, this algorithm cannot be directly implemented in RASP-L, because RASP-L does not allow loops— fundamentally because one forward-pass of a Transformer cannot directly simulate n sequential iterations of the loop. However, if the internal state is written on the scratchpad each iteration, then the Transformer only needs to simulate *one* iteration in one forward-pass, which is now possible.

We leverage this intuition to design a scratchpad for parity. Similar to addition, we add index hints to the prompt to simplify the indexing operation. In the scratchpad output, we locate index hints that precede each 1 in the prompt, and keep track of the running parity with symbols $+$ (even) and $-$ (odd). The last output token corresponds to the final answer. For example: `a 0 b 0 c 1 d 1 e 0 > + c - d +`. Figure 2c shows the exact match performance of the proposed parity scratchpad. We find that some of the runs trained with sequences up to 30 in length can generalize perfectly on sequences of length 50. When training length reaches 40, all models achieve perfect length generalization on length 50. Lastly, in Figure 6b, we compare the training curves of parity using various scratchpads, and we show that training speed is also correlated with how difficult the task is under RASP-L. Details can be found in Appendix F. To our knowledge, these results demonstrate the first instance of strong length generalization on parity for Transformer models trained from scratch.

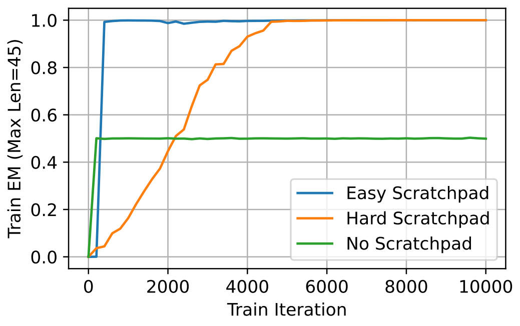
Scratchpad *hurts* generalization on mode. Now we consider the mode task and look at how scratchpad might affect a task that a Transformer is naturally amenable to. A natural algorithm one might come up with is to calculate the counts of each unique token in the sequence, then output the token with the maximum count. To encourage the model to learn this algorithm, we might utilize the following scratchpad, where we output the frequency of each token in ascending order: `a b b c b a c b > 2 a 2 c 4 b b`. The last token in the scratchpad is then the correct answer. However, although this scratchpad provides more supervision for what algorithm the model should learn, it is a more difficult task when considered in terms of RASP-L. Finding and comparing the frequency is simple to do internally, but converting this implicit representation into an integer token adds additional complexity. We show in Figure 5c that the scratchpad performs significantly worse than no scratchpad, both when measured on exact match and also on the accuracy of the final answer.

Nonetheless, one may consider the given scratchpad, which orders the intermediate counts in order of frequency, to be overly demanding. This scratchpad requires the model to know the order of the frequencies before outputting them. Another variant of this could output the scratchpad in order of appearance in the original sequence. Moreover, we can output the token first before outputting their count, which may help the model reference the relevant token for this step. An example is `a b b c b a c b > a 2 b 4 c 2 b` .

The performance of this scratchpad is shown in Figure 7. We see that utilizing this scratchpad still results in much worse length generalization performance than using no scratchpad.



(a) Training speed comparison for addition with index hints



(b) Training speed comparison for parity with different scratchpads

Figure 6: We compare the training speed as measured by exact match on a maximum training length of 45. **(a)** compares the convergence speed of models trained on forward addition vs reverse addition with index hints. **(b)** compares the convergence speed of models trained on different scratchpads on parity.

F Training Speed

The RASP Generalization Conjecture suggests that simple-to-represent programs are also more likely to be learned. Intuitively, we should expect that simple-to-represent programs also exhibits faster optimization during training. In this section, we evaluate different strategies of presenting the addition and the parity task, and see whether the training speed of these variants correspond to the simplicity of their corresponding RASP-L programs (and by extension their length generalization performance).

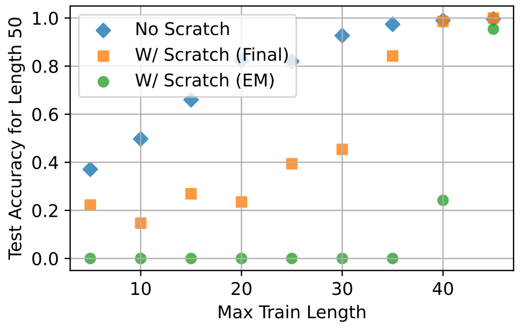
As discussed in Section 4, reverse addition simplifies the function that the model needs to learn when compared to forward addition. Figure 6a shows the training curves for each of these settings. Consistent with intuition, we see that reverse addition converges more quickly than forward addition.

For the parity task, we introduce an additional scratchpad format for comparison. This scratchpad outputs the sum-mod-10 of the parity sequence before outputting the final parity. An example is `0 0 1 1 0 > 2 , 0`. This scratchpad does not simplify the problem as much as the main scratchpad presented in Section 4 because it does not leverage the autoregressive inference to process the task sequentially. However, it is still simpler than parity without any scratchpad because it helps to simplify the final operation of getting the parity of the sum. Instead of doing this internally, the model can now reference the output of the sum-mod-10 and learn a simple mapping between that and the corresponding parity. Figure 6a shows the training curves for each of these settings. We see that the main scratchpad ("Easy Scratchpad") optimizes much more quickly than the sum-mod-10 scratchpad ("Hard Scratchpad"). We also observe that Easy Scratchpad exhibits significantly stronger length generalization than Hard Scratchpad, shown in Figure 7b. Both scratchpads optimizes much better than parity with no scratchpad, which is unable to even fit the training set and demonstrates no length generalization.

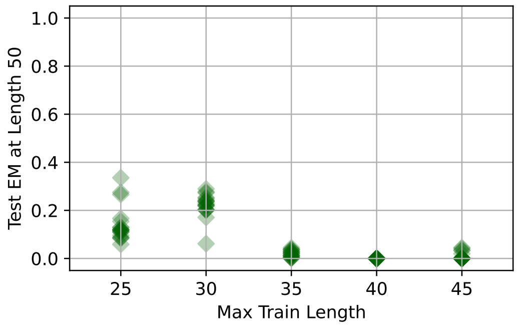
G RASP Details

G.1 RASP: A Primer

The original RASP language can be thought of as a domain-specific-language for specifying Transformer weights, in human-readable form (Weiss et al., 2021). Importantly, RASP was designed for the computational model of Transformers, so short RASP programs define functions which are "easy to represent" for Transformers. Although RASP was conceived as a separate language with its own syntax, we can also realize RASP as a restricted subset of Python where only a few operations are allowed. We show how to do this explicitly in Appendix G.4.1, and just include a few examples here. Every RASP program accepts an input sequence of length n, for all $n \in \mathbb{N}$, and returns an output sequence of the exact same length— just like a Transformer.

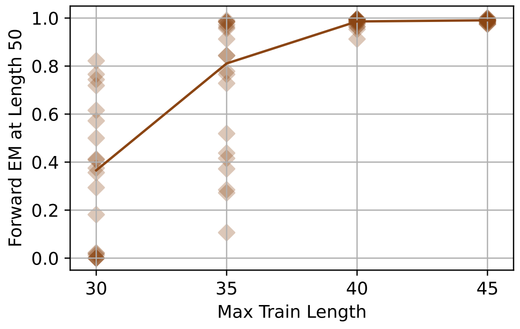


(a) Mode using scratchpad ordered by appearance

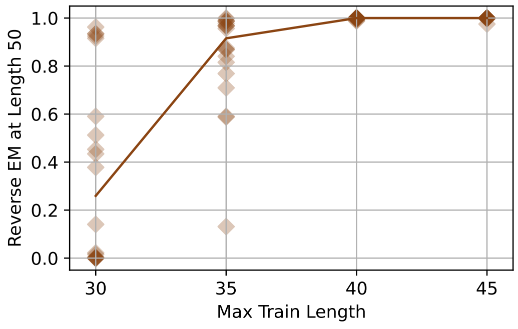


(b) Parity with sum-mod-10 scratchpad

Figure 7: **(a)** compares test performance of mode with or without scratchpad. In this case, we use the scratchpad presented in order of appearance. We see that no scratchpad significantly outperforms scratchpad, whether measured on the final answer accuracy or the exact match of the entire scratchpad output. **(b)** illustrates the generalization performance for parity with scratchpad on length 50. We see that no runs show significant length generalization in this setting.



(a) Forward addition on easy carry examples



(b) Reverse addition on easy carry examples

Figure 8: Length generalization on addition with index hints. Each diamond shows the performance of one of 20 runs, illustrating the spread of different training runs. **(a)** illustrates the generalization performance for forward addition with index hints on easy carry examples of length 50. **(b)** illustrates the generalization performance for reverse addition with index hints on easy carry examples of length 50. Easy carry examples consist of addition questions where the two numbers are sampled randomly and independently, which is the setting considered in prior works studying addition. We see that both settings demonstrate strong length generalization, thus demonstrating the usefulness of the index hints.

The core operations allowed in RASP are: arbitrary elementwise operations over sequences (`map` and `seq_map`), and a very particular type of non-elementwise operation `kqv`, which simulates a causal Attention layer. Moreover, no control flow is allowed; all programs must be straight-line programs, with no branching or loops.

For example, suppose we want to write a causal RASP program which always outputs the second-to-last token of the input sequence, e.g. in Python: `def f(x): return x[-2]`. To represent this function in a causal sequence-to-sequence manner, we need to take in the sequence `x` and output the same sequence but shifted by 2. In pure Python, the function is `lambda x: [0]*2 + x[2:]`, where we pad the shifted `x` with 2 tokens in the front to maintain the same dimension. This lambda function can be implemented in RASP as: `lambda x: kqv(indices(x)+2, indices(x), x, equals)`. This RASP program uses the fact that positional embeddings can be specially constructed such that shift-by-two is an *elementwise* operation on these embeddings. Then, `kqv` function simulates an attention layer where the current position indices (as query) are matched to the shifted-by-2 indices (as key), and applied to input sequence `x` (as values).

Crucially, since we study autoregressive decoder-only Transformers, we must use the *causal* version of RASP, where all seq-to-seq operations are executed causally. Moreover, while RASP programs define sequence-to-sequence functions, we interpret them as sequence-to-token functions, by taking the last token of output sequence as the next-token prediction (in the standard autoregressive manner). This setting differs from most prior literature on RASP, which typically consider non-causal models, and these differences significantly change the nature of RASP programming.

G.2 RASP Specification

Here we describe how to realize RASP as a restricted subset of Python (with numpy). First, every RASP program accepts an input sequence of length n, for arbitrary $n \in \mathbb{N}$, and returns an output sequence of the exact same length— just like a transformer. The restrictions on Python are: All variables are either numpy arrays of length n ("sequences"), or binary matrices in $\{0,1\}^{n \times n}$ ("selectors"). No control flow is allowed; all programs are straight-line programs, with no branching or loops. Finally, every line of the program must be a call to either one of the core functions defined in Listing 1, or to another RASP program.

It is easy to confirm that this is equivalent to the original presentation of RASP in Weiss et al. (2021).

G.3 RASP-L: Learnable RASP

The original RASP technically allows for certain operations which are possible to represent, but not "easy" to represent or learn. For example, any arbitrarily-complex tokenwise operation $\mathbb{R} \to \mathbb{R}$ is allowed. To disallow such pathologies, we define a "learnable" subset of RASP which we call RASP-L. RASP-L enforces the following additional restrictions on RASP.

First, all non-index values, including the input and intermediate variables, must be of type `int8`. This constraint handles issues with both infinite-precision and with learnability of the tokenwise operations (since all tokenwise functions now have small finite domains and co-domains, `int8` $\to$ `int8`, and so can be easily memorized). Although disallowing floating-point operations and unbounded numbers may seem like a strong restriction, we find these are not necessary for most symbolic programs.

Moreover, token indices are treated specially. Standard RASP allows arbitrary arithmetic operations on indices, such as division-by-two: $i \mapsto \lfloor i/2 \rfloor$. However, transformers must decode index information from their positional embeddings, so any RASP-L operation on indices must be "easy" to perform on the appropriate positional embeddings. We find that empirically, such operations involving index-arithmetic are often not learned in a length-generalizing manner. This is potentially because it is difficult to learn globally valid arithmetic structures (such as division by two) from essentially local examples, i.e. short contexts. To reflect this in RASP-L, we only allow the following operations on indices: order comparisons with other indices, and computing successor/predecessor. Formally, the RASP-L core function `indices(x)` returns a special type `IndexInt`, which can take values in $\mathbb{N}$, but only allows these restricted operations. That is, we allow adding 1 to an `IndexInt`, but we do not allow adding two `IndexInts`, nor casting between `IndexInt` and `int8`.

There is one additional restriction on RASP, involving the "selector width" core operation. Selector-width in standard RASP returns the number of prior elements that are selected by a binary Attention matrix, for each token position. The return type of Selector-width in RASP-L inherits from `IndexInt`: thus it can represent unbounded numbers of selected elements, but can only operate on them in restricted ways. Moreover, every call to `sel_width` in RASP-L returns a *new type* which inherits from `IndexInt`, and these types cannot be compared to each other. That is, the sequences returned by two different calls to `sel_width` are incomparable. The reason for these restrictions, which may otherwise seem contrived, is that `sel_width` can be used to simulate `indices`, by calling it on the all-ones selector matrix. Thus, we must restrict the output of `sel_width` sufficiently to not allow bypassing the intended restrictions on index-arithmetic. There may also be more mechanistic motivations for such restrictions, since the Transformer implementation of selector width requires weights which grow linearly with sequence length (Lindner et al., 2023).

```
def map(x, func):
    return np.array([func(xi) for xi in x])

def seq_map(x , y, func):
    return np.array([func(xi, yi) for xi, yi in zip(x,y)])

def select(k, q, pred):
    s = len(k)
    A = np.zeros((s, s), dtype=bool)
    for i in range(s):
        # k_index <= q_index due to causality
        for j in range(i+1):
            A[i, j] = pred(k[j], q[i])
    return A

def sel_width(A):
    return np.dot(A, np.ones(len(A)))

def aggr(A, v):
    out = np.dot(A, v)
    norm = sel_width(A)
    return np.divide(out, norm, out=np.zeros_like(v), where=(norm != 0))

def indices(x):
    return np.arange(len(x))

def fill(x, const):
    return np.array([const] * len(x))

# Convenience function:
def kqv(k, q, v, pred):
    return aggr(select(k, q, pred), v)
```

Listing 1: RASP-L core functions.

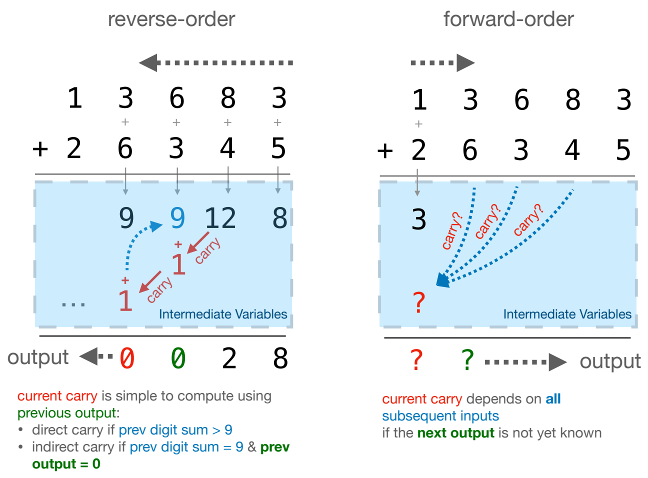

reverse-order
forward-order
1 3 6 8 3
+ 2 6 3 4 5
9 9 12 8
1 carry
1 carry
...
Intermediate Variables
output 0 0 2 8
current carry is simple to compute using previous output:
• direct carry if prev digit sum > 9
• indirect carry if prev digit sum = 9 & prev output = 0
3
carry?
carry?
carry?
?
? ? output
current carry depends on all subsequent inputs
if the next output is not yet known

Figure 9: Intuition for why reverse order addition is simpler to implement in RASP than forward order. In reverse order, the carry calculation is simple with the help of the most recent output digit, which acts like a built-in scratchpad. In forward order, the carry requires a more complex calculation involving all remaining input digits. This intuition is reflected in the RASP-L program for forward addition, which is much longer than that of reverse addition (see Listings 7 and 8).

G.4 RASP Programs

In this section, we provide the RASP-L core (1) and library (2); RASP-L programs for the tasks discussed in the paper, namely counting (3), mode (4), copy-with-unique-tokens (5), addition with reverse-order and index-hints (7); and a naive non-RASP addition algorithm (6).

```python
def where(condition, x_if, y_else):
    # equivalent to np.where(condition, x_if, y_else)
    x_masked = seq_map(x_if,    condition, lambda x, m: x if m else 0)
    y_masked = seq_map(y_else,  condition, lambda y, m: y if not m else 0)
    return seq_map(x_masked, y_masked, lambda x, y: x if y == 0 else y)

def mask(x, bool_mask, mask_val=0):
    # equivalent to x*bool_mask + default*(~bool_mask)
    return where(bool_mask, x, fill(x, mask_val))

def cumsum(bool_array):
    # returns number of previous True elements in bool_array
    return sel_width(select(bool_array, bool_array, lambda k, q: k))

def shift_right(x, n, default=0):
    # shifts sequence x to the right by n positions
    return kqv(indices(x) + n, indices(x), x, equals, default=default)

def maximum_xv(x, v):
    # returns out[i] = v[argmax(x_{<= i})]
    which_gt = select(x, x, gt)
    num_gt = sel_width(which_gt)
    highest = (num_gt == 0)
    idx_hold = cumsum(highest)  # zero-order hold of the number of the highest incides so far.
    idx_hold_if_highest = seq_map(idx_hold, highest, lambda i, h: i if h else 0)
    z = kqv(idx_hold_if_highest, idx_hold, v, equals)
    return z

def maximum(x):
    return maximum_xv(x, x)

def minimum(x):
    return -maximum(-x)

def argmax(x):
    return maximum_xv(x, indices(x))

def argmin(x):
    return argmax(-x)

def num_prev(x, queries):
    # output[i] = number of previous elements of x equal to queries[i], inclusive
    return sel_width(select(x, queries, equals))

def firsts(x, queries, default=0):
    # find the index of the first occurrence of each query[i] in x
    # out[i] := np.flatnonzero(x[:i+1] == queries[i]).min()
    NULL_VAL = -1 # special token, cannot appear in x
    has_prev = kqv(shift_right(x, 1), x, fill(x, 1), equals) # if x[i] has occured previously
    first_occ = 1-has_prev
    first_occ_only = where(first_occ, x, fill(x, NULL_VAL)) # out[i] = x[i] if first_occ[i] else NULL_VAL
    return kqv(first_occ_only, queries, indices(x), equals, default=default)

def find_last_tok(x, tok):
    # finds the index of the last occurrence of tok in sequence x (causally, inclusive)
    toks = fill(x, tok)
    matches = (x == toks)
    nprev = sel_width(select(x, toks, equals))
    nprev_matches_only = mask(nprev, matches) # zero-out non-toks
    idxs = kqv(nprev_matches_only, nprev, indices(x), equals)
    return idxs

def index_select(x, idx, default=0):
    # indexes into sequence x, via index sequence idx
    # i.e. return x[idx] if idx[i] <= i else default
    return kqv(indices(x), idx, x, equals, default=default)

def first_true(x, default=-1):
    # returns the index of the first true value in x
    seen_true = kqv(x, fill(x, 1), fill(x, 1), equals, default=0)
    first_occ = seq_map(seen_true, shift_right(seen_true, 1), lambda curr, prev : curr and not prev)
    return kqv(first_occ, fill(x, 1), indices(x), equals, default=default)


def induct(k, q, offset, default=0, null_val=-999):
    # get value of k at index of first occurrence of q (if found) plus offset
    # null_val is a special token that cannot appear in k or q; used to prevent accidental matches
    indices_to_copy = firsts(shift_right(k, offset, default=null_val), q, default=null_val)
    # copy values of k at indices_to_copy (use requested default for invalid indices)
    copied_values = index_select(k, indices_to_copy, default=default)
    return copied_values
```

Listing 2: RASP-L library functions.

```
def count(seq):
    # First, find the index of most-recent START_TOK (begninng of current sequence)
    start_idx = find_last_tok(seq, START_TOK)

    # Then, compute the start/end numbers of the current sequence
    start_nums = index_select(seq, start_idx+1)
    end_nums   = index_select(seq, start_idx+2)

    # Bool arrays: whether we're predicting the first / last tokens of the current sequence
    pred_first_pos = seq_map(indices(seq), start_idx+2, equals)
    pred_final_pos = (~pred_first_pos) & seq_map(seq, end_nums, equals)

    next_tok = where(pred_first_pos,             # if predicting the first token:
                     start_nums,                 #    next_tok = starting num
                     where(pred_final_pos,       # else if predicting the final token:
                           full(seq, END_TOK),   #    next_tok = END_TOK
                           seq + 1))             # else: next_tok = prev_tok + 1
    return next_tok
```

Listing 3: RASP-L program for Count.

```
def mode(x):
    num_prev_matching = sel_width(select(x, x, equals))
    idx = argmax(num_prev_matching)
    return index_select(x, idx)

def binary_mode(x):
    num_prev_zeros = sel_width(select(x, fill(x, 0), equals))
    num_prev_ones  = sel_width(select(x, fill(x, 1), equals))
    mode_val = seq_map(num_prev_ones, num_prev_zeros, gt)*1
    return mode_val
```

Listing 4: RASP-L program for Mode.

```
def copy_unique(seq):
    return induct(seq, seq, offset=1)

def copy_unique_ar(x):
    START, END = -1, -2
    prompt = np.concatenate(([START], x, [END], [START]))
    seq = prompt.copy()

    while seq[-1] != END:
        next_tok = copy_unique(seq)[-1]
        seq = np.concatenate((seq, [next_tok]))
    return seq

copy_unique_ar(np.array([8, 3, 4, 2, 1, 5]))
>> [-1  8  3  4  2  1  5 -2 -1  8  3  4  2  1  5 -2]
```

Listing 5: RASP-L program for Copy with unique tokens.

```
def add_illegal(inp):  # inp = array of zero-padded digits of x0, x1; returns z = x0 + x1
    num_digits = int(len(inp)/2)  # ILLEGAL: no division on index types
    z = np.zeros(num_digits)
    carry = 0
    reversed_range = range(num_digits)[::-1]  # ILLEGAL: reversal is non-causal
    for i in reversed_range:  # ILLEGAL: no for loops
        x0, x1 = inp[i], inp[num_digits+i]  # ILLEGAL: variables cannot be used as indices
        digit_sum = x0 + x1
        z[i] = (digit_sum + carry) % 10
        carry = 1 if digit_sum > 9 else 0
    return z
```

Listing 6: An addition program that is illegal in RASP-L for several reasons.

```
## Constants and helper functions
START_PROMPT = -1
PLUS = -2
EQUALS_SIGN = -3
END_RESPONSE = -5
NONE = -88

def mask_between_tokens(seq, tok0, tok1):
    seen_tok0 = has_seen(seq, full(seq, tok0))
    seen_tok1 = has_seen(seq, full(seq, tok1))
    ind_between = seq_map(seen_tok0, seen_tok1, lambda a, b: a and not b)  # ind(tok0) <= (*) < ind(tok1)
    return ind_between

def _add_safe(x, y):
    return x + y if (x >= 0) else x # preserve index-hints

## Next-token function
def next_tok_rev_addition_hinted(seq):
    prompt_mask = 1-has_seen(seq, full(seq, EQUALS_SIGN))
    second_summand_mask = mask_between_tokens(seq, PLUS, EQUALS_SIGN)
    prompt = mask(seq, prompt_mask)

    # let's first align the 1st summand with the second.
    other_summand_digit = induct(k=prompt, q=shift_right(prompt, 1), offset=1)
    pairsums = seq_map(seq, other_summand_digit, _add_safe)  # this aligns pairsums with the 2nd summand
    pairsums = mask(pairsums, second_summand_mask, NONE)
    pairsums_nh = mask(pairsums, (seq >= 0), NONE) # no hints: only keep digits

    curr_output_digit  = shift_right(seq, 1)
    curr_pairsum = induct(pairsums, shift_right(seq, 2), offset=1) # pairsum that generated curr_output_digit
    next_pairsum = induct(pairsums, seq, offset=1)

    ## START CHANGES
    direct_carry = curr_pairsum > 9  # previous sum gives carry
    indirect_carry = (curr_pairsum == 9) & (curr_output_digit == 0)  # previous sum is 9 and earlier sum gave carry
    next_tok_gets_carry = direct_carry | indirect_carry

    # (simple) index-hint computations:
    final_hint = full(seq, -100) # final hint output is always -100
    first_hint =  induct_prev(seq, full(seq, EQUALS_SIGN), offset=-2) # first hint is 2 places before '='
    next_hint = shift_right(seq, 1) + 1
    eos = (next_hint > final_hint)
    ## END CHANGES

    next_tok = next_pairsum
    next_tok += next_tok_gets_carry
    next_tok = next_tok % 10

    ## Finally, handle the case of outputing index-hints
    next_tok_is_index_hint = (seq > -100) # all index-hints are <= -100
    eos = (eos & next_tok_is_index_hint)

    next_tok = where( next_tok_is_index_hint, next_hint, next_tok)
    next_tok = where( eos, full(seq, END_RESPONSE), next_tok)
    next_tok = where( (seq == EQUALS_SIGN), first_hint, next_tok)
    return next_tok
```

Listing 7: RASP-L program for addition, with output in reverse order, and index-hints. See Section 4 for details on prompt format. For addition in forward order, the highlighted codeblock is replaced with Listing 8.

```
    ## START CHANGES
    gives_carry = tok_map(pairsums_nh, lambda _x: 1 if _x > 9 else 0)
    z = cumsum((pairsums_nh != 9) & (pairsums_nh != NONE))
    u = mask(z, gives_carry, mask_val=NONE)
    v = tok_map(u, lambda _x: _x - 1)
    chain_end_idxs = firsts(z, v, default=NONE)   # (left) ending indices of carry-chain

    curr_tok_got_carry = ((curr_pairsum % 10) != curr_output_digit)
    next_tok_inside_carry_chain =  (next_pairsum == 9) & curr_tok_got_carry
        # in the middle of a carry-chain? (NOTE: assumes the pairsums has first element 0)

    next_tok_idx = kqv(pairsums, seq, indices(seq), equals) + 1
        # which answer-position are we at? (indices aligned to pairsums)
    next_tok_chain_end = kqv( chain_end_idxs , next_tok_idx , full(seq, 1), equals, default=0)
        # does the next_tok get a carry from the end of a carry-chain?
    next_tok_gets_carry = next_tok_inside_carry_chain | next_tok_chain_end

    # (simple) index-hint computations:
    final_hint = induct_prev(seq, full(seq, EQUALS_SIGN), offset=-2) # final hint is 2 places before '='
    first_hint = full(seq, -100)
    next_hint = shift_right(seq, 1) - 1
    eos = (next_hint < final_hint)
    ## END CHANGES
```

Listing 8: The required patch to Listing 7, to produce a program for addition in forward order. This code replaces the highlighted block in Listing 7. See Section 4 for details on prompt format.