CodeSeq: Enhancing Inductive Reasoning in Large Language Models via Number to Code Synthesis

Anonymous ACL submission

Abstract

Large language models make much progress in reasoning tasks, with inductive reasoning attracting increasing interest due to its better alignment with human learning. However, several challenges remain. First, existing data are difficult to extend and are widely used, lacking scalability. Second, current works do not use the inherent characteristics of inductive thinking. We address these challenges by introducing *CodeSeq*, a synthetic post-training data built from number sequences. We package number sequences into algorithmic problems to discover their general terms via code solutions to preserve case-based supervision signals. Our pipeline generates supervised finetuning data by reflecting on failed test cases and incorporating iterative corrections, thereby teaching LLMs to learn autonomous case generation and self-checking. Also, it leverages reinforcement learning with a novel Case-Synergy Solvability Scaling Reward based on both solvability, estimated from the problem pass rate, and the success rate of self-directed case generation, enabling models to learn more effectively from both successes and failures. Experimental results show that the models trained with *CodeSeq* improve on various inductive reasoning tasks and related code tasks.

1 Introduction

Recent groundbreaking advances in large-scale natural language processing (NLP) make remarkable progress in reasoning capabilities of large language models (LLMs) (Xu et al., 2025; Ke et al., 2026).

Deductive reasoning (Cai et al., 2025) and inductive reasoning (Chen et al., 2026) are two major paradigms of reasoning. The former, including mathematical (Wang et al., 2025b) and code reasoning (Yang et al., 2025b), utilizes general principles to achieve specific conclusions logically. It has already been extensively studied in recent years (Li et al., 2025c,e). In contrast, the latter involves drawing general rules from specific observations or cases. Given that this inductive mode is key to knowledge generalization and better aligns with human learning, it is relatively essential and thus attracts increasing interest.

However, research on inductive reasoning faces certain challenges. **I. Limited scalability.** Current inductive data (Chen et al., 2025b, 2026; Foundation, 2026), such as puzzles and symbols, largely rely on human-designed rules, and some contain only limited observations. These data are hard to extend and are widely used for modern LLMs (Yang et al., 2025a; Kimi, 2026). To further improve such capability through rule scalability and data diversity, it is crucial to construct high-quality data from new sources. **II. Weak inductive specificity.** Many studies merely prompt closed-source LLMs (He et al., 2025; Li et al., 2025a) or fine-tune on simple prompt–response pairs (Lee et al., 2025), but they do not leverage the inherent characteristics of inductive thinking. Hence, they cannot fundamentally enhance the such capacity of LLMs.

To address **challenge I**, we employ number sequences as a source of inductive reasoning data, requiring the identification of the general terms from the given number series. The general terms (rules) can be automatically extended through math symbolic combinations, and each theoretically corresponds to infinitely many cases, thereby ensuring the scalability (Appendix A.3). Specifically, we build a number sequence synthetic data (Bauer et al., 2024) pipeline, forming a post-training dataset denoted as *CodeSeq*. For evaluating, a general term generation (GTG) task is proposed. Towards **challenge II**, we believe the core of inductive thinking lies in *validating rules with cases*. So we aim to train LLMs for a human-like case-based reflection process. Each number (each case) in the number sequences should serve as a supervision signal. Considering LLMs' weak representation ability for pure numbers (Zhou et al., 2024; Marjieh

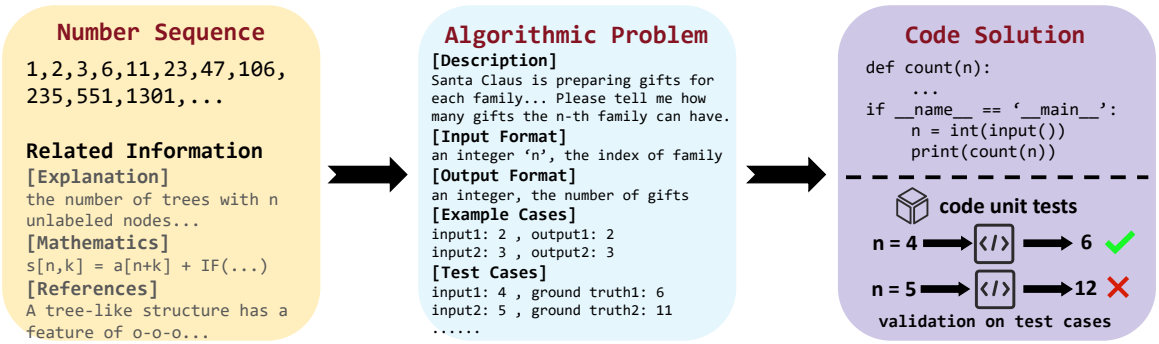


Figure 1: Number sequences are packaged into algorithmic problems with the help of their related information. We ensure each algorithmic problem takes the index of a sequence term as input and outputs the number corresponding to that index (if the input is 2, what we actually want is to output the 2-nd term of the number sequence). Therefore, the code solution is the computational process of the number sequence's general term. LLMs generate the code solution by means of the example cases (usually the first few terms, which help with thinking), and we can then verify the correctness of the solution on the test cases through code unit tests.

et al., 2025) and the inherent difficulty in expressing general terms through mathematical formulas (Barry, 2012), we package number sequences into algorithmic problems to find the general term of each sequence through a code solution (Figure 1). This allows us to use code to express the general terms of many number sequences that are hard to capture directly with conventional mathematical notations. Code execution (Hui et al., 2024) can verify whether a code solution (a general term) satisfies all cases (one number sequence) (Havrilla et al., 2024), preserving the case-based signals.

Our synthetic data pipeline has three steps. **(1) Sequence Algorithmization.** We scrape thousands of number sequences with their related information from websites. A working agent is leveraged to package each number sequence into an algorithmic problem, accompanied by two example cases. Another more powerful guiding agent directly generates the outputs of the example cases based on the problem description to validate whether the problem description meets the example cases. We divide the validated problems into two groups, preparing them separately for supervised finetuning (SFT) and reinforcement learning (RL) data. **(2) Case-based Reflection Injection.** The working agent generates code solutions for the first group. We verify whether the code solution holds for all test cases via code unit tests. The guiding agent provides modification suggestions for the failed test cases and asks the working agent to make corrections iteratively. We inject this reflection process into the Chain-of-Thought (CoT) and form the SFT data, so that LLMs can learn to generate cases autonomously and perform self-checking with them. **(3) Solvability-Estimated Selection.** For the second group, each problem is sampled multiple times to measure its pass rate, providing a solvability estimation. To ensure the model's learnability, we pick up those problems that could only be solved correctly after multiple rollouts, making up the RL data. Meanwhile, we design a Case-synergy Solvability Scaling Reward (CSSR) based on the solvability and the success rate of autonomous case generation, to guarantee a controllable learning process and further improve the model's proficiency in self-directed case generation.

To verify the effectiveness of *CodeSeq*, we train two LLMs, applying the SFT data followed by the RL data. Experimental results show that the models tuned with *CodeSeq* perform well on the in-domain GTG task and show transferability on some general inductive tasks and related code tasks.

The main contributions are listed as follows:

- This is the first work to utilize number sequences for training LLM inductive ability, providing a scalable new source of inductive reasoning data.
- We propose a synthetic data pipeline, forming a post-training dataset *CodeSeq*. The SFT data teaches LLMs to learn autonomous case generation and self-checking. The RL data and the CSSR facilitate controllable learning and further improve the proficiency in self-directed case generation.
- Our data is proven effective for various inductive reasoning tasks and related code tasks.

2 Related Work

Inductive reasoning involves drawing general conclusions from specific observations or cases (Lu, 2024), which is the most universal and essential approach in knowledge discovery (Han et al., 2024). The methods to improve such ability of LLMs can be categorized into three areas (Chen et al., 2026). (1) Post-training enhancement (Tie et al., 2025) directly using algorithms such as SFT and RL. (2) Test-time exploration (Zhang et al., 2025b) only works during the inference stage, which proposes hypotheses and continuously iterates based on validation results. (3) Data augmentation (Zhang et al., 2022) signifies enriching the LLMs' input with additional knowledge or structured signal.

Current inductive data (Li et al., 2025b,d) are difficult to extend and are widely learned, lacking scalability. Also, current works do not use the inherent characteristics of inductive thinking (He et al., 2025; Li et al., 2025a; Lee et al., 2025). To address these issues, we novelly employ number sequences as the source of data and design algorithms tailored for induction to provide high-quality post-training. More discussions are in Appendix A.1.

3 Synthetic Data Pipeline

In this paper, we employ number sequences as the source of inductive reasoning data and design a three-step synthetic data pipeline in Figure 2: Sequence Algorithmization (Section 3.1), Case-based Reflection Injection (Section 3.2), and Solvability-Estimated Selection (Section 3.3). For more detailed information on the principles, purposes, and prompt usages of each step, see Appendix A.4.

3.1 Sequence Algorithmization

This part refers to obtaining, filtering, and transforming the number sequence data into algorithmic problems, as well as verifying their correctness.

Sequence Data Filtering We scrape many number sequences and their related information from various websites. The related information includes the references, formulas, descriptions, and so on. Since we plan to package them into algorithmic problems by an LLM working agent (Wang et al., 2025a; Xu and Yan, 2026), our first need is to judge whether each number sequence has sufficient relevant information to be correctly packaged. We manually write rules to remove number sequences with insufficient information, such as those without the calculation process or evolve from other number sequences (requiring additional webpage links for references). Then we prompt the working agent to self-plan (Jiang et al., 2024) the steps for generating an algorithmic problem and self-reflect (Wang et al., 2024d) on whether each step contains enough information. The above operations result in a batch of number sequences with high information density. See more in Appendix A.4.1.

Algorithmic Problem Generation and Validation We next have the working agent generate an algorithmic problem about the general terms for each high-density number 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 themselves, we utilize a more powerful guiding agent. We feed the problem description and the two example cases' inputs into the guiding agent, which produces the outputs directly. By comparing these outputs with the ground-truth outputs in the example cases, we can determine whether the current problem description matches the example cases, thus verifying the correctness of the problem. In this way, we obtain a series of well-formatted and correct number sequence algorithmic problems, which we refer to as seed sequence data. We divide it into two groups, preparing them separately for SFT and RL data construction. More details are in Appendix A.4.2.

3.2 Case-based Reflection Injection

In this part, we introduce the construction of SFT data that incorporates a case-driven reflection reasoning process, enabling the LLMs to determine automatically whether the current code solution is correct and learn the general thinking paradigm of inductive reasoning (Appendix A.4.3).

After obtaining the seed sequence data, we let the working agent directly generate the code solutions for the algorithmic problems in the first group. Since the problem description involves a natural language story about the general term of a number sequence, the code solution represents the computational process of the general term. Like example cases, we manually select 5 to 7 items from the original number sequence randomly as test cases to validate the correctness of the code solution.

Imitating previous code unit tests (Hui et al., 2024), we use test cases to test the correctness of

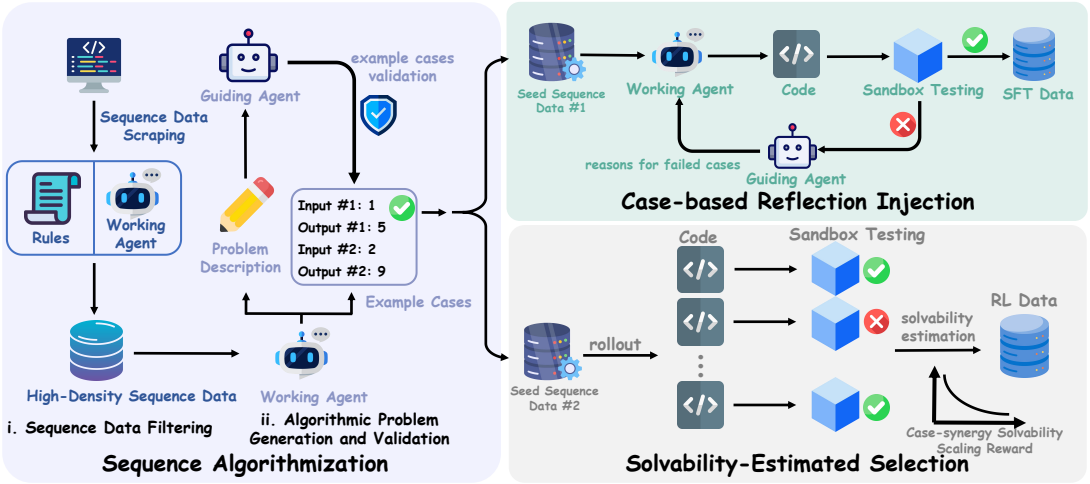


Figure 2: The pipeline of number sequence synthetic data. Sequence Algorithmization performs algorithmic problem generation and verifies the correctness of the problems. Case-based Reflection Injection generates SFT training data through case-based reflection and incorporates the process into CoT. Solvability-Estimated Selection estimates problem solvability using pass rates and selects RL training data accordingly. These two parts together form the post-training dataset for our *CodeSeq*.

each code solution in an isolated sandbox environment. If a code solution fails on a test case, we ask the guiding agent to reflect and provide the reason for the failure. We then give that reason, along with the failed test case, back to the working agent, who regenerates the code solution iteratively.

Ultimately, via continuous reflections (Huang et al., 2024; Wang et al., 2024d) and modifications, we achieve a code solution that passes all the test cases. The reflection process is injected into the CoT. To ensure data diversity, we perform multiple samplings of both the problem descriptions and the initial attempt code solutions, thereby constructing the SFT training data, named $CodeSeq_{\text{SFT}}$.

3.3 Solvability-Estimated Selection

This part introduces our method for estimating the solvability of algorithmic problems based on pass rates to filter RL training samples. We see the pass rate as an estimation of solvability. For assigning different rewards to a model depending on problem solvability, and further improving the model's proficiency in self-directed case generation for reflection, a new Case-synergy Solvability Scaling Reward (CSSR) is proposed to assist the RL training process. In this way, we hope to improve the upper limit of inductive reasoning further. More introductions are in Appendix A.4.4.

We first estimate the difficulty. We perform N rollouts on each algorithmic problem in the second group, using a model with a similar number of parameters as the base LLMs to be trained. This allows us to approximate the difficulty of the current problem as accurately as possible. The greater the difficulty of a problem, the lower its solvability for the current model. For each sampled code solution of the same problem, we also use sandbox verification to determine whether it passes all test cases. Suppose there are N_p code solution samples that pass all test cases, then the estimation of the solvability $\hat{Sov}$ can be expressed as $\frac{N_p}{N}$.

To maximize the model's potential while ensuring learnability, we mostly pick up those number sequences that could only be solved correctly after multiple rollouts. In other words, we choose that could only be solved correctly after several tries, which implies that they have an inferior solvability for the LLMs, as the RL training data. We refer to this part of the data as $CodeSeq_{\text{RL}}$.

We then design a new reward function CSSR:

$$\mathcal{R} = \begin{cases} -1, & \text{if formatted error} \\ 0, & \text{if test fails} \\ -\lambda \log(\hat{Sov} + \epsilon) + \\ (1-\lambda)\frac{N_{tc}}{N_c}, & \text{if test passes} \end{cases} \tag{1}$$

where ϵ is used to prevent the reward from becoming infinitely large when solvability approaches 0. N_c and N_{tc} denote the number of cases generated by the model's self-reflection in CoT and the number of correctly generated cases, respectively.

In the CSSR formula, if a code solution has an

	# Sample	# Pattern	Max. R tokens	# R sample	Avg. rounds	Max. rounds
$CodeSeq_{\text{SFT}}$	$5,487$	$1,271$	4.5K	$3,348$	2.56	5
$CodeSeq_{\text{RL}}$	**# Sample**	**# Pattern**	**Max. R tokens**	**Min. Sov**	**Max. Sov**	**Avg. Sov**
	$1,543$	$1,543$	1.5K	0.00	0.46	0.28

Table 1: Data statistics of our post-training datasets *CodeSeq*. '# Sample', '# Pattern', and 'Max. R tokens' mean the number of training samples, the number of sequence patterns, and the maximum response length, respectively. '# R sample' represents the number of training samples with case-based reflection, for which we record the average and maximum reflection rounds.

incorrect format, the reward is set to -1. If any of the test case fails, the reward is set to 0. If one code solution passes all test cases, the reward is divided into two components. The first part is the scaling of solvability, in which the logarithmic function is used for smoothing, stabilizing the model's learning and convergence: the lower the solvability, the higher the reward is. The second part represents the success rate of autonomous case generation in CoT. λ is employed to balance the two components.

3.4 Synthetic Data Statistics

Table 1 shows the data statistics of *CodeSeq*, which demonstrates the diversity of our data. The SFT and RL data together encompass about $3,000$ number sequence patterns, capturing the underlying rules that govern how each number sequence general term is formed, and thus reflecting diverse inductive patterns. The model can enhance its inductive reasoning ability through such a wide variety of patterns. The maximum response length for the SFT and RL training samples is 4.5K, which can almost be adapted to the training of all open-source models. In the SFT data, it has an average of about 2.56 reflection rounds. This proves that we effectively incorporate case-based reflection signals into the number sequence inductive reasoning data. In the RL data, we select the majority of algorithmic problems with solvability in the range of $(0, 0.46]$. This approach ensures the difficulty of getting the right code solutions, while also maintaining the potential for learnable possibilities. Other details and statistics are in Appendix A.5.

4 Experiments

To prove the effectiveness of our synthetic data *CodeSeq*, we employ it to perform SFT and RL on existing open-sourced LLMs. We test its performance on the in-domain GTG task, three general inductive reasoning benchmarks, and three related code reasoning benchmarks.

4.1 Training Implementation

We conduct training on two widely used LLM backbones: `LLaMA3-8B` (Grattafiori et al., 2024) and `Qwen3-8B` (Yang et al., 2025a). To maintain the models' instruction-following (Zhu et al., 2024) and other inherent abilities when SFT, we mix $CodeSeq_{\text{SFT}}$ with the latest post-training corpus Tülu 3 (Lambert et al., 2025). During the RL stage, to preserve the general capabilities of the LLMs as much as possible, we train the two models on $CodeSeq_{\text{RL}}$, applying the GRPO algorithm (Ramesh et al., 2024). To avoid randomness, we train the models with different seeds and take the averaged results. About LLM backbones and training parameters are in Appendix A.6.

4.2 Benchmarks and Evaluation

In the main experiments, to evaluate two LLMs after training on the GTG task, we construct 200 new algorithmic problems to serve as our test set, which is also validated via code unit tests. To verify whether our data can improve the general inductive reasoning ability of LLMs, we assess on three general inductive reasoning tasks under an OOD setting: symbolic induction DEER (Yang et al., 2024b), list induction ListFunctions (Li et al., 2025b), and program induction CodeARC (Wei et al., 2025). Since the number sequences are converted into code problems, we also utilize three related code benchmarks (OOD): Humaneval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2024). Finally, we utilize OpenCompass (Contributors, 2023) to help evaluate. More information is in Appendix A.6.

4.3 Main Results

Table 2 presents the results of the two models, `LLaMA3-8B` and `Qwen3-8B` on seven benchmarks after training with *CodeSeq*. We can draw the following conclusions. (1) Both base models significantly improve the in-domain GTG task after training with *CodeSeq*. This explains the back-

Models	In-domain	OOD Inductive Tasks			OOD Code Tasks		
	GTG	DEER	ListFun*	CodeArc	Heval*	MBPP	LCBench*
`InternLM2.5-7B`	14.74	42.38	18.22	15.59	67.68	61.86	16.11
`InternLM2.5-20B`	15.82	48.12	20.10	20.80	73.78	68.43	17.95
`DeepSeek-R1-7B`	27.87	53.90	35.52	43.77	84.39	-	16.66
`DeepSeek-R1-32B`	29.75	60.14	39.63	45.82	85.22	-	24.33
`LLaMA3-8B`	9.27	36.05	17.43	16.06	59.15	63.81	11.26
w/ *CodeSeq*$_{\text{SFT}}$	13.44$_{\uparrow}$	37.44$_{\uparrow}$	20.02$_{\uparrow}$	16.52$_{\uparrow}$	65.24$_{\uparrow}$	65.79$_{\uparrow}$	15.14$_{\uparrow}$
w/ *CodeSeq*$_{\text{RL}}$	40.35$_{\uparrow}$	39.28$_{\uparrow}$	21.46$_{\uparrow}$	18.11$_{\uparrow}$	62.80$_{\uparrow}$	67.29$_{\uparrow}$	14.33$_{\uparrow}$
w/ *CodeSeq*	44.22$_{\uparrow}$	40.15$_{\uparrow}$	22.90$_{\uparrow}$	19.74$_{\uparrow}$	65.85$_{\uparrow}$	68.45$_{\uparrow}$	16.14$_{\uparrow}$
Δ	+34.95	+4.10	+5.47	+3.68	+6.70	+4.64	+4.88
`Qwen3-8B`	45.63	55.22	45.66	51.20	88.43	69.80	50.12
w/ *CodeSeq*$_{\text{SFT}}$	48.52$_{\uparrow}$	59.78$_{\uparrow}$	46.87$_{\uparrow}$	54.92$_{\uparrow}$	89.19$_{\uparrow}$	71.25$_{\uparrow}$	51.44$_{\uparrow}$
w/ *CodeSeq*$_{\text{RL}}$	71.63$_{\uparrow}$	58.39$_{\uparrow}$	47.43$_{\uparrow}$	53.10$_{\uparrow}$	89.95$_{\uparrow}$	70.25$_{\uparrow}$	52.70$_{\uparrow}$
w/ *CodeSeq*	75.23$_{\uparrow}$	60.34$_{\uparrow}$	47.54$_{\uparrow}$	55.56$_{\uparrow}$	90.72$_{\uparrow}$	71.32$_{\uparrow}$	52.83$_{\uparrow}$
Δ	+29.60	+5.12	+1.88	+4.36	+2.29	+1.52	+2.71

Table 2: Results of `LLaMA3-8B` and `Qwen3-8B` on our GTG test set and other six OOD benchmarks after training with *CodeSeq*. 'Δ' indicates the performance difference between the trained models and the base models. We use abbreviations for some benchmark names and mark them with '*'.

bones' neglect of the number sequence data. Although problems are transformed into code formats that are more familiar to LLMs, base models still perform worse on them than on the other three inductive tasks. (2) *CodeSeq* enables the two models to achieve good transfer performances on OOD inductive tasks and code tasks (with overall improvement of 4.91 and 2.98 points for each model). This indicates that our case-based reflection data fundamentally improves the inductive reasoning ability of LLMs, while the code format further enhances code reasoning capability. This phenomenon, to some extent, demonstrates that inductive reasoning is the key to knowledge generalization. (3) Base models trained with *CodeSeq* could achieve performance in GTG and certain OOD tasks comparable to models with three times more parameters. In the GTG task, our small-parameter models significantly outperform `InternLM2.5-20B` and `DeepSeek-R1-32B` after training. At the same time, our *CodeSeq* enables `Qwen3-8B` to surpass `DeepSeek-R1-32B` on DEER, and makes `LLaMA3-8B` outperform `Internlm2.5-20B` on ListFunctions and MBPP.

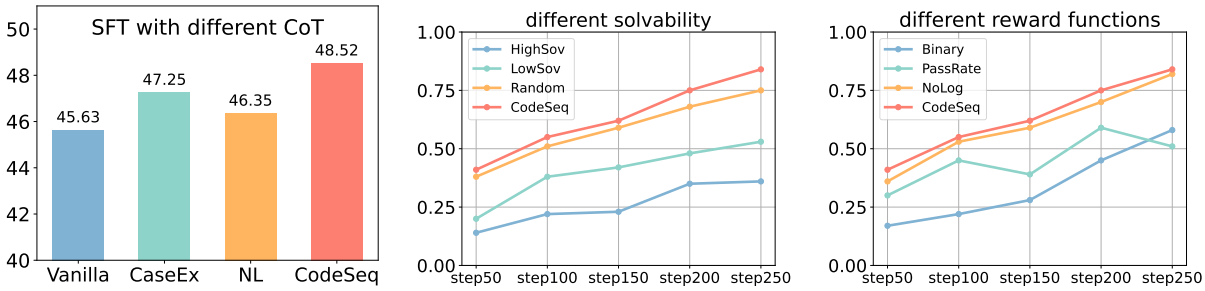
4.4 Abaltion Studies

To demonstrate the effectiveness of our synthetic data, we conduct ablation studies separately during the SFT and RL stages in Table 3. We choose three

	GTG	DEER	LCB*
`LLaMA3-8B`	9.27	36.05	11.26
w/ *CodeSeq*$_{\text{SFT}}$	13.44	37.44	15.14
w/o Tülu 3. (SFT)	8.28	35.09	12.12
w/ Tülu 3. (SFT)	9.35	35.87	10.88
w/o Reflec. (SFT)	10.58	36.88	13.90
w/ *CodeSeq*$_{\text{RL}}$	40.35	39.28	14.33
w/o Sov. (RL)	11.33	37.74	11.13
w/o CaseSuc. (RL)	39.22	38.77	12.50
w/o CSSR. (RL)	35.39	37.92	13.04

Table 3: Results of ablation studies on `LLaMA3-8B`. '(SFT)' and '(RL)' stand for the model only undergoing the SFT and RL process, respectively.

benchmarks: GTG, DEER, and LiveCodeBench. (1) 'w/o Tülu 3.' and 'w/ Tülu 3.' express training without Tülu 3 and just training on Tülu 3 during SFT, separately. Case studies show that Tülu 3 helps the model retain its instruction-following ability, but using such data alone does not improve the model's performance on the three tasks. (2) 'w/o Reflec.' means that during SFT, CoT without case-based reflection is used. This leads to a significant performance drop on the GTG task, indirectly demonstrating that this type of CoT is beneficial for inductive reasoning. (3) 'w/o Sov.' and 'w/o CaseSuc.' speak for directly removing the first and the second term in Formula 1 separately,



(a) different CoT (b) different sovability (c) different reward functions

Figure 3: Results on different training settings with model `Qwen3-8B`.

	GTG	ListFun*	LCB*
`LLaMA3-8B`	9.27	17.43	11.26
w/ OCL	8.89	16.55	12.12
w/ HS3P	8.13	17.80	14.73
w/ *CodeSeq*	44.22	22.90	16.14

Table 4: Results of the other two code training corpora with the model `LLaMA3-8B`.

which both lead to a drop in model performance, denoting the importance of these two components. We can clearly observe that the solvability term plays a more important role in the reward function. (4) 'w/o CSSR.' signifies just using the 'Binary' reward function to train the RL data. This can also not achieve the best performance. We will elaborate on these settings in Section 4.5.

4.5 Effects of Different Training Settings

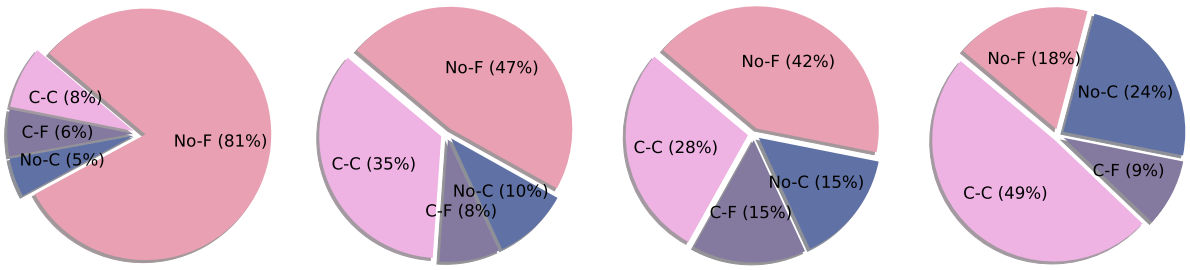
We adopt different training settings (detailed descriptions of each setting are in Appendix A.8.4) to validate the effectiveness of our synthetic data in Figure 3 on `Qwen3-8B`. (1) We investigate the impacts of different CoT formats on SFT. 'Vanilla' means the result without training. Our case-based reflection CoT ('CodeSeq') outperforms both the methods of just providing partial number sequence terms ('CaseEx') and presenting the general terms purely through natural language guidance ('NL') during reflections. (2) We explore the impacts of training samples with different solvability during RL process. We define 0 to 0.3 as low solvability ('LowSov') and 0.7 to 1.0 as high solvability ('HighSov'), partitioning the original training data accordingly while keeping the number of training samples fixed. Results demonstrate that the solvability ranged from 0 to 0.46 (our choice in Table 1), which indicates training with moderately low-solvability samples yields the highest reward during the RL process. (3) 'Binary', 'PassRate', and 'NoLog' represent the following, respectively: assigning 1 only if all test cases pass, otherwise 0; just using the solvability; and not using the logarithmic function in Formula 1. Results in the figure and ablation studies both indicate that our specially designed CSSR reward function is the optimal choice.

4.6 Is Code Format A Decisive Role?

To further demonstrate that our data can simultaneously improve the inductive reasoning and code reasoning abilities of LLMs, we additionally select two other commonly used code training corpora: OpenCodeInstruct (Ahmad et al., 2025) (OCL) and HelpSteer3-Preference (Wang et al., 2025c) (HS3P). The same amount of training data as *CodeSeq* is randomly sampled from these two datasets, respectively. We test on GTG, ListFunctions, and LiveCodeBench. The first two are induction tasks, while the last is a code task. Results in Table 4 show that applying only code corpora can improve code-related capability but not inductive ability. In summary, the code format in our data does not fundamentally affect the main results; the truly critical factor is the number sequence data.

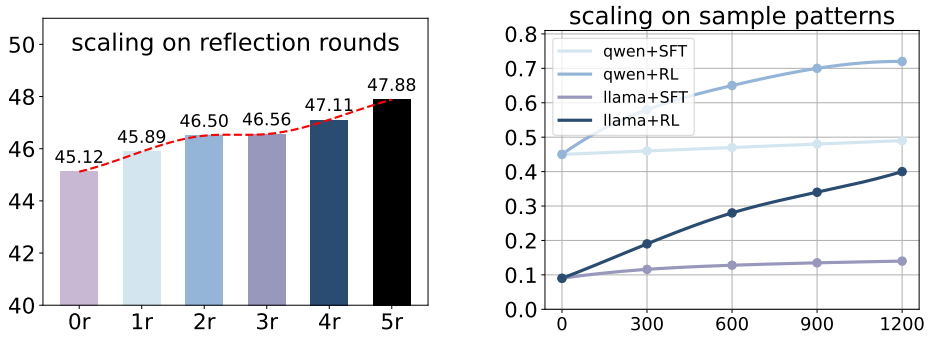
4.7 Can Models Learn to Reason with Cases?

Since our goal is to enable LLMs to learn to self-generate cases for reflection, we perform step-level inspection of the cases generated in the CoT. We investigate the relationship between LLMs' success rate of autonomous case generation and the accuracy of the code solution, thereby determining whether models can actually reason with cases. We quantify this relationship in Figure 4. Results imply that *CodeSeq* improves both the models' success rate in constructing cases and the overall problem-solving accuracy ('C-C'). The models exhibit the emergent ability (Berti et al., 2025) to guess out



(a) LLaMA-vanilla (b) LLaMA-trained (c) Qwen-vanilla (d) Qwen-trained

Figure 4: We study the relationship between case generation in CoT and solution accuracy. 'C-C' and C-F': when all cases in the CoT exist in the original number sequence, the code solution is correct or false; 'No-C' and 'No-F': when at least one real case fails to generate, the code solution is correct or false.



(a) reflection round scaling (b) number sequence pattern scaling

Figure 5: Scaling results on the number of reflection rounds and the number of number sequence patterns.

some true cases (besides the example cases), which we believe is a potential reason for the improvements on the GTG task. This indicates that the error-checking behavior is gradually internalized into the models. More explanations about how to think with cases are shown in Appendix A.8.6.

4.8 The Scaling Behaviors of *CodeSeq*

We investigate two scaling laws (Kaplan et al., 2020; Isik et al., 2024) introduced by *CodeSeq* for the two models. The results are in Figure 5. (1) We first examine how training samples with different reflection rounds affect models' SFT performance on the GTG task with `Qwen3-8B`. '0r' represents training via SFT data that is correct on the first rollout (with 0 reflection rounds). We ensure the same amount of SFT training data for different reflection rounds. As the number of reflection rounds increases, the model's performance on the GTG task improves. This suggests that utilizing more challenging samples (with more reflection rounds) is more beneficial for SFT. (2) We also dig the scaling behavior of the diversity of number sequence patterns. Each number sequence corresponds to a specific pattern. The accuracy on the GTG task improves as the number of sequence patterns increases with the two models. Results indicate that more patterns help the models raise their inductive reasoning ceiling. More explanations of this part are in Appendix A.8.7.

5 Conclusion

Faced up with current challenges of inductive reasoning, we novelly employ number sequences as the source of data and build a synthetic data pipeline, forming a post-training dataset denoted as *CodeSeq*. We package number sequences into algorithmic problems to find the general terms through code solutions, proposing a GTG task. We construct SFT data by reflecting on the failed cases and iteratively making corrections, teaching LLMs to learn autonomous case generation and self-checking. We use the pass rate as an estimate of solvability to construct the RL data, and propose a reward CSSR based on both solvability and the success rate of self-directed case generation to facilitate learning. Experimental results show that the models trained with *CodeSeq* improve on various inductive reasoning tasks and related code tasks.

Limitations

Our data has already been used in the training of several strong, well-known LLM backbones in industry. We gain the insight that data construction can serve as a unifying mechanism for diverse cognitive processes, enabling a closer simulation of real-world learning. Since number sequence problems are relatively challenging, we utilize powerful models' API to assist our data pipeline in order to efficiently obtain high-quality data, which therefore incurred considerable costs. In the future, we will further investigate how to effectively elicit case generation capability in models across a wider range of inductive tasks, and theoretically study the relationship between case generation accuracy and final task success rate.

Ethics Statements

Our data is constructed using LLMs, focusing on the scientific task of number sequences. We ensure correctness through a rigorous verification process, and there are no security concerns involved throughout the entire pipeline. We obtain all the synthetic data with API Keys through a paid subscription. The entire process and outcomes are free from intellectual property and ethical legal disputes, incorporating ethical considerations.

Reproducibility Statments

We package the code and data into a zip file as the supplementary materials in OpenReview. Once our paper is accepted, we will release all relevant materials publicly on GitHub.

Acknowledgments

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

Use of AI Assistants

We primarily use AI assistants to improve and enrich our writing, especially by leveraging LLMs to help us write taxonomy in LaTeX.

References

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. 2025. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *Preprint*, arXiv:2504.04030.

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.

Paul Barry. 2012. On the hankel transform of c-fractions. *arXiv*.

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.

Reza Bayat, Mohammad Pezeshki, Elvis Dohmatob, David Lopez-Paz, and Pascal Vincent. 2024. The pitfalls of memorization: When memorization hurts generalization. *Preprint*, arXiv:2412.07684.

Leonardo Berti, Flavio Giorgi, and Gjergji Kasneci. 2025. Emergent abilities in large language models: A survey. *Preprint*, arXiv:2503.05788.

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.

Chengkun Cai, Xu Zhao, Haoliang Liu, Zhongyu Jiang, Tianfang Zhang, Zongkai Wu, Jenq-Neng Hwang, and Lei Li. 2025. The role of deductive and inductive reasoning in large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 16780–16790. Association for Computational Linguistics.

Fredrik Carlsson, Fangyu Liu, Daniel Ward, Murathan Kurfali, and Joakim Nivre. 2025. The hyperfitting phenomenon: Sharpening and stabilizing llms for open-ended text generation. *Preprint*, arXiv:2412.04318.

Bowen Chen, Rune Sætre, and Yusuke Miyao. 2024a. A comprehensive evaluation of inductive reasoning capabilities and problem solving in large language models. In *Findings of the Association for Computational Linguistics: EACL 2024, St. Julian's, Malta, March 17-22, 2024*, pages 323–339. Association for Computational Linguistics.

Jiangjie Chen, Qianyu He, Siyu Yuan, Aili Chen, Zhicheng Cai, Weinan Dai, Hongli Yu, Qiying Yu, Xuefeng Li, Jiaze Chen, Hao Zhou, and Mingxuan Wang. 2025a. Enigmata: Scaling logical reasoning in large language models with synthetic verifiable puzzles. *Preprint*, arXiv:2505.19914.

Kedi Chen, Dezhao Ruan, Yuhao Dan, Yaoting Wang, Siyu Yan, Xuecheng Wu, Yinqi Zhang, Qin Chen,

Jie Zhou, Liang He, Biqing Qi, Linyang Li, Qipeng Guo, Xiaoming Shi, and Wei Zhang. 2026. A survey of inductive reasoning for large language models. *Preprint*, arXiv:2510.10182.

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

Si Chen, Richong Zhang, and Xu Zhang. 2025b. Geoilp: A synthetic dataset to guide large-scale rule induction. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yulong Chen, Yang Liu, Jianhao Yan, Xuefeng Bai, Ming Zhong, Yinghao Yang, Ziyi Yang, Chenguang Zhu, and Yue Zhang. 2024b. See what llms cannot answer: A self-challenge framework for uncovering LLM weaknesses. *CoRR*, abs/2408.08978.

Kewei Cheng, Jingfeng Yang, Haoming Jiang, Zhengyang Wang, Binxuan Huang, Ruirui Li, Shiyang Li, Zheng Li, Yifan Gao, Xian Li, Bing Yin, and Yizhou Sun. 2024. Inductive or deductive? rethinking the fundamental reasoning abilities of llms. *CoRR*, abs/2408.00114.

Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4.

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, 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.

Daniel Fernandes, João Pedro Matos-Carvalho, Carlos M. Fernandes, and Nuno Fachada. 2025. Deepseek-v3, gpt-4, phi-4, and llama-3.3 generate correct code for lorawan-related engineering tasks. *CoRR*, abs/2502.14926.

ARC Prize Foundation. 2026. Arc-agi-3: A new challenge for frontier agentic intelligence. *Preprint*, arXiv:2603.24621.

Panagiotis Giadikiaroglou, Maria Lymperaiou, Giorgos Filandrianos, and Giorgos Stamou. 2024. Puzzle solving using reasoning of large language models: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 11574–11591. Association for Computational Linguistics.

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.

Jan Hauke and Tomasz Kossowski. 2011. Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. *Quaestiones geographicae*, 30(2):87–93.

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.

Kaiyu He, Mian Zhang, Shuo Yan, Peilin Wu, and Zhiyu Chen. 2025. IDEA: enhancing the rule learning ability of large language model agent through induction, deduction, and abduction. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 13563–13597. Association for Computational Linguistics.

Thomas G. Holzman, James W. Pelligrino, and Robert E. Glaser. 1982. Cognitive dimensions of numerical rule induction. *Journal of Educational Psychology*, 74:360–373.

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, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Berivan Isik, Natalia Ponomareva, Hussein Hazimeh, Dimitris Paparas, Sergei Vassilvitskii, and Sanmi Koyejo. 2024. Scaling laws for downstream task performance of large language models. *CoRR*, abs/2402.04177.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

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

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

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

Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, Caiming Xiong, and Shafiq Joty. 2026. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *Preprint*, arXiv:2504.09037.

Team Kimi. 2026. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

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, and 4 others. 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*.

Kang-il Lee, Hyukhun Koh, Dongryeol Lee, Seunghyun Yoon, Minsung Kim, and Kyomin Jung. 2025. Generating diverse hypotheses for inductive reasoning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 8461–8474. Association for Computational Linguistics.

Zhikai Lei, Tianyi Liang, Hanglei Hu, Jin Zhang, Yunhua Zhou, Yunfan Shao, Linyang Li, Chenchui Li, Changbo Wang, Hang Yan, and Qipeng Guo. 2024. Gaokao-eval: Does high scores truly reflect strong capabilities in llms? *CoRR*, abs/2412.10056.

Chunyang Li, Weiqi Wang, Tianshi Zheng, and Yangqiu Song. 2025a. Patterns over principles: The fragility of inductive reasoning in llms under noisy observations. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 19608–19626. Association for Computational Linguistics.

Jiachun Li, Pengfei Cao, Zhuoran Jin, Yubo Chen, Kang Liu, and Jun Zhao. 2025b. MIRAGE: evaluating and explaining inductive reasoning process in language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Peiji Li, Jiasheng Ye, Yongkang Chen, Yichuan Ma, Zijie Yu, Kedi Chen, Ganqu Cui, Haozhan Li, Jiacheng Chen, Chengqi Lyu, Wenwei Zhang, Linyang Li, Qipeng Guo, Dahua Lin, Bowen Zhou, and Kai Chen. 2025c. Internbootcamp technical report: Boosting llm reasoning with verifiable task scaling. *Preprint*, arXiv:2508.08636.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Wei-Long Zheng, Yewen Pu, and Kevin Ellis. 2025d. Combining induction and transduction for abstract reasoning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

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.

Yunxin Li, Zhenyu Liu, Zitao Li, Xuanyu Zhang, Zhenran Xu, Xinyu Chen, Haoyuan Shi, Shenyuan Jiang, Xintong Wang, Jifang Wang, and so on. 2025e. Perception, reason, think, and plan: A survey on large multimodal reasoning models. *Preprint*, arXiv:2505.04921.

Junteng Liu, Yuanxiang Fan, Zhuo Jiang, Han Ding, Yongyi Hu, Chi Zhang, Yiqi Shi, Shitong Weng,

Aili Chen, Shiqi Chen, Yunan Huang, Mozhi Zhang, Pengyu Zhao, Junjie Yan, and Junxian He. 2025. Synlogic: Synthesizing verifiable reasoning data at scale for learning logical reasoning and beyond. *Preprint*, arXiv:2505.19641.

Zhou Lu. 2024. When is inductive inference possible? 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*.

Haozhe Ma, Zhengding Luo, Thanh Vinh Vo, Kuankuan Sima, and Tze-Yun Leong. 2025a. Highly efficient self-adaptive reward shaping for reinforcement learning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yichuan Ma, Yunfan Shao, Peiji Li, Demin Song, Qipeng Guo, Linyang Li, Xipeng Qiu, and Kai Chen. 2025b. Unitcoder: Scalable iterative code synthesis with unit test guidance. *CoRR*, abs/2502.11460.

Mikolaj Malkinski and Jacek Mandziuk. 2025. Deep learning methods for abstract visual reasoning: A survey on raven's progressive matrices. *ACM Comput. Surv.*, 57(7):166:1–166:36.

Raja Marjieh, Veniamin Veselovsky, Thomas L. Griffiths, and Ilia Sucholutsky. 2025. What is a number, that a large language model may know it? *CoRR*, abs/2502.01540.

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.

Andrew Y. Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, pages 278–287. Morgan Kaufmann.

Joongun Park, Seunghyo Kang, Sanghyeon Lee, Taehoon Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2024. Hardware-hardened sandbox enclaves for trusted serverless computing. *ACM Trans. Archit. Code Optim.*, 21(1):13:1–13:25.

Anuj Pasricha and Alessandro Roncone. 2024. The virtues of laziness: Multi-query kinodynamic motion planning with lazy methods. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, pages 14286–14292. IEEE.

Rolf Pfister and Hansueli Jud. 2025. Understanding and benchmarking artificial intelligence: Openai's o3 is not AGI. *CoRR*, abs/2501.07458.

Yu Qiao, Phuong-Nam Tran, Ji Su Yoon, Loc X. Nguyen, and Choong Seon Hong. 2025. Deepseek-inspired exploration of rl-based llms and synergy with wireless networks: A survey. *CoRR*, abs/2503.09956.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Shyam Sundhar Ramesh, Yifan Hu, Iason Chaimalas, Viraj Mehta, Pier Giuseppe Sessa, Haitham Bou Ammar, and Ilija Bogunovic. 2024. Group robust preference optimization in reward-free rlhf. *Preprint*, arXiv:2405.20304.

Zhiwen Ruan, Yun Chen, Yutao Hou, Peng Li, Yang Liu, and Guanhua Chen. 2025. Unveiling over-memorization in finetuning llms for reasoning tasks. *Preprint*, arXiv:2508.04117.

Christopher Rytting and David Wingate. 2021. Leveraging the inductive bias of large language models for abstract textual reasoning. In *Neural Information Processing Systems*.

Ute Schmid and Marco Ragni. 2015. Comparing computer models solving number series problems. In *Artificial General Intelligence*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *Preprint*, arXiv:1707.06347.

Christian R. Shelton. 2000. Balancing multiple sources of reward in reinforcement learning. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pages 1082–1088. MIT Press.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Rob Sullivan and Nelly Elsayed. 2024. Can large language models act as symbolic reasoners? *CoRR*, abs/2410.21490.

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.

Intern Team. 2025. Intern-s1: A scientific multimodal foundation model. *CoRR*, abs/2508.15763.

Guiyao Tie, Zeli Zhao, Dingjie Song, Fuyang Wei, Rong Zhou, Yurou Dai, Wen Yin, Zhejian Yang, Jiangyue Yan, Yao Su, Zhenhan Dai, Yifeng Xie, Yihan Cao, Lichao Sun, Pan Zhou, Lifang He, Hechang Chen, Yu Zhang, Qingsong Wen, and 7 others. 2025. A survey on post-training of large language models. *Preprint*, arXiv:2503.06072.

Harm van Seijen, Mehdi Fatemi, and Arash Tavakoli. 2019. Using a logarithmic mapping to enable lower discount factors in reinforcement learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14111–14121.

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.

Kun Wang, Guibin Zhang, Zhenhong Zhou, Jiahao Wu, Miao Yu, Shiqian Zhao, Chenlong Yin, Jinhu Fu, Yibo Yan, Hanjun Luo, and so on. 2025a. A comprehensive survey in llm(-agent) full stack safety: Data, training and deployment. *Preprint*, arXiv:2504.15585.

Peng-Yuan Wang, Tian-Shuo Liu, Chenyang Wang, Yi-Di Wang, Shu Yan, Cheng-Xing Jia, Xu-Hui Liu, Xin-Wei Chen, Jia-Cheng Xu, Ziniu Li, and Yang Yu. 2025b. A survey on large language models for mathematical reasoning. *Preprint*, arXiv:2506.08446.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D. Goodman. 2024b. Hypothesis search: Inductive reasoning with language models. 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. 2024c. Large-scale multi-modal pre-trained models: A comprehensive survey. *Preprint*, arXiv:2302.10035.

Yutong Wang, Jiali Zeng, Xuebo Liu, Fandong Meng, Jie Zhou, and Min Zhang. 2024d. 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.

Zhilin Wang, Jiaqi Zeng, Olivier Delalleau, Hoo-Chang Shin, Felipe Soares, Alexander Bukharin, Ellie Evans, Yi Dong, and Oleksii Kuchaiev. 2025c. Helpsteer3-preference: Open human-annotated preference data across diverse tasks and languages. *Preprint*, arXiv:2505.11475.

Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. 2025. Codearc: Benchmarking reasoning capabilities of llm agents for inductive program synthesis. *Preprint*, arXiv:2503.23145.

Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, Chenyang Shao, Yuwei Yan, Qinglong Yang, Yiwen Song, Sijian Ren, Xinyuan Hu, Yu Li, Jie Feng, Chen Gao, and Yong Li. 2025. Towards large reasoning models: A survey of reinforced reasoning with large language models. *Preprint*, arXiv:2501.09686.

Renjun Xu and Yang Yan. 2026. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *Preprint*, arXiv:2602.12430.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and so on. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian McAuley. 2025b. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *Preprint*, arXiv:2502.19411.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2024a. 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. 2024b. Language models as inductive reasoners. *Preprint*, arXiv:2212.10923.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and so on. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Charlie Zhang, Graham Neubig, and Xiang Yue. 2025a. On the interplay of pre-training, mid-training, and rl on reasoning language models. *Preprint*, arXiv:2512.07783.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, and Chen Ma. 2025b. A survey on test-time scaling in large language models: What, how, where, and how well? *Preprint*, arXiv:2503.24235.

Yue Zhang, Hongliang Fei, and Ping Li. 2022. End-to-end distantly supervised information extraction with retrieval augmentation. In *SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022*, pages 2449–2455. ACM.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. *Preprint*, arXiv:2312.07104.

Zhejian Zhou, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Scaling behavior for large language models regarding numeral systems: An example using pythia. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 3806–3820. Association for Computational Linguistics.

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 Additional Related Work

A.1.1 Inductive Reasoning

LLM Reasoning can be categorized into deductive reasoning (Johnson-Laird, 1999; Li et al., 2025c) and inductive reasoning (Hayes et al., 2010; Lu, 2024). Deductive reasoning, such as well-defined tasks like mathematical reasoning and code reasoning, utilizes general principles and axioms to achieve specific goals, pursuing logical certainty. While inductive reasoning is quite the opposite. It involves drawing general conclusions from specific observations, which is the most universal and essential approach 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. Thus, the inductive reasoning (Chen et al., 2026) of LLMs attracts increasing interest among researchers.

A.1.2 Code Generation

Code serves as a crucial link between humans and machines. Code programs are marked by several notable traits: precision, logical structure, modular design, and maintainability (Wan et al., 2023; Sun et al., 2025). In the era of AI, code generation mainly goes through three stages: (1) code embedding (Girdhar et al., 2023), (2) code pretrained models (Wang et al., 2024c), and (3) code generation in LLMs. These three stages have corresponding relationships with the development of NLP. The most prominent feature of code generation is learning with execution feedback (Yang et al., 2023). It enables compilers or interpreters to produce accurate feedback on cases automatically. This process can also be called the code unit tests (Le et al., 2022; Ma et al., 2025b), which typically runs in a manually written, isolated sandbox environment (Park et al., 2024).

In the era of LLMs, there are three main methods for enhancing code ability: (1) decoding-enhanced, that is, using methods such as self-planning (Jiang et al., 2024), self-filling (Martínez-Magallanes et al., 2023), and Program of Thought (PoT) (Bi et al., 2024) to improve generation accuracy. (2) feedback-driven, which is similar to tree search (Dainese et al., 2024; Matute et al., 2024) and

applies unit tests to provide supervised signals. (3) natural-language (NL) guidance (Wang et al., 2024a), which means deploying natural language to guide the generation process of code.

A.2 Preliminary Experiments

Number sequences are excellent for inductive reasoning because deriving the general term formula of a number sequence requires inferring an abstract and universal representation based on the specific terms of the number sequence.

After the process in Section 3, we obtain the SFT and RL number sequence synthetic data. In addition to the post-training dataset and the test set, we first construct 200 samples as a prior test set and conduct the next number prediction experiments. The next number prediction can be viewed as a simplified version of the GTG task, making it easier to assess how well existing LLMs understand number sequences and to quantify their inductive reasoning abilities. We conduct these experiments with the three powerful reasoning LLMs: `o3-mini-high`, `Claude-Sonnet-4`, and `DeepSeek-R1`. We provide each model with the first few terms of a number sequence and require it to directly output the next term.

We use the prompt in Figure 6 to have LLMs predict the next number in the given sequence, and the results are in Table 5.

`o3-mini-high`	38.0
`Claude-Sonnet-4`	43.0
`DeepSeek-R1`	32.0

Table 5: Results of the next number prediction task for the three top close-sourced LLMs.

The results show that even the most powerful existing LLMs still fail to achieve high accuracy on this task, indirectly reflecting their weakness in inductive reasoning ability.

It is worth emphasizing that although this task is a simplified version of the GTG task, it is actually more difficult. This is because the code problem descriptions in GTG contain implicit hints about the general terms of number sequences, and the transformed code tasks are more familiar to LLMs compared to direct inductive reasoning tasks.

A.3 Number Sequences

Although some recent works (Chen et al., 2024a; Cheng et al., 2024) begin to explore inductive reasoning, there still remain some challenges. In addition to the two challenges discussed in Section 1, one of the biggest challenges is that existing inductive data mostly focus on superficial regularities while lacking more complex internal patterns.

Commonly used data only emphasizes the superficial correspondences between observations. These correspondences consist of transformations of shapes, element substitutions, local pattern similarities, and so on. They focus more on enabling models to discover immediately visible 'pattern matches' rather than deep logical rules.

Take the GTG task as an example, the complexity of a rule can be quantified from the following aspects: (1) If a problem, after being transformed into an algorithmic task by a strong agent, exhibits inconsistency between its problem description and the generated example cases, this indicates that the underlying rule is difficult for the agent to comprehend. (2) If, after going through our pipeline, even a strong agent fails to produce the correct solution to the constructed algorithmic task, the rule can be considered complex for LLMs. (3) If a problem shows low solvability after multiple rollouts by the target model, this also suggests that the underlying rule is complex.

Number sequence data, although superficially just a series of consecutive numbers, often contain complex mathematical formulas or generative rules that cannot be directly inferred from their appearance. Such data is inherently clean and noise-free, ensuring correctness. Furthermore, number sequence data is highly interpretable, with strong inductive relationships between elements, making it well-suited for constructing supervised synthetic data. Lastly, superficial format similarities (Chen et al., 2024a; Cheng et al., 2024) or toy/game-like inductive signals in other inductive tasks are not essential enough for real-world applications. In contrast, our number sequence pipeline enables natural transfer to tasks like math or code reasoning, offering strong practical utility.

In the following of the current subsection, we will use separate paragraphs to provide more detailed explanations about number sequences.

The mathematics underlying number sequence induction. The regularities in number sequences often involve multi-level recursions or nonlinear operations, resulting in a depth of pattern structure (Holzman et al., 1982). Their generative rules can form complex inductive structures

The Prompt for Next Number Prediction

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 6: The prompt for the next number prediction task.

through combinations of operations, periodic nesting, and hierarchical dependencies (Schmid and Ragni, 2015), making them a deeper and more sophisticated source of data with high information density. LLMs perform well on short-term pattern recognition but show a significant drop in accuracy when dealing with complex recursions and closed-form inference, indicating that number-sequence tasks are more challenging at the abstract reasoning level (Rytting and Wingate, 2021).

Can the number sequence task truly reflect a model's inductive reasoning ability? Yes, they can. From a mathematical perspective, computing the general term of a number sequence requires the model to summarize patterns and regularities from the sequence's terms, which aligns with the process of inductive reasoning. This is consistent with the definition of inductive reasoning. At the same time, we also test our data via three general inductive reasoning benchmarks. Experimental results in Table 2 and Table 4 demonstrate that our data possesses generalization.

Some advantages over other inductive data. Our synthetic data can be fully automatically generated and verified at a lower cost compared to other tasks. For the types of inductive problems, applying the same data pipeline may lead to several issues: (1) Graph-structured data may not be easily convertible into textual forms familiar to LLMs, and graph rules cannot be executed and verified in the same straightforward way as code. (2) Textual induction (or linguistic induction) is susceptible to context, and multiple answers may be equally valid. It is hard to validate the correctness. Pure semantic matching is neither as rigorous nor as convenient as executable code for verification. (3) Puzzle-style problems (or some Visual inductive problems) are feasible, but as puzzle complexity increases (e.g., in terms of the number of constraints or the scale of variables), the cost of expansion and verification rises significantly. Because the rules of the puzzle problems rely on manual design. (4) Most relational datasets rely on a limited and fixed set of operations (e.g., map/reduce/sort). Because the underlying rules are highly enumerable, models tend to memorize template-like mappings rather than truly learning inductive reasoning.

Why do number sequences have practical use? (1) For inductive reasoning. Inductive reasoning is more aligned with how humans learn and can also enhance deductive reasoning capability. Therefore, it is a fundamental skill for LLMs. Although number sequences typically appear in everyday life within pedagogical settings, in our work, they merely serve as a starting point. Our main goal is to construct training data for inductive reasoning. Most existing LLM synthetic reasoning datasets and benchmarks focus on deductive reasoning (e.g., math and code), while the training data for inductive reasoning is relatively scarce. The main reason is that current data for inductive reasoning heavily relies on manual expert intervention, and the reasoning process is difficult to express in natural language. Our proposed pipeline can automatically generate synthetic data for inductive reasoning, thereby filling this gap. (2) For LLM reasoning. Recent LLM training efforts increasingly focus on non-deductive reasoning tasks, such as toy/puzzles/chess tasks (Giadikiaroglou et al., 2024), symbolic reasoning (Sullivan and Elsayed, 2024), and abstract logical reasoning (Malkinski and Mandziuk, 2025). While these tasks may not be as prevalent as mathematical reasoning, they are equally important for evaluating the overall reasoning ability of LLMs, and they are all fundamentally based on inductive thinking. (3) For LLM backbone training in industry. Our dataset

has already been used in the training of two famous open-source LLMs in 2025, one at the 8B scale and one exceeding 200B (we will disclose these names in the camera-ready version). In the training of these two models, our data is evaluated on over 10 reasoning benchmarks, resulting in an average improvement of 2.1 points and nearly 1 point, respectively. Therefore, it holds much practical value. (4) For real life. The number sequence task aligns with real-world applications because it comes from the sequences, patterns, and dependencies found in the real world. Solutions of the sequence problems can be directly applied to fields such as finance, technology, and healthcare, meeting human demands for automation, efficiency, and accuracy. Such sequences naturally arise from real-world observations, including economic indicators, sensor readings, medical records, and user behavior logs. Therefore, solving number sequence problems is not merely an abstract mathematical exercise, but a simple way to model and predict structured phenomena in real environments. Overall, due to the scalability, interpretability, verifiability, and practicality of number sequences, they serve as an excellent source of data.

More explanations related to the real-world scenarios. (1) Existing works lack real-world data and are primarily based on manually crafted but not automatically constructed synthetic data. Current inductive reasoning studies also lack evaluation in real-world scenarios. In contrast, number sequences provide a natural bridge between synthetic reasoning data and real-world inductive demands. Sequential numerical patterns widely exist in practical environments, such as financial market indicators, industrial sensor measurements, medical vital signs, and user activity logs. In these domains, the core challenge is often to infer hidden rules or dependencies from a limited set of observations and generalize them to unseen future cases, which closely matches the inductive reasoning objective of sequence tasks. (2) In natural language scenarios, (i) there is a large amount of redundant information, sentence perplexity is relatively high, and ambiguity may arise with strong dependence on context; (ii) the 'patterns' are weakly expressed, making it difficult for LLMs to learn and induce specific rules or conclusions effectively. Number sequences provide a clean and structured environment where the underlying patterns are explicitly reflected through numerical cases, with minimal redundant noise or contextual ambiguity. This makes them an ideal medium for LLMs to focus on rule induction and hypothesis validation.

A.4 Supplementary Information for Synthetic Data Pipeline

In this subsection, we provide more detailed information to clarify the pipeline.

For the working agent, considering the need to make frequent calls, and for cost-saving purposes, we chose `DeepSeek-V3`[1] (DeepSeek-AI et al., 2024), while for the guiding agent, we select one of the most powerful reasoning models currently, `o3-mini-high` (Pfister and Jud, 2025)[2], so that the reflection process will be more accurate. In fact, small models can also accomplish this task; they just require more attempts to obtain qualified data. Larger models enable more efficient completion of these tasks. We will demonstrate how these strong agents work under the guidance of prompts with detailed instructions.

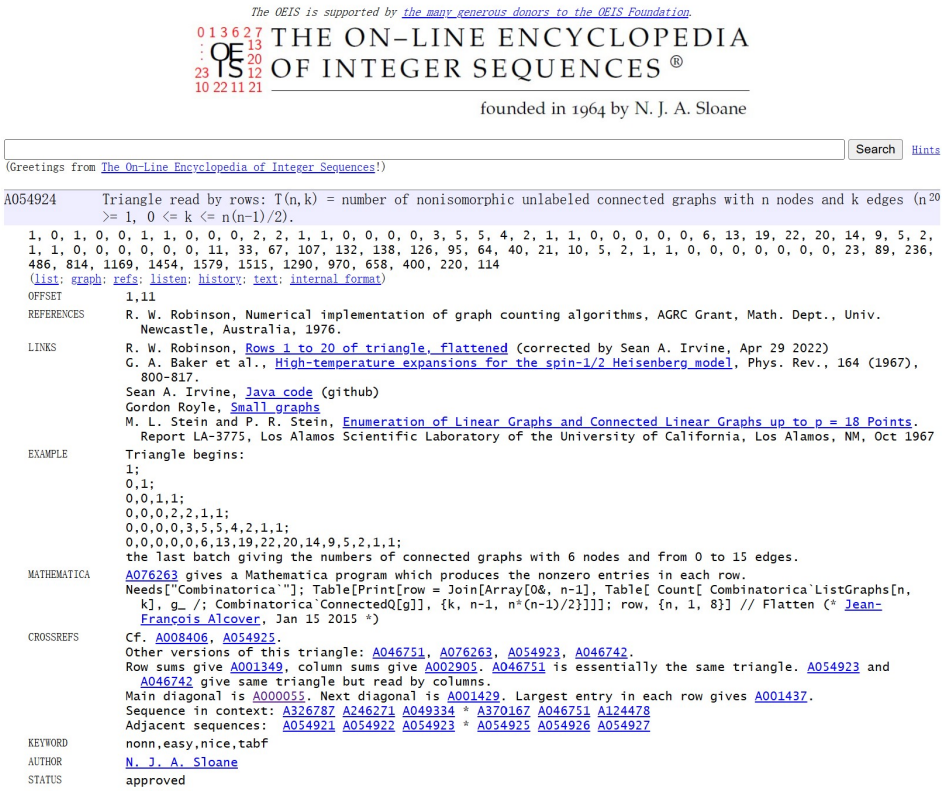
The data in this paper consists of inputting a number sequence and outputting the general term of the number sequence. However, since LLMs' weak representation ability for pure numbers and the general term of the number sequence cannot be fully described using pure mathematical language, and we aim to construct interpretable and verifiable thinking processes for LLMs to learn, we package the number sequences into algorithmic problems. Ultimately, we input an algorithmic problem and output the corresponding code solution and the thinking process.

A.4.1 Sequence Algorithmization: Sequence Data Filtering

We scrape a large number of number sequences and their related information from three math-related websites: OEIS[3], Euler[4], and Fenbi[5].

OEIS OEIS is currently the most comprehensive website for number sequences. Due to the diversity and complexity of its data, over 80% of our data comes from it. The number sequences here are all derived from basic number sequences through math symbolic composition. Here, the symbols are not limited to mathematical operators but also include a wide range of logical operators. As a

[1] https://chat.deepseek.com/
[2] https://openai.com/index/openai-o3-mini/
[3] https://oeis.org/
[4] https://projecteuler.net/
[5] https://www.fenbi.com/spa/tiku/guide/home/xingce/xingce



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

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).

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 7: An example of one OEIS webpage. The identifier in this page is 'A054924'. This webpage includes the number sequence, number sequence offsets, number sequence references, number sequence links to other supplementary information, mathematical explanations, cross-references with other number sequences, and so on.

result, the general term of a number sequence cannot always be expressed solely by a mathematical formula; however, this design ensures a high level of diversity.

Euler Project Euler is a challenge-based platform that integrates mathematics and programming, well-suited for learners who prefer practical engagement. It offers a wide range of problems related to number sequences. This portion accounts for approximately 10% of the data.

Fenbi Fenbi is a Chinese platform for national civil service examinations, featuring a large number of multiple-choice questions on number sequence patterns. These questions require the calculation of decimals and fractions, while we only need number sequence problems with integers. Therefore, the proportion of data from Fenbi is relatively small.

Take OEIS as an example, Each page on this website corresponds to a number sequence and all its information, including the source, formula, general term description, and so on. Normally, each number sequence has a unique identifier, such as 'ABCD'. Using these identifiers, we can also effectively deduplicate the data. The URL for the number sequence on the website takes the form 'https://oeis/ABCD', so we can conveniently scrape these number sequences via their URLs. We give an example of one OEIS webpage in Figure 7.

We need to filter the information for each candidate number sequence to ensure the accuracy of the algorithmic problem generation process. The lack of information or explanations prevents us from correctly converting the number sequences into algorithmic problems. We first manually write rules to filter out number sequences with insufficient information, including: (1) those with too few terms, which will result in any powerful agent being unable to understand the mathematical logic of the whole sequence thoroughly. (2) those that evolve from other number sequences, which will result in people being unable to scrape enough information about the current sequence from the existing website (requiring additional webpage links for references). (3) those without 'mathematics' or 'programming' fields. This is for the working agent to initially filter information, making it easier to generate the code problems.

We then prompt the working agent to self-plan the steps for generating a problem and self-reflect

The Prompt for Checking Enough Information

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

(Wang et al., 2024d) on whether each step contains enough information. This prompt is shown in Figure 8. The above operations result in a batch of number sequences with high information density.

A.4.2 Sequence Algorithmization: Algorithmic Problem Generation and Validation

We next have the working agent generate an algorithmic problem about the general term for each high-density number sequence, along with two example cases. The prompt for problem generation is in Figure 9. Example cases provide the standard input and output formations of this algorithmic problem to help the problem solvers understand it better. We also present an example of a number sequence algorithmic problem in Figure 10.

We ensure that the algorithmic problem takes the index of a number sequence term as input and outputs the number corresponding to that index (if the input is 2, what we actually want is to output the 2-nd term of the number sequence). Therefore, the code solution is the computational process of the general term.

To further verify the correctness of the algorithmic problems, we utilize another powerful LLM as a guiding agent. We feed the problem description and the two example cases' inputs into the guiding agent, which produces the results directly (prompt in Figure 11). By comparing these results with the outputs in the generated example cases, we can determine whether the current problem description matches the example cases, thus verifying the correctness of the problem. Take the algorithmic problem in Figure 10 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 generated problem is correct.

We still manually select 5 to 7 items from the original number sequence randomly as test cases, so that we can validate the correctness of the code solution. The test cases do not overlap with the example cases, and it is guaranteed that the second example case's next term in the original number sequence corresponds to the first test case (maintaining consistent bias).

In fact, we also test whether the generated code solutions can pass all the test cases (not just the selected ones). Additionally, we select 50 number sequences of length greater than 10 and use all of their terms as test cases. The experimental results show that the code solutions for these 50 algorithmic problems pass all corresponding test cases. This confirms the selected test cases are representative. The detailed evaluation process for the test cases is presented in the next subsection.

In this way, we obtain a series of well-formatted and correct number sequence algorithmic problems, which we refer to as seed sequence data. We divide

The Prompt for Problem Generation

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).
Please output your response in JSON dictionary format:
{
"thinking based on steps": xxx,
"description": xxx,
"input_format": xxx,
"output_format": xxx,
"example cases": [{"input1":, "output1":, "explanation1":},
{"input2":, "output2":, "explanation2":}] ,
}

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

Figure 9: The prompt for algorithmic problem generation.

the sequences that are correctly constructed into two groups, preparing them separately for SFT and RL data construction.

Why not use the same batch of data to construct both SFT and RL data? This is because we aim to cover as many types of number sequences (i.e., general term patterns) as possible.

How can we ensure that the algorithmic problems are truly packaged? (1) In the High-Density Sequence Data, the background introductions and mathematical descriptions of the current number sequence are already included. Therefore, the working agent can intuitively understand the mathematical logic of the number sequence and, while ensuring the correctness, only needs to fabricate a story to wrap it. (2) For the generated example cases, we use human-written rules to determine whether they exist in the original number sequence and calculate the corresponding offsets. (3) To ensure the consistency between the example cases and the story descriptions, we have the guiding agent generate the output answers based on the input of the example cases and the problem descriptions. We only retain the problems whose answers are generated correctly, thereby ensuring the consistency between example cases and the story description. (4) If we use the same agent that generates problem descriptions and solves the problems, this may lead to bias or limited diversity issues. So, we apply different LLM as agent cores. (5) To ensure the accuracy of the above processes, we do use powerful LLMs. Through these steps, we can rigorously ensure the alignment between the problem descriptions (stories) and all cases, as well as the correctness of all cases, and therefore no additional supervision is needed.

A.4.3 Case-based Reflection Injection

After obtaining the seed sequence data, we allow the working agent to directly generate the code solution for each algorithmic problem in the first group, as shown in Figure 12.

Since the problem description involves a story-based portrayal of the general term, the code solution represents the computational process for the general term of the number sequence.

Sandbox and Unit Tests Imitating previous unit tests (Hui et al., 2024), we apply test cases to test the correctness of each code solution in an isolated sandbox environment. A sandbox environment for executing code (Cohn et al., 2024; Li et al., 2014) 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

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 10: A generated example of a number sequence algorithmic problem.

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. Finally, we restore the system's original state after execution. This approach ensures safe, isolated execution of potentially risky code. Our sandbox code is adapted from the evaluation code of LiveCodeBench[6] (Jain et al., 2024) in the OpenCompass project (Contributors, 2023). The code unit tests within this framework are designed to assess code tasks across various scenarios, and are therefore inherently intended for general tasks. We use exactly the same settings, so there is hardly any risk that the tests provide shortcuts specific to number-sequence tasks while failing on general tasks.

If a code solution fails on a test case, we ask the guiding agent to reflect and provide the reason for the failure (Figure 13). We then give that reason, the failed test case, along with the original code, back to the working agent to regenerate and correct the code solution. The prompt for the working agent to regenerate and correct the code is in Figure 14. Ultimately, through this continuous reflection process, we achieve a code solution that passes all the test cases. We inject each step of the case-driven reflection into the CoT. To ensure data diversity, we perform multiple samplings of both the problem descriptions and the initial attempt code solutions.

The whole reflection process is strictly verified.

[6]https://github.com/open-compass/opencompass/datasets/livecodebench/

The Prompt for Example Cases' Outputs Generation

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

The Prompt for Code Solution Generation

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 12: The prompt for the code solution generation.

Each reflection corresponds to a piece of code, which is executed and validated on cases through independent unit tests, regardless of the LLM. Therefore, the correctness of the code demonstrates the correctness of the reflection. If a reflection is correct, it produces the final correct code, which can be used as ground truth for training. If the code generated after reflection fails the unit tests, further reflection is conducted on the failing test cases until the final correct answer is obtained.

A.4.4 Solvability-Estimated Selection

Our data is trained with two LLMs: `LLaMA3-8B` and `Qwen3-8B`. To more accurately estimate the difficulty of each code problem, we chose `DeepSeek-R1-7B`, a model of comparable scale, to perform the rollouts (Fernandes et al., 2025).

Before training, we randomly sample $5,00$ training instances and estimate pass rates using three models with similar parameter scales (DeepSeek, Qwen, and LLaMA), each with 8 rollouts per instance. The results show that the difference in average pass rates across models is within 15%, indicating strong consistency among models of similar size. We also analyze samples with pass rates below 0.5 (i.e., relatively difficult instances). The overlap ratio is approximately 88% between Qwen and DeepSeek, and around 91% between LLaMA and DeepSeek, suggesting low deviation and high stability in identifying difficult samples.

Moreover, this estimate is used only to select relatively challenging training samples to improve RL efficiency, rather than for precise probability modeling. For modern robust RL algorithms, the presence of a small portion of misclassified easy samples (i.e., potential selection bias) does not materially affect the overall training outcome or model capability (Qiao et al., 2025; Zhang et al., 2025a).

To mitigate randomness as much as possible, we

The Prompt for Giving Reasons

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]
##failed case input## : [slot] (failed case input) [slot]
##failed case output## : [slot] (failed case output) [slot]

Figure 13: This prompt is inputted into the guiding agent to generate the reason why such a case fails on the current code solution.

set the number of rollouts N to 32. If $N_c = 0$, the second term is directly assigned a value of 0. The rollout for each problem is independently generated by `DeepSeek-R1-7B` before training, while the rollout used in the RL algorithm is produced by the model being trained.

The design rationale of CSSR The CSSR reward function consists of a solvability term with an additional case-specific reward term. Ng et al. (1999) shows that if the additional reward exists in a weighted form, the theoretical optimal policy remains unchanged. On this basis, such multi-component rewards (Shelton, 2000) are common and do not disrupt the fundamental optimization objective. In Ma et al. (2025a), the reward is constructed using historical success rates estimated from experience, and van Seijen et al. (2019) demonstrates that applying a log function can stabilize training. So, in the first term, we apply a log-based shaping operation to the solvability.

Can we use the length of the CoT as an estimation of problem solvability? When generating algorithmic problems with the prompt shown in Figure 9, we also have it output the CoT for packaging. Some previous studies (Chen et al., 2024b; Lei et al., 2024) show that when using LLMs to construct problems, the longer the CoT, the more difficult the generated problem tends to be. So, can we use the length of the CoT to represent the solvability of a problem? Or, in other words, is there a correlation between the length of the CoT and the solvability of the problem? We record the length of the CoT for each problem as well as the pass rate of the problem to calculate the correlation between the two in Figure 15. We normalize the thinking CoT into $[0, 1]$ and compute the Pearson correlation coefficient (Cohen et al., 2009) and Spearman correlation coefficient (Hauke and Kossowski, 2011) between the two variables. The values of the two are -0.10 and -0.21, respectively, and it can be seen that the correlation between the two is not significant in this experiment. Therefore, we do not use the length of the thinking CoT to represent the solvability of a problem.

A.5 Supplementary Information of the *CodeSeq* Dataset

Based on the above process, we construct SFT and RL data with number sequences.

A standard SFT input format in *CodeSeq* is shown in Figure 16, and a standard SFT output format in *CodeSeq* is shown in Figure 17. As other powerful reasoning models, we use the CoT technique (Yang et al., 2024a) to guide the model's deep reasoning process. In the output format, we store the CoT field and the final answer field separately. As for RL data, we use the same 'input' as in Figure 16, but in the output, we only retain contents in the 'code' field.

It is worth noting that our SFT data includes a small number of short training samples (e.g., cases where a single line of 'print' suffices to express the general formula). This is intended to increase

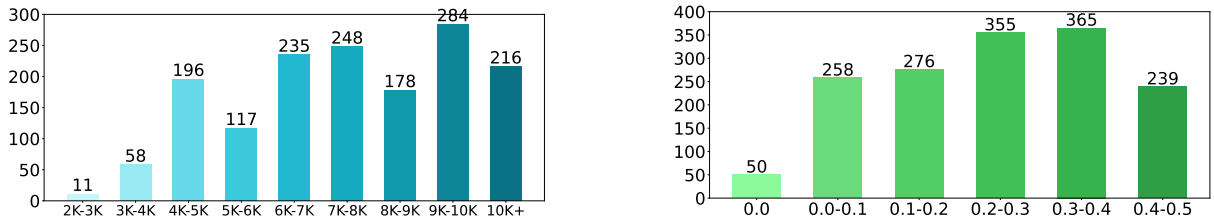
The Prompt for Code Solution Updation

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



(a) distribution of CoT length

(b) distribution of solvability

Figure 15: Exploring the relationship between the length of the thinking CoT and the solvability.

the diversity of our data, helping the model learn problem-solving primitives. There aren't many such short training samples, and this is intended to avoid bias and prevent issues such as lazy planning (Pasricha and Roncone, 2024) (making the model believe that longer outputs are better).

Why 0.46? The value 0.46 represents an approximation of 15/32. We chose this number primarily to keep the amount of RL training data around 1.5K and ensure the data quality. On one hand, we show that training with easy samples is less effective (Figure 3 (b)), so we prioritize selecting difficult samples (with solvability less than 0.5) for the model to learn from. In contrast, if we use fewer data samples (such as less than 1.2K), our prior experiments and the results in Figure 3 ((b) only hard problems) demonstrate that it is difficult to achieve optimal performance. On the other hand, the total number of samples with solvability less than 0.5 is around 1.7K. If all 1.7K data are used for training, the results on GTG are approximately 41.89 on `LLaMA3-8B`, which is lower than that of *CodeSeq*. In summary, for the choice of the solvability upper bound, our constraints are: (1) filtering out simple training samples with solvability more than 0.5; (2) the number of training samples is at least 1.2K (since performance keeps improving when the sample size is less than 1.2K); (3) choosing 0.5 yields 1.7K training samples, but the performance drops. Therefore, we chose 0.46 as the threshold, corresponding to 1.5K training samples for RL.

Data Statistics After going through the strict and complex data generation process shown in Figure 2, we perform statistical analysis for each step in Table 6. For each algorithmic problem derived from a number sequence, if it fails at any step in the pipeline, we discard that data. In the end, we retained 20K valid algorithmic problems for generating SFT and RL data. Our final post-training

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 16: The standard input of one SFT training data in *CodeSeq*.

synthetic data undergoes rigorous filtering and is of relatively high quality.

# scapped number sequence	270K
# high-density number sequence	100K
# validated problems	20K
# seed sequence data #1	10K
# seed sequence data #2	10K
# passed SFT data	2K
# resampled SFT data	5.5K
# passed RL data	4K
# seleted RL data	1.5K

Table 6: The sample statistics at each stage in the synthetic data pipeline. The data presented in this table are not the accurate values.

Pricing When constructing the synthetic data, we primarily use the API Keys of `DeepSeek-V3` and `o3-mini-high` for our experiments. The total cost is approximately USD $2,500$ and RMB 400. We invest a significant amount of funding in trial-and-error and data synthesis, which helps ensure the quality and effectiveness of the data. The expenses mentioned above are not solely for data synthesis. A small part of them is also included in the evaluation, which we will explain in detail in the experimental subsection.

A.6 Details for Training and Evaluation

A.6.1 LLM Backbones

We conduct both the SFT and RL experiments on two widely used LLMs: `LLaMA3-8B` and `Qwen3-8B` with our post-training dataset *CodeSeq*. These two LLM backbones are highly representative in the academic community.

A.6.2 Mix Training during SFT

To maintain the models' instruction-following and other inherent abilities when conducting SFT, we mix $\textit{CodeSeq}_{\text{SFT}}$ with one of the latest post-training corpora: Tülu 3 (Lambert et al., 2025).

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 17: The standard output of one SFT training data in *CodeSeq*.

Tülu 3 This post-training corpus is a comprehensive dataset developed by the Allen Institute to advance the post-training of LLMs. The Tülu 3 dataset is designed to enhance language models' performance through SFT and RL. 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. The training data of Tülu 3 comes from various sources, like the instruction dataset: FLAN v2, OpenAssistant, WildChat, and No-Robots; the math instruction dataset: Numina-Math, SciRIFF, and OpenMathInstruct; also, some code instruction datasets: CodeAlpaca; the rest of the data is from human-made instruction data with `GPT4o` and `Claude3`.

We need to ensure that it does not conclude any data related to evaluation benchmarks, preventing data leakage. During the training process, we remove samples longer than 3K tokens and exclude all training samples related to code (since code reasoning is highly relevant to our inductive tasks) with `Qwen3-8B` filtering such code-area data. Finally, we retain over 800K training samples of Tülu 3. It is worth noting that we only perform around $1,000$ steps of SFT, so not all of the 800K Tülu 3 data is trained. To improve the models' reasoning ability while maintaining their other capabilities, particularly instruction-following ability, we calculate the average number of tokens in the Tülu 3 and *CodeSeq* and assign a weight ratio of 5:1 to these two datasets for mixed training.

A.6.3 Training Implementations and Parameters

We conduct both SFT and RL on two widely used LLMs. We conduct SFT and RL based on the InternTrainer[7] framework and VERL[8] (Sheng et al., 2024) framework with 8∗NVIDIA-L20Y separately. We not only compare the performance differences of the models before and after training, but also include `InternLM2.5` and `DeepSeek-R1`, each coming in two parameter sizes, as additional baselines. The training parameters of SFT and RL are shown in Table 7 and Table 8, respectively. λ and ϵ in Formula 1 is set to 0.9 and 0.1.

A.6.4 OpenCompass

We employ OpenCompass[9] (Contributors, 2023), which is an LLM evaluation platform, supporting a wide range of models, to evaluate the results of three code benchmarks. 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.

For the GTG task, we chose `gen pass@1` (Chen et al., 2021) (where $n = 32$ in our settings) as the evaluation metric. For the three general reasoning tasks: DEER, Listfunctions, and CodeArc, we apply `METEOR`, `accuracy`, and `success rate` as evaluation metrics, respectively. For three code

[7] https://github.com/interntrainer
[8] https://github.com/volcengine/verl
[9] https://github.com/open-compass/opencompass

total-steps	$1,000$
epochs	1
bsz	16
gradient-accumulation	16
micro-bsz	1
seq-len	$5,120$
max-length-per-sample	$5,120$
min-length	50
num-worker	4
loss-label-smooth	0
lr	1e−6
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	$1,000$
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 7: The training parameters of SFT.

total-steps	300
bsz	64
actor.ppo-mini-bsz	64
promot-max-length	$2,048$
response-max-length	$2,048$
lr	1e−6
rollout.n	8
warmup-step	0.1
warmup-style	cosine
actor.ulysses-sequence-parallel-size	1
actor.use-kl-loss	True
actor.entropy-coeff	0.0
actor.kl-loss-coef	0.0
actor.ppo-micro-bsz-per-gpu	4
ref.log-prob-micro-bsz-per-gpu	64
rollout.log-prob-micro-bsz-per-gpu	64
rollout.tensor-model-parallel-size	1

Table 8: The training parameters of RL.

reasoning tasks: Humaneval and LiveCodeBench use `gen pass@1`, while MBPP utilizes `accuracy`.

Although our evaluation results may differ somewhat from the original benchmark results, for the same benchmark, we use the same evaluation framework and settings, so the comparison is fair.

A.6.5 Explanations on LLMs with an Enormous Number of Parameters

For all open-source models, we perform local deployment and inference. For larger models such as `DeepSeek-R1` and `DeepSeek-V3`, we deploy them on 16∗80GB GPUs and perform inference using SGLang[10] (Zheng et al., 2024). For closed-source models, we access them via API calls.

A.7 Some Clarifications

Whether LLM-generated data introduces bias and noise? Our pipeline minimizes bias as much as possible. (1) Length bias. The length of a number sequence is independent of the data: each term in the number sequence corresponds to one case in the algorithmic problem (Figure 1). We ensure that every problem contains exactly two example cases and 5 to 7 test cases, and this quantity is strictly controlled. (2) Pattern bias. Each number sequence corresponds to a distinct inductive pattern. We ensure that each pattern appears once in the training or test sets, preventing the base model or the test distribution from being skewed toward any particular pattern. What's more, we run all code in a sandbox unit test independent of the LLM, ensuring that every case executes correctly. This guarantees that the intermediate processes are noise-free and that the final ground-truth answers are completely accurate. Through the rigorous steps described above, we can ensure consistency between the process and the results, which allows us to: (1) guaranteeing that each problem is correct, (2) achieving rich diversity in problem samples (more than $2,500$ number sequence patterns with various sampling

[10] https://github.com/sgl-project/sglang

strategies), (3) avoiding superficial edits: the correctness of the final answers shows that the LLMs engage in a natural and proper reflection process, (4) retaining the LLM's natural reflection process, and training LLMs to generate cases automatically.

How to avoid overlap or overfitting? There is no overlap between the training and test sets. Each number sequence represents a distinct inductive pattern and corresponds to a unique ID; simply deduplicating by IDs is sufficient to prevent any overlap. Every data point in our SFT training set, RL training set, and GTG test set is unique with no repetition. As a result, the model encounters entirely new number sequences during the testing phase. The SFT stage does not lead to overfitting, then. As a result, the model encounters an entirely new number sequence pattern during testing.

Potential risks. Inductive reasoning carries certain risks. (1) Confidently wrong outputs: When performing induction, LLMs may 'hypothesize a pattern', generating a self-consistent but incorrect answer even when the data does not follow any true rule. (2) If the training data itself is biased (e.g., some patterns appear more frequently), the model may tend to produce these common patterns, resulting in systematic bias. However, in our work, we have taken steps to mitigate such risks as much as possible. First, the ground truth for all training data is verified using unit tests independent of any LLM, ensuring both answer correctness and process consistency. Each sample in our training and test sets is a unique number sequence, and we ensure that no number sequence is repeated. Moreover, each number sequence embodies a completely different inductive pattern, which helps to alleviate this bias.

Not distillation. This work does not directly distill powerful LLMs. Rather, it merely has LLMs generate algorithmic problems based on number sequences and produce reflection contents. The construction and validation of the synthetic data are carried out using human-designed rules and independent sandbox unit tests. The ablation experiments demonstrate that the data, the CoT design, and the algorithms themselves are the key factors for success. Another reason we do not directly distill larger LLMs is that their performance on number sequence problems is inherently limited. Table 5 shows the results of the three powerful reasoning models on the next number prediction task—a simplified but more difficult version of the GTG task—which reflects that even the strongest models can only achieve around 0.4 accuracy. Using powerful, closed-source LLMs to synthesize data is a standard practice in mainstream LLM training. But this is not one of our innovations, and our pipeline does not distill any LLMs.

The main contribution. Our main contribution lies in proposing synthetic data for inductive reasoning. It uses number sequences as the data source, injecting precise reflection mechanisms into the model, and implements difficulty control. Our method is built around the data, including the case-based reflection paradigm used in SFT and the CSSR reward function used in RL.

Why do we need to collect data from new sources? Some common inductive training data, like Li et al. (2025b), are constructed from common puzzles, lists, strings, etc. These datasets, as typical LLM reasoning sources, are likely to suffer from data leakage—that is, the model may have already seen similar content during training, preventing further improvement in capabilities. Therefore, Li et al. (2025b) cannot enhance OOD inductive reasoning ability (Table 15). The `Intern` series (Team, 2025), as well as the `Minimax-M1` (Liu et al., 2025) and `Seed-thinking` models (Chen et al., 2025a), all make extensive use of large amounts of toy-style inductive data. Moreover, numerous studies (Ruan et al., 2025; Bayat et al., 2024; Carlsson et al., 2025) have shown that training on previously seen samples can cause overfitting and harm a model's OOD capabilities. The above discussion further illustrates that, in order to improve a model's reasoning ability while maintaining OOD performance, it is crucial to construct high-quality data from new data sources. This highlights the effectiveness of our number sequence data and pipeline.

About data leakage. (1) There is no overlap between the training and test sets. Each number sequence represents a distinct inductive pattern and corresponds to a unique ID; simply deduplicating by IDs is sufficient to prevent any overlap. Every data point in our SFT training set, RL training set, and GTG test set is unique with no repetition. As a result, the model encounters entirely new number sequences during the testing phase. The SFT stage does not lead to overfitting, then. As a result, the model encounters entirely new number sequences during testing. (2) The results in Table 5 show that

SFT	**whole time**	**time per** 100 **step**	**weight convertion time**
	$7,860s$	$786s$	$30s$
RL	**whole time**	**val time portion**	**weight convertion time**
	$17,124s$	0.2	$70s$

Table 9: The training time for both the SFT and RL stages with `LLaMA3-8B`.

even the most powerful existing LLMs still fail to achieve high accuracy on the number sequence task, indirectly reflecting their weakness in inductive reasoning ability. This suggests that current models are unlikely to be trained on this type of number sequence data or are not trained well, and therefore, our approach indeed obtains relatively new data with properly designed algorithms.

A.8 Supplementary Information on the Experiments

A.8.1 Trained Models for Next Number prediction

We mention the next number prediction task in the Appendix above. Now, we also test our trained models `LLaMA3-8B` and `Qwen3-8B` on this task. Results are in Table 10. The experimental results show that our training data are also effective for this task. It is worth noting that our results are relatively lower compared to the Table 2. This is mainly because our training data are presented in the form of code, whereas the tasks in this table require the model to directly output numerical values. Meanwhile, the descriptions of code problems in the GTG task's training data contain hints for computing the general term. As a result, the setting here can be regarded as OOD, which further provides indirect evidence that LLMs have relatively weaker capabilities in expressing and manipulating pure numbers.

	next number prediction
`LLaMA3-8B`	8.0
w/ *CodeSeq*	33.0
`Qwen3-8B`	28.0
w/ *CodeSeq*	54.0

Table 10: Results of trained models on the next number prediction task.

A.8.2 The Training Time

We record the training time for both the SFT and RL stages with `LLaMA3-8B`. As shown in Table 9, although our model achieves significant improvements in both inductive reasoning and code reasoning, it requires no more than 5 hours of training, whether in SFT or RL, to reach optimal performance. This clearly expresses the effectiveness of our data and algorithm.

rollouts	32
temperature	$min(max_temp, 1.0)$
max-response-length	$min(max_len, 10 * 1024)$
rolling-timeout	10

Table 11: The settings for evaluating the GTG task on various models.

A.8.3 Explanations on Main Results

The term 'domain' in Table 2 refers to the number sequence task. 'in-domain' refers to the GTG task, which involves calculating the general term of a number sequence using code. Therefore, the evaluations on the other six benchmarks can all be regarded as OOD. However, since there is no overlap between the number sequence patterns in our training and test sets, our GTG task can also be viewed as 'out-of-pattern' to some extent.

In the main experiments, we use the same configuration for each benchmark to ensure fairness. For the three general inductive tasks and the three code tasks, we adopt evaluation code from their GitHub repositories and the default evaluation settings provided by OpenCompass separately. For the GTG task, we apply the settings in Table 11.

Moreover, both `DeepSeek-R1-7B` and `DeepSeek-R1-32B` models fail to produce valid predictions on the MBPP benchmark when evaluated using the same configuration. Therefore, to ensure fairness, we do not report their results.

A.8.4 Explanations on Different Training Settings

Different CoT When using different types of CoT for SFT training, we ensure that the number of training data is kept consistent. Figure 18 and Figure 19 demonstrate the CoT example for 'CaseEx' and 'NL', meaning providing the first several cases

with explanations in the CoT to enhance the models' ability to generate cases and just using natural language explanations instead of providing failed cases during the reflection process, separately.

Different Sovability Under the four settings, we ensure that the number of training samples is fixed at 1K. For 'HighSov', we randomly select samples with solvability between 0.7 and 1.0; for 'LowSov', we sample instances with solvability between 0.0 and 0.3; for 'Random', we randomly choose samples from the entire solvability range; and for 'CodeSeq', we sample instances with solvability between 0.0 and 0.46.

Different Reward Functions Below, we sequentially present the three reward functions we compare: 'Binary', 'PassRate', and 'NoLog'.

$$\mathcal{R}_{\text{Binary}} = \begin{cases} 1 & \text{if all cases pass} \\ 0 & \text{else} \end{cases} \tag{2}$$

$$\mathcal{R}_{\text{PassRate}} = 1 - \hat{Sov} \tag{3}$$

$$\mathcal{R}_{\text{NoLog}} = \begin{cases} -1, & \text{if formatted error} \\ 0, & \text{if test fails} \\ \lambda(1-\hat{Sov}) + & \\ (1-\lambda)\frac{N_{tc}}{N_c} & \text{if test passes} \end{cases} \tag{4}$$

Experimental results show that using only the pass rate leads to a volatile RL training process. Although a binary reward function can increase the final reward, the improvement is relatively slow because it does not provide fine-grained differentiation between different code solutions. 'NoLog' mode also can not get the best results.

Whether the reward we design would cause reward design flaws? There is a potential viewpoint that our designed reward function has a bias. If the number sequence is long, then the polynomial that fits it is essentially unique. In this case, there will only be one valid pattern. However, if the number sequence is short and we allow higher-degree polynomials, then many different patterns can all fit the given numbers. If the reward is based on the number of valid patterns, it will naturally favor situations where multiple patterns exist. That means the model is encouraged to get better at handling cases that are inherently ambiguous, since many different polynomials can explain them. At the same time, it becomes worse at handling cases where the sequence is longer and the polynomial is uniquely determined. In short, the reward design pushes the model toward ambiguous patterns rather than guiding it toward the correct and more meaningful ones. We can approach this question from two perspectives. (1) Most of our GTG samples do not involve polynomials. One of the reasons we package number sequences into algorithmic problems is that many of the underlying math rules cannot be intuitively expressed using mathematical formulas (such as polynomials), as in the introduction. We understand the bias described above, but the fact remains that most of our underlying formulas are indeed not polynomial-based. (2) If a number sequence has multiple patterns that satisfy it during rollout, we can consider that the number sequence has multiple code solutions. Therefore, for the LLM currently attempting the problem, it is a relatively easy task. One characteristic of inductive reasoning is that a unique answer does not necessarily exist, so this is not an issue. Consequently, in Formula 1, we assign a lower reward to this type of problem.

	in-domain	**OOD code tasks**		
Models	GTG	Heval*	MBPP	LCB*
`Qwen3-8B`	45.63	88.43	69.80	50.12
w/ GRPO	71.63	89.95	70.25	52.70
w/ DAPO	65.22	89.18	68.10	50.06
w/ DPO	50.09	87.65	67.71	48.87
w/ PPO	-	-	-	-

Table 12: Results on training with different RL algorithms on `Qwen3-8B`.

A.8.5 RL Methods

We primarily use the GRPO algorithm to train our RL data. Meanwhile, we also compare it with other RL algorithms, DAPO (Yu et al., 2025), PPO (Schulman et al., 2017) and DPO (Rafailov et al., 2024), to draw more convincing conclusions on the model `Qwen3-8B`. To quickly verify the effectiveness of each RL algorithm, we select GTG and three coding tasks that can be evaluated with one-click execution using OpenCompass.

As shown in Table 12, the PPO algorithm leads to model collapse, while the DPO algorithm performs worse overall compared to GRPO. For the DAPO algorithm, although the model can maintain OOD performance to some extent, its performance on the GTG task and the three code benchmarks is inferior to GRPO. Therefore, we ultimately chose the GRPO algorithm.

```
output:
{
"cot":
"
   [slot](one term in the number sequence)[slot]
   [slot](the explannation of the term)[slot]

   [slot](one term in the number sequence)[slot]
   [slot](the explannation of the term)[slot]

......
   [slot](one term in the number sequence)[slot]
   [slot](the explannation of the term)[slot]
",

"code":
   [slot](answer code)[slot]
}
```

Figure 18: An example of the 'CaseEx' CoT.

num_0	$1,045$
num_1	$1,141$
num_2	742
num_3	630
num_4	$1,110$
num_5	819

Table 13: The number of training samples corresponding to different numbers of reflection rounds among the SFT data.

For GRPO, we perform multiple training runs and report the average of the highest rewards. The confidence interval is approximately 0.025 (qwen for GTG task), which does not fundamentally affect our conclusions. We believe this variation mainly arises from differences in model rollouts. For the GRPO hyperparameters, we set the batch size and rollout number to fully utilize available GPU memory in order to maximize training efficiency. For λ, we sweep values from 0.5 to 1.0 and select 0.9, which can achieve the best results. For ϵ, we set it to 0.1 to prevent the reward from becoming excessively large when the solvability is very small.

A.8.6 Explanations on Reasoning with Cases

The case-based error-checking behavior in inductive reasoning is a natural behavior of people, and our data are used to train the model to internalize this habit. We prompt two LLMs to make cases themselves while producing code solutions, to verify the correctness of the code. This approach allows us to assess whether *CodeSeq* enables the model to learn how to generate correct cases. We append an additional natural language instruction to the prompt in Figure 16: "You need to ensure that the cases you examine do not come from the example cases", which allows us to evaluate the model's intrinsic emergent capability for case generation. We force the LLMs to generate at least one case. If all of the cases exist in the current number sequence, we consider it to be correct. If the LLMs fail to generate at least one case, it is prompted to regenerate until the output satisfies our requirements.

A.8.7 Explanations on Scaling Behaviors

Reflection Round Scaling We also ensure consistency in the number of SFT training samples. Specifically, within the final 5.5K SFT training dataset, we count the number of samples corresponding to different numbers of reflection rounds: num_0, num_1, num_2, num_3, num_4, and num_5. We then select the minimum among these as the SFT data size for our experiment. The statistics are in Table 13, and the final selected number is 630.

Number Sequence Pattern Scaling We also randomly select 300, 600, 900, and $1,200$ training samples (patterns) from *CodeSeq* for our experiments. We make sure that each of these training samples is just utilized for the SFT or RL stage.

output:
{
"cot":
"The Python code for this algorithm problem, based on my initial thoughts, is:
[slot](code1)[slot]

I don't think this code is correct, because: xxxxx
So, the revised solution code is:
[slot](code2)[slot]

I don't think this code is correct, because: xxxxx
So, the revised solution code is:
[slot](code3)[slot]
...

I think this is the correct solution.",

"code":
[slot](answer code)[slot]
}

Figure 19: An example of the 'NL' CoT.

	MIRAGE	*CodeSeq*
Data Type	rule-based	synthetic data
Training Method	SFT	SFT and RL
Prompt Formation	ICL	case-driven
Difficulty Control	No	Yes

Table 14: Comparisons between MIRAGE and our number sqeuence data *CodeSeq*.

	GTG	DEER	LCB*
`LLaMA3-8B`	9.27	36.05	11.26
w/ MIRAGE	10.33	36.10	10.10
w/ *CodeSeq*	44.22	40.15	16.14

Table 15: Comparisons between MIRAGE- and our *CodeSeq*- trained model on DEER and MMLU.

Therefore, the data in this part is obtained by deduplication using number sequence IDs.

A.8.8 The Advantages of Our Training Data

We compare the advantages of our training data with a recent, widely recognized inductive training corpus MIRAGE (Li et al., 2025b) in Table 14. The table shows that our data can not only be automatically synthesized, but also includes case-driven SFT data as well as difficulty-controlled RL data.

We train `LLaMA3-8B` with MIRAGE. Since MIRAGE is derived from ARC and ListFunctions, we do not evaluate the model on these two benchmarks, but on DEER. We also test its generalization performance on the OOD code task LiveCodeBench. Results in Table 15 demonstrate that our data not only improves LLMs' overall inductive reasoning abilities, but can also transfer to OOD scenarios.

A.8.9 The Advantages of Our Training Process

We compare the advantages of our training process with a widely recognized work. Wang et al. (2024b) (HS) is similar to our SFT data construction process, but their approach is more dependent on human intervention. Meanwhile, their evaluation tasks are limited, and they only use non-trainable models (e.g., `GPT4`). Our data not only includes SFT and RL data but also validates the performance of seven sub-tasks across different domains on trainable models. More importantly, the core of our work lies in ensuring the fully automated construction of training data that incorporates the correct reasoning process, with the goal of enhancing the fundamental inductive reasoning ability of LLMs—rather than merely prompting LLMs. We list the differences in Table 17.

	In-domain	OOD Inductive Tasks			OOD Code Tasks		
Models	GTG	DEER	ListFun*	CodeArc	Heval*	MBPP	LCBench*
`Qwen3-14B`	48.12	56.77	55.44	62.94	95.41	73.40	54.20
w/ *CodeSeq*	77.66	58.80	56.03	63.90	96.18	74.25	57.26
`Qwen2.5-Coder-7B`	30.11	41.68	33.23	35.35	88.41	78.99	44.57
w/ *CodeSeq*	70.05	42.39	38.15	40.60	89.93	79.55	45.60

Table 16: Results on training with a larger model `Qwen3-14B` and a reasoning model `Qwen2.5-Coder-7B`.

	HS	Ours
Human Intervention	Yes	No
Model Trainability	No	Yes
Multi-domain Validation	No	Yes
Practicality Validation	No	Yes

Table 17: Comparison of the training process between one previous work and our approach.

Model	**GTG** (200)	**GTG** $(1,000)$
`LLaMA3-8B`	9.27	8.30
w/ *CodeSeq*	44.22	44.89
`Qwen3-8B`	45.63	46.15
w/ *CodeSeq*	75.23	74.08

Table 18: Performance on GTG with different test set sizes.

A.8.10 Training with Larger Models and Reasoning Models

To further demonstrate the effectiveness of our data, we conduct the same training as in the main experiments on a larger model `Qwen3-14B` and a reasoning model `Qwen2.5-Coder-7B`. The results in Table 16 show that our training data are effective for the same model of various sizes and some specialized reasoning models. The results here are consistent with those in Table 2.

A.8.11 The representativeness of the test set

We expand the test set to $1,000$ test cases to cover more complex patterns and rerun the evaluation. The results are shown in Table 18 below, and the conclusions are consistent with those in the original Table 2: the *CodeSeq* training set indeed enhances LLMs' inductive reasoning performance on new number sequences. We can also see that the representativeness of these 200 test samples is sufficient. For fast and convenient evaluation on the GTG task, we choose to use a smaller one.

A.8.12 Falure Analysis

We conduct a statistical analysis and investigation on the cases where the Qwen model fails both before and after training.

We find that the majority of failures (about 60%) are due to the general term formulas of the number sequences involving compositional rules. For example, the number sequence 10, 3, 2, 2, 3, 3, 3, ..., is generated by a formula that combines square root operations, recursion, and modulo operations. Another portion (about 30%) of the number sequences contains hidden variables, where the sequence alone is insufficient to infer the rule, requiring reconstruction of the hidden state. For instance, the sequence 1, 1, 2, 4, 3, 7, 5, 12, 7, ... actually represents a robot moving in the directions right, down, left, and up in order, with each step's displacement equal to the sum of the previous two steps. The general term of the sequence corresponds to the robot's Manhattan distance (the hidden variable in this case) from the starting point.