Code-Driven Inductive Synthesis: Enhancing Reasoning Abilities of Large Language Models with Sequences

Anonymous ACL submission

Abstract

Large language models make remarkable progress in reasoning capabilities. Existing works focus mainly on deductive reasoning tasks (e.g., code and math), while another type of reasoning mode that better aligns with human learning, inductive reasoning, is not well studied. We attribute the reason to the fact that obtaining high-quality process supervision data is challenging for inductive reasoning. Towards this end, we novelly employ number sequences as the source of inductive reasoning data. We package sequences into algorithmic problems to find the general term of each sequence through a code solution. In this way, we can verify whether the code solution holds for any term in the current sequence, and inject case-based supervision signals by using code unit tests. We build a sequence synthetic data pipeline and form a training dataset CodeSeq. Experimental results show that the models tuned with CodeSeq improve on both code and comprehensive reasoning benchmarks.

1 Introduction

Recent advances in AI, including openai-o1 (Zhong et al., 2024) and deepseek-r1 (DeepSeek-AI, 2025) make remarkable progress in reasoning capabilities of large language models (LLMs) (Xi et al., 2023; Xu et al., 2024; Jin et al., 2024; Franceschelli and Musolesi, 2023), such as mathematical reasoning (Ahn et al., 2024; Chen et al., 2024) and code reasoning (Liu et al., 2023; Jiang et al., 2024a).

Existing works focus mainly on deductive reasoning tasks (e.g., code and math) (Wang et al., 2024b; Lu et al., 2024), utilizing general principles and axioms to logically achieve specific conclusions. In contrast, another mode of reasoning, inductive reasoning (Han et al., 2024), involves drawing general conclusions from specific patterns. This paradigm is key to knowledge generalization and better aligns with human learning. However, limited research are conducted in this area.

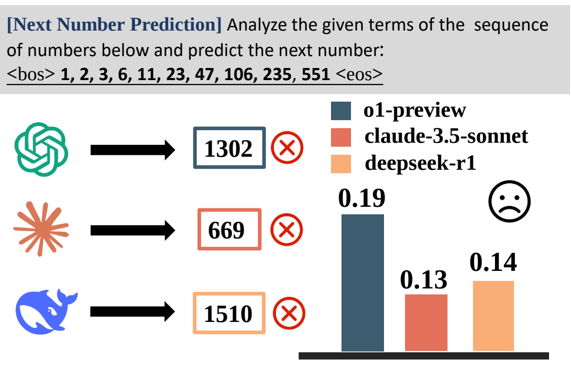


Figure 1: We select 200 sequences and prompt three powerful models for next number prediction (more details in Appendix A.1). The results demonstrate that existing LLMs perform poorly in inductive reasoning, indicating significant research potential in this area.

We attribute the reason to the fact that obtaining high-quality process supervision data (Havrilla et al., 2024) is quite challenging. In math-type problems, each step of the derivation process can be annotated and verified (Yang et al., 2024a). However, the intermediate steps in inductive reasoning are relatively open, making it difficult to determine correctness. This leads to challenges in data construction and, consequently, hardness in model learning.

In this paper, we novelly employ number sequences as the source of inductive reasoning data. Sequence problems require generalizing from previous observations to predict future elements, which can reflect the inductive ability (see Figure 1). We package sequences into algorithmic problems to find the general term of each sequence through a code solution. In this way, we can verify whether the code solution holds for any term in the current sequence, and inject case-based supervision signals via code unit tests (Hui et al., 2024). Specifically, we build a sequence synthetic data (Bauer et al., 2024) pipeline guided by code unit tests, then forming a training dataset **CodeSeq**.

The pipeline consists of three steps. **(1) Data filtering**. We scrape many sequences and their related information from websites. We use manually written rules and a language model working agent to filtrate the sequences that have enough information to be packaged into algorithmic problems. **(2) Problem generation**. We leverage the working agent to generate an algorithmic problem about the general term for each selected sequence, along with two example cases. Another guiding agent directly generates the output based on the problem description and the input of example cases to verify whether the algorithmic problem itself is correct. **(3) Supervision injection**. The working agent generates code solutions for the correct problems. We verify whether the code solution holds for any term in the sequence through code unit tests. The guiding agent provides modification suggestions and asks the working agent to regenerate the answers for the failed solutions. Through this pipeline, we inject case-based supervision signals while searching for general term code solutions for sequences, forming the complete synthetic dataset CodeSeq.

To verify the effectiveness, we apply it to perform supervised fine-tuning (SFT) on two LLMs. Experimental results show that the models tuned with CodeSeq improve on two code benchmarks and three comprehensive reasoning benchmarks.

Our contributions can be listed as follows:

- To our knowledge, we are the first to utilize sequences as the inductive reasoning data and study their impact on LLMs.
- We package the sequences into algorithmic problems, which can be injected with case-based supervision signals to improve data quality for the inductive reasoning task.
- Our synthetic data CodeSeq is proven effective for various reasoning tasks, demonstrating the potential of inductive reasoning.

2 Sequence Synthetic Data Pipeline

In this paper, we employ sequences as the source of inductive reasoning data. We package sequences into algorithmic problems to find the general term of each sequence through a code solution. In this way, we can verify whether the code solution holds for any term in the current sequence, and inject case-based supervision signals by using code unit tests. The whole pipeline consists of three steps in Figure 2. More details can refer to Appendix A.3.

2.1 Sequence Data Filtering

We scrape a large number of sequences and their related information from websites[1]. Each page on the website corresponds to a sequence and all its information, including the source, formula, general term description, and so on.

We will package the sequences into algorithmic problems by a powerful language model working agent. To ensure the accuracy of this process, we need to filter the information for each candidate sequence. We first manually wrote rules to filter out sequences with insufficient information, such as those with too few terms, or those that evolve from other sequences (requiring additional webpage links for reference). Then we prompt the working agent to self-planning (Jiang et al., 2024b) the steps for generating an algorithmic problem and self-reflecting (Wang et al., 2024c) on whether each step contains enough information. The above operations result in a batch of sequences with high information density.

2.2 Sequence Algorithmic Problem Generation and Validation

We next have the working agent generate an algorithmic problem about the general terms for each sequence, along with two example cases. Example cases provide the standard input and output cases for this algorithmic problem to help the problem solvers understand it better.

To further verify the correctness of the algorithmic problems, we utilize another powerful LLM as a guiding agent. We input the problem description and two example cases' inputs into it and let it directly output the results. By comparing these results with the ground truth outputs generated by the working agent, we can determine whether the current problem is correct. Seed sequence data is gained via this example case validation.

2.3 Case-based Supervision Signal Injection

After obtaining the seed data, we let the working agent directly generate the code solution for the algorithmic problem. Since the problem description involves the general term of a sequence, the code solution represents the computational process for the general term of the sequence. Unlike the example cases, we also set 5 to 7 test cases for each sequence to ensure the correctness of the code solution.

[1] https://oeis.org/

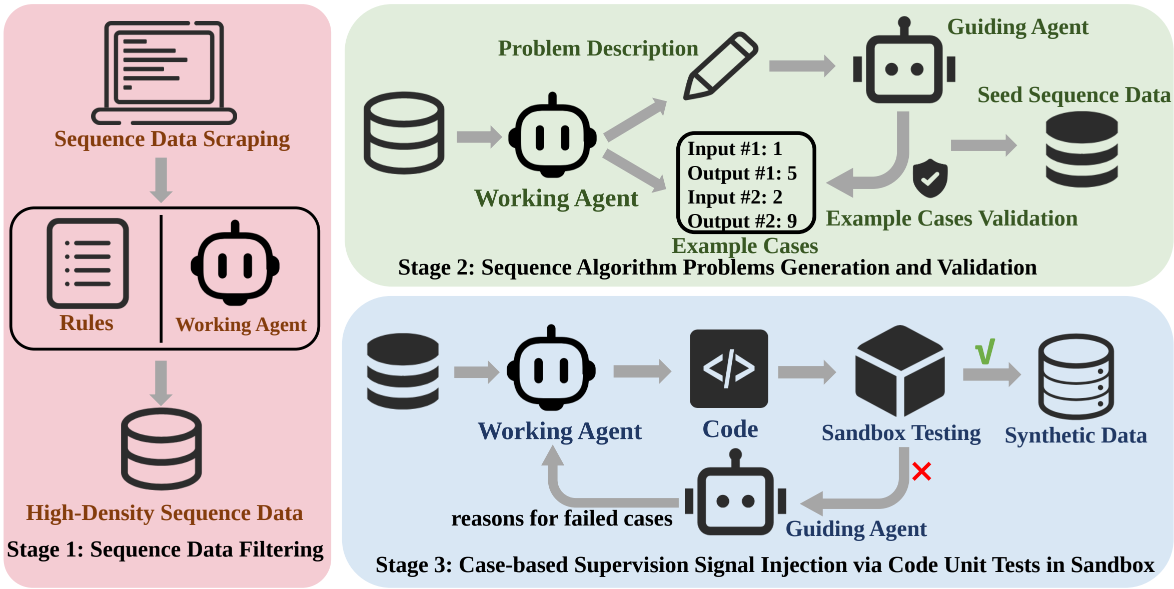


Figure 2: The sequence synthetic data pipeline consists of three steps, and then forming our CodeSeq.

Imitating previous unit tests (Hui et al., 2024), we use test cases to test the correctness of each code solution in an isolated sandbox environment. If a code solution fails on a test case, we ask the guiding agent to provide the reason for the failure. We then give that reason along with the test case back to the working agent to correct the code solution. Ultimately, through continuous self-correcting (Huang et al., 2024), we achieve a code solution that passes all the test cases.

2.4 Synthetic Data Statistics

Based on the above process, we record the code of the current version each time a modification is made and generate synthetic data for each sequence, then forming a training dataset CodeSeq. The data organization details of CodeSeq are provided in the Appendix A.4.

To ensure the diversity of the training data, we perform resampling (Hirota et al., 2024) on the problem descriptions and the initially generated code solutions. This operation is equivalent to resetting the starting point of the reasoning data, thereby obtaining a richer training corpus. We use LLaMA3-8B model as the tokenizer and the final data statistics of CodeSeq can be found in Table 1. From the table, we can see that our CodeSeq has a rich set of tokens available for training with an average of about 3 correction rounds. This proves that we effectively incorporate supervision signals into the sequence inductive reasoning data.

Sample Form	SFT form
Sample Numbers	9242
All Tokens	15.3M
Output Tokens	9M
Output Max Tokens	4273
First Hit Rate	0.52
Avg Correction Rounds	2.93
Max Correction Rounds	5

Table 1: The data statistical information of CodeSeq. 'First Hit Rate' indicates the probability that the first-generated code can pass all test cases.

3 Experiments

To prove the effectiveness of our sequence inductive reasoning synthetic data CodeSeq, we employ it to perform SFT on existing LLMs. We test its performance on code and other comprehensive reasoning benchmarks. We also explore whether CodeSeq could enhance the models' inductive reasoning capabilities.

3.1 Training, Benchmarks, and Evaluation

We conduct SFT on two widely used LLMs: LLaMA3-8B (Grattafiori et al., 2024) and Qwen2.5-7B (Qwen et al., 2025). To maintain the models' instruction-following ability (Zhu et al., 2024), we mix CodeSeq with the latest post-training (Williams and Aletras, 2024) corpus Tulu3 (Lambert et al., 2025) for SFT. We then test the tuned models on two code benchmarks: Humaneval

	Heval	MBPP	MMLU	BBH	GK
GPT4o	92.70	87.60	88.70	83.10	72.20
LLaMA3-8B	56.70	63.81	51.80	63.03	29.64
+ CodeSeq	57.32	65.79	60.62	64.40	29.71
Δ	+0.62	+1.98	+8.82	+1.37	+0.07
Qwen2.5-7B	71.34	71.59	68.23	66.05	63.29
+ CodeSeq	78.05	73.93	70.74	69.70	63.77
Δ	+6.71	+2.34	+2.51	+3.65	+0.48

Table 2: Both models have improvements on five benchmarks, finetuned by CodeSeq. ‘Heval’ and ‘GK’ represent Humaneval and GaoKaoBench respectively.

	Heval	MBPP	MMLU	BBH	GK
LLaMA3-8B	56.70	63.81	51.80	63.03	29.64
+ CodeSeq	57.32	65.79	60.62	64.40	29.71
- Tulu3	49.68	56.33	54.14	60.26	23.18
- CodeSeq	54.65	61.72	51.91	62.88	28.45
- test cases	55.90	62.47	50.88	62.59	29.53

Table 3: We conduct ablation studies with LLaMA3-8B. ‘-Tulu3’, ‘-CodeSeq’ and ‘-test cases’ mean only SFT with CodeSeq, only SFT with Tulu3 and deleting stage 3 in Figure 2, respectively.

(Chen et al., 2021) and MBPP (Austin et al., 2021), along with three comprehensive reasoning benchmarks: MMLU (Hendrycks et al., 2021), BBH (Suzgun et al., 2022), and GaoKaoBench (Zhang et al., 2024). Finally, we employ OpenCompass (Contributors, 2023), which is an LLM evaluation platform, supporting a wide range of models, to evaluate the results. More details about training and evaluating can refer to Appendix A.5.

3.2 Main Results

Table 2 shows the main results of the two models' performances on five benchmarks after finetuned by CodeSeq. We can summarize that: (1) The sequence inductive reasoning synthetic data can effectively enhance the code generation capabilities of the two LLMs. After being finetuned with CodeSeq, the models achieve an average improvement of 3.67 points on Humaneval and 2.16 points on MBPP respectively. (2) The sequence inductive reasoning synthetic data also demonstrates excellent transfer effects on comprehensive reasoning benchmarks (OOD). In particular, the LLaMA3-8B model improves by more than 8 points on MMLU. It is worth noting that although our CodeSeq data is in English, we still maintain the performance on the Chinese GaoKaoBench.

3.3 Ablation Study

We conduct ablation studies with LLaMA3-8B. From Table 3, we can conclude that: (1) If Tulu3 is not used, the model will break down in terms of instruction-following ability, and the performances on various benchmarks will significantly decline. (2) Training only with Tulu3 does not improve performance, so there will be no data leakage for the five benchmarks. (3) The synthetic data will not improve compared to the original LLaMA3-8B if the case-based supervision signals are not injected.

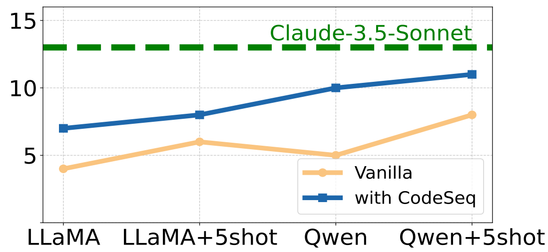


Figure 3: We respectively carry out next number prediction using LLaMA3-8B and Qwen2.5-7B before and after training, to test their inductive reasoning abilities.

3.4 CodeSeq for Next Number Prediction

We respectively carry out next number prediction using LLaMA3-8B and Qwen2.5-7B before and after training, to test their direct inductive reasoning abilities in Figure 3. It can be concluded that in the 5-shot scenario, the accuracy of the models' prediction of the next number in a sequence will increase. It is worth noting that the models trained with our CodeSeq reveal significant improvements in this task. Among them, after being trained with CodeSeq, Qwen2.5-7B's accuracy under the 5-shot setting is already close to the performance of Claude-3.5-Sonnet.

4 Conclusion

In this paper, we novelly employ number sequences as the source of inductive reasoning data. To our knowledge, we are the first to utilize sequences as such kind of data to study their impact on LLMs. We package the sequences into algorithmic problems, hence we can inject case-based supervision signals via code unit tests to improve data quality. Our synthetic data CodeSeq is proven effective for various reasoning tasks, demonstrating the potential of inductive reasoning.

Limitations

This paper takes sequences as a type of inductive reasoning data and explores the impact of this type of data on LLMs. We construct our own pipeline for generating synthetic sequence data and successfully combine it with code to insert process supervision signals. The finally formed CodeSeq training dataset is proven to have good effects on various reasoning tasks. However, this article still has two limitations: (1) Inductive reasoning tasks themselves are still in the initial stage of development. The significance of this type of task, the datasets, and the evaluation methods, etc., have not been systematically organized. Although we conduct preliminary explorations, this is a relatively novel direction and can be regarded as one of the future research works. (2) Using only sequences as the data source for inductive reasoning is relatively limited. It is expected that more synthetic training data for inductive reasoning can be obtained in the future.

Ethics Statements

The pipeline is primarily generated by deepseek-v3 and o1-preview. We obtain all the API Keys through a paid subscription. The data source is the OEIS website, which is a public website. The entire process and outcomes are free from intellectual property and ethical legal disputes.

Acknowledgments

We will finish this part in the camera-ready version.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024: Student Research Workshop, St. Julian's, Malta, March 21-22, 2024*, pages 225–237. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

André Bauer, Simon Trapp, Michael Stenger, Robert Leppich, Samuel Kounev, Mark Leznik, Kyle Chard, and Ian T. Foster. 2024. Comprehensive exploration of synthetic data generation: A survey. *CoRR*, abs/2401.02524.

Zhen Bi, Ningyu Zhang, Yinuo Jiang, Shumin Deng, Guozhou Zheng, and Huajun Chen. 2024. When do program-of-thought works for reasoning? In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 17691–17699. AAAI Press.

Kedi Chen, Qin Chen, Jie Zhou, Yishen He, and Liang He. 2024. Diahalu: A dialogue-level hallucination evaluation benchmark for large language models. *CoRR*, abs/2403.00896.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and so on. 2021. Evaluating large language models trained on code.

Elizabeth Cohn, Frida Esther Kleiman, Shayaa Muhammad, S. Scott Jones, Nakisa Pourkey, and Louise Bier. 2024. Returning value to the community through the *All of Us* research program data sandbox model. *J. Am. Medical Informatics Assoc.*, 31(12):2980–2984.

OpenCompass Contributors. 2023. Opencompass: A universal evaluation platform for foundation models. `https://github.com/open-compass/opencompass`.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating code world models with large language models guided by monte carlo tree search. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and so on. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Giorgio Franceschelli and Mirco Musolesi. 2023. On the creativity of large language models. *CoRR*, abs/2304.00008.

Rohit Girdhar, Alaaeldin El-Nouby, Zhuang Liu, Mannat Singh, Kalyan Vasudev Alwala, Armand Joulin, and Ishan Misra. 2023. Imagebind one embedding space to bind them all. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 15180–15190. IEEE.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and so on. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Simon Jerome Han, Keith J. Ransom, Andrew Perfors, and Charles Kemp. 2024. Inductive reasoning in humans and large language models. *Cogn. Syst. Res.*, 83:101155.

Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. 2024. Teaching large language models to reason with reinforcement learning. *CoRR*, abs/2403.04642.

Brett K Hayes, Evan Heit, and Haruka Swendsen. 2010. Inductive reasoning. *Wiley interdisciplinary reviews: Cognitive science*, 1(2):278–292.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. *Preprint*, arXiv:2009.03300.

Yusuke Hirota, Jerone Theodore Alexander Andrews, Dora Zhao, Orestis Papakyriakopoulos, Apostolos Modas, Yuta Nakashima, and Alice Xiang. 2024. Resampled datasets are not enough: Mitigating societal bias beyond single attributes. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 8249–8267. Association for Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024a. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *Preprint*, arXiv:2303.06689.

Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. 2024. Large language models on graphs: A comprehensive survey. *IEEE Trans. Knowl. Data Eng.*, 36(12):8622–8642.

Philip N Johnson-Laird. 1999. Deductive reasoning. *Annual review of psychology*, 50(1):109–135.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. 2025. Tulu 3: Pushing frontiers in open language model post-training. *Preprint*, arXiv:2411.15124.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Yanlin Li, Jonathan M. McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, pages 409–420. USENIX Association.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. *CoRR*, abs/2410.08196.

Mario Martínez-Magallanes, Enrique Cuan-Urquizo, Saúl E Crespo-Sánchez, Ana P Valerga, Armando Roman-Flores, Erick Ramírez-Cedillo, and Cecilia D Treviño-Quintanilla. 2023. Hierarchical and fractal structured materials: Design, additive manufacturing and mechanical properties. *Proceedings of the Institution of Mechanical Engineers, Part L: Journal of Materials: Design and Applications*, 237(3):650–666.

Gabriel Matute, Wode Ni, Titus Barik, Alvin Cheung, and Sarah E. Chasins. 2024. Syntactic code search with sequence-to-tree matching: Supporting syntactic search with incomplete code fragments. *Proc. ACM Program. Lang.*, 8(PLDI):2051–2072.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and so on. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. 2025. A survey of neural code intelligence: Paradigms, advances and beyond. *Preprint*, arXiv:2403.14734.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *Preprint*, arXiv:2210.09261.

Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip S. Yu. 2023. Deep learning for code intelligence: Survey, benchmark and toolkit. *Preprint*, arXiv:2401.00288.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024a. CLAP: learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 503–515. ACM.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2024b. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Xiao Wang, Guangyao Chen, Guangwu Qian, Pengcheng Gao, Xiao-Yong Wei, Yaowei Wang, Yonghong Tian, and Wen Gao. 2023. Large-scale multi-modal pre-trained models: A comprehensive survey. *Mach. Intell. Res.*, 20(4):447–482.

Yutong Wang, Jiali Zeng, Xuebo Liu, Fandong Meng, Jie Zhou, and Min Zhang. 2024c. Taste: Teaching large language models to translate through self-reflection. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 6144–6158. Association for Computational Linguistics.

Miles Williams and Nikolaos Aletras. 2024. On the impact of calibration data in post-training quantization and pruning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 10100–10118. Association for Computational Linguistics.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and so on. 2023. The rise and potential of large language model based agents: A survey. *CoRR*, abs/2309.07864.

Derong Xu, Wei Chen, Wenjun Peng, Chao Zhang, Tong Xu, Xiangyu Zhao, Xian Wu, Yefeng Zheng, Yang Wang, and Enhong Chen. 2024. Large language models for generative information extraction: a survey. *Frontiers Comput. Sci.*, 18(6):186357.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024a. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *Preprint*, arXiv:2409.12122.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2024b. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Trans. Software Eng.*, 50(9):2437–2457.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zonglin Yang, Li Dong, Xinya Du, Hao Cheng, Erik Cambria, Xiaodong Liu, Jianfeng Gao, and Furu Wei. 2024c. Language models as inductive reasoners. *Preprint*, arXiv:2212.10923.

Xiaotian Zhang, Chunyang Li, Yi Zong, Zhengyu Ying, Liang He, and Xipeng Qiu. 2024. Evaluating the performance of large language models on gaokao benchmark. *Preprint*, arXiv:2305.12474.

Tianyang Zhong, Zhengliang Liu, Yi Pan, Yutong Zhang, Yifan Zhou, Shizhe Liang, Zihao Wu, Yanjun Lyu, Peng Shu, Xiaowei Yu, and so on. 2024. Evaluation of openai o1: Opportunities and challenges of AGI. *CoRR*, abs/2409.18486.

Yutao Zhu, Peitian Zhang, Chenghao Zhang, Yifei Chen, Binyu Xie, Zheng Liu, Ji-Rong Wen, and Zhicheng Dou. 2024. INTERS: unlocking the power of large language models in search with instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 2782–2809. Association for Computational Linguistics.

A Appendix

A.1 Next Number Prediction

Sequences are an excellent type of data for inductive reasoning because deriving the general term formula of a sequence requires inferring an abstract, universal representation based on the specific terms of the sequence.

After the process in the Sequence Synthetic Data Pipeline Section, we obtain sequence synthetic data. We randomly select 200 sequences and conduct the next number prediction experiments with the three most powerful LLMs in terms of reasoning ability: o1-preview, claude-3.5-sonnet, and deepseek-r1. We ensure that these 200 test data are not used in the construction of CodeSeq.

We use the following prompt in Figure 4 to have it predict the next number in the given sequence.

> I will give you a sequence now.
> Please predict the next number based on the terms I provide.
> Please respond in JSON dictionary format: {“thought”: xxx, “answer”: xxx},
> where the “thought” section represents your inductive reasoning process for the sequence, and the “answer” section should directly give a number representing the final predicted answer.
>
> <bos> (the sequence) <eos>

Figure 4: The prompt for the next number prediction task.

A.2 Related Work

A.2.1 Inductive Reasoning

Reasoning can be mainly divided into two modes: deductive reasoning (Johnson-Laird, 1999) and inductive reasoning (Hayes et al., 2010). Deductive reasoning, such as well-defined tasks like mathematical and code reasoning (Wang et al., 2024b; Lu et al., 2024), utilizes general principles and axioms to achieve specific goals, pursuing logical certainty. While inductive reasoning is quite the opposite.

Inductive reasoning, involving drawing general conclusions from specific patterns, is the most universal and essential method in knowledge discovery (Han et al., 2024): (1) Deriving general conclusions from specific cases, allowing it to cover and generalize to a wider range of applications, which aligns with the human learning process. (2) Adaptive adjustments augment its reasoning ability in uncertain and complex scenarios, where inductive outcomes may not always be unique.

Despite its significance, existing works of LLMs reasoning are limited to deductive reasoning (Ahn et al., 2024; Chen et al., 2024; Liu et al., 2023; Jiang et al., 2024a). This is because obtaining high-quality process supervision data is quite challenging for inductive reasoning (Yang et al., 2024c). So this paper aims to overcome such limitation.

A.2.2 Code Reasoning

Code serves as a crucial link between humans and machines. It is ultimately converted into specific programs that can replace human labor in fulfilling diverse tasks. These programs are marked by several notable traits, including precision, logical structure, modular design, and excitability (Wan et al., 2023; Sun et al., 2025).

In the era of AI, code generation mainly consists of three stages: (1) the code embedding (Girdhar et al., 2023), (2) code pre-trained models (Wang et al., 2023), and (3) code generation in LLMs. These three stages have corresponding relationships with the development of natural language processing.

The most prominent feature of code generation is learning with execution feedback (Yang et al., 2023). Code has an inherent property of being compliant and executable. This enables compilers or interpreters to automatically produce accurate feedback. This process can be called the code unit tests (Le et al., 2022).

In the era of LLMs, there are three main methods for enhanced code generation: (1) Decoding-enhanced, that is, using methods such as self-planning (Jiang et al., 2024b) and self-filling (Martínez-Magallanes et al., 2023), and guiding the generation of code in combination with the Program of Thought (PoT) (Bi et al., 2024) technology. (2) Feedback-drive, which is similar to tree search (Matute et al., 2024; Dainese et al., 2024) and uses unit tests to provide supervision signals. (3) Natural-language (NL) guidance (Wang et al., 2024a), that is, using natural language to guide the generation of code.

In this paper, we explore injecting case-based code supervision signals to improve inductive reasoning data quality.

A.3 The Sequence Inductive Reasoning Synthetic Data Pipeline

In this section, we will provide more detailed information and more examples to clearly explain the sequence synthetic data pipeline. For the working agent, considering that we need to make frequent calls, and for cost-saving purposes, we chose deepseek-v3[2] (DeepSeek-AI et al., 2024), while for the guiding agent, we select the currently most powerful reasoning model, o1-preview[3], so that the self-correction process will be more accurate. We will demonstrate how these strong instructions-following agents work under the guidance of prompts with detailed instructions.

A.3.1 Sequence Data Filtering

We scrape a large number of sequences and their related information from the OEIS website. Each page on the website corresponds to a sequence and all its information, including the source, formula, general term description, and so on. We give an example of one OEIS webpage in Figure 5.

We need to filter the information for each candidate sequence to ensure the accuracy of the algorithmic problem generation process. We first manually wrote rules to filter out sequences with insufficient information, including: (1) those with too few terms, which will result in any powerful agent being unable to thoroughly understand the mathematical logic of the sequence. (2) those that evolve from other sequences, which will result in us being unable to crawl enough information about the current sequence from the existing website. (3) those without "mathematical" or "programming" fields, this is for the working agent to initially filter information, making it easier to generate algorithm problems. Then we prompt the working agent to self-planning the steps for generating an algorithmic problem and self-reflecting on whether each step contains enough information. This prompt are shown in Figure 6. The above operations result in a batch of sequences with high information density.

A.3.2 Sequence Algorithmic Problem Generation and Validation

We next have the working agent generate an algorithmic problem about the general terms for each sequence, along with two example cases. The prompt for problem generation is in Figure 7. Example cases provide the standard input and output cases for this algorithmic problem to help the problem solvers understand it better. We also give a generated example in Figure 8.

To further verify the correctness of the algorithmic problems, we utilize another powerful LLM as a guiding agent. We input the problem description and two example cases' inputs into it and let it directly output the results (prompt in Figure 9). By comparing these outputs with the ground truth outputs generated by the working agent, we can determine whether the current problem is correct. Seed sequence data is gained via this example case validation. Take the algorithmic problem in Figure 8 as an example, if the guiding agent outputs 7 for the first example case, it matches the ground truth. If both the answers match the ground truth in example cases, we can say that the current generated problem is correct.

A.3.3 Case-based Supervision Signal Injection

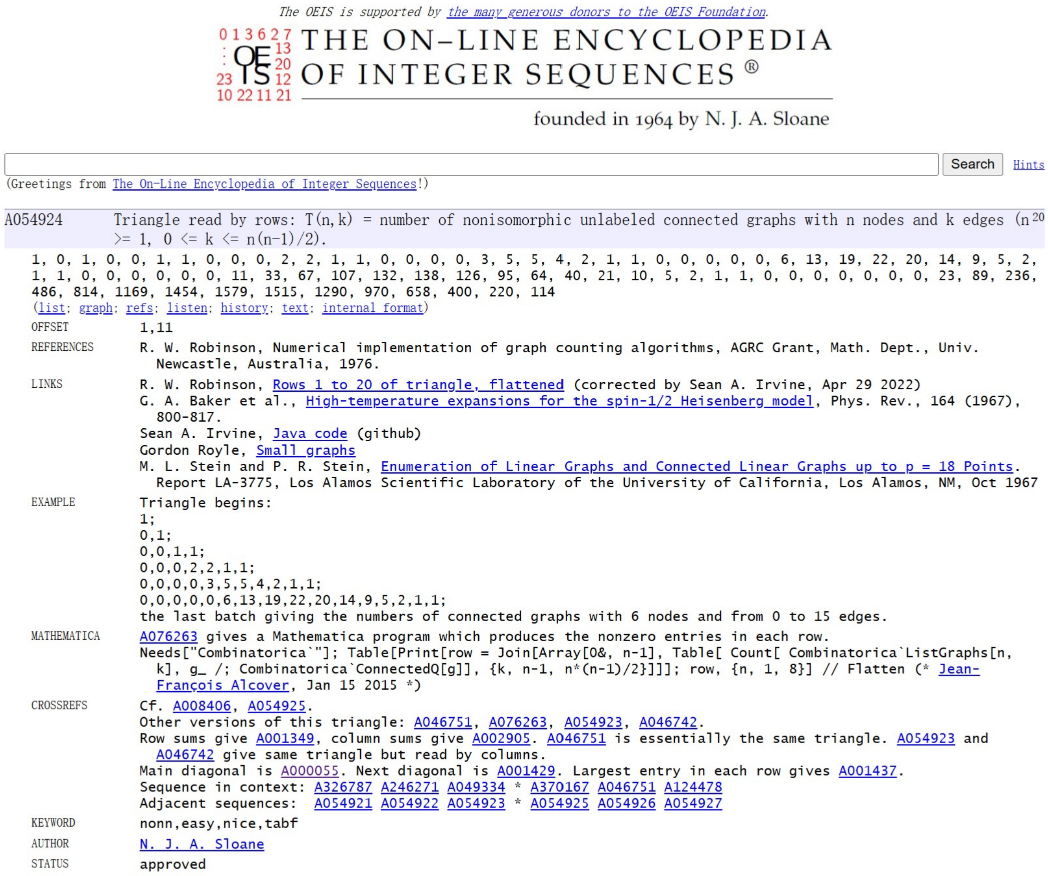
After obtaining the seed data, we let the working agent directly generate the code solution for the algorithmic problem with the prompt in Figure 10.

Since the problem description involves the general term of a sequence, the code solution represents the computational process for the general term of the sequence. Unlike the example cases, we also set 5 to 7 test cases for each sequence to ensure the correctness of the code solution, as illustrated in Figure 7.

Imitating previous unit tests (Hui et al., 2024), we use test cases to test the correctness of each code solution in an isolated sandbox environment. A sandbox environment for executing code (Li et al., 2014; Cohn et al., 2024) is a controlled and isolated setting where code can be run without affecting the host system or other applications. In this environment, the code is executed within a restricted space, preventing it from accessing sensitive resources, files, or system-level operations outside the sandbox. Sandboxes are commonly used for testing, experimentation, and security purposes, as they allow developers to execute potentially untrusted or experimental code safely. The goal is to mitigate risks, such as malware or unintentional system damage, by containing the code's actions and ensuring it can not interfere with critical parts of the system. Our code sets up a sandbox environment to safely execute user-provided Python code. It isolates the code by removing access to potentially dangerous built-in functions like open, exec, and eval, and replaces the print function with a safe version. We also redirect input and output to custom streams to capture them. The code is executed in a controlled environment with only a limited set of built-in functions available. If errors occur, they are caught and formatted with details, including the line number.

[2]https://www.deepseek.com/

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


The OEIS is supported by the many generous donors to the OEIS Foundation.

0 1 3 6 2 7 13 20 12 21 11 22 10 23 OEIS THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES ®

founded in 1964 by N. J. A. Sloane

Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

A054924 Triangle read by rows: T(n,k) = number of nonisomorphic unlabeled connected graphs with n nodes and k edges (n >= 1, 0 <= k <= n(n-1)/2). 20

1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 2, 2, 1, 1, 0, 0, 0, 0, 3, 5, 5, 4, 2, 1, 1, 0, 0, 0, 0, 0, 6, 13, 19, 22, 20, 14, 9, 5, 2, 1, 1, 0, 0, 0, 0, 0, 0, 11, 33, 67, 107, 132, 138, 126, 95, 64, 40, 21, 10, 5, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 23, 89, 236, 486, 814, 1169, 1454, 1579, 1515, 1290, 970, 658, 400, 220, 114

(list; graph; refs; listen; history; text; internal format)

OFFSET	1,11
REFERENCES	R. W. Robinson, Numerical implementation of graph counting algorithms, AGRC Grant, Math. Dept., Univ. Newcastle, Australia, 1976.
LINKS	R. W. Robinson, Rows 1 to 20 of triangle, flattened (corrected by Sean A. Irvine, Apr 29 2022) G. A. Baker et al., High-temperature expansions for the spin-1/2 Heisenberg model, Phys. Rev., 164 (1967), 800-817. Sean A. Irvine, Java code (github) Gordon Royle, Small graphs M. L. Stein and P. R. Stein, Enumeration of Linear Graphs and Connected Linear Graphs up to p = 18 Points. Report LA-3775, Los Alamos Scientific Laboratory of the University of California, Los Alamos, NM, Oct 1967
EXAMPLE	Triangle begins: 1; 0,1; 0,0,1,1; 0,0,0,2,2,1,1; 0,0,0,0,3,5,5,4,2,1,1; 0,0,0,0,0,6,13,19,22,20,14,9,5,2,1,1; the last batch giving the numbers of connected graphs with 6 nodes and from 0 to 15 edges.
MATHEMATICA	A076263 gives a Mathematica program which produces the nonzero entries in each row. Needs["Combinatorica`"]; Table[Print[row = Join[Array[0&, n-1], Table[ Count[ Combinatorica`ListGraphs[n, k], g_ /; Combinatorica`ConnectedQ[g]], {k, n-1, n*(n-1)/2}]]]; row, {n, 1, 8}] // Flatten (* Jean-François Alcover, Jan 15 2015 *)
CROSSREFS	Cf. A008406, A054925. Other versions of this triangle: A046751, A076263, A054923, A046742. Row sums give A001349, column sums give A002905. A046751 is essentially the same triangle. A054923 and A046742 give same triangle but read by columns. Main diagonal is A000055. Next diagonal is A001429. Largest entry in each row gives A001437. Sequence in context: A326787 A246271 A049334 * A370167 A046751 A124478 Adjacent sequences: A054921 A054922 A054923 * A054925 A054926 A054927
KEYWORD	nonn,easy,nice,tabf
AUTHOR	N. J. A. Sloane
STATUS	approved

Figure 5: An example of one OEIS webpage. This webpage includes the sequence, sequence offsets, sequence references, sequence links to other supplementary information, examples in the explanation process, mathematical explanations, the relationship between sequences, and so on.

Finally, we restore the system's original state after execution. This approach ensures safe, isolated execution of potentially risky code.

If a code solution fails on a test case, we ask the guiding agent to provide the reason for the failure (Figure 11). We then give that reason along with the test case back to the working agent to correct the code solution. The prompt for the working agent to regenerate and correct the code is in Figure 12. Ultimately, through continuous self-correcting, we achieve a code solution that passes all the test cases.

A.4 The CodeSeq Dataset

Based on the above process, we record the code of the current version each time a modification is made and generate synthetic data for each sequence, then form a training dataset CodeSeq.

Our training data is primarily used for model training in the post-trained stage (especially SFT), so our dataset is organized in the SFT format. A standard SFT input format in CodeSeq is shown in Figure 13, and a standard SFT output format in CodeSeq is shown in Figure 14. As with other powerful reasoning models, we use the Chain-of-Thought (CoT) technique (Yang et al., 2024b) to guide the model's deep reasoning process. In the output format, we store the CoT field and the final answer field separately.

A.5 Details for Training and Evaluation

A.5.1 LLM Backbones

We conduct SFT on two widely used LLMs: LLaMA3-8B-Instruct and Qwen2.5-7B-Instruct.

LLaMA3-8B-Instruct (Grattafiori et al., 2024) LLaMA3-8B is an advanced LLM developed by Meta, featuring 8 billion parameters. It is part of the Llama 3 family. This model is built on an optimized Transformer architecture and trained on a diverse dataset of over 15 trillion tokens. The training dataset includes a significant amount of code and covers over 30 languages, with more than 5% of the data being non-English. LLaMA3-8B is particularly designed to excel in instruction-based

I will give you a sequence and all the relevant information about it.
I would like to turn this sequence into an algorithmic problem about its general term formula.
The problem must consist of the problem statement, the format requirements for the input and output, and two examples for input and output.
Now, please first plan the steps required to generate an algorithm problem, and then evaluate whether the information I provided can meet the conditions for generating an algorithm problem by following those steps.
Please output your response in JSON dictionary format: {“step”: xxx, “step_judge”: xxx, “is_able”: xxx}.
where “step” represents the steps you planned, “step_judge” represents the thought process for each step's evaluation, and “is_able” indicates whether it is possible to generate an algorithm problem based on the provided information (True or False).

<bos> (the sequence) <eos>
[slot] (the relevant information) [slot]

Figure 6: The prompt for the working agent to conduct self-planning on the problem generation and self-reflecting on whether each step contains enough information.

tasks, making it highly effective for scenarios requiring precise and context-aware responses.

Qwen2.5-7B-Instruct (Qwen et al., 2025) Qwen2.5-7B is a powerful LLM developed by Alibaba's ModelScope team, featuring 7.6 billion parameters. It is designed to excel in various natural language processing tasks, with notable strengths in long-context understanding, multilingual support, and specialized capabilities for coding and mathematical tasks. This model supports up to 128K tokens for context understanding and can generate up to 8K tokens of text, making it highly effective for long-text generation and structured data processing. What's more, Qwen2.5-7B is trained on a massive 18T dataset.

A.5.2 Mix Training Details

To maintain the models' instruction-following ability, we mix CodeSeq with the latest post-training dataset Tulu3 (Lambert et al., 2025) for SFT.

Tulu3 is a comprehensive dataset and training framework developed by the Allen Institute to advance the post-training of LLMs. The Tulu3 dataset is designed to enhance language models' performance through SFT and reinforcement learning. It includes a mixture of data from various sources, covering a wide range of natural language processing tasks such as instruction following, mathematical reasoning, and code generation.

Due to the timeliness of Tulu3, we ensure that it is not used for any backbone model training. During the training process, we removed samples longer than 5120 tokens and excluded all samples related to mathematics and code (since we focus on code and comprehensive reasoning tasks). Finally, we retain over 800k training samples of Tulu3.

To improve the models' reasoning ability while maintaining its other capabilities, particularly instruction-following ability, we calculate the average number of tokens in the Tulu3 and CodeSeq datasets. We assign a weight ratio of 5:1 to these two datasets for mixed training. During training, we wrap all inputs and outputs with chat templates to prevent the loss of instruction-following capabilities.

A.5.3 Training Parameters

We conduct SFT on two widely used LLMs: LLaMA3-8B and Qwen2.5-7B based on InternTrainer[4] framework with 8 NVIDIA-L20Y. The training parameters are shown in Table 4.

A.5.4 Benchmarks

We test the tuned models on two code benchmarks: Humaneval (Chen et al., 2021) and MBPP (Austin et al., 2021), along with three comprehen-

[4] https://github.com/interntrainer

I will give you a sequence and all the relevant information about it.
I would like to turn this sequence into an algorithmic problem about its general term formula.
The problem must consist of the problem statement, the format requirements for the input and output, two example cases of input, output and their explanations (make it easier for problem solvers to understand), and not more than five test cases of input, output and their explanations (facilitate backend sandbox testing).
Please output your response in JSON dictionary format:
{
“description”: xxx,
“input_format”: xxx,
“output_format”: xxx,
“example cases”: [{“input1”:, “output1”:, “explanation1”:}, {“input2”:, “output2”:, “explanation2”:}] ,
“test cases”: [{“input1”:, “output1”: ,“explanation1”: },…]
}

##sequence##: <bos> (the sequence) <eos>
##relevant information##: [slot] (the relevant information) [slot]

Figure 7: The prompt for algorithmic problem generation.

sive reasoning benchmarks: MMLU (Hendrycks et al., 2021), BBH (Suzgun et al., 2022), and GaoKaoBench (Zhang et al., 2024).

Humaneval consists of 164 hand-crafted programming challenges that are comparable to simple software interview questions, each with a function signature, natural language description, and unit tests to validate the correctness of generated code.

MBPP The MBPP (Mostly Basic Python Problems) benchmark consists of around 1,000 crowd-sourced Python programming problems, each with a task description, code solution, and three automated test cases.

MMLU The MMLU (Massive Multitask Language Understanding) benchmark is a comprehensive evaluation tool designed to assess the knowledge and reasoning capabilities of LLMs across a wide range of academic and real-world subjects.

BBH The Big Bench Hard (BBH) benchmark is a collection of challenging tasks designed to evaluate the reasoning and logical abilities of LLMs.

GaoKaoBench The GAOKAO-Bench is an evaluation framework that uses Chinese college entrance examination (Gaokao) questions as its dataset to assess the language understanding and logical reasoning capabilities of LLMs. It includes a comprehensive collection of questions from 2010 to 2023. For convenience in evaluation, we select only objective questions for testing.

A.5.5 Compared Models

We chose GPT4o as the target baseline because it undergoes the most systematic evaluations across various benchmarks. Since the parameters in our models are much fewer than that of GPT4o, it is difficult for our model to outperform it in all aspects. Nevertheless, we can still present the corresponding results. This will facilitate everyone’s understanding of the gap and motivate us to strive for catching up.

GPT4o GPT4o[5] is an advanced AI model that provides more accurate and efficient language processing capabilities by OpenAI. It builds upon the strengths of previous models while incorporating new optimizations to enhance performance. With its ability to understand and generate human-like text, GPT4o aims to assist users in various tasks such as writing, problem-solving, and information retrieval.

A.5.6 OpenCompass

We employ OpenCompass[6] (Contributors, 2023), which is an LLM evaluation platform, supporting a wide range of models, to evaluate the results.

[5]https://openai.com/index/hello-gpt-4o/
[6]https://github.com/open-compass/opencompass

description: In the Kingdom of Numeria, the wise mathematician Elara has devised a unique festival where citizens create beautiful mosaic patterns using tiles of various sizes. However, to honor an ancient tradition, no tile in a mosaic can have a size that is a multiple of 3. The size of a tile is represented by a positive integer. Given a total area `n`, Elara challenges the citizens to determine how many distinct ways they can partition the area into tiles following the tradition. Each partition must consist of positive integers where none of the integers are multiples of 3, and the order of tiles does not matter (i.e., partitions are considered the same regardless of the order of tiles). Help the citizens of Numeria by writing a program that calculates the number of valid partitions for a given area `n`.Note: Two partitions are different if they have a different set of tile sizes, regardless of the order.
input_format: The input consists of a single integer `n` (1 ≤ n ≤ 5000) representing the total area to be partitioned.
output_format: Output a single integer representing the number of distinct valid partitions of `n` where no part is a multiple of 3.
example cases: [{"input1": 6, "output1": 7, "explanation1": For n=6, the valid partitions are:\\n[6], [5,1], [4,2], [4,1,1], [2,2,2], [2,2,1,1], [1,1,1,1,1,1]\\nThere are 7 valid partitions.}, …]
test cases: …

Figure 8: A generated example for one sequence algorithmic problem.

It features a wide range of capabilities, including language understanding, reasoning, coding, and long-text generation, and provides a fair and reproducible benchmark for model evaluation.

We apply the Hugging Face framework to infer the models. For code generation, the settings are: *{max-out-len: 1024, max-seq-len: 2048, batch-size:4, min-new-tokens: 50, num-return-sequences: 1, top-p: 0.9, num-beams: 10}*. For other generation, the settings are: *{max-out-len: 1024, batch-size:8, min-new-tokens: 10}*

For Humaneval, MBPP, and comprehensive reasoning benchmarks, we use pass@1, pass score, and selection accuracy as metrics separately. We ensure that all baselines are tested with the same settings for a fair comparison.

total-steps	1000
epochs	1
bzs	16
gradient-accumulation	16
micro-bsz	1
seq-len	5120
max-length-per-sample	5120
min-length	50
num-worker	4
loss-label-smooth	0
lr	1e-5
warmup-ratio	0.1
weight-decay	0.01
adam-beta1	0.9
adam-beta2	0.95
adam-eps	1e-8
fp16-initial-scale	2**14
fp16-min-scale	1
fp16-growth-interval	1000
fp16-growth-factor	2
fp16-backoff-factor	0.5
fp16-max-scale	2**24
zero1-size	8
tensor-size	1
pipeline-size	1
weight-size	1

Table 4: The training parameters.

I will now give you an algorithmic problem along with two input examples (numbers). Please directly provide the corresponding answers (numbers) for these two inputs.
Please output your response in JSON dictionary format:
{“reason1”: xxx, "answer1": xxx, “reason2”: xxx, "answer2": xxx}
where “reason1” and “reason2” represent your thought process for the two input examples, and "answer1" and "answer2" are your answers (please provide the numbers directly, with no extra output).

problem description##: [slot] (problem description) [slot]
input1##:[slot] (input1) [slot]
input2##:[slot] (input2) [slot]

Figure 9: The prompt for the guiding agent directly outputs the results so that we can determine whether the current problem is correct.

I will now give you an algorithmic problem.
Please give me your code solution with Python.
Please respond in JSON dictionary format: {“thought”: xxx, “code”: xxx},
where the “thought” section represents your reasoning process for the problem, and the “code” section should directly give a python code.

##problem description## : [slot] (problem description) [slot]
##input format## : [slot] (input format) [slot]
##output format## : [slot] (output format) [slot]
##example1## : [slot] (example1) [slot]
##example2## : [slot] (example2) [slot]

Figure 10: The prompt for the first time code solution generation.

I will now give you an algorithmic problem, its python code and one case.
Please tell me why the case works and why the code fails on the case.
Please output your response in JSON dictionary format:{“work_reason”: xxx, “failed_reason": xxx}
where “work_reason” and “failed_reason” represent why the case works and why the code fails on the case.

##problem description## : [slot] (problem description) [slot]
##input format## : [slot] (input format) [slot]
##output format## : [slot] (output format) [slot]
##example1## : [slot] (example1) [slot]
##example2## : [slot] (example2) [slot]
##python code##: [slot] (code) [slot]
##case input1## : [slot] (case input1) [slot]
##case output2## : [slot] (case output1) [slot]

Figure 11: This prompt is inputted into the guiding agent to generate the reason why such case fails on the current code solution.

I will now give you an algorithmic problem, the code solution for this problem and a test case that the code fails .
Please give me your updated code solution with Python.
Please respond in JSON dictionary format: {“thought”: xxx, “code”: xxx},
where the “thought” section represents your reasoning process for the problem, and the “code” section should directly give a python code.

##problem description## : [slot] (problem description) [slot]
##input format## : [slot] (input format) [slot]
##output format## : [slot] (output format) [slot]
##example1## : [slot] (example1) [slot]
##example2## : [slot] (example2) [slot]

##origin code##: [slot] (origin code) [slot]
#case input##: [slot] (case input) [slot]
#case output##: [slot] (case input) [slot]
#work reason##: [slot] (work reason) [slot]
#failed reason##: [slot] (failed reason) [slot]

Figure 12: The prompt for the working agent to regenerate and correct the code.

input: You are now an algorithm competition participant. Based on the following algorithm problem, you need to provide the problem-solving approach and the Python code.
Here is the problem statement: [slot] (problem description) [slot]
input format for this problem is: [slot] (input format) [slot]
output format for this problem is: [slot] (output format) [slot]
I will give you two examples to help you better understand the problem and come up with the solution approach and Python code.
example 1
input1: [slot] (input1) [slot]
output1: [slot] (output1) [slot]
The explanation for this example is: [slot] (explanation1) [slot]
example 2
input2: [slot] (input2) [slot]
output2: [slot] (output2) [slot]
The explanation for this example is: [slot] (explanation2) [slot]
When you output your solution approach, you can write an initial solution code based on your first thoughts. Then, please create examples and test them on your own. Afterward, reflect and make corrections to your code based on the examples.
Now, please provide the Python solution approach and code for this problem in {\"cot\":xxx,\"code\":xxx} dictionary format, where \"cot\" represents your solution approach, and \"code\" represents the final answer code.

Figure 13: The standard input of one sft training data in CodeSeq.

output:

{“cot”:
“The Python code for this algorithm problem, based on my initial thoughts, is:
[slot](code1)[slot]
The solution code above did not pass the following test case:
[slot](input1)[slot]
[slot](output1)[slot]
The reason this test case is valid is:
[slot](valid rason1)[slot]
The reason the code fails for this case is:
[slot](failed reason1)[slot]
Based on this test case, the revised solution code is:
[slot](code2)[slot]
…

This code passes all the test cases, so it is the correct solution.”,
“code”:
[slot](answer code)[slot] }

Figure 14: The standard output of one sft training data in CodeSeq.